TEST CASE SELECTION AND PRIORITZATION FOR OBJECT-ORIENTED SOFTWARE BASED ON SLICING AND COUPLING

UMAR FAROOQ KHATAK

A thesis submitted in
fulfilment of the requirements for the award of the
Doctor of Philosophy in Information Technology

Faculty of Computer Science and Information Technology
Universiti Tun Hussein Onn Malaysia

AUGUST 2022

DEDICATION

I dedicate my thesis to my parents for their support and prayers, despite the difficult circumstances they faced, which gave me the fortitude to overcome the hurdles entrenched throughout my academic challenges. This thesis is dedicated to my parents for their patience, support, and prayers for my achievement. I also dedicate this thesis to my supervisor and friends who helped me come up with the concepts for this research.

ACKNOWLEDGEMENT

In the name of Allah, the Most Generous, the Most Forgiving. I thank Allah for His many bounties in my life and for His willingness to help me finish this research. Without the instruction, assistance, and support of a large number of people, this dissertation simply would not have been conceivable to begin with. This is a chance to convey my gratitude and admiration to everyone involved.

I would like to thank my supervisor, Dr Aida Binti Mustapha, for her openness and encouragement. My thesis advisor, Dr Aida Binti Mustapha, was an encouragement to me while I struggled to complete it. She is a true leader and an excellent role model. Without her help from the beginning of the research process, I would not have been able to complete this thesis. I owe her a debt of gratitude for the wonderful opportunities she gave me for my professional growth and development. Having her as a teacher is a great honour. I would also like to express my gratitude to my co-supervisor Dr Zainuri, whose honesty and support will never be forgotten by me. I am grateful to him for the wonderful experiences he provided and for giving opportunities for me to develop professionally.

Thank you, Mom and Dad, for your unfailing love and support. You inspire me and give me self-assurance. My success and achievements are directly linked to their confidence in me. My deepest gratitude goes out to my siblings, who serve as a constant source of support and stability in my life and serve as constant reminders of the important things in life. During the process of writing my thesis and every day, I am grateful for the love and support I received from my family and friends. I will always be grateful.

ABSTRACT

Regression testing has become more prevalent with the increasing use of iterative development as software artefacts are reused in different software development projects. The objective of regression testing is to detect fault after the software is changed. It is done by reducing the amount of time required for the test cases to run, the number of test cases in the test suite, or the selection of test cases that have been previously run on the system under test. However, determining the suitable test cases in regression testing is challenging, especially when managing the retesting process within a limited budget and timeframe. To address this issue, this research proposes using program slicing and coupling metrics to improve the selection and prioritisation of regression test cases specific to the affected segments of the program. In order to determine these dependencies among the program parts, this research proposed an approach for regression testing, which generated a suitable intermediate graph for object-oriented programs. In this study, the scalability of intermediate graphs was significantly improved by reducing redundant edges approximately 4.1%. Next, this study proposes regression test case selection with Optimal Hierarchical Decomposition Slice (OHDS) strategy to obtain complete coverage information nodes for the affected slice graph. Once the impactful test cases have been selected, the test cases should be prioritised to enhance the ability of the retesting process to detect early errors. In this research, the coupling metrics are used to prioritise the test cases by using an export-import factor for the affected program parts. The evaluation strategy measured the Average Percentage of Fault Detection (APFD), and the experiments produced an increase of 2.8% in APFD value. This result showed that the test cases executed only on the affected portions identified as having a high degree of Export/Import coupling are likely to detect faults earlier than other test cases within the test suite.

ABSTRAK

Ujian regresi bertambah popular dengan peningkatan pembangunan perisian secara berulang dimana artifak perisian digunakan semula pada projek pembangunan perisian yang berbeza. Objektif ujian regresi adalah untuk mengesan perubahan yang telah dilakukan pada aplikasi. Ia dilaksanakan dengan mengurangkan jumlah masa yang diperlukan untuk kes ujian dijalankan, bilangan kes ujian dalam suite ujian, atau pemilihan kes ujian yang telah dijalankan sebelum ini pada sistem yang sedang diuji. Walau bagaimanapun, menentukan kes ujian yang sesuai dalam ujian regresi adalah tugas yang mencabar terutamanya apabila menguruskan proses ujian semula dalam anggaran dan jangka masa yang terhad. Untuk menangani isu ini, penyelidikan ini mencadangkan penggunaan penghirisan program dan metrik gandingan untuk menambah baik pemilihan dan keutamaan kes dalam regresi kes ujian khusus untuk segmen program yang terkesan. Bagi menentukan kebergantungan diantara bahagian program, penyelidikan ini mencadangkan pendekatan model seni bina bagi regresi pemilihan kes ujian yang akan menghasilkan graf perantaraan yang sesuai bagi program berorientasikan objek. Perwakilan graf perantaraan ini mengenal pasti semua potensi kebergantungan antara bahagian program. Dalam kajian ini, kebolehskalaan graf perantaraan telah dipertingkatkan dengan mengurangkan lewah sebanyak 4.1% untuk semua program tanpa menjejaskan semantik. Seterusnya, kajian ini mencadangkan pemilihan kes ujian regresi dengan Hirisan Penguraian Hierarki Optimum (OHDS) untuk mendapatkan liputan maklumat yang lengkap bagi graf hirisan yang terjejas. Sebaik sahaja kes ujian yang terkesan dikenal pasti, kes ujian akan diberi keutamaan untuk meningkatkan keupayaan proses ujian semula bagi mengesan ralat lebih awal. Dalam penyelidikan ini, metrik gandingan digunakan untuk memberi keutamaan kes ujian dengan menggunakan faktor eksport-import bagi program yang terkesan. Strategi penilaian mengukur Purata Peratusan Pengesanan Kegagalan (APFD) dan eksperimen menghasilkan peningkatan 2.8% dalam nilai APFD. Keputusan ini menunjukkan bahawa kes ujian yang dilaksanakan hanya pada bahagian yang terjejas yang dikenalpasti mempunyai gandingan eksport/import tahap tinggi dapat mengesan kesalahan lebih awal berbanding lain-lain kes ujian dalam sut ujian.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ACC	-	Affected Coupling Component
APFD	-	Average Percentage of Fault Detection
ASG	-	Affected Slice Graph
ASG-RTS	-	Affected Slice Graph Regression Test Selection
CIA	-	Change Impact Analysis
CIDG	-	Class Dependence Graph
COSDG	-	Call-based Object-oriented System Dependence Graph
DFG	-	Data Flow Graph
EI-WACC	-	Export/Import Weighted Affected Component Coupling
EI-ACTCP	-	Export/Import Affected Coupling Test Case Prioritization
EOOSDG	-	Extended Object-Oriented System Dependence Graph
ESDG	-	Extended System Dependence Graph
HRTS	-	Hierarchical Regression Test Case Selection
INS-FT	-	Import Nodes Selection with Functional and Transitive
JSDG	-	Java System Dependence Graph
LOC	-	Line Of Code
TCP	-	Test Case Prioritization
OHDS	-	Optimal Hierarchical Decomposition Slice
OOP	-	Object-Oriented Programming
PDG	-	Program Dependence Graph

RTS	-	Regression Test Case Selection
SDG	-	System Dependence Graph

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Overview

In order to satisfy the changing needs of the users and the increasing expectations of the customers, it has become essential for the software to evolve throughout a given time. In addition to increasing software complexity, there is also an increase in the cost and effort associated with its maintenance (Ba-Quttayyan *et al*., 2018). After modifying a program, regression testing should be performed to ensure that the changed component is valid and that the changes do not adversely affect other program components. Software maintenance has become increasingly dependent on regression testing. The need to make changes to a program that has already been tested cannot be overstated. Regression testing plays a significant role in the retest of the program. As a result of these modifications, regression testing was carried out without compromising the time and cost while keeping the same level of testing coverage. Thus, this research proposes a component slicing and coupling-based approach to establish the affected program parts or components. This approach will help to improve test case selection and prioritization.

1.2 Research Background

In the field of Software Engineering, certain methods and scientific principles are applied to design and develop software products. The development of a software product involves following certain processes and resulting in an authentic and efficient product. Software engineering can be defined as systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software (IEEE Computer Society, 2017).

Software maintenance is also part of the Software Development Life Cycle (SDLC), which maintains software that involves fixing and improving existing software issues, making the software compatible with new hardware and software requirements, and resolving complex issues (Tiky, 2016). In recent years, software development cost has increased compared to other component computer system project. In the same way, the cost of maintaining the software system keeps increasing (Ali *et al.*, 2020). IEEE Standard defined software maintenance as "the process of modifying a software system after delivery to correct faults, improve the performance or adapt it to a changing environment" (IEEE Computer Society, 2017). Mainly, software products go through changes in code and related documentation because of a fault or some improvement in the performance (Kaur & Singh, 2015). A significant part of what is spent on software production is spent on maintenance, which represents nearly 50% of the total cost (Rava &Wan-Kadir, 2016).

According to Ngah (2012), software states may change and advance over time, and any software development project that does not require modification is impossible. There is a need to retest the software system to validate these modifications to find whether the software behaves as intended. This process of selective retesting is referred to as regression testing (Chauhan, 2010). Regression Testing (RT) is a retest activity to ensure that system modifications do not affect other parts of the system and that the unchanged parts of the system are still working as it did before a change (Minhas *et. al*, 2020).

By selecting a subset of the existing test suite that is relevant to regression testing, the cost of regression testing can be reduced. Regression Test Selection (RTS) techniques are primarily meant to reduce regression test costs and increase fault detection possibilities (Musa *et al.*, 2014). Instead of testing all of the program's components to verify that every change is valid, it is preferable to select test cases that cover the aspects of the programme that have been modified. In test case selection, the issue is to select a subset of test cases that can be used to test the parts of the software that have been modified. In addition to Regression Test Selection (RTS), the Test Case Prioritization (TCP) approaches aim to determine the order in which test cases should be executed to maximize the detection of defects at an early stage during regression testing (Elbaum *et al*., 2004). Test suite prioritization seeks to discover faults as early as possible within the system under test by reordering test cases. However, rather than reducing the number of tests in the suite, it simply rearranges them according to fault-detection capabilities.

According to Mössenböck (2012), OO programming can be defined as programming with abstract data types (classes) employing inheritance and dynamic binding. With OO programming, the complexity shifts from interactions between methods to object relationships and communications among objects. The dependencies among the program parts play a vital role in detecting the critical parts of the program during software maintenance (Panda, 2016). Thus, it is crucial to analyse the dependencies between the different programming constructs and identify these critical elements in the programs. Most system dependence graph-based slicing techniques for analysing interdependencies between various programs have been used with partially object-oriented C++. It does not have such features as dynamic methods, static method dispatch, interfaces, exception handling, and multi-threading, which are present in Java language, thus making maintenance even more challenging (Shu *et al*., 2013). In OOP, one of the fundamental concepts is coupling and its measures are proven to strongly correlate with fault-proneness (Meyers & Binkley, 2007).

The main goal of regression testing is to improve the effectiveness by increasing the rate of fault detection and identifying change specific faults. The regression testing process is recognized as part of the validation process and poses many challenges in testing the software. The two basic challenges in regression testing are selecting relevant test cases and test case prioritization (Minhas *et al*., 2020). It is challenging to manage retesting in terms of time and cost, especially when an extensive test suite is involved (Musa *et al*., 2015). Selecting test cases based on the source code is an evident approach to determine which tests are suitable because of the modifications made to the program (Musa *et al*., 2014). The primary cause of regression testing is change. Regarding the frequency of regression testing, 53.3% of the organizations repeat regression testing for every new version of the product, and 28.9% reported regression testing after every change (Minhas *et al*., 2020). Program slicing is one of the top three techniques for regression testing (Kazmi *et al.*, 2017), which identifies affected parts based on change analysis.

1.3 Problem Statement

In the software development life cycle (SDLC), software maintenance and evolution is the process that involves fixing and improving existing software issues (Ogheneovo, 2013). This allows the software to be compatible and adapted to meet new hardware and software requirements and meet user expectations. As software develops through a series of changes, it is necessary to perform regression tests to validate the changes. In order to perform regression testing, it is necessary to identify what parts of the program would be affected by any changes made to them as part of the maintenance process. In software maintenance activities, regression testing is crucial to ensure that bug fixes or enhancements do not impair the current functionality and the original design requirement (Kaur & Singh, 2015).

In regression testing, the tester ensures the program is not affected by any additional problems by using one of the most straightforward methodologies available (Orso *et al.*, 2003). Despite its advantage of being the safest method, it can only be used in the test suite that is relatively small. All test cases may be randomly selected to reduce the size of the test suite; however, most randomly selected test cases to provide the same results as unrelated test cases or have nothing to do with the modified programme. A subset of existing tests relevant to the testing process is selected to reduce the cost of regression testing in a regression test selection strategy. A primary purpose of RTS is to decrease regression testing costs and maximize potential fault detection (Musa *et al.*, 2014). In the ideal case, the subset of tests is intended to identify the same number of errors as the original test suite with less effort (Yang *et al.*, 2014).

In addition, dependencies among the program parts play a vital role in detecting the critical parts of the program during software maintenance (Panda, 2016). A graphical representation of the program is essential to determine these dependencies among the program parts. In existing studies by Panigrahi & Mall (2013) and Musa *et al.* (2015) used the coverage information of affected nodes for object-oriented programs via the Extended System Dependence Graphs (ESDG) to graphically represents the internal structure of the Program under Test (PuT) of Java program in RTS. However, it may not capture the exact structure of all the possible dependencies among the program parts, as it only represents six edges.

An architectural model for selecting regression tests based on hierarchical structures has been previously proposed using the Extended Object-Oriented System Dependency Graph (EOOSDG) for Java program structural representation (Panda, 2016). However, graph representation for OO programing causes cost redundancy problems with a large number of edges redundant, which have to be filtered by some approaches like the transitive technique (Panda, 2016). The transitive technique used was able to reduce the cost of redundant edges but caused information loss around the edges, which may have a semantic effect on other nodes. The semantic effect refers to the fact that the type of redundant edge selection for removal edges is unknown, which results in an incomplete edge representation among nodes.

The literature has shown that the techniques used to identify the affected parts of the OO programs plays a critical effort in defining the real coverage information. The program can select the correct subset of the test suite. Musa *et al.* (2015) proposed that the forward slicing approach used the coverage information of affected nodes to select test cases. However, the coverage information for the affected nodes was incomplete. Panda *et al.*, (2016) also used the forward/backward slicing approach in RTS via hierarchical slicing. However, due to dependency restriction on forwarding slicing, this approach cannot detect the complete coverage information for the affected part of the program. Furthermore, the additional sub-edge comparison in backward slicing increases the effort to select affected nodes.

Given these scenarios, it is always a challenge for software testers to improve detecting faults. To the least of effort, prioritising regression test cases is necessary to detect faults early in the retesting process (Campos *et al*., 2017). In Test Case Prioritization (TCP), the order in which the selected regression test cases are performed optimized error detection rates at a lower cost and time. It has been found that regression TCP is a topic that is often discussed for procedural programs but is limited for OO programs (Farooq *et al*., 2019). A TCP strategy aims to achieve some performance goals, such as detecting faults early by finding a suitable order to execute each test case in a test suite. In the literature, regression test cases are prioritized based on change impact analysis using a slice approach for OO programming, including export coupling (Panda *et al*., 2016). However, it does not include another significant coupling, the import coupling, which does not explicitly define the affected parts of the program.

1.4 Research Questions

The research background has provided sufficient context that leads the following research questions:

(i) How to make the graph representation scalable for Object-Oriented Programming (OOP) without losing the semantic effect of the edges?

(ii) How to identify the affected part of the program more efficiently i.e. coverage information?

(iii) How does the Export/Import couple influence the Test Case Prioritization (TCP)?

1.5 Research Objectives

In order to achieve a comparable rate of fault detection and have confidence in the quality of the software, this research proposes an approach for regression testing, in particular the Regression Test Case Selection (RTS) and Test Case Prioritization (TCP) in Object-Oriented (OO) software that focuses on reducing the execution of existing tests based on slicing and coupling approach. In addition, this research proposes to develop a mechanism to help testers decide which changes in a program need immediate attention, thereby reducing regression testing time. In order to accomplish this broad purpose, the following objectives are to be undertaken.

(i) To analyse and identify the graph representation for java structure and reduce the semantic effects of the edges

(ii) To enhance the technique for identifying and selecting the affected part of the program using the forward/backward slicing approach for regression test case selection.

(iii) To design algorithm using export/import factor coupling metrics to prioritise test cases.

1.6 Research Scope

This research has the following scopes:

(i) Based on slice-based affected nodes, this study examines regression test selection and test case prioritization at the code level of object-oriented programs and the coverage information derived from the source code.

(ii) Two adequacy criteria, all-nodes and all-edges, have selectively been used in this study for the experimental programs.

(iii) This research's experimental proposal has taken ten benchmark programs from Software-artefact Infrastructure Repository (SIR).

(iv) This research focuses on OO programs written in Java, a programming language; therefore, it does not consider programs created with other languages such as C and C#.

(v) This research does not include an empirical analysis of the prioritization time and will be addressed in future work.

1.7 Thesis Organization

This thesis is arranged in six chapters as follows:

Chapter 1 introduces the thesis. It provides background information, the problem statement, the research objectives, the scope of the research, and the contribution of the research.

Chapter 2 presents the reviews on the current status of the regression testing. It presents the fundamentals and terminology of Object-Oriented (OO) programming, an overview of OO software testing techniques, and a detailed study of existing regression testing techniques for OOP.

Chapter 3 presents a general overview of the research procedure and materials used to define regression test selection. Test case prioritization techniques for OO programs based on affected parts of the program and implement the prototype tool support of the proposed technique have been briefly described in this chapter.

Chapter 4 presents the implementation of the proposed regression test case selection and test case prioritization technique and an illustrative example of how the proposed technique selected and prioritized the tests.

Chapter 5 presents the experimental results of the empirical evaluation and provides them analyses and interpretation.

Chapter 6 covers the summary and conclusions of this work with future directions.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Whenever a program is developed to implement an algorithm or logic, its developers are always concerned about its performance and correctness. The developers must be certain that the software satisfies a certain quality level. Program testing may be done to assure a specific level of software quality. Software testing is defined as executing a program to find errors (Myers *et al*.,2011). In order to assure quality, the software has to compute results for the entire domain of input, with all the results that it computes specified. Therefore, software needs to be thoroughly tested to validate the input domain. Software testing approaches can only imply the existence of faults but are not able to demonstrate their absence if the processes are not exhaustive. Homes (2013) stated that exhaustive testing is not possible due to the following reasons:

(i) There are too many possible implementation paths in a program, so the difficulty alluded to by this assertion is made worse because certain execution paths could fail (Homes, 2013).

(ii) Design and specifications are subject to change during software development; therefore, testing is difficult since software testing is an algorithmically insoluble problem, and specification errors can lead to serious design errors (Chauhan, 2010; Jorgensen, 2013).

Orso & Rothermel (2014) defined the aims of a software testing approach, but Dalal & Chhillar (2012) highlighted its limitations including predefined testing time not allocated when testing phase begins, 100% testing not possible in case of complex systems, lack of formal testing as well as reviews at requirement and design stage, lack of formal unit testing methodology, Lack of efficient and effective automation testing as selection etc. Ruthruff & Rothermel (2010) observed that defects were always undesirable; therefore, a trade-off has always existed between exhaustive testing and computation costs. As a result, no testing method can be completely accurate and applicable to all programs. Although various restrictions limit the testing process, the constant and intelligent application of a testing approach can offer a sufficient level of software quality. The cornerstones of testing techniques are verification and validation. Static testing involves verification without code execution, while dynamic testing involves verification and code execution.

The hierarchical breakdown of testing techniques and their connections with various test adequacy criteria is depicted in Figure 2.1 based on the definitions by Chauhan (2010) and Homès (2013). This thesis involves the use of execution-based testing. There are three types of execution-based testing techniques: program-based, specification-based, and a merger of both as shown in Figure 2.1.

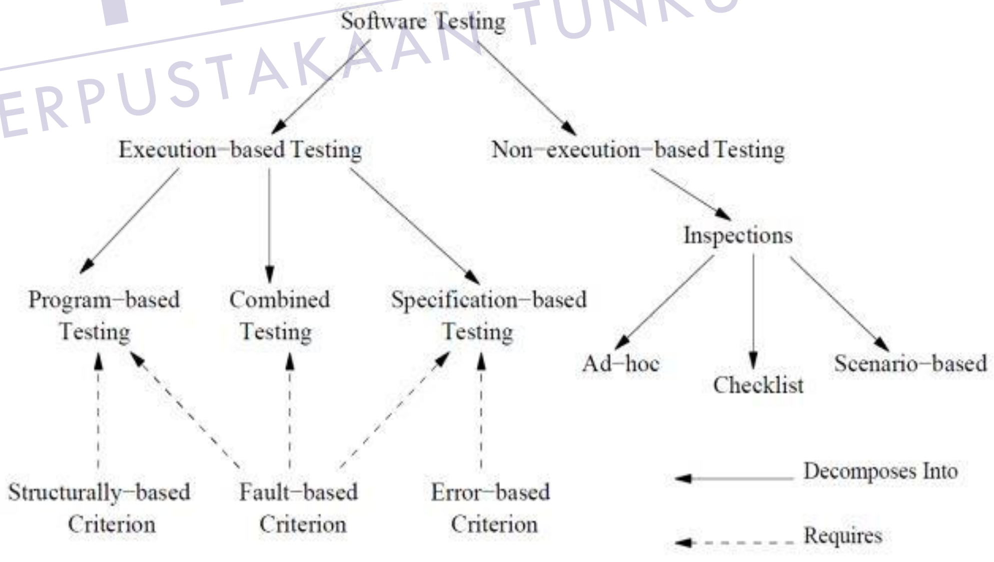


Figure 2.1: A Hierarchy of Software Testing (Chauhan, 2010; Homès, 2013)

Let *C* define the kind of test cases included in the test suite *T*. *T* is produced by evaluating the source code of a programme, *P*, based on its structure and characteristics, using a program-based testing approach. In order to create the required test suite, a specification-based testing approach utilizes *P*'s functional or non-functional specifications. By contrast, integrated testing utilizes both program-based and specification-based procedures to generate *T*. The test cases techniques are divided into three categories based on the testing approach used to generate them:

(i) Black box test cases are developed without knowledge of *P*'s source code and are entirely based on functional specifications. As a result, the cases' input and output behaviour are unaffected by *P*'s underlying structure. Black box testing is typically performed using two methods: boundary value analysis. and equivalence class partitioning

(ii) White box test cases are based on heuristics and are written utilising *P*'s whole source code. In order to carry out unit testing effectively, this approach is essential. There are various types of white box testing, such as data flow-based testing, fault-based testing, and coverage-based testing.

(iii) Testing grey box scenarios involves only using the design models of programme *P* as a basis. Class diagram-based testing, State-model-based testing, use case-based testing, and sequence diagram-based testing are examples of grey box testing.

As shown in Figure 2.1, different testing techniques are also linked to different test adequacy criteria. Following a structurally based criterion, *T* must cover specific control structures and variables within *P*, such as path coverage, statement coverage, condition coverage, branch coverage. Implementing a structurally based test adequacy criterion requires using a program-based testing approach. The fault-based test adequacy criteria ensure that *T* identifies the mistakes that programmers frequently introduce into *P*. Finally, error-based testing is based on the assumption that *T* does not deviate from the requirements in any way. As a result, error-based adequacy requirements inspire specification-based testing techniques.

The primary objective of software testing is to detect faults and errors in software before it is published, therefore improving software reliability. According to Myers *et al.* (2011), over half of the elapsed time and more than half of the total expense were spent testing the software or system being created in a typical programming project. Retesting the product gets more expensive and time-consuming when software projects are modified throughout the maintenance phase. An important aspect of software maintenance is regression testing, which guarantees that modifications do not adversely affect the software's correctness.

2.2 Regression Testing

Regression testing is included in the validation process and looks to be a significant problem in software testing. Managing the retesting process in terms of time and cost becomes a fundamental problem, particularly as the test suite expands in size. Changes to a software system may include bug fixes or the addition or removal of functionality. Regression testing is recognized as a critical component of software development. The practice of regression testing is defined as the process of thorough testing a system or component to ensure that modifications have not resulted in unintended consequences and that it still fulfils the requirement (Chauhan, 2010). Generally, a system is said to regress when (1) a new component has been introduced or (2) a modification made to a current component influences other aspects of the software. As a result, it is essential to retest the modified code and any possibly impacted code resulting from the change.

Regression testing is a costly task that often accounts for 50% of all software maintenance costs (Sandeep & Solanki, 2018). It is often expensive to conduct regression testing due to executing the test suite. Completion of regression testing of software containing 20,000 lines of code, according to Dahiya & Solanki, (2018), takes roughly seven weeks of continuous running. This involves creating various strategies to improve regression testing efficiency including test cases selection, minimization and prioritization. It is critical to reduce the expense of retesting software by identifying and retesting only those elements of the programme that are affected by the change. Orso and Rothermel (2014) have identified two major challenges in selective regression testing: (a) determining which existing tests must be rerun because it may exhibit different behaviour in the changed programme, and (b) determining which programme components must be retested to meet some coverage criterion. Thus, following Orso & Rothermel (2014), two problems can be extended as a process consisting of the following steps: Selecting a set of test cases *T* to be executed on a program *P*,

(i) Selecting *T*' ≤ *T* and retesting *P*' with *T*' to establish the correctness of *P*' with respect to *T*', where P' is the modified version of program *P*.

(ii) Creating *T*'', a set of new test cases for *P*', if required, and retesting *P*' with *T*'', so that still get the exact correctness of *P*' with respect to *T*''.

(iii) Creating *T*''' from *T*, *T*', *T*'' and adding some new test cases, if required, to test the correctness of *P*'.

During regression testing, the following significant issues are addressed, which are (1) test suite execution, (2) regression test selection, (3) coverage identification, and (4) test suite maintenance. The following approaches may be used to address the problem of software regression testing (Chauhan, 2010), as illustrated in Figure 2.2.

(i) Retest all approach: To test the updated version of the software, all test cases in the test suite are performed. The updated programme P' is effectively covered by test suite T.

(ii) Regression test selection: This technique reduces the time required to retest a changed programme by selecting a subset of the given test suite. Regression test selection techniques attempt to identify just those test cases that can exercise the modified sections of the programme and the parts affected by the alteration to decrease the cost of testing.

(iii) Test case prioritization: Prioritizing test cases is concerned with changing the order in which these are run. The test cases in a given test suite are organised according to rules. The higher-priority test case is executed first, followed by the lower-priority test case.

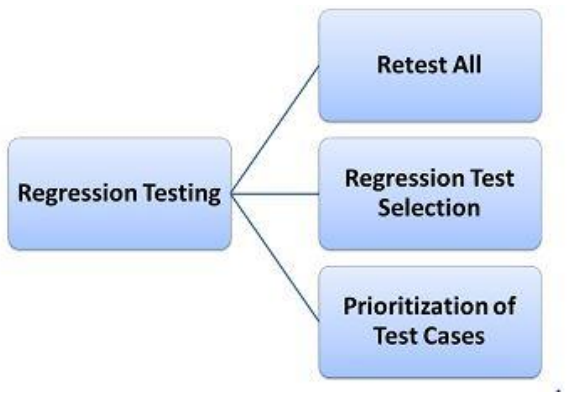


Figure 2.2: Regression Testing Approaches

Testing is a time-consuming and labour-intensive task accounting for over half of the development cost, with software maintenance accounting for the other 80% (Surendran *et al*.,2016). The most significant problem among the many stages of software testing, such as planning, designing, and execution, is designing and regression testing. During practical structural testing, testers are confronted with several obstacles (Mohanty *et al*., 2017). One key concern is the unrestricted size of source code, which impacts the scalability, consistency, and integrity of software systems during regression testing (Mansor & Ndudi, 2015). During such scenarios in structural testing, the supplied problem can be reduced into a reasonable number of sub-problems utilising the divide and conquer technique (Orso *et al*.,2001). This is related to Weiser’s concept of programme slicing, which he developed in his PhD thesis in 1979 (Singh & Singh, 2014).

2.3 Program Slicing

The size and complexity of software get harder to understand, maintain and test (Singh *et al*., 2014). Multiple software maintenance studies show that around half of the time is spent understanding the program code that is supposed to be maintained. So, the aim is to simplify the program code for better understanding, and the approach is to break the code into smaller pieces. Program slicing is a technique for extracting the portions of a programme relevant to a particular calculation (Alokush *et al*., 2018).

Consequently, a program slice is a set of statements that modify a variable at a particular point in time. Program slicing is a method for automatically dissecting programmes by analysing its data and control flow relationships beginning from a subset of the behaviour (Baber *et al*., 2020).

The first step in slicing a programme is to specify a point of interest, also known as the slicing criterion, represented as (*s*, *v*), where *s* is the statement number and *v* is the variable used or defined at *s*. Several scholars have made contributions to the field of programme slicing during the last few decades. Since Weiser introduced programme slicing as a debugging tool in 1984, other techniques have improved efficiency, precision, speed, and the usefulness of programme slicing for different purposes (Ngah & Selamat, 2014). Program slicing has been applied to both unstructured and structured programs and Object-oriented, Aspect-Oriented, and Feature-Oriented programs (Sasirekha *et al*., 2011). Furthermore, comparable slicing methods have been utilised to tackle various issues.

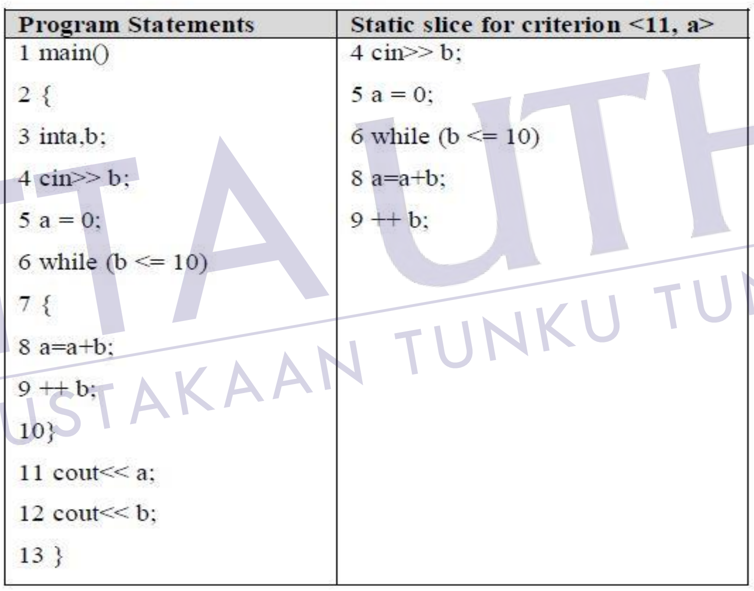
2.3.1 Types of Program Slices

The two basic forms of slicing criterion are static and dynamic slice, whereas the two main types of slicing direction are forward and backward slice. These slicing approaches are discussed in this section.

2.3.1.1 Slicing Criterion

In static slicing, program is analysed statically (without execution) for computing slices. It can be approached in terms of dataflow equations, information-flow relations, dependence graph or graph reachability (Singh *et al*., 2014). Weiser's slicing approach is based on the iteration of dataflow equations. The slices are computed using an iterative process by calculating consecutive sets of relevant variables for each node in the control flow graph. Any input value can be utilised with a static slice. *C* = (*x*, *y*) denotes a static slicing criterion, where x denotes a programme statement, and y denotes a subset of programme variables. In Figure 2.3, which is an example programme, shows the static slice condition as <11, a>. The end outcome is a collection of statements <4, 5, 6, 8, 9>.

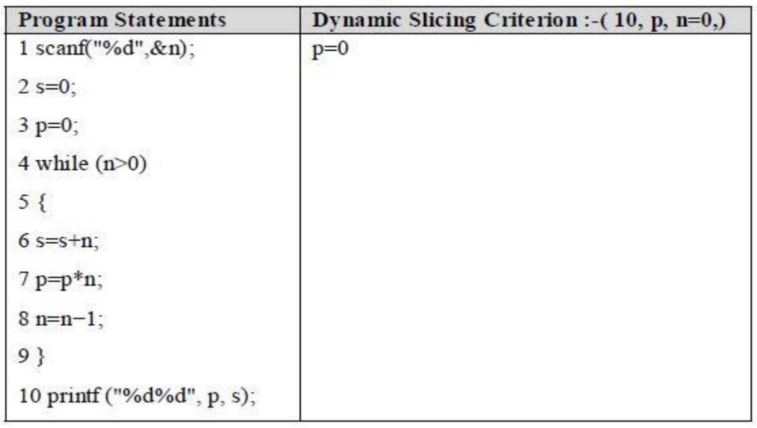
Tracing errors may be difficult since static slices include all possible executions. To address this, the notion of dynamic slices was created. Korel proposed the notion of the dynamic slice in 1988 (Sikka & Kaur, 2013). Dynamic slice being created for distinguished input variable is more précised over static slice and overcome the problems faced by static slicing in arrays and pointers of not knowing the result information about a specific element of an array because of the array being treated as a single variable in static approach (Sikka & Kaur, 2013). Dynamic slices are programme statements that only alter the slicing criterion for a single input run.



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 }	

Figure 2.3: Static Slice

Next, Figure 2.4 shows a sample programme to be sliced. The variable for which slicing is to be performed is *p*, the slicing point is the program's finish, and the input supplied is *n* = 0.



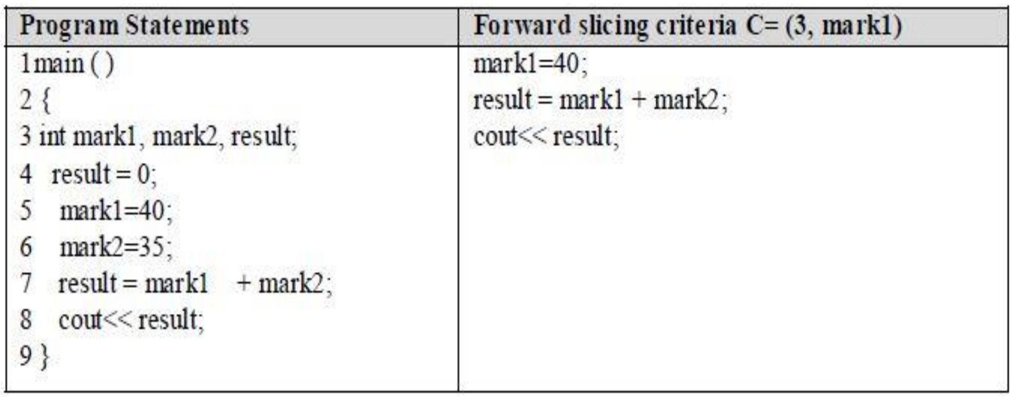
Program Statements	Dynamic Slicing Criterion :-(10, p, n=0,)
1 scanf("%d",&n);	p=0
2 s=0;	
3 p=0;	
4 while (n>0)	
5 {	
6 s=s+n;	
7 p=p*n;	
8 n=n−1;	
9 }	
10 printf ("%d%d", p, s);	

Figure 2.4: Dynamic Slice

2.3.1.2 Slicing Direction

Slicing direction can be forward or backward. Forward slice covers any parts of a programme that are possibly affected by the slicing requirement since it rely on it. This technique helps in knowing the effect of modifications in a part of the program on other parts. This technique was first time used by Reps and Bricker in 1989 and defined as: "A forward slicing of a program with respect to a program point p and set of program variables V consists of all statements and predicates in the program that may be affected by the value of variables in V at p" (Singh & Singh, 2014). In the forward slice, the programme is traversed in the forward direction. The forward slice lists all the programme statements impacted by the slicing criterion. *C* is an abbreviation for the letter *C* (*x*, *y*), *x* is the statement number, and *y* is the slice variable. Examine the sample programme in Figure 2.5. *C* = stands for the forward slicing criteria (3, mark1). The slice contains all of the statements in the programme that are impacted by the variable 'mark1' defined in statement number 3.

Meanwhile, the backward slide includes all aspects of a programme that may impact the slicing criterion due to the reliance on those parts. Weiser defined backwards slicing as: "A backward slice concerning a program point p and set of program variables V consists of statements and predicates in the program that may affect the value of variables in V at p" (Munjal, 2015). It contains the statements and control predicates of the program having some effect on the slicing criterion.

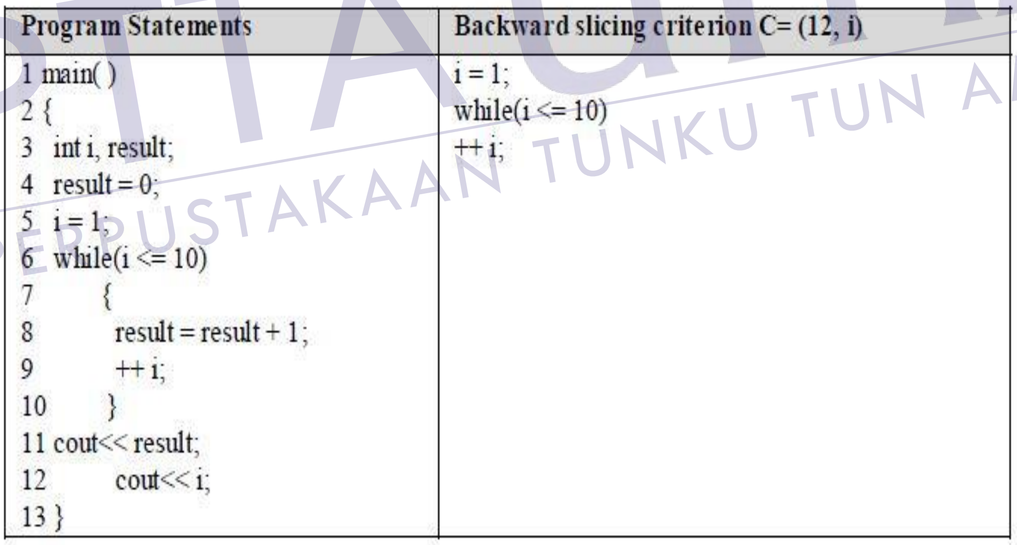


Program Statements	Forward slicing criteria C= (3, mark1)

1 main ( )
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;

Figure 2. 5: Forward Slicing

Note that *C* = (*x*, *y*) is the backward slicing criteria. The statement number is '*x*', and the slice variable is '*y*'. Take a look at the example programme in Figure 2.6 where *C* = (12, *i*). All programme statements that affect the value of the variable *i* in statement number 12 are shown.



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;

Figure 2.6: Backward Slicing

2.3.2 Slicing in Java Programing

Object-oriented (OO) programme slicing has distinct challenges than procedure-oriented programme slicing. Classes, dynamic binding, encapsulation, inheritance, message passing, and polymorphism are special OO programmes that require extra attention since it create new dependencies between instructions (Alokush *et al*., 2018). Although these features benefit OO programmes, it may affect slice precision.

Many researchers, like Allen & Horwitz (2003), Chen & Xu (2001), Hammer & Snelting (2004), Li *et al.* (2004), and Wang & Roychoudhury (2004), have proposed different approaches for calculating Java programme slices. A number of the slicing mechanisms are based on dependency graphs, such as Program Dependency Graph (PDG) and System Dependency Graph (SDG), while others are based on an analysis of Java bytecode. To overcome the restrictions and improve the efficiency of OO slicing techniques, Kovacs *et al.* (1996) developed a static inter-procedural slicing of Java programmes. This approach focuses on expressing particular Java features to increase the efficiency of the slicing process (Chandra *et al*., 2015). The suggested slicing approach supports static variables, multiple packages, and interfaces. It also enhanced the program's SDG by providing polymorphic calls that eliminated the need for extra nodes.

Chen *et al.* (2001) created a novel approach for visually presenting the OO Java programme. Alokush *et al.*(2018), discussed the many dependencies that might occur in a Java programme and recommended class slicing depending on the programme dependency Graph (PDG). In this technique, the programme dependency graph comprises a collection of separate PDGs.

Based on this new model for representing programs, Chen *et al.* (2001) introduced the concept of class slicing, partial slicing and object slicing. Allen *et al.* (2003) utilized SDG to improve on the work of Chen *et al.* (2001) on programme slicing (Panda & Mohapatra, 2013). In this article, they proposed programme slicing to be applied in the presence of exceptions. Because the programme had try-catch and throw blocks, the emphasis was primarily on establishing control and data dependencies. Other Java-specific features (including template classes, interface, super, and polymorphic calls) have not been tested for slicing.

Wang *et al.* (2004) proposed a compressed byte-code tracing method for slicing Java programmes. Wang *et al*., (2017) extended his work to represent the byte-code associated with an execution trace for a Java application. Next, it reverse-engineered the execution route to discover the control and data dependencies on the slicing criterion. In order to implement this approach, a trace table must be constructed for each technique. This way of computing slices is inefficient if a programme has too many methods. This is due to the increased execution cost involved in maintaining execution trace tables.

Li *et al.* (2004) developed a hierarchical slicing approach for slicing Java programmes. The slicing algorithm is implemented using the JATO tool. The hierarchical slices are computed level by level, beginning at the package level and ending at the statement level (Li *et al*., 2013). Work by Li *et al.* (2004) resulted in a level-by-level graphical representation of object-oriented programmes at various levels of programme organisation, including package, class, method, and statement. The four distinct graphs generated at each level are the Package Level Dependence Graph (PLDG), Class Level Dependence Graph (CLDG), Method Level Dependence Graph (MLDG), and System Level Dependence Graph (SLDG). To achieve hierarchical slicing, it is necessary to develop four distinct slicing criteria, one for each level of hierarchy. A graph traversed from package level to statement level produced imperfect slices.

All of the primary research, including Allen & Horwitz (2003), Chen & Xu (2001), Hammer & Snelting (2004), Li *et al.* (2004), and Wang & Roychoudhury (2004), advocated slicing Java programmes by taking a specific characteristic or kind of dependence inherent in a Java programme into consideration. The total impact of features on dependencies, such as dependencies caused by packages and other particular Java features, is ignored. The proposed technique went great to analyse all possible dependencies in OO programmes and calculate a more exact slice. Slicing is one of the way to do regression testing. First, it is needed to determine which statements are affected by the new statement and which might be affected by the change. However, most methods currently available are based on forwarding or backward traversal. In this study, the proposed approach was more effective for regression testing because of the following factors: forward and backward slicing, which may both be used to gather effective change impact information, since it identifies precisely those program parts that may be affected or get affected by the change. The input that the slicing algorithm takes is usually an intermediate representation of the program under consideration. Normally, the intermediate representation of the program under consideration is a dependency graph (Selamat & Ngah, 2017).

2.4 Program Graph Representation

Based on the literature, it is suggested not to evaluate the entire source code for efficient and effective regression testing but instead to focus on the parts of the code failing at a particular execution point (Fang *et al.*, 2014). In order to determine those high chances areas, there is a need to construct an intermediate representation that identifies precisely the program's conditions. In addition to pseudocode, flowcharts, high-level source code, a collection of instructions in the memory of a computer. Each of these representations serves a different purpose, depending on the context in which it is used. The use of different representations may be required, for example, to aid human reading, the annotation for verification, and transformation for the processing of a program on multiprocessors and distributed computers. About program slicing, program representations are employed to aid in effective slicing automation (Mall & Kumar, 2004).

A program graph is a graphical representation of a program source code consisting of a combinatorial structure composed of vertices (also known as points or nodes) linked by edges (Arora *et al*., 2012). Consider the vertices to be the destinations and the edges to be the paths. A graph can illustrate the flow of control between statements in a program. However, a dependence graph displays the characteristics and dependencies of the program across numerous objects. In order to discover interactions and relationships among program parts, it is crucial to graphically model the program under test using an intermediate graph representation (Mohapatra *et al.*, 2004; Walkinshaw *et al*., 2003).

Among program graphs that exist include the Data Flow Graph (DFG), Program Dependence Graph (PDG), System Dependence Graph (SDG), Extended System Dependence Graph (ESDG), and Call-based Object-Oriented System Dependence Graph (COSDG). Figure 2.7 shows the graph representation for a computer program, consisting of three main types – the Control Flow Graph, the Program Dependency Graph, and the System Dependency Graph.

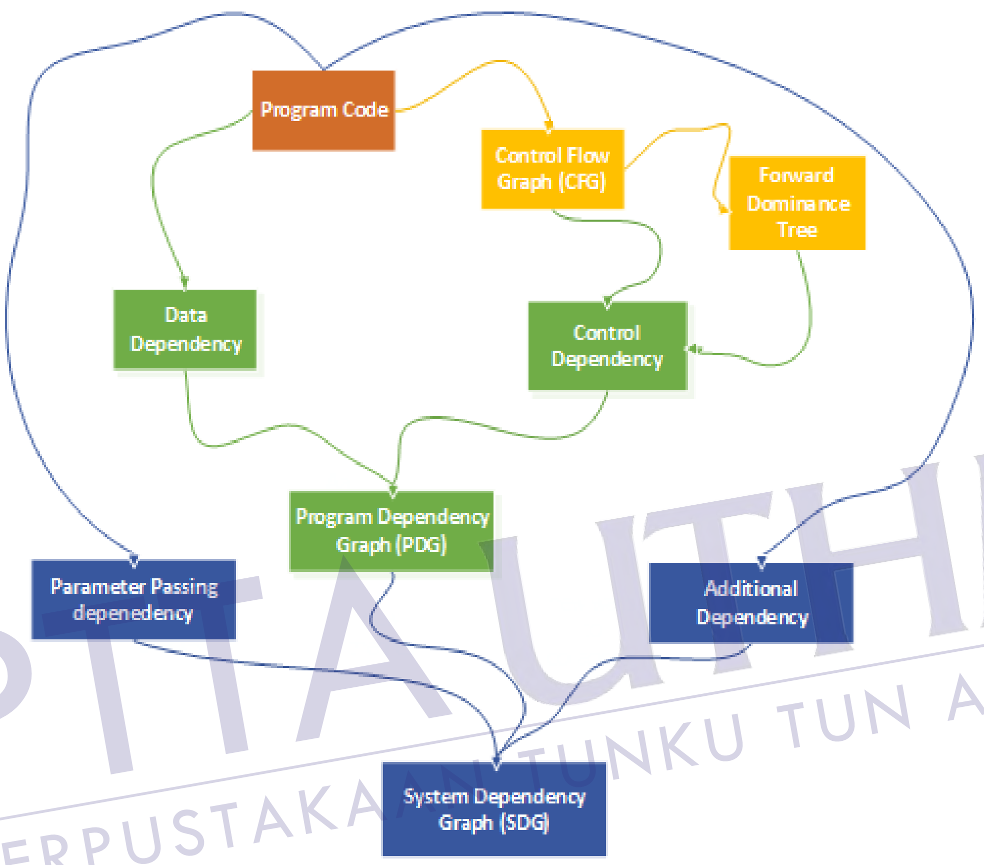


Figure 2.7: Program Presentation using Graph

2.4.1 Control Flow Graph

A Control Flow Graph is a graphical representation of control flow or computation during the execution of programmes (CFG). A Control Flow Graph (CFG) represents the program with nodes and edges from the start node to the end node. A CFG represents the control dependencies of the program (Ngah *et al.*, 2014). The control flow graph (CFG) is an intermediate programme model that may be utilised for data flow analysis and code optimizations such as common sub-expression elimination, copy propagation, and loop invariant code mobility. It is simple to encapsulate data for each fundamental block. A control flow graph may quickly find inaccessible portions of a programme and uncover syntactic patterns such as loops. Consequently, the control flow graph comprises all flow diagram components, such as the start node, end node, and flows between nodes.

```
int a, b, sum;
1.  read(a);
2.  read(b);
3.  sum = 0;
4.  while(a<8)
5.      sum = sum + b;
6.      a = a + 1;
7.  write(sum);
8.  sum = b;
9.  write(sum);
```

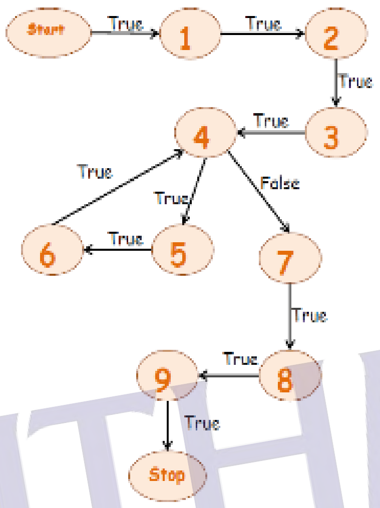


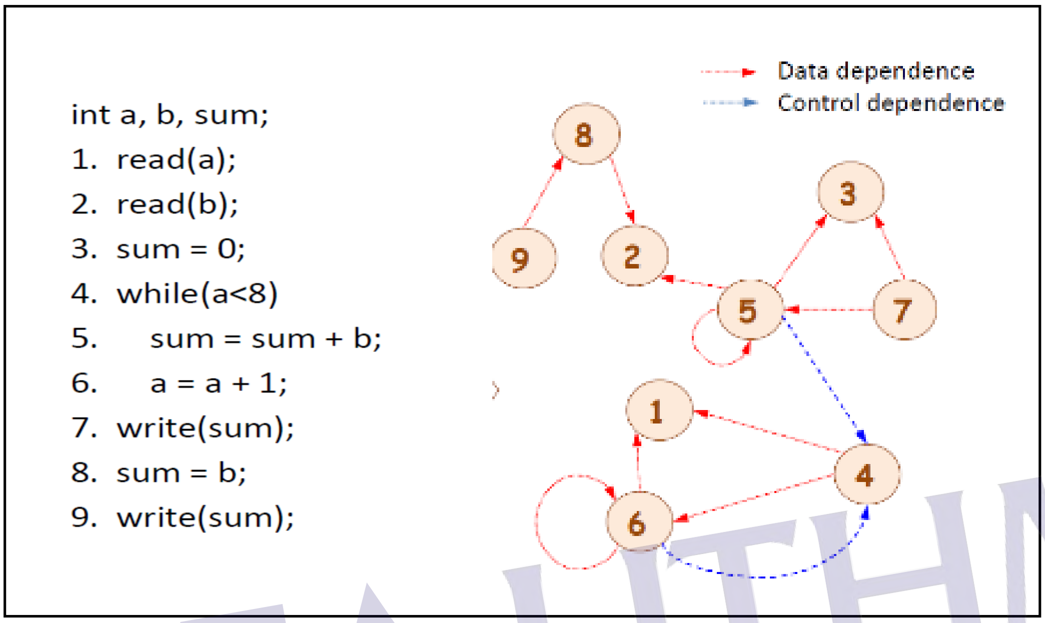
Figure 2.8: Call Flow Graph Presentation

Control Flow Graph will start from node 1 to read a variable, sequentially following node 2, and 3. At node 4 if the condition is true the program will proceed to node 5, and 6. Otherwise the program will move forward to node 7, further moving to end point as shown in Figure 2.8.

2.4.2 Program Dependency Graph

Program dependency analysis is a popular approach in software testing and debugging, and it is essential for programme comprehension (Shu *et al*., 2013). Dependency on the Program Graph represents a programme as a graph, with nodes representing statements and predicate expressions (Horwitz, Reps & Binkley, 2004). The edges occurrence on the node reflects the data values on which the node's activities are reliant and the control conditions under which the operations are performed. A

Program Dependency Graph (PDG) may express both control and data dependency in a single graph, as illustrated in Figure 2.9.



```
int a, b, sum;
1.  read(a);
2.  read(b);
3.  sum = 0;
4.  while(a<8)
5.      sum = sum + b;
6.      a = a + 1;
7.  write(sum);
8.  sum = b;
9.  write(sum);
```

Figure 2.9: Program Dependence Graph Presentation

Consider the program in Figure 2.9 that uses the code fragment to find the factorial of an integer. The control predicate at Statement 4 is required to execute Statement 5 and 6. Statement 4 relies on data from statements 1. The related PDG of the program is seen in Figure 2.9. The edge 5, and 6 have control dependency on edge 4 as can be seen in program. It is highlighted in blue line. The remaining all edges have data dependency and highlighted in red line as can be seen in Figure 2.9.

2.4.3 System Dependence Graph

The System Dependence Graph (SDG) is a version of the Program Dependence Graph (PDG) that displays a programme with a large number of procedures and procedural calls (Chandra *et al*., 2015). SDG is a programming language in which parameters are supplied as values. A complete system consists of a single (main) programme plus a collection of auxiliary operations (Horwitz *et al*., 2004).

REFERENCES

Aggarwal, K. K., Singh, Y., Kaur, A., & Malhotra, R. (2007). Investigating effect of Design Metrics on Fault Proneness in Object-Oriented Systems. *J. Object Technol*., 6(10), pp. 127-141.

Aggarwal, K. K., Singh, Y., Kaur, A., & Malhotra, R. (2009). Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study. *Software process: Improvement and practice*, 14(1), pp. 39-62.

Al-Dallal, J. (2013). Object-Oriented Class Maintainability Prediction using Internal Quality Attributes. *Information and Software Technology*, 55(11), pp. 2028–2048.

Al-Dallal, J. (2006). An Efficient Algorithm for Computing all Program Forward Static Slices. *Transaction on Engineering, Computing and Technology*, 16, pp. 108-111.

Ali, S. S., Zafar, M. S., & Saeed, M. T. (2020). Effort Estimation Problems in Software Maintenance–A Survey. In *2020 3rd International Conference on Computing, Mathematics and Engineering Technologies* (*iCoMET*), pp. 1-9.

Alokush, B., Abdallah, M., Alrifaee, M., & Salah, M. (2018). A proposed Java static slicing approach. *Indonesian Journal of Electrical Engineering and Computer Science*, 11(1), pp. 308-317.

Alpuente, M., Ballis, D., Frechina, F., & Romero, D. (2014). Using conditional trace slicing for improving Maude programs. *Science of Computer Programming*, 80, pp. 385-415.

Allen, M., & Horwitz, S. (2003). Slicing Java programs that throw and catch exceptions. *ACM SIGPLAN Notices*, 38(10), pp. 44-54.

Arora, V., Bhatia, R. K., & Singh, M. (2012). Evaluation of flow graph and dependence graphs for program representation. *International Journal of Computer Applications*, 56(14).

Arisholm, E., Briand, L. C., & Foyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on software engineering*, 30(8), pp. 491-506.

Awad, K., Abdallah, M., Tamimi, A., Ngah, A., & Tamimi, H. (2019). A proposed forward clause slicing application. *Indonesian Journal of Electrical Engineering and Computer Science*, 13(1), pp. 401-407.

Badri, L., Badri, M., & St-Yves, D. (2005, December). Supporting predictive change impact analysis: a control call graph based technique. In 12th Asia-Pacific Software Engineering Conference (APSEC'05), pp. 177-184.

Bader, R., Alokush, B., Abdallah, M., Awad, K., & Ngah, A. (2020). *A proposed java forward slicing approach. Telkomnika*, 18(1), pp. 311-316.

Ba-Quttayyan, B., Mohd, H., & Baharom, F. (2018). Regression testing–A protocol for systematic literature review. *In AIP Conference Proceedings* Vol. 2016, No. 1, pp. 020032.

Bidve, V. S., & Sarasu, P. (2016). Tool for measuring coupling in object-oriented Java software. *IACSIT International Journal of Engineering and Technology*, 8(2), pp. 812-820.

Bidve, V. S., & Khare, A. (2012). A survey of coupling measurement in object oriented systems. *International Journal of Advances in Engineering & Technology*, 2(1), pp. 43.

Biswas, S., Mall, R., Satpathy, M., & Sukumaran, S. (2011). Regression test selection techniques: A survey. *Informatica*, 35(3), pp. 289–321.

Briand, L. C., Daly, J. W., & Wust, J. K. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1), pp. 91-121.

Briand, L. C., Wust, J., & Lounis, H. (1999). Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings IEEE International Conference on Software Maintenance-1999* (*ICSM'99*), pp.475-482.

Briand, L. C., Wüst, J., Daly, J. W., & Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3), pp. 245-273.

Cabot, J., & Gogolla, M. (2012). Object constraint language (OCL): a definitive guide. *In International school on formal methods for the design of computer, communication and software systems*, pp. 58-90.

Chae, H. S., & Bae, D. H. (2011). A coupling measure for object-oriented classes. *Software Practice and Experience* (30).

Chandra, A., Singhal, A., & Bansal, A. (2015). A study of program slicing techniques for software development approaches. *In 2015 1st International Conference on Next Generation Computing Technologies* (*NGCT*), pp. 622-627.

Chauhan, N. (2015). Regression test selection for object oriented systems using OPDG and slicing technique. In *2015 2nd International Conference on Computing for Sustainable Global Development* (*INDIACom*), pp. 1372-1378

Chauhan, N. (2010). Software Testing: Principles and Practices. *Oxford university press.*

Chechik, M., Lai, W., Nejati, S., Cabot, J., Diskin, Z., Easterbrook, S., Sabetzadeh, M., & Salay, R. (2009). Relationship-based change propagation: A case study. In *2009 ICSE Workshop on Modeling in Software Engineering*, pp.7-12.

Chen, Z., & Xu, B. (2001). Slicing object-oriented Java programs. *ACM Sigplan Notices*, 36(4), pp. 33-40.

Dalal, S., & Chhillar, R. S. (2012). Software Testing-Three P'S Paradigm and Limitations. *International Journal of Computer Applications*, 54(12).

Dahiya, O., & Solanki, K. (2018). A systematic literature study of regression test case prioritization approaches. *International Journal of Engineering & Technology*, 7(4), pp. 2184-2191.

De AG Saraiva, J., de França, M. S., Soares, S. C., Fernando Filho, J. C. L., & de Souza, R. M. (2015). Classifying metrics for assessing object-oriented software maintainability: A family of metrics' catalogs. *Journal of Systems and Software*, 103, pp. 85-101.

De Lucia, A., Oliveto, R., & Vorraro, L. (2008). Using structural and semantic metrics to improve class cohesion. In *2008 IEEE International Conference on Software Maintenance*, pp. 27-36.

de S. Campos Junior, H., Araújo, M. A. P., David, J. M. N., Braga, R., Campos, F., & Ströele, V. (2017). Test case prioritization: a systematic review and mapping of the literature. *In Proceedings of the 31st Brazilian Symposium on Software Engineering*, pp. 34-43.

Do, H., Rothermel, G., & Kinneer, A. (2004). Empirical studies of test case prioritization in a JUnit testing environment. In *15th international symposium on software reliability engineering*, pp. 113-124.

Do, H., & Rothermel, G. (2006). On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9), pp. 733-752.

Eder, J., Kappel, G., & Schrefl, M. (1994). Coupling and cohesion in object-oriented systems, *Technical Report, University of Klagenfurt*, pp. 264-272

Elbaum, S., Rothermel, G., & Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 235-245.

Elbaum, S., Rothermel, G., Kanduri, S., & Malishevsky, A. G. (2004). Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3), pp. 185-210

Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2002). Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2), pp. 159-182.

El-Emam, K., Melo, W., & Machado, J. C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of systems and software*, 56(1), pp. 63-75.

Ellis, T. J., & Levy, Y. (2010). A guide for novice researchers: Design and development research methods. In *Proceedings of Informing Science & IT Education Conference* (*InSITE*) Vol. 10, pp. 107-118.

Fang, C., Chen, Z., Wu, K., & Zhao, Z. (2014). Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2), pp. 335-361.

Gao, K., Khoshgoftaar, T. M., Wang, H., & Seliya, N. (2011). Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5), pp. 579-606.

German, D. M., Hassan, A. E., & Robles, G. (2009). Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10), pp. 1394-1408.

Gethers, M., Dit, B., Kagdi, H., & Poshyvanyk, D. (2012). Integrated impact analysis for managing software changes. In *2012 34th International Conference on Software Engineering* (*ICSE*), pp. 430-440.

Green, P. D., Lane, P. C., Rainer, A., & Scholz, S. (2009). An introduction to slice-based cohesion and coupling metrics.

Hammer, C., & Snelting, G. (2004). An improved slicer for Java. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering,* pp. 17-22.

Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., & Gujarathi, A. (2001). Regression test selection for Java software. *ACM Sigplan Notices*, 36(11), pp. 312-326.

Hattori, L., Guerrero, D., Figueiredo, J., Brunet, J., & Damásio, J. (2008). On the precision and accuracy of impact analysis techniques. In *Seventh IEEE/ACIS International Conference on Computer and Information Science* (*icis 2008*), pp. 513-518.

Hitz, M., & Montazeri, B. (1995). *Measuring coupling and cohesion in object-oriented systems,* pp. 25-27.

Homès, B. (2013). Fundamentals of software testing. John Wiley & Sons.

Horwitz, S., Reps, T., & Binkley, D. (2004). Interprocedural slicing using dependence graphs. *Acm Sigplan Notices*, 39(4), pp. 229-243.

Hou, S. S., Zhang, L., Xie, T., Mei, H., & Sun, J. S. (2007). Applying interface-contract mutation in regression testing of component-based software. In *2007 IEEE International Conference on Software Maintenance,* pp. 174-183.

ISO/IEC/IEE. (2017). Systems and software engineering - vocabulary (*ISO/IEC/IEE* 24765).

Jayant, D. K., & Sagar, D. B. (2016). An Empirical View of Regression Test Case Generation Based On UML. *Globus an International Journal of Management & IT*, 8(1), pp.1-9.

Jeffrey, D., & Gupta, N. (2006). Test case prioritization using relevant slices. In *30th Annual International Computer Software and Applications Conference* (*COMPSAC'06*) Vol. 1, pp. 411-420.

Jorgensen, P. C. (2013). Software testing: a craftsman's approach. Auerbach Publications.

Joyner, W. D., & Melles, C. G. (2017). *Adventures in graph theory*. Springer International Publishing.

Kaur, U., & Singh, G. (2015). A review on software maintenance issues and how to reduce maintenance efforts. *International Journal of Computer Applications*, 118(1).

Kazmi, R., Jawawi, D. N., Mohamad, R., & Ghani, I. (2017). Effective regression test case selection: A systematic literature review. *ACM Computing Surveys* (*CSUR*), 50(2), pp. 1-32.

Khatri, M., Goswami, D.P., & Scholar, M.T. (2017). A Retrospective Study on Cohesion and Coupling Metrics of OO Software Systems.

Khatibsyarbini, M., Isa, M. A., Jawawi, D. N., & Tumeng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93, pp. 74-93.

Korel, B., Koutsogiannakis, G., & Tahat, L. H. (2007). Model-based test prioritization heuristic methods and their evaluation. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pp. 34-43.

Korel, B., Koutsogiannakis, G., & Tahat, L. H. (2008). Application of system models in regression test suite prioritization. In *2008 IEEE International Conference on Software Maintenance* pp. 247-256.

Korel, B., Tahat, L. H., & Harman, M. (2005). Test prioritization using system models. In *21st IEEE International Conference on Software Maintenance* pp. 559-568.

Kozlov, D., Koskinen, J., & Sakkinen, M. (2013). Fault-proneness of open source software: Exploring its relations to internal software quality and maintenance process. *Open Software Engineering Journal*, pp.07-20.

Larprattanakul, A., & Suwannasart, T. (2013). An approach for regression test case selection using object dependency graph. In *2013 5th International Conference on Intelligent Networking and Collaborative Systems*, pp. 617-621

Li, B. (2001) A Hierarchical Slice-Based Framework for Object-Oriented Coupling Measurement. Citeseer, pp. 01-16.

Li, B. X., Fan, X. C., Pang, J., & Zhao, J. J. (2004). A model for slicing Java programs hierarchically. *Journal of Computer Science and Technology*, 19(6), pp. 848-85.

Li, B., Sun, X., Leung, H., & Zhang, S. (2013). A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8), pp. 613-646.

Lisper, B., Masud, A. N., & Khanfar, H. (2015). Static backward demand-driven slicing. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* pp. 115-126.

Malishevsky, A. G., Ruthruff, J. R., Rothermel, G., & Elbaum, S. (2006). Cost-cognizant test case prioritization. *Technical Report TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln*. pp. 97-106

Malhotra, R., & Chug, A. (2016). Software maintainability: Systematic literature review and current trends. *International Journal of Software Engineering and Knowledge Engineering*, 26(08), pp. 1221-1253.

Maia, C. L. B., do Carmo, R. A. F., de Freitas, F. G., de Campos, G. A. L., & de Souza, J. T. (2010). Automated test case prioritization with reactive GRASP. *Advances in Software Engineering, 2010*.

Mansor, Z. U. L. K. E. F. L. I., & NDUDI, E. E. (2015). Issues, Challenges and Best Practices of Software Testing Activity. *In Proc. 14th Conf. Appl. Comput. Eng.*(*ACE15*), *South Korea*, pp. 42-47.

Marcus, A., & Poshyvanyk, D. (2009). The conceptual coupling of classes. In *Proc. 21th IEEE International Conference on Software Maintenance*, pp. 453-45.

Ma, Y. S., Offutt, J., & Kwon, Y. R. (2006). MuJava: a mutation system for Java. In *Proceedings of the 28th international conference on Software engineering*, pp. 827-830.

Meyers, T. M., & Binkley, D. (2007). An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 17(1), pp. 1-27.

Minhas, N. M., Petersen, K., Börstler, J., & Wnuk, K. (2020). Regression testing for large-scale embedded software development–Exploring the state of practice. Information and Software Technology, 120, pp. 106-254

Mohapatra, D. P., Mall, R., & Kumar, R. (2004). An edge marking technique for dynamic slicing of object-oriented programs.

Mohanty, H., Mohanty, J. R., & Balakrishnan, A. (Eds.). (2017). *Trends in software testing. Springer Singapore*.

Mössenböck, H. (2012). *Object-oriented programming in Oberon-2*. Springer Science & Business Media.

Munjal, D. (2015). Analysis of Slice-Based Metrics for Aspect-Oriented Programs (*Doctoral dissertation*).

Musa, S., Sultan, A. B. M., Ghani, A. B. A., & Bahaarom, S. (2015). Regression Test Cases Selection for Object-Oriented Programs based on Affected Statements. *International Journal of Software Engineering and Its Applications*, 9(10), pp. 91-108.

Musa, S., Sultan, A. B. M., Abd-Ghani, A. A. B., & Baharom, S. (2015). Software regression test case prioritization for object-oriented programs using genetic algorithm with reduced-fitness severity. *Indian Journal of Science and Technology*, 8(30), pp. 1-9.

Musa, S., Sultan, A. B. M., Abd Ghani, A. A. B., & Baharom, S. (2014). A regression test case selection and prioritization for object-oriented programs using dependency graph and genetic algorithm. *Int J Eng Sci*, 4(7), pp. 54-64.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

Najumudheen, E. S. F., Mall, R., & Samanta, D. (2009). A dependence graph-based representation for test coverage analysis of object-oriented programs. *ACM SIGSOFT Software Engineering Notes*, 34(2), pp. 1-8.

Najumudheen, E. S. F., Mall, R., & Samanta, D. (2010). A Dependence Representation for Coverage Testing of Object-Oriented Programs. *J. Object Technol.*, 9(4), pp. 1-23.

Najumudheen, E. S. F., Mall, R., & Samanta, D. (2011). Test coverage analysis based on an object-oriented program model. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(7), pp. 465-493.

Ngah, A. (2012). Regression test selection by exclusion (*Doctoral dissertation, Durham University*).

Ngah, A., & Selamat, S. A. (2014). A Brief Survey of Program Slicing. *In International Symposium on Research in Innovation and Sustainability 2014* (*ISoRIS'14*), pp. 1467-1470.

Nicolaescu, A., Lichter, H., & Xu, Y. (2015). Evolution of object oriented coupling metrics: a sampling of 25 years of research. *In 2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics* pp. 48-54.

Orso, A., Apiwattanapong, T., & Harrold, M. J. (2003). Leveraging field data for impact analysis and regression testing. *ACM SIGSOFT Software Engineering Notes*, 28(5), pp. 128-137.

Orso, A., Sinha, S., & Harrold, M. J. (2001). Incremental slicing based on data-dependences types. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pp. 158-167

Orso, A., & Rothermel, G. (2014). Software testing: a research travelogue (2000–2014). *In Future of Software Engineering Proceedings*, pp. 117-132

Otieno, C., Okeyo, G., & Kimani, S. (2015). Coupling measures for object oriented software systems-a state-of-the-art review. *International Journal of Engineering and Science* (*IJES*), 4, pp. 01-10.

Otieno, C. (2016). Unified Class Coupling Model for Coupling Measurement in Object Oriented Software Systems (*Doctoral dissertation*, *Information Technology*, *JKUAT*).

Panda, S. (2016). Regression Testing of Object-Oriented Software based on Program Slicing (*Doctoral dissertation*).

Panda, S., Munjal, D., & Mohapatra, D. P. (2016). A slice-based change impact analysis for regression test case prioritization of object-oriented programs. *Advances in Software Engineering*, *2016*.

Panda, S., & Mohapatra, D. P. (2013). Application of hierarchical slicing to regression test selection of Java programs.

Panigrahi, C. R., & Mall, R. (2013). An approach to prioritize the regression test cases of object-oriented programs. *CSI transactions on ICT*, 1(2), pp. 159-173.

Panigrahi, C. R., & Mall, R. (2014). A heuristic-based regression test case prioritization approach for object-oriented programs. *Innovations in Systems and Software Engineering*, 10(3), pp. 155-163.

Panigrahi, C. R., & Mall, R. (2016). Regression test size reduction using improved precision slices. *Innovations in Systems and Software Engineering*, 12(2), pp. 153-159.

Pujari, N., Ray, A., & Singh, J. (2018). Web Slicing Incorporating Component-Oriented Programming. *In 2018 Second International Conference on Computing Methodologies and Communication* (*ICCMC*), pp. 334-339.

Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8), pp. 1397-1418.

Rajlich, V. (2014). Software evolution and maintenance. *In Future of Software Engineering Proceedings*, pp. 133-144.

Rava, M., & Wan-Kadir, W. M. (2016). A review on prioritization techniques in regression testing. *International Journal of Software Engineering and Its Applications*, 10(1), pp. 221-232.

Razafimahatratra, H., Mahatody, T., Razafimandimby, J. P., & Simionescu, S. M. (2017). Automatic detection of coupling type in the UML sequence diagram. *In 2017 21st International Conference on System Theory, Control and Computing* (*ICSTCC*), pp. 635-640

Ren, X., Chesley, O. C., & Ryder, B. G. (2006). Identifying failure causes in Java programs: An application of change impact analysis. *IEEE transactions on software engineering*, 32(9), pp. 718-732.

Rothermel, G., & Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8), pp. 529-551.

Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (1999). Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999*, pp. 179-188

Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10), pp. 929-948.

Rothermel, G., & Harrold, M. J. (1994). Selecting Regression Tests for Object-Oriented Software. I*n ICSM*, Vol. 94, pp. 14-25.

Ruthruff, J. R., Elbaum, S., & Rothermel, G. (2010). Experimental program analysis. *Information and Software Technology,* 52(4), pp. 359-379.

Selamat, S. A., & Ngah, A. (2017). Slicing for Java Program: A Preliminary Study. *Journal of Telecommunication*, *Electronic and Computer Engineering* (*JTEC*), 9(3-5), pp. 147-151.

Sandeep Dalal, S., & Solanki, K. (2018). Experimental Analysis of "BA-TPF" Technique for Regression Test Optimization. *International Journal of Engineering & Technology*, 7(4), pp. 3135-3141.

Sajeev, A. S. M., & Wibowo, B. (2003). Regression test selection based on version changes of components. *In Tenth Asia-Pacific Software Engineering Conference*, pp. 78-85

Sahakyan, A. (2019). Software change impact analysis with respect to data protection.

Sasirekha, N., Robert, A. E., & Hemalatha, D. M. (2011). Program slicing techniques and its applications. arXiv preprint arXiv, pp. 1108.1352.

Shaik, A., Reddy, K., & Damodaram, A. (2012). Object oriented software metrics and quality assessment*:* Current state of the art. *International Journal of Computer Applications*, 37(11), pp. 6-15.

Sharma, S., & Srinivasan, S. (2013). A review of Coupling and Cohesion metrics in Object Oriented Environment. *International Journal of Computer Science & Engineering Technology (IJCSET),* 4(8), pp. 1105-1111.

Sherriff, M., & Williams, L. (2008). Empirical software change impact analysis using singular value decomposition. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 268-277.

Shu, G., Sun, B., Henderson, T. A., & Podgurski, A. (2013). Javapdg: A new platform for program dependence analysis. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* pp. 408-415.

Sikka, P., & Kaur, K. (2013). Program slicing techniques and their need in aspect oriented programming. *International Journal of Computer Applications*, 70(3), pp. 11-14.

Singh, S. N., & Singh, L. (2014, September). Study of current program slicing techniques. *In 2014 5th International Conference-Confluence The Next Generation Information Technology Summit* (*Confluence*), pp. 810-814.

Sun, X., Li, B., Li, B., & Wen, W. (2012). A comparative study of static CIA techniques. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*, pp. 1-8.

Srikanth, H., Banerjee, S., Williams, L., & Osborne, J. (2014). Towards the prioritization of system test cases. *Software Testing, Verification and Reliability*, 24(4), pp. 320-337.

Surendran, A., Samuel, P., & Jacob, K. P. (2016). *Program slicing techniques for software testing* (Doctoral dissertation, Cochin University of Science and Technology).

Tao, C., Li, B., Sun, X., & Zhang, C. (2010). An approach to regression test selection based on hierarchical slicing technique. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pp. 347-352

Tiky, Y. T. (2016). Software development life cycle. Hongkong: *THe Hongkong University of Science and Technology.*

Tip, F. (2015). Infeasible paths in object-oriented programs. *Science of Computer Programming*, 97, pp. 91-97.

Tonella, P. (2003). Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE transactions on software engineering*, 29(6), pp. 495-509.

Umudova, S. (2019). Analysis of software maintenance phases. *Noble International Journal of Scientific Research*, 3(6), pp. 62-66.

Wang, T., & Roychoudhury, A. (2004, May). Using compressed bytecode traces for slicing Java programs. In *Proceedings. 26th International Conference on Software Engineering*, pp. 512-521.

Wang, X., Zhang, Y., Zhao, L., & Chen, X. (2017). Dead code detection method based on program slicing. *In 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery* (*CyberC*), pp. 155-158

Walkinshaw, N., Roper, M., & Wood, M. (2003). The Java system dependence graph. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 55-64

Wen, W. (2012). Software fault localization based on program slicing spectrum. In *2012 34th International Conference on Software Engineering* (*ICSE*), pp**.** 1511-1514

Xi, L., Li, M., Dan, Z., & Wei, L. (2011). An approach of coarse-grained dynamic slice for Java program. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pp. 670-674.

Yang, Y., Zhou, Y., Lu, H., Chen, L., Chen, Z., Xu, B., & Zhang, Z. (2014). Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study. *IEEE Transactions on Software Engineering*, 41(4), pp. 331-357.

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2), pp. 67-120.

Yu, P., Systa, T., & Muller, H. (2002). Predicting fault-proneness using OO metrics. An industrial case study. In *Proceedings of the sixth european conference on software maintenance and reengineering*, pp. 99-107.

Zarrad, A. (2015). A Systematic Review on Regression Testing for Web-Based Applications. *J. Softw*., 10(8), pp. 971-990.

Zhang, L., Hao, D., Zhang, L., Rothermel, G., & Mei, H. (2013). Bridging the gap between the total and additional test-case prioritization strategies. In *2013 35th International Conference on Software Engineering* (*ICSE*), pp. 192-201.