TEST CASE PRIORITIZATION USING SOFTWARE REQUIREMENTS, DESIGN DIAGRAMS AND SOURCE CODE COLLABORATION

by

MD SAEED SIDDIK
REG: HA 638
SESSION: 2009-2010

A thesis
submitted in partial
fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

TEST CASE PRIORITIZATION USING SOFTWARE REQUIREMENTS, DESIGN DIAGRAMS AND SOURCE CODE COLLABORATION

MD SAEED SIDDIK

Approved:

Signature	*Date*
Supervisor: Dr. Kazi Muheymin-Us-Sakib	
Committee Member: Dr. Zerin Begum	
Committee Member: Dr. Kazi Muheymin-Us-Sakib	
Committee Member: Dr. Muhammad Mahbub Alam	
Committee Member: Shah Mostafa Khaled	

To those children, who did not get the educational opportunities.

Abstract

Test case prioritization is a technique for selecting particular test cases, which are expected to detect faulty modules earlier. Since at least 40% of software budget is allocated for software testing, test case prioritization can reduce that by detecting maximum faults in minimum test case execution. To prioritize test cases, closely related information to test cases are usually considered. Testing is a part of Software Development Life Cycle (SDLC), so, information from different phases of SDLC is a better option to determine the priority. However, SDLC information are recently introduced and the analysis of existing literature shows that no one integrates all phase information to prioritize test cases.

This thesis proposes a framework named as *Requirements, Design diagrams, and source Code Collaboration (RDCC)* to accumulate the SDLC information for efficient test case prioritization. Requirements are taken as input and split into words or terms excluding stop words and those collected words are used to calculate term frequency - inverse document frequency (tf-idf). The sum of tf-idf values of words belonging to a requirement defines the priority value. Design diagrams are extracted as readable Extensible Markup Language (XML) format to calculate the dependencies and connectivities among software modules, corresponding to each requirements. The degree of each module is used to determine the design diagrams priority. Whereas classes corresponding to each requirements are prioritized by analyzing code metrics.

Final RDCC priorities are constructed by the multiplication of those calculated values and their assigned priority constants. Test cases are prioritized based on those collaborated information by using requirements traceability matrix where the test cases and their related requirement IDs are listed. If one test case is connected to multiple requirements, cumulative RDCC priorities are assigned to that test case.

The proposed RDCC framework is experimented on different software projects and the results are compared to several prominent software test case prioritization schemes

which are natural order, requirements based prioritization technique and source code based prioritization technique. On average, proposed collaborative approach performs 22.77% and 29.01% better than any other implemented test case prioritization schemes in terms of average fault detection and number of test case execution respectively.

Acknowledgments

First and above all, I praise Allah, the Almighty and the Guardian, for providing me this opportunity and granting me the capability to proceed successfully. I would like to express my deep and sincere gratitude to my thesis supervisor Dr. Kazi Muheymin-Us-Sakib, Director and Associate Professor, IIT University of Dhaka, for providing invaluable guidance throughout my research. His dynamism, vision, sincerity and motivation have deeply inspired me. He has not only contributed as my technical supervisor but also helped as my philosophical mentor.

Many thanks to the faculty members of IIT, Dr. Md. Mahbubul Alam Joarder, Professor and Mr. Shah Mostafa Khaled, Assistant Professor, IIT University of Dhaka for helping me by providing feedbacks. Many thanks to Mr. Alim Ul Gias and Mr. Asif Imran, Lecturer, IIT, University of Dhaka for their mentoring and technical support. Thanks to Mr. Md. Selim, Mr. Rayhanul Islam, Nadia Nahar, Sohel Rana, A S Rifat and Kazi Solaiman for their total cooperation.

This research is partially supported by the fellowship of ICT Division, Bangladesh. I am grateful to the following -

- The ministry of Post, Telecommunication and Information Technology, Bangladesh under Information and Communication Technology Fellowship No - 56.00.0000.028.33.019.14-54

Publications

Publications during the Masters period

- Siddik, M.S.; Sakib, K., "RDCC: An effective test case prioritization framework using software requirements, design and source code collaboration", 17th International Conference on Computer and Information Technology (ICCIT), pp. 75 - 80, Dec. 2014, IEEE, doi: 10.1109/ICCITechn.2014.7073072

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Test case prioritization re-orders the test cases for providing earlier feedback to software testers and managers about faulty modules [1, 2]. It can reduce the time and cost in testing phase by working on critical sections earlier [1, 3]. Testing is a vital phase of Software Development Life Cycle (SDLC) ensuring the correctness and quality of final products. It is a process for evaluating software by detecting differences between given input and expected output. However it costs more than 40% of total development budget and time [1, 3]. Although test case prioritization can minimize the testing time and cost, without executing the whole test suite, test case prioritization is really difficult to achieve.

To achieve an efficient prioritization technique, several issues connected to test cases such as requirements for test case generation, source code for test case execution, etc. are needed to be addressed. These issues are distilled into research question and an efficient prioritization framework is proposed to solve that. This chapter presents the motivation of this thesis with the research question and contribution. Finally, this chapter is concluded by providing the thesis organization list.

1.1 Motivation

Test case prioritization is a process to sort out test cases in an optimal order which can detect faults earlier. Since testing phase consumes at least 40% of whole project time and cost [4], test case prioritization can reduce software budget by early fault detection and minimum test case execution.

In prioritization schemes, priority values are assigned to all test cases based on their accuracy of fault detection. Test cases are prioritized before whole software deployment

by using related information to that software, like requirements, source code, risks and complexities etc. To determine an effective test case prioritization scheme, information which are closely related to test cases, are needed to be considered [5]. Since testing is a vital part of SDLC, other phases of SDLC are intimately related to test cases. Major issues which are closely related to test cases are briefly described below.

- **Requirements and Test Cases**

  Requirement specification is a part of SDLC which are used to write test cases [5]. Requirements are usually constituted by the customers' viewpoints and expectation about a software, whereas test cases confirm those customers' expectations. Final products are validated by those requirements in the testing phase. That is why analysis of software requirements for prioritizing test cases is one of the prominent approaches of prioritization.

  Since each of the test cases is related to at least one requirement, clustering similar requirements are used as a variation of test case prioritization scheme by grouping related test cases together [6]. Risks and severity of requirements are also being analyzed to find out the potential threats earlier [7]. These types of prioritization schemes are developed by using pre-execution information from software requirements only.

- **Design Diagrams and Test Cases**

  Software design is one of the major phases of SDLC, containing the software internal connectivities and dependencies, which are important for determining dependent test cases. Because, if one software module contains any fault, the dependent and connected modules become potentially fault prone. By selecting the connected and dependent test cases together, related faults are detected earlier. That is why dependency values from design diagrams are needed to be incorporated for an optimal test case prioritization scheme.

- **Source Code and Test Cases**

  Since test cases are executed on software source code, incorporating source code and test cases can lead to an efficient prioritization scheme. Several prominent test case prioritization approaches are developed by analyzing source code only [8, 2, 9]. In those approaches, various types of information from source code are used for prioritization such as line of codes, number of functions and statements, etc. These types of prioritization schemes increase the efficiency of fault detection, because source code are closely related to test cases.

Those issues which are described above, basically represent the relationship between test cases and different phases of SDLC. Requirements are used to write test cases and final product validation. On the other, hand design diagrams are used to determine test cases priority by using software modules' dependencies and connectivities, where source code are used for test case execution. The importance of relating requirements, design diagrams and source code to test cases are also denoted by those issues which are needed to be addressed collaboratively for an efficient prioritization scheme.

1.2 Research Question (RQ)

One of the common goals of test case prioritization schemes is to find the appropriate set of test cases earlier which are important in terms of fault detection and percentage of test case execution to detect those faults. Those terms are considered as the criteria of prioritization performance. For test case prioritization, various issues which are presented in previous section can play vital roles together. This lead to the primary research question.

- **RQ:** How to prioritize test cases in software testing, using collaborative information from software requirement specification, design diagrams and source code ?

In the beginning, all representative information are needed to be collected from SDLC phases. Those information are collaborated according to the relationship with the test cases. Finally, test cases are prioritized based on those information. More specifically, this research intends to answer following sub-questions.

- How to parse and collaborate representative information from requirements, design diagrams and source code ?

Representative information are extracted from requirements, design documents and source code because irrelevant sections can influence the selected results such as stop words in requirements, comments in source code, etc. Requirements are parsed as textual similarities and their priorities are calculated using term documents matrix [10]. Design diagrams are extracted as readable XML to detect the connectivities and dependencies among different software modules, corresponding to each requirements. Classes or functions related to each requirements, are used as idiosyncratic element of source code and prioritized by using several code metrics which are line of codes, cyclomatic complexity, nested block depth and weighted method per class. These information are collaborated by assigning similar priority to all phases.

- How to prioritize test cases based on collaborative information from different phases of SDLC ?

The set of selected collaborative information are needed to be linked with test suits for allocating the relationship to requirements and test cases, using traceability matrix where every test cases are assigned to one or more requirements. If one test case is assigned to multiple requirements, the cumulative requirement priorities are used as final priority. At the end, the test cases are sorted in descending order based on their assigned priority values.

1.3 Research Contributions

This research presents an effective test case prioritization framework named as Requirements, Design diagrams and source Code Collaboration (RDCC) to address the question that has been raised in the previous section. RDCC used collaborative priority values to reorder test cases in descending order. Every requirement IDs are uniquely identified as RDCC IDs for test case mapping. Collaboration from requirements to design diagrams, requirements to source code and requirements to test cases are used to detect priority values. In a nutshell, the major contributions of RDCC framework are listed below.

- RDCC offers an efficient prioritization scheme by collaborating different phases of SDLC to test cases. Requirements (a phase of SDLC) are analyzed as textual similarity by calculating term frequency and inverse document frequency [10]. Design diagrams are extracted as readable XML format to detect the connectivity among software modules corresponding to each requirements. Source code are parsed as software code metrics to calculate code priority for either classes or functions related to each requirements. Those three priorities are used to assign final priority values to test cases for early fault detection and minimization of test case execution.

- RDCC assigns equal priority to every phases of SDLC, because different phases of SDLC can represent the whole software in different point of views. Priority constants of different phases are also normalized to 0 to N for computational simplicity. If any of those phases is absent, the priority constant value of that phase will be assigned to 0. The cumulative priority values of different phases of SDLC are assigned to related test cases. Finally test cases are sorted in descending order based on their priority values which are assigned by collaborative SDLC information.

- It has been demonstrated experimentally that the collaborative information from different phases of SDLC are more effective for test case prioritization than considering

any individual phase (such as requirements [7] and source code [8]). The experimental results are measured by two different point of views named as - percentage of fault detection and percentage of test case execution. Proposed RDCC scheme overcomes the limitation of analyzing individual SDLC phase by integrating those priorities together.

1.4 Organization of the Thesis

Rest of the thesis is organized as follows.

- **Chapter 2: Background of Test Case Prioritization** This chapter discusses a background of software testing and test case prioritization. It also provides about the core concepts of SDLC for prioritizing test cases. It illustrates the challenges and assumptions for prioritization schemes.

- **Chapter 3: Test Case Prioritization in Literature** This chapter provides the researches which are related to test case prioritization. Different prioritization approaches including and excluding SDLC information are described with their research issues and drawbacks.

- **Chapter 4: RDCC: A Test Case Prioritization Framework using Requirements, Design diagrams, and source Code Collaboration** In this chapter proposed framework for test case prioritization with architecture and internal activities are presented. The RDCC framework is explained as a layer based architecture. Moreover, it has been discussed by a case study, that how RDCC framework is executed step by step for prioritizing test cases.

- **Chapter 5: Experimental Setup and Results Analysis** This chapter describes the experimental setup and the results which are obtained by executing both proposed

and compared prioritization schemes. Results are analyzed and discussed in terms of early fault detection and test case execution percentage.

- **Chapter 6: Conclusion and Future Work** The concluding remarks about this research are presented in this chapter. Initially a discussion is presented regarding RDCC and different experimental results. The achievements of this research are also presented in this chapter. This chapter is concluded by providing the scope of further directions of test case prioritization researches.

Chapter 2

Background of Test Case Prioritization

Test case prioritization is a vital part of software testing which is executed to find the optimal order of test cases for early fault detection. These techniques are performed before the test case execution. This is a supporting element of software testing to reduce time and cost by detecting early faults. Test cases are prioritized based on various information from SDLC, other connectivity among test cases, etc. This chapter presents an overview of test cases with their prioritization approaches and related information. Since testing is a part of software development life cycle, the connection between test case prioritization and different phases of SDLC are also briefly explained.

2.1 Testing as a Phase of SDLC

Software testing [1, 11] is an inseparable part of SDLC to validate the final product based on users' expectations. Although testing phase is executed at the end of SDLC, about 30% to 40% time and cost are allocated for this phase [1, 4]. In this phase, software source code are tested using test cases. Test cases are basically executed for finding faults and unexpected outputs from either a software or one of its features. Test cases are prioritized for finding those faults and unexpected outputs earlier which can reduce the whole testing time and cost [1]. Brief description of test cases and its' prioritization approaches are presented below.

2.1.1 Test Cases

A test case, in software testing, is a set of conditions, inputs and expected outputs to determine an application or a feature of that application is working as exception or not [11]. IEEE Standard 610 [12] defines test case as follows:

- "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement ".
- "Documentation specifying inputs, predicted results, and a set of execution conditions for a test item ".

According to Ron Patton et al. [13], "Test cases are the specific inputs that you will try and the procedures that you will follow when you test the software " .

Test case is one of the vital parts of SDLC which is used to validate the final product before release. Test cases are written based on requirements collected from customers. The additional information that may be included to a standard test case are listed below:

- test case ID
- test case description
- execution status (pass or fail)
- related requirement(s)
- test step or order of execution number
- test category
- author

2.1.2 Test Case Prioritization

Test case prioritization [1, 8, 14] determines the proper re-orderings of test cases so that faults can be detected early. It does not always confirm to detect faults early but maximizes the probability to detect early faults every time [15]. The objectives of test case prioritization techniques are to identify the optimal order of the test cases that are effective (in terms of capability of early detecting faults) and efficient (in terms of test case execution numbers) [16]. More specifically, the test case prioritization is a process to identify maximum faults in minimum test case execution.

Table 2.1: Test Case Prioritization Sample

Test Case	Fault1	Fault2	Fault3	Fault4	Fault5	Fault6	Fault7	Fault8
A	*							
B				*	*	*	*	
C		*	*					*

A sample test case prioritization problem with three test cases A, B, C and their corresponding 8 faults are presented in Table 2.1. Since the number of test cases is three, the permutation order of those test cases is 6. There is six different order to execute those test cases for fault detection. The problem is to find the optimal order of test cases which can detect faults earlier.

Test cases are prioritized based on various information like code coverages [8, 9, 17], test cases similarity [18], requirements risk factors [7, 3], etc. to detect the optimal test cases order. According to Table 2.1, the optimal test cases order is B $\rightarrow$ C $\rightarrow$ A, which can detect faults earlier than any other order.

2.2 Types of Test Case Prioritization

Test case prioritization is performed by various approaches like processing requirements [7, 19, 6], using source code dependencies [17], implementing random approaches [20], etc. Those prioritization techniques are validated by different performance matrices named as average percentage of fault detection, average percentage test case execution, etc. [21].

2.2.1 Requirement Based Prioritization

Requirements processing is a very familiar type of software test case prioritization because final products are validated by customer driven requirements. Since requirements specification is a part of SDLC, engineers list well defined and verified requirements for validation and further design diagrams creation. Those verified requirements are prioritized for early fault detection.

A traceability matrix is needed to be generated by the testing engineers containing the requirement ID and related test case ID in the same row for requirements based prioritization. Various requirement based prioritization schemes are already proposed by existing researches. A list of some potential approaches, on how to prioritize test cases using requirements, are presented below. The details explanation of requirement based prioritization are presented in Chapter 3 Section 3.1.

- **Requirements Clustering:**

  Test cases are prioritized by clustering similar requirements to detect related faults earlier. This process is executed by using textual cluster [6], requirements frequency cluster [3], etc. It is assumed for requirements based prioritization that, complex and frequent requirements contain potential faults in a software.

- **Risk and Severity:**

  Test cases are also prioritized by allocating risks and severity values to every requirements [7, 22]. Those values are collected from customers, developers and managers.

- **Requirements Complexity:**

  Based on the hypothesis on complex requirements lead to the fault prone product modules, requirements are prioritized based on their assigned complexity [19].

2.2.2 Source Code Based Prioritization

Source code have a heart and soul relationship to test cases, because test cases are executed on the source code. Faults are occurred in the source code of software. That is why prioritizing test cases using software source code is one of the efficient and common approaches in test case prioritization researches [8, 9, 17]. In this type, test cases are prioritized based on interior information of software source code. Some of those source code based prioritization schemes are listed below.

- **Code Coverage:**

  Source code coverage information is used to prioritize test cases for early fault detection [1, 8]. It is assumed for code based prioritization that, test cases which cover maximum code segments, can detect maximum faults in source code.

- **Code Metrics:**

  Code metrics are used to prioritize test cases, because those contains all interior information of every software source code.

- **Functional Dependency:**

  Functional dependencies are used to prioritize test cases by indicating the dependent modules of a software [17]. Those dependencies may lead to the dependent faults.

2.3 Assumptions and Prerequisite of Test Case Prioritization

Test case prioritization is executed as a module of software testing, that is why some prerequisite and assumptions are needed. Some important processing tasks and assumption values are listed below.

Test Case Generation

Test cases must be generated based on user requirements for test case prioritization. Without generating good test cases, no prioritization approach can work successfully. A requirement test case mapping matrix is also needed to be created for determining the relationship of test cases and requirements.

Requirement and Diagram Generation

Requirements and design diagrams are two major parts of SDLC which are related to test case prioritization. Since test cases are written based on requirements, correction and validation are necessary for every requirements. An XML format of design diagram is mandatory to calculate software design priority. Diagram to XML converters are used to generate XML files containing the inter connectivities and dependencies among the design modules corresponding to every requirements.

Traceability Matrix

A traceability matrix is a document, usually a tabular form, containing the relationship between test cases and requirements. Test case ID including it's corresponded requirement

IDs are denoted by the row of that table. a traceability matrix must be generated for implementing any test case prioritization schemes. A sample traceability matrix is presented in Table 2.2, containing 3 test cases and 4 requirements.

Table 2.2: Traceability Matrix Sample

Test Case	Req1	Req2	Req3	Req4
1	*			
2				*
3		*	*	

Assumptions for Implementation

For implementing any test case prioritization schemes, it is assumed that software source code, requirements, diagrams, test cases are well prepared. Random prioritization schemes are implemented for E numbers of execution where average results of those executions are used for further experiments. In the proposed framework of this thesis, the maximum prioritization values for any item is bounded to N for experimental and computational simplicity. User may vary this value on the basis of software quality. The detail explanation about this normalization values for experiments are presented in Chapter 5 Subsection 5.2.4.

2.4 Summary

This chapter presents the background study of test cases and its' prioritization approaches. A variety of test cases prioritization techniques with their assumptions are also reported. Since the proposed framework is developed based on information from various phases of SDLC, the connection of SDLC to test case prioritization are also presented in this chapter. The literature reviews of test case prioritization techniques are presented in the next chapter.

Chapter 3

Literature Review of Test Case Prioritization

Because of its importance and effectiveness in large scale software testing, in recent years, researchers have investigated different approaches of test case prioritization. It runs based on some potential industrial goals like early fault detection, minimum testing time, maximum code coverage etc, [1, 8]. Existing researches on test case prioritization can be divided into different groups such as analyzing software requirements to detect faulty modules earlier [6, 7], executing heuristic search algorithms to get maximum test suite coverage [23, 24], etc. This chapter describes various test case prioritization schemes, where following section identifies how software requirements are effectively related to prioritization.

3.1 Prioritization by Analyzing Software Requirements

Test cases are usually written based on software requirements which are the key part of Software Requirement Specification (SRS). Final products are validated against customer requirements, which is stored in SRS documents. Analysis of SRS documents is very important for test case prioritization, because those are prepared based on customers' feedbacks and priorities, requirement engineers' viewpoints etc. [25]. That is why, researchers pointed the use of requirements during software test prioritization [6, 7, 3, 26]. Following subsections present some potential researches with their experiments and drawbacks, which emphasized the need of software requirements specification to prioritize test cases.

3.1.1 Clustering Software Requirements

Since each of the test cases are related to at least one requirement, clustering similar requirements can help to prioritize test cases effectively by grouping related test cases together [6]. If one code segment contains any faults, its' related segments become potentially fault prone. Making a software product cohesive, related and similar requirements are implemented in the same classes or packages. So it is important to group related test cases in the same cluster for detecting related faulty modules earlier.

In the early phase of requirements based clustering approaches, similar requirements are grouped together based on their textual similarities [27]. Test cases are also grouped based on their associated requirement clusters. This approach can be split into five main activities [6], which are listed bellow.

1. Create requirement clusters by textual similarity [27]
2. Generate a matrix by requirement-test mapping
3. Build a group of test cases based on their associated requirement clusters
4. Calculate importance value of all requirements in a cluster for prioritization
5. Select test cases from those prioritized clusters

To perform prioritization, several types of information have been used such as requirements-traceability matrix [28], requirements modification history [6], etc. Arafeen et al. experimented two in-house developed Java programs named as iTrust and Capstone containing 42 and 142 test cases respectively, to investigate the effectiveness of clustering approach [6]. The empirical results show that clustering approach significantly outperformed the techniques without clustering. However reported results would be more accurate, if those were generated by considering the corresponding design diagrams for prioritizing requirements, because design diagrams have presentation uniqueness of a software.

3.1.2 Requirements and Risk Factors

Risks are the possibility of negative and undesirable outcomes in a software, where severity is the impact of those undesirable outcomes [7]. Collaborating requirements' risk and severity can present a better view for prioritizing test cases, because this collaboration defines the potential threated modules [7]. In a software project, all requirements are not equally important, approximately 45% of the software functions are rarely used, 19% are average used, and only 36% of the software functions are frequently used [3]. Frequently used requirements are the most risky segments among others. This approach finds those requirements earlier to prioritize test cases, that are frequently or always used.

Collaborating risk and severity approach basically inputs two types of prioritization factors for each requirement from different software stakeholders which are customers, managers and developers. Those factors are listed below.

1. Priorities, collected from customers, developers and managers for every requirements

2. Probability and severity values of risk factors for every requirements

Different values are assigned to every software requirements from customers and developers which are added to the risk factor of those requirements which are assigned manually. Srivastva et al. assigned probability factors from 0 to 10 to each requirement based on their importance [7]. This process finally delivers a numeric value of weighted risk and requirements priority by multiplying associated risk and probability values.

Risks are disclosed by using some factors named as Loss of Power (LOP), Corrupted File Data (CFD), Database Not Synchronized (DNS) etc [7]. By integrating the probability of occurrence and severity of impact, this approach presents a prioritized requirements list. Testing based on those prioritized requirements can improve the effectiveness of fault detection. This process was validated by an in-house devolved software containing 6 requirements and 6 test cases.

However, there were no exact direction how to map those prioritized requirements with test cases. There is also be a lack of integrating different software information such as requirements, source code etc. which makes this result questionable. Because, according to Islam et al., without considering source code and requirements, prioritization approaches can not cover all possible segments [15].

3.1.3 Requirements of System-level Test Cases

Prioritization Of Requirements for Test (PORT) is an effective system-level test case prioritization approach using customer assigned requirements priority [19, 26]. It integrates customer-assigned priority to identify important requirements. It considers both new and regression test cases for prioritization. The main objective of this strategy is to propose and validate a system-level prioritization scheme to reveal severe faults earlier and improve customer-perceived software quality [26].

PORT approach introduced four distinct factors for prioritizing user requirements. Those factors are listed below.

1. Customer-assigned requirements' priority
2. Requirements' volatility
3. Developer-perceived implementation complexity
4. Fault proneness of requirements

Based on the project and customers need, the development team assigned weight to those factors. A default value is assigned for giving each factors equal weight. After assigning weights to each requirement, the requirements list is sorted based on those weights. Finally the test cases are sorted based on their corresponding requirements' weights. To

determine the effectiveness of PORT approach, two phased feasibility study have been conducted in four similar student projects [26].

Srikanth et al. noticed that, this approach outperforms in terms of random testing [26]. Although those approaches can incorporate customers requirement priority with the test suite, there is no direction for relating requirements to the design diagrams or implementation modules. Without incorporating implementation modules with requirements, prioritization approaches failed to acquire details information about a software [29].

3.2 Prioritization by using Heuristic Search Algorithms

Heuristic is an special type of approach which is designed for finding an approximate solution when classic methods fail to find any exact solution in a short time [30]. It is a difficult challenge to find the appropriate order of test cases in testing phase of regression testing. Researchers introduced some heuristic algorithms to predict the approximate order of test cases for prioritizing such as greedy algorithm [23], hill climbing [24], genetic algorithm [31] etc. Following subsections describe those existing literature of test case prioritization, which are developed based on heuristic algorithms.

3.2.1 Multi Objective Test Case Prioritization

Most of the test case prioritization approaches exploit a single objective function, like minimum time allocation, code or requirements coverage etc. Where multi-objective prioritization approaches are experimented to achive multiple prioritization objectives [15, 29]. A software tool named Multi Objective Test Case Prioritization Technique (MOTCP) are presented to achieve both code and requirements coverage [29]. Genetic algorithm is used to identify appropriate test case ordering with respect to three different dimensions which

are structure, function and cost. Where the structural dimension was related to codes and test cases under analysis. On the other hand, the functional dimension was about how test cases exercise requirements, whereas the cost dimension was concerned to the test case execution time.

MOTCP works on requirements specification and source code to gather relative information from those. It recovers links among test cases, source code and requirements specifications by using Latent Semantic Indexing [32]. The whole technique consists of the following steps:

1. Collecting the software artifacts
2. Recovering the links among requirements and source code
3. Injecting 15 faults in the application code
4. Prioritizing the test cases using MOTCP
5. Executing the prioritized test case orderings in the buggy versions of the application
6. Collecting the achieved results

This technique has been experimented on 4 in house developed Java software having 10 to 15 requirements and 47 to 426 test cases. Islam et al. claimed that MOTCP performed better than random and adaptive random approach in every 30 iteration of test case executing [29]. Results would have been more accurate by incorporating design diagram information, because design diagrams have some representative uniqueness of a software such as software module connectivities and dependencies etc.[5].

3.2.2 Graph Theoretic Framework

Graph theory is one of the most important approaches to solve mathematical and computational problems. Ramanathan et al. introduced graph-theoretic framework for test case prioritization named PHALANX [18]. This approach has been presented by addressing the limitations of implementation complexity in testing process. It is the first test case prioritization framework using graph theory by considering the dissimilarity of a test case with others in the test suite. Because by executing dissimilar test cases upfront, different aspects of the software are tested earlier [18]. Whole PHALANX technique can be divided into three folds, which are listed bellow.

1. Develop test case dissimilarity graph
2. Compute longest common sub-sequences from that graph
3. Case study showcase from SIR repository [33].

It abstracts test cases into a test case dissimilarity weighted graph. In this graph, nodes represent test cases and weighted edges represent user defined proximity measures between test cases. Edge weight of that graph has been calculated using the edit distance [34] between two test cases. Greedy and spectral ordering algorithm [35] have been explored for finding the maximum weighted path. Experimental results noticed that this framework can detect major faults after executing the first 20% of the ordered test cases. For using graph theoretic framework, testing engineers must follow their predefined test case formation. This work could be extended by using common test case generation trend [11].

3.2.3 Additional and Optimal Greedy

Meta heuristics algorithms, basically greedy algorithms, are used to search any information in optimization problem. In the context of maximum code coverage, Li et al. performed a

simulation experiment to select those set of test cases which can cover maximum code segments [24]. A variety of greedy approaches named total, additional and 2-optimal greedy etc. have been proposed to search optimal result in terms of maximum code coverage.

The connectivity between test cases and requirements are calculated from requirement connectivity matrix, where test case ID and their connected requirement IDs are listed [24]. In this matrix, every cell represents the number of requirements satisfied by a test case. Then the mean and standard deviation of every cell is calculated for creating clusters of test cases. In the simulation experiment, large quantities of data are analyzed to compare the performance of these search algorithms. Finally, it was reported that the search algorithm named additional greedy algorithm outperforms the other search algorithms in most cases. The result would have been more efficient by considering source code information, because source code represent the implemented viewpoint of a software.

3.3 Prioritization by Analyzing Software Source Code

Source code is the most important part of SDLC, because it contains programmed logic, scripts, code, etc. [36]. Performing upgrades, installing components or fixing faults, developers need it to make any changes. Test cases are run on the software source code. That is why test cases are highly interrelated to source code. For fulfillment the prioritization goal named maximum code coverage, source code analysis is mandatory. It was researched that, information form source code analysis can lead efficient test case prioritization approaches [17, 8]. Following subsections present some researches, where analysis of source code is used for test case prioritization.

3.3.1 Dependency Structure of Test Suite

Haidry et al. proposed a functional test case prioritization technique based on the inherent structure of dependencies among test cases [17]. Because test cases are inter-related based on their executing dependencies. These are usually developed for finding faults in source code. Since functions of source code are dependent among each other, test cases are also dependent [37]. It was claimed that faults are identified earlier by considering highly dependent modules, because scenarios containing more relationships are complex and potentially fault prone.

Based on the functional dependency a direct acyclic graph is developed for test cases. In this graph $G = (V, E)$, the set V defines a set of test cases and the set E defines the functional dependencies between two test cases. Dependency values are assigned to every test cases that means every vertex v. After that, the longest path is calculated from this graph and proposed the path order as prioritized test case order. This approach runs pretty good, but manually generated dependency structure graph may vary from person to person. To find a common and usable solution, automatically extracting dependency structures from test suite is recommended.

3.3.2 Critical Components Identification

Critical component identification is a newly proposed test case prioritization approach based on software source code analysis [38]. In every software, some components become critical and fault prone, because of their high functionality and dependability with other components. On the other hand, poor quality of any software components adversely affects the quality of the overall system [39].

The critical component identification process is executed by using source code dependency. The dependency values are calculated in terms of external and internal dependency

metrics. Based on these calculated values, a graph called component execution sequence graph is drawn. Then the components are analyzed and prioritized based on the critical measure of the software. However, this approach does well for component based architecture using software source code, there is no direction to incorporate requirements or design diagrams with test cases. By incorporating requirements to prioritization scheme would have been more accurate, because test cases are written based on software requirements.

3.4 Summary

The review of the existing literature has shown that various prioritization schemes have been proposed for software testing like source code analysis, heuristics approach etc. Very few researchers addressed the software requirement information for test case prioritization which are incomplete because, none of the work directly propose any method that incorporates all phases of SDLC such as software requirements, design diagrams and source code. The proposed test case prioritization framework which collaborates all the phases of SDLC is presented in the next chapter.

Chapter 4

RDCC: A Test Case Prioritization Framework using Requirements, Design Diagrams, and Source Code Collaboration

Requirements, Design diagram, and source Code Collaboration (RDCC) is the proposed prioritization framework, integrating every phase of SDLC to detect faulty modules earlier. Reviewing the literature shows that without considering information from all phases of SDLC, prioritization approach cannot work properly. In SDLC, requirement engineers assemble SRS documents using direct interaction with customers or end-users. Software designers prepare design diagrams for the development phase on the basis of these SRS documents. Finally software developers develop source code based on these design diagrams. These three phases of SDLC represent the total software from different point of views with individual priority. That is why collaborative SDLC information can illustrate more detail of software than analyzing any single one. The proposed RDCC framework collaborates all phases of SDLC to identify efficient test case ordering. This chapter describes the details RDCC framework including it's architecture, activities, processing algorithms and prioritization approach. The architectural view of the proposed framework is presented in the next section.

4.1 Architecture of RDCC Framework

A layer based architecture is proposed for RDCC framework where different layers are allocated different responsibilities of prioritization process. The architectural component

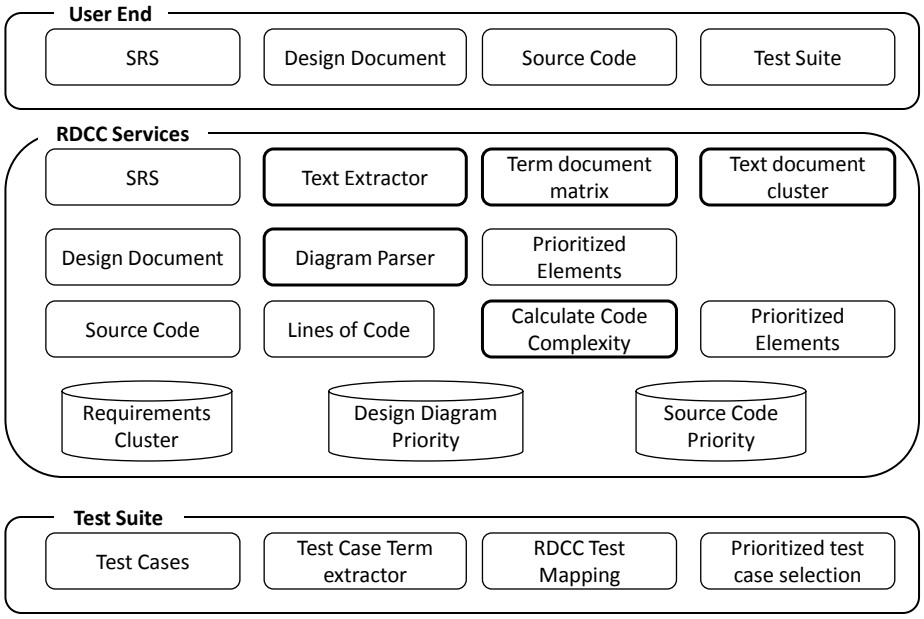


Figure 4.1: Architectural Component Stack of RDCC

stacks of RDCC framework are presented in Figure 4.1, where boxes and cylindrical shapes are depicted the activities and data storage respectively. Boxes, having thick border in Figure 4.1, are the core elements of RDCC, which focus on the overall major activities named as text extraction, term document matrix generation, diagram activities module connection, and code metrics calculation presented in Section 4.3. Boxes containing thin boarder in Figure 4.1 are the supporting elements of the main activities such as stop-word removal, xml parser and non executable code filter which are also presented in Section 4.3. The storage of calculated values from different phases of SDLC named as requirements, design and source code priority value are noted by cylindrical shape in Figure 4.1. RDCC framework is separated into three different layers. Those are listed below.

1. User End Layer
2. RDCC Service Layer
3. Test Case Processing Layer

4.2 Layer 1: User End Layer

The top layer of RDCC framework is named as user end, because it is composed of those components which are directly gathered from end users like customers, developers, requirements and testing engineers. In this layer, requirements, design diagrams, source code and test cases are collected as input from customers, designers, developers and test engineers respectively for whole RDCC framework. Those inputs are the processing elements of RDCC service layer. The top level overview of user end layer is consisted of four core sections of SDLC, named as -

1. Software Requirement Specification (SRS)
2. Software Design Documents
3. Software Source Code
4. Software Test Cases

4.3 Layer 2: RDCC Service Layer

The principle processing layer of this framework is named as RDCC Service layer. It takes different SDLC phases information as input from user end layer and calculates their priority value as output. Requirements are parsed from SRS documents using text extractor. Term document matrix [40] is generated using these extracted text. Connections and dependencies among different modules corresponding to each requirements of a software are parsed from design diagrams. Code metrics are calculated for defining code priority values.

After calculating all priorities from requirements, design diagrams and source code, a new value is generated by using these. This new collaborative priority value is the final outcome of RDCC service layer which is used as the input of test suite processing layer.

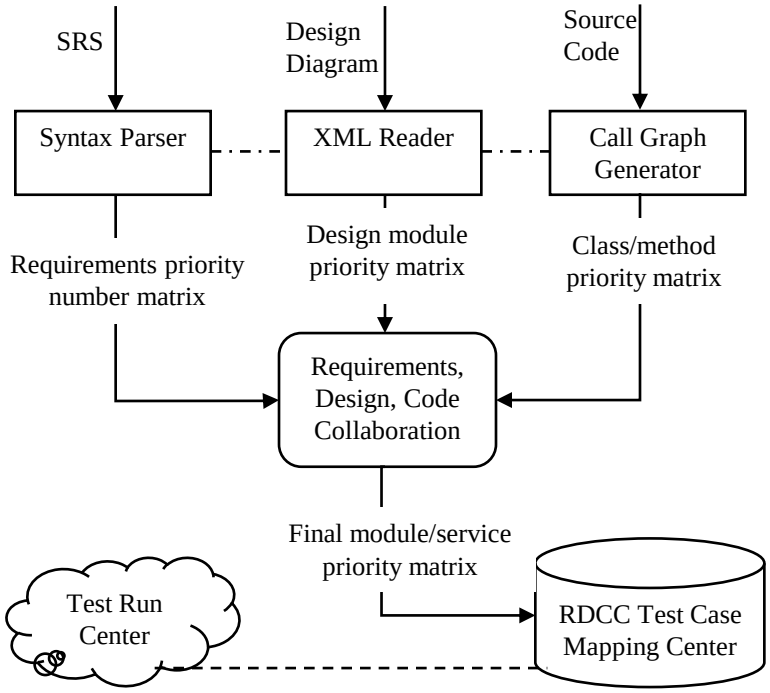


Figure 4.2: Internal Interaction of RDCC Activities

RDCC framework works on different phases of SDLC named as SRS, design, development and testing. The internal interaction among the major activities including their dependencies are presented in Figure 4.2. Requirements, design diagrams and source code are parsed using syntax parser, XML reader and call graph generator respectively. These parsed information are individually prioritized and finally collaborated each other. RDCC test case mapping center (presented in Figure 4.2) uses these collaborative matrix to generate a test case priority list. The whole process is divided into four major activities which are listed below.

1. Requirements prioritization

2. Design diagram prioritization

3. Source code prioritization

4. RDCC module prioritization

The following subsections describe those RDCC service layer activities in details.

4.3.1 Requirement Prioritization Technique

SRS documents usually contain the listed software requirements, which are provided by customers. A list of well documented requirements is mandatory for this phase to parse the representative information. Requirement engineers should follow the naming convention and gathered quality requirements in SRS documents. Because according to Weieger et al. without writing quality requirements in SRS document, a software project may fail to achieve its' success [41]. Next subsections describe how to process and prioritize requirements for test case prioritization.

Requirements Processing

Requirements are analyzed for prioritization because test cases are written based on software requirements. Each requirement is consisted of different structured words like articles, prepositions etc. and non structured words such as nouns, verbs etc. Relationships among all words in every requirement are calculated using textual similarities [42]. Textual similarity is basically used for calculating words relationships in the field of data mining and information retrieval [42, 40].

All the requirements are parsed excluding the stop words which are not related to major words. Priority values are calculated among those parsed requirements using term frequency - inverse document frequency (tf-idf) [10] described later in this subsection. A list of requirements for experimental case study named *Scientific Calculator* are presented in

Table 4.1. There are 8 distinct requirements for processing in that list. This whole requirement processing tasks are split into two major tasks: text extraction, term document matrix creation prioritization.

Table 4.1: Requirements for Scientific Calculator Case Study

Index	Description
1	In any situation the calculator has to produce a correct result defined by the well-known arithmetic rules
2	On encountering a division by 0 the display should read ”Infinity” and typing the key Clear should reset the calculator
3	On calculating the square root value of a negative operand the display should read ”NaN”
4	On erroneous operand or operation keys the display should read Reset (Clear) to continue as appropriate
5	On calculating arcsine, arccosine value input should be within -1 to 1 otherwise the display should read NaN
6	On calculating Ln and Log10 value must be greater than 0 otherwise the display should read -Infinity
7	This application will run and close properly
8	In this application all keys are functional.

a. Text Extraction

Each requirement is considered as a pool of structured words and non structured words. Requirements are inputed as string type which are split into words. According to the example from Table 4.1 Index 1, first requirement of that project contains 17 words or terms. Those words are split and stored into a list including stop words. The stop words that have no specific meaning or that are not related to RDCC are eliminated. There are various word processing and stop word removal libraries like Java Wordnet library [43], Python nltk [44] etc. Hence this framework is implemented in Java programming language, Wordnet library [43] is used for stop word removing and text processing. The split result of first requirement from Table 4.1 Index 1 is presented in Table 4.2. Stop words are eliminated from that list to generate term document matrix which is described in next phase.

Table 4.2: Split Word List for Table 4.1 Index 1

Split Word	In	any	situation	the	calculator	has	to	produce	
Stop Word (Y/N)	Y	Y	N	Y	N	Y	Y	N	
Split Word	a	correct	result	defined	by	the	well-known	arithmetic	rule
Stop Word (Y/N)	Y	N	N	N	Y	Y	N	N	N

Table 4.3: Symbols Using for Equation (4.1), (4.2) and (4.3)

Symbol	**Description**
t	Term or word in a requirement
r	Individual requirement
N	Number of requirements containing term t
R	Set of all requirements
tf	Term Frequency
idf	Inverse Document Frequency
$tf-idf$	Term Frequency Inverse Document Frequency

b. Term Document Matrix Creation and Prioritization

The selected distinct terms obtained from the previous step are used to create a term document matrix. In this matrix the rows and columns correspond to the requirements and the distinct terms respectively. This matrix also contains the frequency of words in the corresponding requirements. Those frequency values of all terms are used for requirements prioritization. The proposed RDCC framework uses term frequency-inverse document frequency (tf-idf) [10] to create term document matrix.

$$tf(t,r) = \frac{f(t,r)}{max\{f(w,r) : w \in r\}} \tag{4.1}$$

$$idf(t,r) = log\frac{N}{|\{r \in R : t \in r\}|} \tag{4.2}$$

$$tf-idf(t,r,R) = tf(t,r) \times idf(t,r) \tag{4.3}$$

The term frequency $tf(t,r)$, is the number of occurrences of a term t in a requirement r. On the other hand, the inverse document frequency, $idf(t,r)$ is the logarithm of the ratio for the total number of requirements and the number of requirements containing the term t. The mathematical model of term frequency $tf(t,r)$ and inverse document frequency $idf(t,r)$ are presented by Equation (4.1) and (4.2) respectively. Table 4.3 presents the symbols and description for Equation (4.1) and (4.2). The tf-idf(t,r,R) value of a requirement is calculated by multiplying the term frequency and inverse document frequency which is presented by Equation (4.3). Final calculated tf-idf values of every requirement r, is used for requirements prioritization. Requirements list is sorted in descending order where higher value of requirement priority denotes the higher possibilities of fault detection.

Every requirement has a requirement ID which is used as RDCC ID in this framework. The priority of each RDCC ID is calculated by the division of the term priority values and the sum of all priorities which is presented by Equation (4.4) where i and j represent the Requirement ID and term ID number respectively. The number of terms containing by a requirements are denoted by t. The final value is normalized from 0 to N by dividing the maximum priority value and multiplying the result into N which are presented in Equation (4.5). The symbols used in Equation (4.4) and (4.5) are presented in Table 4.3. The value of N is may be any unsigned integer which may vary from project to project. For case study 'Scientific Calculator', the value of N is considered as small value (for example N=10), because this project is a tinny project containing 8 requirements.

$$Req_i = \sum_{j=1}^{t} TermID_{ij} \tag{4.4}$$

$$ReqPriority_i = \frac{Req_i}{Max\sum_{i=1}^{r} Req_i} \times N \tag{4.5}$$

Table 4.4: Term Document Matrix Scientific Calculator Case Study

Term	situation	calculator	result	... ...
Req# 1	0.301	0	0	... ...
Req# 2	0	0.145	0.11	... ...
Req# 3	0.191	0.14	0	... ...
Req# 4	0.189	0.3	0.31	... ...

Table 4.5: Requirement Priorities Values

Index	**Priority Value**
1	10
2	7.7951269855
3	7.2567611435
4	7.6043005037
5	7.9889158883
6	7.2567611435
7	9.5192307692
8	7.4019663453

Table 4.4 shows an example of the term-document matrix of four case study requirements (presented in Table 4.1). After performing term extraction, a set of distinct terms including *situation*, *calculator*, *result* are identified. The $tf \times idf$ values of every term for each requirement are calculated. For instance, the calculated $tf \times idf$ value of term *calculator* for Req.# 2, 3, and 4 are 0.145, 0.14, and 0.3 respectively. The final requirement priorities list of Scientific Calculator case study is presented in Table 4.5 containing the requirement index and its corresponding priority value. For this case study the maximum and minimum priority values are 10 and 7.256. Based on Table 4.5, the descending order sorted requirements are Req 1, Req 7, Req 5, Req 2, Req 4, Req 8, Req 3 and Req 6.

Algorithm for Assigning Requirement Priorities

The requirement priorities assigning algorithm defines the requirements processing and prioritization approach. Requirements are vital part for test case prioritization, because final

Table 4.6: Symbols for Requirement Priorities Algorithm

Symbol	Description
R	Set of all requirements
r	Individual requirement
$\mathcal{P}1$	Set of requirement priorities
p_i	Individual requirement priorities
t	Term or word in a requirement
tf	Term Frequency
idf	Inverse Document Frequency
$tf-idf$	Term Frequency Inverse Document Frequency

products are validated by software requirements. Algorithm 1 presents the requirements processing algorithm. This algorithm takes the set of requirements R, priority list $\mathcal{P}1$, priority function: $R_i \rightarrow \mathcal{P}1_i$ as input. It returns an assigned priority list. This algorithm uses Equation (4.1 and 4.2) for requirements processing. The symbols used in Algorithm 1 are noted in Table 4.6.

In the very beginning of this process, every requirements are split into words. These words are the combination of stop words and necessary words. Stop words are removed from requirements. After removing the stop words, only important words are listed for execution. The sentence split and stop word removing process are presented in Algorithm 1 Line 2. These words are usually named as terms for creating term document matrix. Initially the requirement priority list $\mathcal{P}1$ is initialized to 0 for every requirements whih is presented in Algorithm 1 Line 3.

The term list which is generated from stop word filtering process, are used to calculate term frequency. Equation (4.1) is used to calculate term frequency value for all terms in a requirement presented in Algorithm 1 Line 7. Inverse document frequency is also calculated using Equation (4.2) for every requirement in the document presented in Algorithm 1 Line 8. Finally the summation of term tf-idf values are calculated using Equation (4.3) and stored in priority list $\mathcal{P}1$. The term document matrix is generated using these tf-idf values. The whole processing section is presented in Algorithm 1 Line 7-11.

The priority values are updated by dividing the maximum value of $\mathcal{P}1$. Priority values for each requirements in $\mathcal{P}1$ are normalized from 0 to N by multiplying final priority value with N. If the priority contains negative value, 0 is assigned to these requirements. This process confirms the priority value range from 0 to N. The priority assigning process is presented in Algorithm 1 Line 15-19.

Algorithm 1 Algorithm for Assigning Requirement Priorities

Input: Set of requirements R, Priority list $\mathcal{P}1$, Priority function: $R_i \rightarrow \mathcal{P}1_i$
Output: Assigned priority list for requirements $\mathcal{P}1$

1: **Begin**
2: $R \leftarrow filterstopword(R)$
3: $\mathcal{P}1 \leftarrow \{\}$
4: **for each** requirements $r_i \in R$ **do**
5: $sum \leftarrow \{\}$
6: **for each** term $t \in r_i$ **do**
7: tf = calculatetf(t, r) using Equation (4.1)
8: idf = calculateidf(t, r) using Equation (4.2)
9: tf-idf = $tf \times idf$
10: $sum += tf - idf$
11: **end for**
12: $\mathcal{P}1_i \leftarrow sum$
13: **end for**
14: **for each** priority $p_i \in \mathcal{P}1$ **do**
15: **if** $p_i < 0$ **then**
16: $p_i \leftarrow 0$
17: **else**
18: $p_i \leftarrow \frac{p_i}{max(\mathcal{P}1)}$
19: **end if**
20: **end for**
21: **End**

4.3.2 Software Design Prioritization Technique

Software design prioritization is also an important phase of this framework because design diagrams contain the designers view points. Every view point is necessary to produce an efficient test case ordering. To process the design diagrams, information from different

Table 4.7: State Connectivity Values of Scientific Calculator Case Study

	Minus Sign Performed	Power Sign Performed	Plus Sign Performed	Cos Sign Performed	Scientific Calculation
Minus Sign Performed	0	0	0	0	1
Power Sign Performed	1	0	0	0	1
Plus Sign Performed	0	0	0	0	1
Cos Sign Performed	1	0	1	0	0
...	...	...	...	...	...
...	...	...	...	...	...
Scientific Calculation	1	1	1	0	1

diagrams like Unified Model Language (UML) and state transactions are extracted. The diagram reader takes design diagrams from designers and forwards those to the Extensible Markup Language (XML) converter component for generating program readable XML format, because program cannot take any input directly from the diagrams. There are different XML converter tools like eclipse graphical modeling [45], and enterprise architect [46] etc. This framework uses enterprise architecture for converting diagrams to XML files.

Table 4.8: Design Diagrams Priority Values for Case Study Scientific Calculator

State	**Priority Value**
Minus Sign Performed	1.2757731959
Power Sign Performed	1.2757731959
Plus Sign Performed	1.2757731959
Cos Sign Performed	1.2693298969
Scientific Calculation	7.0833333333

The XML Converter produces XML files that can present the major activities and their related sub actions. The inter-connectivity values among the major activities are calculated and prioritized using their relationships. Table 4.7 presents some sample state interconnectivity values from Scientific Calculator case study. Those states priorities are listed in

Table 4.8. According to Table 4.8, state named Scientific Calculation, contains most priority value among all other states. So test cases related to Scientific Calculation state can detect early faults in testing phase.

Table 4.9: Symbols for Design Diagram Priority Algorithm

Symbol	Description
D	Design diagram XLM
E	Set of elements after XML parsing
$\mathcal{P}2$	Set of design element priority
p_i	Individual design element priority
e	Individual design element
d	Individual design element

Algorithm 2 Design Diagram Priority

Input: Design diagram XLM D, Priority list $\mathcal{P}2$, Priority function: $D_i \rightarrow \mathcal{P}2_i$
Output: Assigned priority list for design diagram $\mathcal{P}2$

Begin
$E \leftarrow XMLParser(D)$
$\mathcal{P}2 \leftarrow \{\}$
for each element $e \in E$ **do**
    for each element $d \in E \setminus \{e\}$ **do**
        if $hasRelation(e,d)$ **then**
            $\mathcal{P}2_e \leftarrow \mathcal{P}2_e + 1$
            $\mathcal{P}2_d \leftarrow \mathcal{P}2_d + 1$
        end if
    end for
end for
for each priority $p_i \in \mathcal{P}2$ **do**
    if $p_i < 0$ **then**
        $p_i \leftarrow 0$
    else
        $p_i \leftarrow \frac{p_i}{max(\mathcal{P}2)}$
    end if
end for
End

The algorithm for assigning design priority is illustrated in Algorithm 2. This algorithm takes XML design diagram D and set of design element priority $\mathcal{P}2$ as input. It generates a weighted priority list for design elements corresponding to each requirements. XML Parser

is used to retrieve information form design XML. Table 4.9 presents the important symbols which are used in this Algorithm 2.

Before executing the Algorithm 2, priority list $\mathcal{P}2$ is initialized by 0 for all design elements. This algorithm calculates the relationship between every pair of elements (for example, every state in state-transition diagram). The relationships are calculated among all states in design diagrams which is presented at Algorithm 2, Line 4 and 5. If there is a relation between any two elements, the priority value is updated for both elements which is denoted at Line 7 and 8. For an example, if two states named Power Sign Performed and Scientific Calculation (Table 4.7) are connected to each other, the 2nd and 5th position values in priority list are incremented.

The priority values are updated by dividing the maximum value of $\mathcal{P}2$. Priority values for each state in $\mathcal{P}2$ are normalized from 0 to N by multiplying final priority value with N. If the priority contains negative value, 0 is set for those design states priorities. The priority assigning and normalization process is presented in Algorithm 2, Line 12 to 18. Finally the priority list is sorted decreasingly for finding the important design elements earlier. Because those states may be error prone which are densely related to other elements [5].

4.3.3 Software Source Code Prioritization Technique

Source code is one of the major elements of SDLC containing the real development information. Source code contain the developers' view points about a software which may vary from designers or requirement engineers. Developers need to follow the naming code convention for this framework. If developers failed to follow naming convention, code parser cannot determine the expected phase and fails to track the implementation of required requirements. Source code is prioritized based on source code metrics information.

Source Code Metrics

A software code metrics is a set of quantitative measures that provides overall description about a software [47]. The code metric value is calculated using different types of information obtaining from source code for example executable lines, passing parameters and functions calls. Figure 4.3 presents a sample code metrics list with values. RDCC framework analyzes the most leading paradigm named Object Oriented Programming (OOP), which is developed using class concept where classes are the main code element. Four most prominent code metrics for OOP [48] are used in this research to calculate source code priority. Those code metrics are named as -

1. Lines of Code
2. McCabe's Cyclomatic Complexity
3. Weighted Methods per Class
4. Nested Block Depth

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum
⊞ Total Lines of Code	782				
⊞ Method Lines of Code (avg/max per method)	511	12.775	10.869	43	/TestCasePrioritization/src/codeParser/CodeMetricXmlPars...
⊞ McCabe Cyclomatic Complexity (avg/max per metho		2.85	3.127	14	/TestCasePrioritization/src/codeParser/Related_Info_Colle...
⊞ Nested Block Depth (avg/max per method)		2.05	1.612	10	/TestCasePrioritization/src/codeParser/CodeMetricXmlPars...
⊞ Weighted methods per Class (avg/max per type)	114	9.5	6.007	19	/TestCasePrioritization/src/codeParser/Related_Info_Colle...
⊞ Number of Parameters (avg/max per method)		0.65	1.038	5	/TestCasePrioritization/src/rdccFinalPointCalculation/RDCC...
⊞ Number of Static Attributes (avg/max per type)	0	0	0	0	/TestCasePrioritization/src/codeParser/CodeMetricXmlPars...
⊞ Efferent Coupling (avg/max per packageFragment)		0.714	1.03	3	/TestCasePrioritization/src/codeParser
⊞ Depth of Inheritance Tree (avg/max per type)		1	0	1	/TestCasePrioritization/src/codeParser/CodeMetricXmlPars...
⊞ Specialization Index (avg/max per type)		0	0	0	/TestCasePrioritization/src/codeParser/CodeMetricXmlPars...
⊞ Number of Classes (avg/max per packageFragment	12	1.714	0.7	3	/TestCasePrioritization/src/codeParser

Figure 4.3: Sample Source Code Metrics List

Lines of Code (LOC) [49]

LOC is an important code metric for OOP architecture because it can measure how large a class is. LOC counts all instruction from the source code. The executable lines

of code are only considered in RDCC framework. It ignores the comments and the blank lines inside the code. Listing 4.1 presents a sample OOP source code class written in Java. In this class there are 21 lines of code where the numbers of blank lines, comments and executable lines are 3, 2 and 16 respectively. That means the LOC value of this sample class is 16.

Listing 4.1: Sample Source Code

```
1  public class Related_Info_Collection_using_Method {
2
3  void writeToFile()
4  {
5    try{
6      // Create file
7      FileWriter fstream = new FileWriter("scientificCalculator\\CodeMetric-point.txt");
8      BufferedWriter out = new BufferedWriter(fstream);
9
10     for (Map.Entry<String, Double> entry : finalAllMetric.entrySet()) {
11       out.write(entry.getKey() + " " + entry.getValue() + "\n");
12     }
13
14     //Close the output stream
15     out.close();
16   }
17   catch (Exception e){//Catch exception if any
18     System.err.println("Error: " + e.getMessage());
19   }
20 }
21 }
```

McCabe's Cyclomatic Complexity (MCC) [50]

Cyclomatic complexity is a software metric which is used to calculate the complexity of a program [50]. MCC counts of the number of linearly independent paths in a source code. For calculating MCC, a direct acyclic graph is generated from software source code. The nodes of the graph represent the distinct groups of statements or commands in program. The direct edges connect two nodes, if the second one has executable dependencies on the first one. Cyclomatic complexity can be applied on classes, methods, and functions.

$$M = E - N + 2P \tag{4.6}$$

$$M = E - N + P \tag{4.7}$$

The mathematical formulation of calculating cyclomatic complexity can be divided in two scenario. The first one is when the exit and end nodes are connected. That means the graph is strongly connected. In this scenario the complexity M is defined by Equation (4.6). The second scenario is when the start and end nodes are connected which are presented by Equation (4.7). Where E, N and P denotes the number of edges, nodes and connected components of a graph respectively.

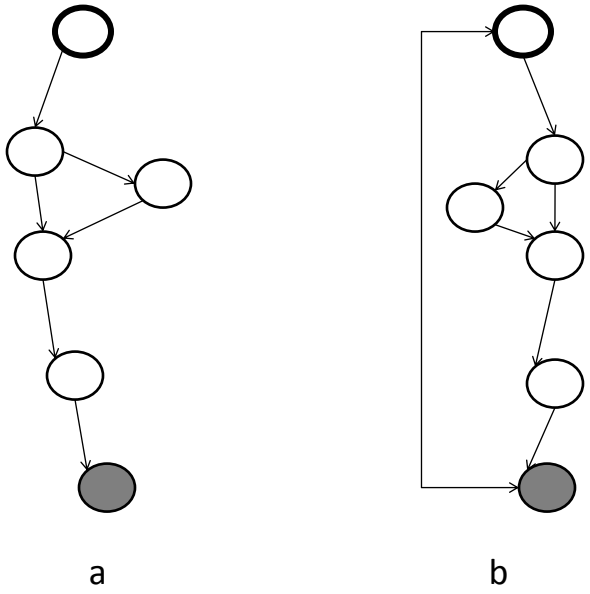


Figure 4.4: McCabe's Cyclomatic Complexity (MCC) Example

In Figure 4.4 a, the calculation of cyclomatic complexity for first scenario using Equation (4.6) is M = 6 - 6 + 2 * 1 = 2. On the other hand, another scenario presented in Figure 4.4 b, the calculate MCC value is M = 7 -6 + 1 = 2. So in every scenario MCC produce exactly same result.

Weighted Methods per Class (WMC) [51]

WMC is the third considered code metric which is only applicable for OOP code. It is an indicator of how much effort is required to maintain and develop a particular class. WMC is calculated by using all the methods in a class which are called by other classes. An example, if one class has 5 methods and 3 are called by other classes using this class object. The WMC of that class is 3. In this process, all the class WMC are calculated one by one. In OOP, a class with a low WMC usually denotes to greater polymorphism. Conversely a class with a high WMC indicates that the class is complex. That means classes containing the higher value of WMC, have the higher possibility of error.

Nested Block Depth (NBD) [51]

NBD can be calculated for OOP source code. It can measure the impact length of any block or function. It is calculated by the numbers of nested defined function or nested calling from a method in a class. The higher value of NBD is usually defined on complex and densely connected one. This types of functions or classes having a lots of dependability which may lead to error prone classes or functions. RDCC basically sorts test cases in such an order, where error and defected classes or functions are early detected.

Source Code Prioritization Equation

Using these variables for each class, Source Code Metric (SCM) is calculated using the Equation (4.8). The values of LOC, MCC, NBD and WMC are divided by their maximum values. Finally, those results are averaged by Equation (4.8), where *i* represent the source code class ID in OOP.

$$\mathcal{P}3 = \frac{\frac{LOC}{max(LOC)} + \frac{MCC}{max(MCC)} + \frac{WMC}{max(WMC)} + \frac{NBD}{max(NBD)}}{count(LOC, MCC, WMC, NBD)} \tag{4.8}$$

Table 4.10: Symbols for Source code Priority Algorithm

Symbol	Description
C	Source Code
c	Individual Class in OOP Code
f	Individual Function in non OOP Code
cm	Code Metrics
LOC_i	Lines of Code for i^{th} class/function
MCC_i	McCabe's Cyclomatic Complexity for i^{th} class/function
WMC_i	Weighted Methods per Class for i^{th} class/function
NBD_i	Nested Block Depth for i^{th} class/function
$\mathcal{P}3$	Set of Source Code Priority
p_i	Individual Source Class/Function Priority

Source Code Prioritization Algorithm

Source code is prioritized based on several code metrics which are LOC, MCC, WMC and NBD. Algorithm 3 presents the whole source code processing and prioritizing approaches to detect vulnerable code sections.

This algorithm is designed for OOP source code. Algorithm 3 takes source code C, priority list $\mathcal{P}3$ and priority function: $C_i \rightarrow \mathcal{P}3_i$ as input. A sorted source code priority list is generated as output to investigate error prone modules earlier. The symbols used in Algorithm 3 are defined in Table 4.10.

In the beginning of Algorithm 3, four different code metrics priority list LOC_i, MCC_i, WMC_i, NBD_i are initialized by 0 values using Line 2-5. If the source code is developed by OOP, all four metrics are being calculated for every class presented at Algorithm 3 Line 7-13. On the other hand, if the source code is developed by non OOP, all other code metrics are calculated without Weighted Methods per Class (WMC) for every function presented at Algorithm 3 Line 14 to 19.

Those calculated code metrics are normalized by dividing their current and maximum value among their priority list. This process is executed for every code metric in the priority list which is presented at Algorithm 3 Line 21-26. Finally the average value of those four

Algorithm 3 Source Code Priority Algorithm

Input: Source Code C, Priority list $\mathcal{P}3$, Priority function: $C_i \rightarrow \mathcal{P}3_i$
Output: Assigned priority list for requirements $\mathcal{P}3$

Begin
$LOC_i \leftarrow 0$
$MCC_i \leftarrow 0$
$WMC_i \leftarrow 0$
$NBD_i \leftarrow 0$
$\mathcal{P}1 \leftarrow \{\}$
if Source Code follows OOP **then**
    for each class $c \in C$ **do**
        $LOC_i \leftarrow calculateLOC(c)$
        $MCC_i \leftarrow calculateMCC(c)$
        $WMC_i \leftarrow calculateWMC(c)$
        $NBD_i \leftarrow calculateNBD(c)$
    end for
else
    for each function $f \in C$ **do**
        $LOC_i \leftarrow calculateLOC(f)$
        $MCC_i \leftarrow calculateMCC(f)$
        $NBD_i \leftarrow calculateNBD(f)$
    end for
end if
for each values $v_a \in LOC, v_b \in MCC, v_c \in WMC, v_d \in NBD$ **do**
    $LOC_i \leftarrow \frac{v_a}{max(LOC)}$
    $MCC_i \leftarrow \frac{v_b}{max(MCC)}$
    $WMC_i \leftarrow \frac{v_c}{max(WMC)}$
    $NBD_i \leftarrow \frac{v_d}{max(NBD)}$
end for
for each priority $p_i \in \mathcal{P}3$
    $p_i \leftarrow$ average(LOC, MCC, WMC, NBD) **do**
end for
End

normalized code metrics are assigned to the source code priority list $\mathcal{P}3$ which is denoted in Algorithm 3 Line 27-29.

4.3.4 RDCC Module Integration and Prioritization

Integration of RDCC module is one of the important phases of this framework, because in this phase the different SDLC phases priorities are combined together. After calculating the priority of every term in requirements, design diagrams and source code, integrated priority needs to be calculated.

Table 4.11: Symbols for RDCC Priority Equation (4.9)

Symbol	Description
W_i	Final weight of RDCC ID$_i$
t	Number of RDCC IDs
$\mathcal{P}1$	Requirements prioritization values (0...N)
α	Requirements prioritization weight
$\mathcal{P}2$	Design prioritization values (0...N)
β	Design prioritization weight
$\mathcal{P}3$	Code prioritization value (0...N)
γ	Code prioritization weight

$$W_i = \alpha \times \mathcal{P}1 + \beta \times \mathcal{P}2 + \gamma \times \mathcal{P}3 \quad \forall_{i=1,2,3,\ \ldots\ \ldots,\ t} \tag{4.9}$$

$$Subject\ to: \ 0 \leq \alpha, \beta, \gamma \leq N \ \ and \ \ \alpha + \beta + \gamma = N$$

Equation (4.9) explains the collaborative viewpoints by calculating the values of every RDCC ID. Symbols used in Equation (4.9) are described in Table 4.11. The calculated values from requirement specifications, design diagrams and source code are multiplied by

their weight constants α, β, and γ respectively. The weight constants may vary from 0 to N and the sum of all three constants must be equal to N. According to Pressmen, all phases of SDLC are equally important to represent a whole software development [5]. That is why, equal weights are assigned to every constant α, β, and γ. According to the Equation (4.9), the value of α, β and γ is equal to $N/3$. The sum of all multiplied weighted values calculates the final weight. This calculation are executed until all the RDCC IDs are processed.

Table 4.12: List of Requirement IDs, Design Modules and Classes of Case Study Scientific Calculator

Requirement ID	Design Module	Source Class
1	Plus Sign Performed, Minus Sign Performed	operation.java
2	Button Functional, Scientific Calculation	jframeui.java
...	...	...
...	...	...
8	Scientific Calculation	scicalculator.java

A list of related design modules and source code classes corresponding to each requirements for *Scientific Calculation* case study are presented in Table 4.12. One requirement may be connected to multiple design modules or classes. Since requirement IDs are equally identified as RDCC IDs, the cumulative priorities of related design modules (calculated from Subsection 4.3.2) and classes (calculated from Subsection 4.4.3) are calculated as final RDCC priorities.

Table 4.13 presents the final RDCC priority values which are calculated using Equation (4.9) for case study *Scientific Calculator*. The RDCC IDs and their final priorities are listed in that table. The highest and lowest priority values are 27.401 and 8.441 respectively which are contained by RDCC ID 8 and 4. This result is naturally ordered and used to calculate test case priorities in RDCC - test case mapping section which are presented in next section.

Table 4.13: Priority Values for Case Study Scientific Calculator

RDCC ID	Weighted Priorities
1	24.12236873
2	9.47901193
3	9.120884931
4	8.440920963
5	10.76049509
6	8.517013845
7	11.19247169
8	27.40196635

4.4 Layer 3: Test Case Processing Layer

The third and final layer of RDCC architecture is named as test case processing where every test case is analyzed and mapped with prioritized RDCC module. This layer takes final priority value from RDCC service layer and generates an ordered list of test cases using these values for test case prioritization.

Test cases are written based on the customer requirements to verify the expected final products. That is why every test cases must have a relationship to at least one requirement. Requirements traceability matrix is that matrix where the relations between requirements and test cases are stored. Using the values from that matrix and prioritized RDCC from previous layer, test cases are listed in descending order. The priority value of a test case is proportional to fault detection. That means test cases containing high priority values have high fault detecting possibilities. Test case processing layer can be explained in two different phases, which are explained below.

4.4.1 RDCC - Test Case Mapping Resolution

Test cases are mapped and prioritized in RDCC - test case mapping resolution phase. Test engineers generate the test cases based on requirements and moc user interface [52]. A re-

Table 4.14: RDCC Test Cases Mapping Matrix of Scientific Calculator Case Study

Test Case ID	**RDCC ID**	**Test Case ID**	**RDCC ID**	**Test Case ID**	**RDCC ID**
1	7	13	1,4	26	1,4
2	8	14	1,4	27	2,4
3	7	15	1,4	28	6,4
4	7	16	1,4	29	6,4
5	8	17	1,4	30	5,4
6	8	18	1,4	31	5,4
7	8	19	1,4	32	1,4
8	8	20	1,4	33	1,4
9	8	21	1,4	34	1,4
10	8	22	1,4	35	1,4
11	8	23	1,4	36	1,4
12	1,4	24	2,4	37	3
		25	1,4		

quirements traceability matrix, is generated in test generation phase including requirements and their corresponding test cases. Since, requirements IDs are used as RDCC IDs, that means the RDCC - test cases mapping resolution can be found in that table. This table is called test case traceability matrix where every test case is assigned to at least requirements.

Table 4.14 presents the traceability matrix for case study *Scientific Calculator*. According to Table 4.14 test cases ID 37 is related to RDCC ID 3. That means for partial fulfillment the 3rd requirement, 37th test case is executed.

Since every test case is mapped with requirements in test case matrix (presented in previous section), those requirements priorities can be used for their assigning test cases. If one test case is assigned to multiple requirements, the priority value of that test case is calculated by the summation of all assigned requirement priorities values.

Table 4.15: Natural Order of Test Cases with Priorities and Execution Report for Scientific Calculator Case Study

Test Case ID	**Priority**	**Passed/-Failed**	**Test Case ID**	**Priority**	**Passed/-Failed**
1	11.192	0	19	32.563	0
2	27.401	1	20	32.563	0
3	11.192	0	21	32.563	1
4	11.192	0	22	32.563	0
5	27.401	1	23	32.563	1
6	27.401	1	24	17.919	1
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
16	32.563	0	34	32.563	1
17	32.563	0	35	32.563	1
18	32.563	1	36	32.563	0

4.4.2 Test Case Prioritization and Selection

Test cases are prioritized finally in this phase. Test cases are also ordered based on their assigned priority values. The weighted priority sum of each RDCC IDs are calculated by the summation of the priority values from different phases of SDLC (presented in Table 4.11). Finally, test cases are sorted in descending order based on those priorities.

According to mapping matrix from Table 4.14, test case ID 12 is assigned to RDCC ID 1 and 4. The final RDCC priority values of ID 1 and 4 are 24.122 and 8.441 respectively which is presented in Table 4.13. So the priority value of test case ID 12 is 24.122 + 8.441 = 32.563. All the priority values are calculated based on this process. The natural order of test cases with priorities and execution reports are presented in Table 4.15. The highest and lowest priority values for 'Scientific Calculator' case study are 32.563 and 9.121 respectively.

Table 4.16 presents the final order of test cases for *Scientific Calculator* case study which are sorted in descending order based on assigned priorities. The execution reports 0 and 1 denotes the fault detection and successfully execution respectively. This order of

Table 4.16: RDCC Sorted Order of Test Cases with Priorities and Execution Report for Scientific Calculator Case Study

Test Case ID	**Priority**	**Passed/-Failed**	**Test Case ID**	**Priority**	**Passed/-Failed**
12	32.563	1	2	27.401	1
13	32.563	0	5	27.401	1
14	32.563	0	6	27.401	1
15	32.563	1	7	27.401	1
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
34	32.563	1	4	11.192	0
35	32.563	1	37	9.121	1

test cases can detect faults earlier than any other prioritization techniques like requirements processing only [7], code analysis only [9] and natural order or test cases.

4.5 Summary

In this chapter, a layer based framework for prioritizing test cases named RDCC is presented. Multiple phases of SDLC named requirement specifications, design diagrams and source code are collaborated in this framework. Test cases are prioritized in RDCC framework based on those collaborative information. Requirements are parsed as textual similarity and prioritized using term frequency - inverse documents frequency. Design diagrams are prioritized based on internal state connections corresponding to each requirements. Source code is prioritized using different code metrics named as LOC, MCC WMC, and NBD. The architecture and details procedure of RDCC are also presented in this chapter. The whole framework is explained using a real life case study containing 8 requirements, 37 test cases and 12 faults. Experiments results of RDCC framework and their explanation is presented in very next chapter.

Chapter 5

Experimental Setup and Analysis of Results

This chapter presents the implementation schemes of the proposed test case prioritization framework along with its evaluation on early fault detection. The RDCC framework is tested with six applications which are developed by undergraduate Software Project Lab at IIT, University of Dhaka. Initially a brief description of those applications are presented with their requirements, activities and test cases. The implementation schemes of RDCC are provided having an insight of the used libraries like Wordnet [53], XML parser [46] etc. The performance of RDCC is compared to several most prominent approaches which are natural orders, requirements [7] and source code based [8] test case prioritization. The detail regarding the experimental setup of those schemes are also provided. Finally, the performance of RDCC is evaluated on the basis of early fault detection and percentage of test case execution.

5.1 Experimental Setup

This section presents the experimental setup including environmental and test applications for prioritization. For experimenting, execution environments and test beds are needed to be prepared. Several supporting packages such as Wordnet, metric Plugin etc. are used for experiment. Six different applications developed including offline desktop and online web apps, have been used to evaluate the performance of RDCC. Initially a description of these six applications are provided with configuration details (requirements, activities and test cases). Proposed RDCC and three other approaches are implemented on those datasets using the same computing environmental setup. This section highlights all the test-bed details of this thesis.

5.1.1 Environmental Setup

The proposed RDCC framework and other prioritization schemes are implemented using Java programming language and javac compiler in Eclipse editor. For experimentally implementing RDCC framework, requirements, design diagrams and source code are analyzed by text processing, diagram parsing and code metric calculating. The predefined supporting packages and experimental configurations are briefly described below.

WordNet: A Lexical Database of English [53]

WordNet is a lexical database of English language where nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms [53]. WordNet is used for processing words in requirements. It is implemented to filter stop words and find the base form of verb in the requirement list.

Java API for WordNet Searching (JAWS) [54]

JAWS is an API that provides Java applications with the ability to retrieve data from the WordNet database. JAWS version 3.0 is used to implement RDCC framework. Since the whole framework is implemented in Java programming language, this API is used to integrate WordNet database for processing software requirements using Java.

Enterprise Architect [46]

Enterprise Architect is a design tool for software development where various design diagrams like UML, class diagram etc. can be drawn [46]. This tool is used to draw design diagrams for standardization purposes, because this tool can generate background XML

format of any diagrams. Those XML formats are used to calculate design priority in the experiment of this research.

Eclipse Metric Plugin [55]

Eclipse metric plugin is a metric calculation and dependency analyzer plugin for the Eclipse platform [55]. Since the whole framework is developed in Eclipse editor, this plugin is used to retrieve code metrics from software source code.

Hardware and Software Configuration

The proposed RDCC framework and other prioritization techniques are compiled and executed in a desktop computer at the masters lab of IIT, University of Dhaka. The hardware and software configuration of that computer are listed below.

1. Operating System: Linux Mint15
2. Platform: 32bit
3. Processor Model: Intel Core-i3
4. Processor Speed: 2.20 GHz
5. Cache Memory: 6 MB
6. RAM: 4 GB

5.1.2 Test Applications

The applications chosen for conducting experiment are developed by undergraduate final Software Project Lab at IIT, University of Dhaka. All of those applications are developed by preparing every SDLC phases named as software requirements, design, source code and test cases. A brief description of those applications and their deployment configuration are provided below.

Table 5.1: Dataset Information for Test Case Prioritization

Dataset Name	**Number of Require-ments**	**Design Diagram**	**LOC**	**Number of Test Cases**
News-A: An Online News Portal	14	Yes	356	74
Scientific Calculator	8	Yes	636	37
Sparrow: File Reading Software	20	Yes	752	25
Amghotok: A Platform of Marriage	13	Yes	953	20
Painter: A Canvas for Painting Freely	12	Yes	1021	16
POAS: Program Office Automation Software	18	Yes	4037	62

News-A: An Online News Portal [56]

An online news portal named as News-A is developed for retrieving and displaying all the major information in Bangladeshi newspaper like The Prothom Alo, The Daily Star, etc. This application shows the news in details based on the given headlines with images. It displays the news in different categories named as sports, education, business, etc. It can support both Bangla and English language for scrolling different online news paper.

News-A is developed by Java programming language using jsoup web service version

1.8.1 [57]. It contains 14 requirements and 356 LOCs. News-A has 74 different test cases to cover all expected requirements. All the statistical information are presented in Table 5.1. Appendix A describes the detail explanation of requirements, test cases, diagrams etc of News-A application.

Scientific Calculator [58]

Scientific Calculator is an offline desktop application that provides a graphical calculator from where users can provide different mathematical operations to solve. Users can provide different operators (power, square root, etc) and operands (combination from value 0 to 9). Users can also provide geometrical input like sine, cosine, logarithm etc. Scientific calculator calculates those operations and generates result.

This application contains 8 distinct requirements, single state transition and class diagram. Scientific Calculator is developed by Java programming language where the user interfaces are designed by javax.swing package. This application contains 636 LOCs and 51 different methods for developing source code listed in Table 5.1. Finally 37 distinct test cases are used to detect faults in the testing phase. All the detail information about the Scientific Calculator application are presented in Appendix B.

Sparrow: File Reading Software [59]

Sparrow is an offline desktop speech synthesizer program for visually impaired peoples to use computer. By using Sparrow users can get the voice of formatted and unformatted text file like PDF, DOC, TXT, etc. That means he/she will be able to read a book using the virtual voice from a computer. Users can also use the voice effect to set the frequency of the virtual voice. Only English language is processed by this application for text to speech.

Sparrow contains 20 requirements and state transition diagrams prepared by requirements engineers and designers respectively. This application is developed by Java programming language where Mbrola voices [60] are used for virtual voice generation. It has 752 LOCs and 25 distinct test cases for prioritization which are presented in Table 5.1. The requirements, test cases and other information about this application are presented at Appendix C.

Amghotok: A Platform of Matchmaking [61]

Amghotok is a popular android application which provides a platform of matchmaking. Users can search potential bride and groom information based on their qualifications. Users can also upload new or update old information. Those information are stored in a web server and processed an android application. Users must pay registration fees for every successful processing.

Amghotok is developed by Java language using Android version 4.3 Jelly Bean. This application contains 13 distinct requirements, single state transaction diagram and 953 LOCs. It has 20 test cases for detecting faults from this application. The major information are reported in Table 5.1 and the details are explained at Appendix D.

Painter: A Canvas for Painting Freely [62]

Painter is an offline desktop application for painting freely in any canvas. Users can use different painting tools such as pencil, eraser, brush, smudge, etc. Users can also use various colors and shapes like polygon, triangle, rectangle, etc. for drawing. External images can be uploaded as canvas background or edited manually. Users can save their painting in different format like png, jpeg, etc.

Painter application is developed by using Java programming language where javax.swing package is used for designing user interfaces. This application consists of 12 unique requirements, 508 LOCs and 20 test cases which are presented in Table 5.1. The detail information including requirements, test cases, etc are presented in Appendix E.

POAS: Program Office Automation Software [63]

The Program Office Automation Software (POAS) is an automated accounting software for maintaining the expenditure to conduct different academic programs (such as bachelors and masters) at IIT, University of Dhaka. The academic staffs are the primary stakeholders of this application. Users can store all the academic expenditure information including student payment, examination budget, office stationary etc. in a database for further usage. Users can get an expenditure printable pdf report as output.

POAS contains 18 requirements and multiple state transition diagram to represent the whole scenario. It is developed by Java programming language and MySQL database. User interfaces of POAS are designed by javax.swing package. This application contains 4037 LOCs and 62 test cases which are listed in Table 5.1. The explanation of POAS dataset are presented at Appendix F.

5.2 Implemented Test Case Prioritization Approaches

The proposed RDCC framework is experimented and compared to source code [8] and requirements [7] based test case prioritization approaches, and natural order of test cases for early fault detection. The additional function coverage, and risk and severity analysis techniques are used as the representative of source code and requirements based prioritization approach in experimental setup of this thesis.

5.2.1 Additional Function Coverage: A Code Based Prioritization

The Additional Function Coverage (AFC) is a code based prioritization approach for early fault detection where the functions are the major part to assign weights on a test case [8]. This prioritization technique depends on information relating the test suite to various elements of source code of the original system like statements, classes, functions etc [9, 17, 8]. For example, a particular code based technique can utilize information about the number of functions executed, or the number of blocks of code executed, by a test.

To achieve early fault detection Elbum et al. presented several code based prioritization techniques including Additional Function Coverage (AFC) [8]. This approach is proposed by analyzing source code only, where information from source code are used to prioritize test cases.

In AFC prioritization the test with the maximum number of functions covered is selected for early fault detection. If more than one test has the maximum number of functions coverages, a test is selected randomly from that test suite. Since this AFC prioritization technique is implemented in Java programming language, java rand function is used for random selection. The selected test cases and covered functions are not considered for further selection in AFC approach. A subset of the test suite is created based on the adjusted function coverage information, and a new test is selected from the remaining tests that covers the maximum number of functions. This process is repeated until all functions covered by at least one test case. Finally, AFC generates a sorted list of test cases.

The list of test cases and their related source code functions are the input of this AFC approach. Greedy selection approach is used to find maximum functions coverage test cases. An example of test cases and related covered functions are presented in Table 5.2. According to that table, Test Case ID2 can cover maximum functions, that is why TC2 is

Table 5.2: Example Relationship for AFC

Test Case Number	Related Functions Number
TC1	F2, F1
TC2	F2, F3, F4
TC3	F7
TC4	F5, F6

selected. TC1 and TC4 have the same coverage value, so random selector is used for this case. This approach is being continued until all the function is covered. This final test case order of AFC technique is TC2, TC1, TC4 and TC3.

5.2.2 Risk and Severity Factor: A Requirements Based Prioritization

Analyzing Risk and Severity Factor (RSF) [7] is the most prominent approach for test case prioritization using software requirements only. Because almost, every software projects benefit from risk and their severity analysis. These analysis results allow developers and product managers to pay special attention when designing the applications and generating the test cases to mitigate the risks.

Table 5.3: Requirements Priority Factor (with sample values)

Requirements	Customer	Developer	Manager	Priority Sum
Req-1	4	5	4	13
Req-2	5	4	3	12

Table 5.4: Risk Factor (with sample values)

Requirements	Probability	Severity	Total Risk Exposure
Req-1	4	5	20
Req-2	5	3	15

Table 5.5: Priority Decision Table (with sample values)

Factor	Req-1	Req-2	Weights
Priority	13	12	0.6
Risk	20	15	0.4
Weighted Priority	15.8	13.2	1

In RSF approach, two different prioritization factors are assigned for each requirement. Those factors are collected from customers, developers or managers which are listed below.

1. the requirement priority (value range 0 to 10) is collected from customers, developers and managers

2. risk factors (value range 0 to 10) for every requirement are collected from developers

Requirements and risk based test case prioritization technique can be divided into three major tasks named as calculating requirement priority factors, risk factors and final priority values. Activities and implementation process of those tasks are briefly described below.

a. Calculating Requirement Priority Factors

Collected priority values from three different stakeholders are integrated for calculating priority sum of a requirement. An example of calculating requirement priority sum is presented in Table 5.3. Those priority values are assigned to a range 0 to 10.

b. Calculating Risk Factors

Two risk factors named as probability and severity are collected from developers team and multiplied each other to calculate final risk factor. Table 5.4 represents an example of final risk calculation process. The dataset developer team assign those risk factors for implementing RSF approach.

c. Calculating Final Priority Values

The final priority values are calculated by the multiplication of those priority factors and their assigning weights which are the impact of those calculated phases on prioritization. In RSF technique, the requirement priorities are considered more important than risk factors [7]. Hence the prioritization weights of requirement priorities and risk factors are 0.6 and 0.4 respectively assigned by the authors. An example of final priority calculation process is denoted by Table 5.5. This process is implemented for every requirement. According to Table 5.5, Req-1 is more important than Req-2, that means the test cases related to Req-1 are executed first for early fault detection.

5.2.3 Natural Order for Test Case Execution

Test cases are written by test engineers and stored in test suite for execution. In test suite list, test cases are naturally ordered which is the order of test cases generation. No predefined information or priorities are considered during test cases generation. That is why this order of test cases is called natural order.

5.2.4 RDCC Framework for Test Case Prioritization

Requirements, Design diagrams and source Code Collaboration (RDCC), is the proposed test case prioritization technique where information from all phases of SDLC are integrated for early fault detection. This technique is implemented using Java programming language. The whole RDCC framework can be divided into various implementation phases which are briefly described below.

a. Processing Requirements

In the RDCC scheme, requirements are split into words using java split function. Stopwords are filtered using WordNet database [53] in every requirements. The remaining keywords are stored in requirement lists. Verbs are replaced with their base form using WordNet verb list. Term document matrix is generated based on those remaining words or terms using term frequency - inverse document frequency presented in Chapter 4 Section 4.3.1. In this matrix every row represents the requirement with it's containing term values. Those term values are integrated for calculating the requirement priority values which are presented in Equation 4.4.

The final requirement priority values are normalized by dividing with the maximum priority value and multiplying with the normalized range N which are presented in Equation 4.5. The value of N may vary from case to case, but this normalization value would be fixed for every phase of prioritization. So, the final requirement priority values are assigned between 0 to N range. The value of N for this implementation is given later.

b. Processing Design Diagrams

Design diagrams are processed to retrieve design modules or state corresponding to each requirements. Diagrams need to be converted as readable XML format. Enterprise architect [46] is used to draw design diagrams because this tool can generate background XML file of diagram. The modules interconnections are stored in a list for design priority. A design modules and requirements connectivity list is also created for mapping design diagrams and requirements.

In the beginning of this process, design priority list is initialized to 0. If there is a connection between two states, these state positions in the priority list are incremented. This process is being continued until all the connections are processed. The whole design

priority process is described in Chapter 4 Algorithm 2. The design priority values are normalized as same as requirement priority normalization from range 0 to N.

c. Processing Source Code

Source code are processed using four code metrics named as, Lines of Code [49], McCabe's Cyclomatic Complexity [50], Weighted Methods per Class [49], and Nested Block Depth [49] which are presented in Chapter 4 Subsection 4.3.3. Eclipse metric plugin is used to export code metrics values into a XML file. Those four values are calculated for every classes in OOP to initialize code priority. The whole process is describes in Chapter 4 Algorithm 3. The individual class of function metric values are normalized by dividing the maximum metric value. This process is repeated for all four metrics. The average value among those four metrics are assigned as final code metric value for each class or function. The final value is multiplied by N to normalize with design diagrams and requirements priority for computational and experimental simplicity.

d. RDCC Module Integration and Priority Constant Discussion

Requirements are uniquely identified by RDCC IDs for implementation in this framework. Every design modules and source code classes are related to at least one requirement. The RDCC priorities are calculated by the summation of priority values and weight constants. The integration of different phases' of priorities are discussed at Chapter 4 Section 4.3.4.

e. Assumptions and Priority Constants

The priority weight constants of requirements, design diagrams and source code are α, β and γ respectively. The value of α, β and γ may vary from software to software. The sum of those three constants must be equal to N for normalization (details presented in Chapter 4 Section 4.3.4). For implementing RDCC framework, those constants are assigned the same value to give equal priority for requirements, designs and source code. That means $\alpha = \beta = \gamma = \frac{N}{3}$. The value of N is assumed as 10 in this experiment for computing simplicity only.

Users may change those constant values based on their importance. If any of those three information is absent, that priority constant value of that phase is assigned to 0. For example, if design diagrams are not available for any software, the design priority constant value of β is assigned to 0. All the assumptions and priority constants values for implementation phases of RDCC are listed in Table 5.6.

Table 5.6: Experimental Setup and Determined Constants Values for Implementing different Prioritization Techniques

Prioritization Parameters	Values
Upper limit of single prioritization patmetr (N)	10
Prioritization Range	0 to 10
Requirements priority constant (α)	$\frac{10}{3}$
Design priority constant (β)	$\frac{10}{3}$
Source code priority constant (γ)	$\frac{10}{3}$
Implementing language	Java
Numbers of requirements in dataset	Varied from 8 to 20
Numbers of Line of Code in dataset	Varied from 636 to 4037
Numbers of Test Cases in dataset	Varied from 18 to 74

f. Prioritizing Test Cases

Test cases are prioritized based on the calculated RDCC priority values using RDCC - test case mapping matrix presented at Chapter 4 Section 4.4.2. Test cases priority values are

assigned based on their connected RDCC ID priority values. If multiple RDCC IDs are connected to a test case, the cummulative values of those RDCC ID priorities are assigned to that test case. Finally the test case list is sorted in descending order to detect early faults.

5.3 Performance Analysis and Comparison

To measure the efficiency of test case prioritization technique, fault detection rates and test case execution percentage both should be considered as metrics. Those parameters represent the performance of prioritization techniques in two different point of views. Definition and experimental results of those two performance analysis phases are given below.

1. Phase 1: Percentage of detected faults after different size of test execution
2. Phase 2: Percentage of test case execution to detect different amount of faults

For all of two cases the experimental setup and assumed constants are remaining the same which are listed in Table 5.6. The higher the fault detection, the better the prioritization approach is. On the other hand, the prioritization approach executes the lower numbers of test cases, is better than any other prioritization schemes.

5.3.1 Percentage of Detected Faults after Different Size of Test Case Execution

One of the major goals of test case prioritization is to detect faults earlier. The prioritization techniques which can detect early faults are more efficient than others. The percentage of fault detection shows what sizes of test cases are executed to detect a certain amount of faults. This experimental process is divided into three phases of test case execution named as first quarter (25%), half (50%) and third quarter (75%).

Table 5.7: Percentage of Detected Faults after 25% Test Case Execution

Test Case Prioritization Techniques				
Dataset	Natural Order	RSF [7]	AFC [8]	RDCC
Amghotok	25.00	25.00	25.00	33.33
Scientific Calculator	25.00	25.00	33.33	50.00
Painter	25.00	25.00	25.00	37.50
News A	7.14	7.14	7.14	14.29
Sparrow	16.67	33.33	33.33	33.33
POAS	20.00	20.00	30.00	40.00
Average	19.80	22.58	25.63	34.74

The prioritization approach which can detect maximum faults in every phases is better than other approaches in terms of fault detection. Full test case (100%) execution results are not considered, because after executing 100% test cases all faults should be detected for every prioritization schemes. The different size of test case execution for fault detection are listed and explained below.

1. Phase 1.1: Percent of Detected Faults after 25% Test Case Execution
2. Phase 1.2: Percent of Detected Faults after 50% Test Case Execution
3. Phase 1.3: Percent of Detected Faults after 75% Test Case Execution

Phase 1.1: Percentage of Detected Faults after 25% Test Case Execution

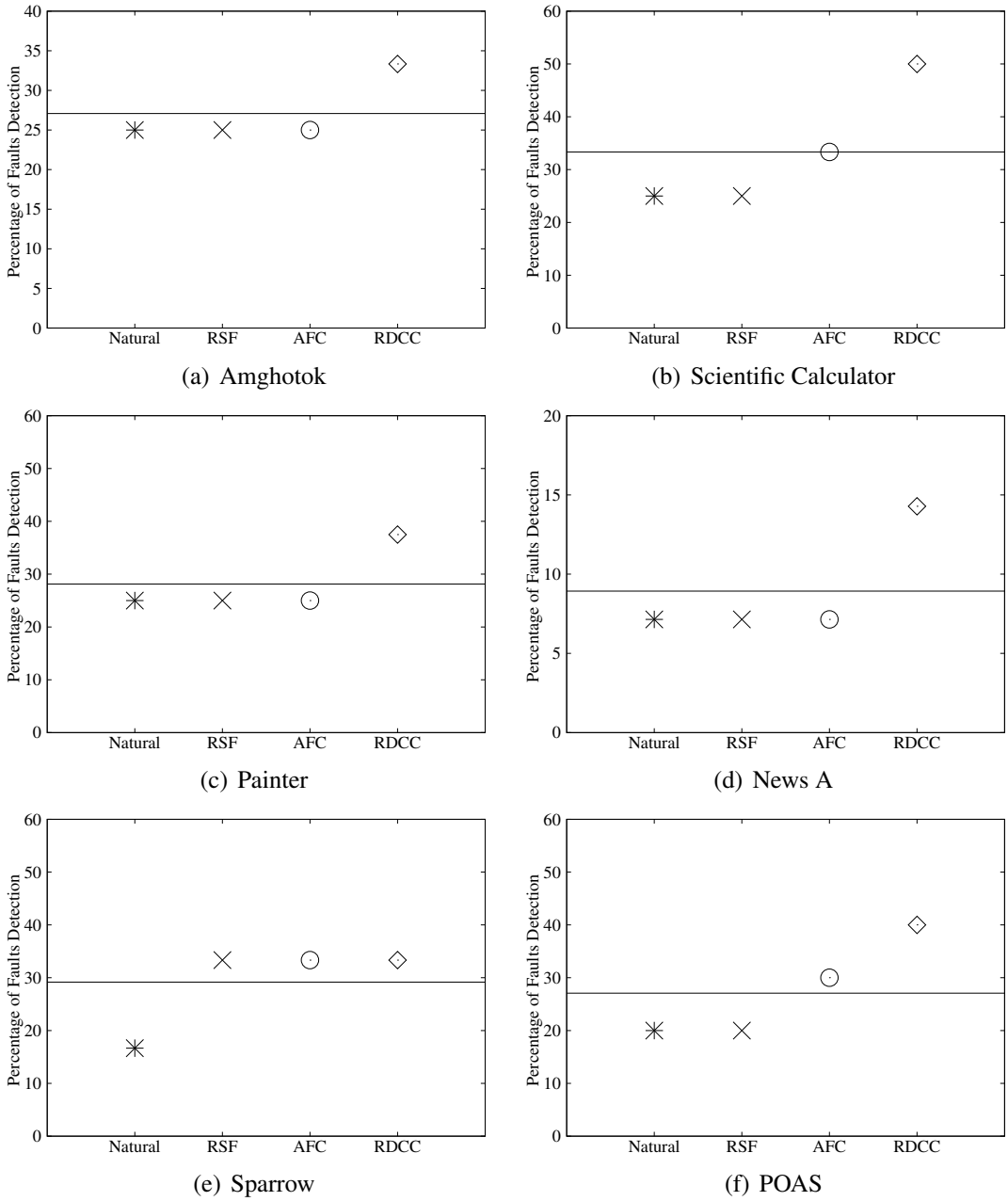
To compare the efficiency of test case prioritization techniques, the percentage of detecting faults were measured after 25% test case execution. Since early fault detection is one of the major goals of test case prioritization, the higher percentage of fault detection in Phase 1.1 can lead a better approach. Table 5.7 presents the result of detected faults after 25% test case execution with the datasets and implemented prioritization algorithms.

According to Table 5.7, the respective average of fault detection percentages for natural order, requirements [7], code based [8] and proposed RDCC approaches are 19.76%, 23.10%, 24.76% and 33.69% respectively. This results denote that proposed RDCC technique performs better than any other implemented prioritization techniques in terms of detecting faults after 25% test case execution.

The graphical representation of fault detection after 25% test case execution on different datasets are presented at Figure 5.1. The six subfigures denote the comparison results of prioritization techniques on six different datasets. The X and Y axis of different subfigures in Figure 5.1 represent the prioritization techniques and fault detection percentage after 25% test case execution respectively. The horizontal lines denote the average fault detection percentage value of different prioritization techniques.

According to Figure 5.1 (a), (b), (c), (d) and (f) RDCC approach performs single best which is the above average result in terms of fault detection in Phase 1.1. However RDCC approach performs the best result jointly with RSF [7] and AFC [8] approach for dataset Sparrow, which is shown in Figure 5.1 (e). That means there is no specific significance of collaborating SDLC information for prioritization in terms of that dataset after 25% test case execution.

This may happens for small size of dataset, because in small software, the classes are not distinguishable and the requirements are analogous to each others. In this situation every phase of SDLC contains quite similar information, that is why collaboration information from every phase of SDLC may not provide better result than other approaches.



(a) Amghotok (b) Scientific Calculator

(c) Painter (d) News A

(e) Sparrow (f) POAS

Figure 5.1: Fault Detection after 25% Test Case Execution

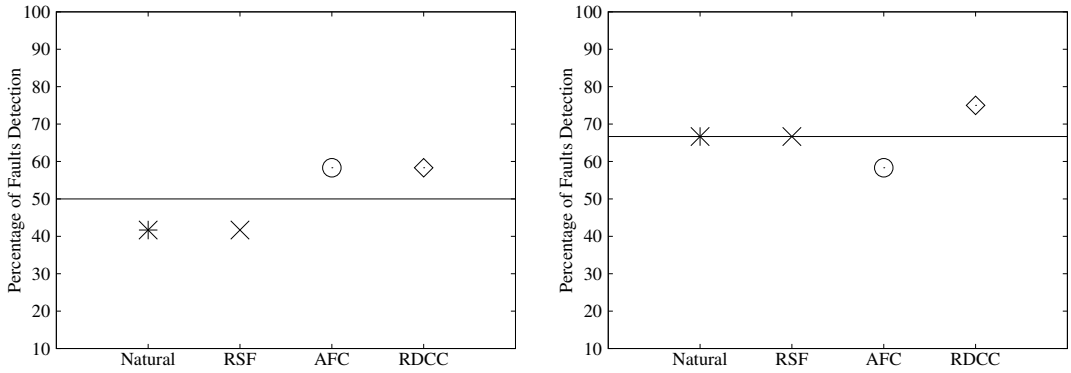
Table 5.8: Percentage of Detected Faults after 50% Test Case Execution

Test Case Prioritization Techniques				
Dataset	Natural Order	RSF [7]	AFC [8]	RDCC
Amghotok	41.67	41.67	58.33	58.33
Scientific Calculator	66.67	66.67	58.33	75.00
Painter	50.00	50.00	50.00	62.50
News A	7.14	35.71	46.43	82.14
Sparrow	33.33	50.00	50.00	55.55
POAS	50.00	60.00	60.00	80.00
Average	41.47	52.76	53.85	63.13

Phase 1.2: Percentage of Detected Faults after 50% Test Case Execution

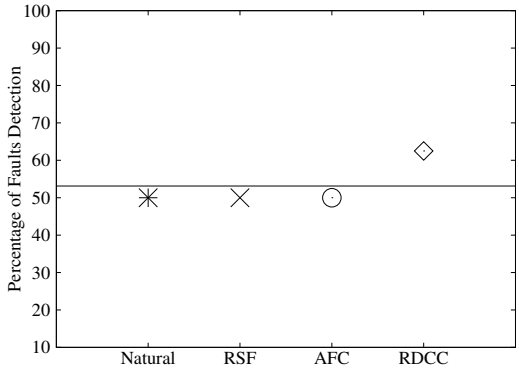
To compare experimental results in Phase 1.2, half of the whole test cases are executed for detecting early faults. Table 5.8 presents the comparison results of different datasets for the percentage of detecting faults after 50% test case execution. The results of different prioritization schemes and their experimented datasets are also presented by Table 5.8. According to Table 5.8, the average fault detection percentages for natural order, requirements [7], code based [8] and RDCC approaches in Phase 1.2 are 41.47%, 52.76%, 53.85% and 63.13% respectively. This results conclude that proposed RDCC technique performs the best among all other implemented prioritization techniques in terms of average fault detection after 50% test case execution.

Figure 5.2 shows the graphical representation of fault detection after 50% test case execution on different datasets in Phase 1.2. The six subfigures of Figure 5.2 represent the comparison results of prioritization techniques on six different datasets during Phase 1.2 execution. The X and Y axis of different subfigures in Figure 5.2 denote the implemented test case prioritization techniques and percentage of fault detection in Phase 1.2 respectively. The horizontal lines present the fault detection average in Phase 1.2 of different

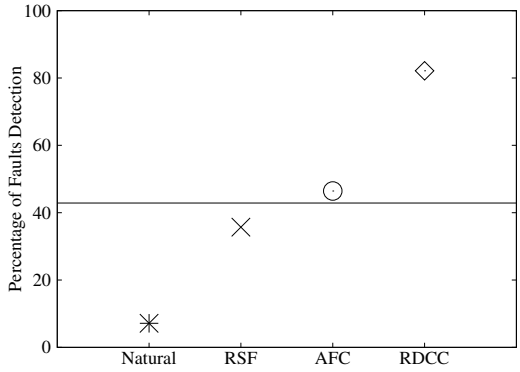


(a) Amghotok

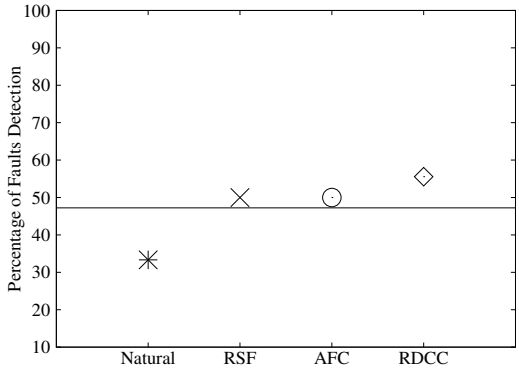
(b) Scientific Calculator



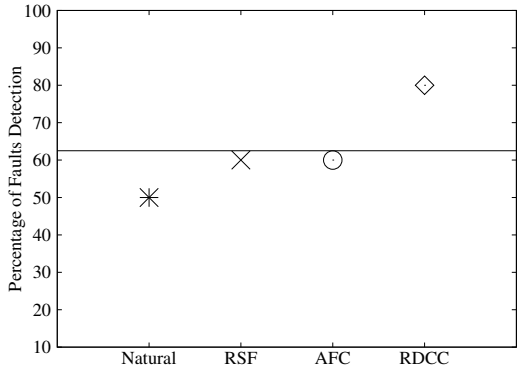
(c) Painter



(d) News A



(e) Sparrow



(f) POAS

Figure 5.2: Fault Detection after 50% Test Case Execution

Table 5.9: Percentage of Detected Faults after 75% Test Case Execution

Test Case Prioritization Techniques				
Dataset	Natural Order	RSF [7]	AFC [8]	RDCC
Amghotok	75.00	58.33	83.33	83.33
Scientific Calculator	75.00	75.00	75.00	75.00
Painter	62.50	75.00	75.00	87.50
News A	32.14	92.86	92.86	92.86
Sparrow	66.67	83.33	83.33	83.33
POAS	80.00	90.00	90.00	100.00
Average	65.22	79.09	83.25	87.00

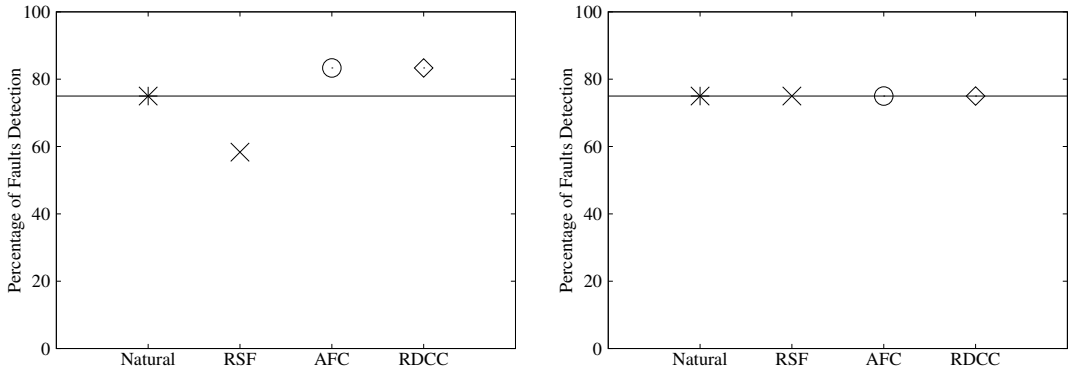
prioritization techniques. According to Figure 5.2 in every case of 50% test case execution, proposed RDCC approach can detect more faults than any other prioritization techniques.

Phase 1.3: Percentage of Detected Faults after 75% Test Case Execution

In the Phase 1.3 of test case execution, implemented test case prioritization techniques are compared for early fault detecting. In this phase, fault detection percentage are calculated after two third means 75% test case execution which are presented at Table 5.9. According to Table 5.9, the respective average of fault detection percentages are 65.22%, 79.09%, 83.25% and 87.00% for natural order, requirements [7], code based [8] and proposed RDCC approaches respectively.

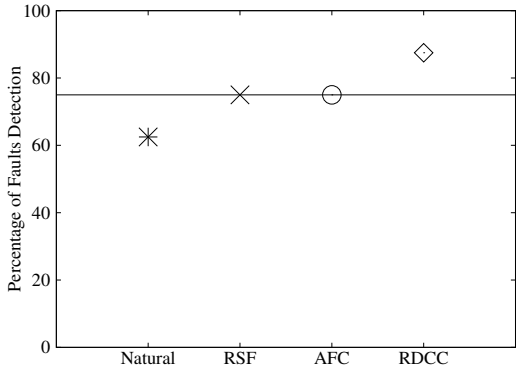
This average results show that, proposed RDCC technique performs the best among all other implemented prioritization techniques in terms of detecting faults after 75% test case execution in Phase 1.3.

The graphical representation of fault detection after 75% test case execution on different datasets are shown at Figure 5.3. The comparison results of prioritization techniques on six different datasets are shown by the six subfigures of Figure 5.3. The X and Y axis of

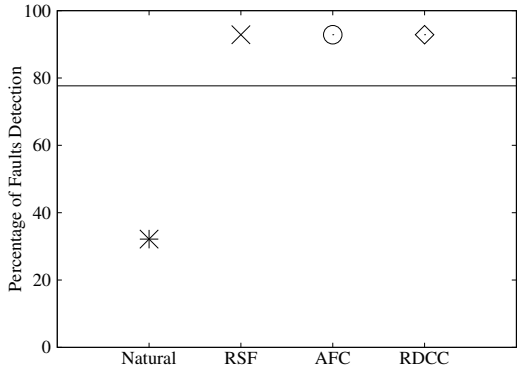


(a) Amghotok

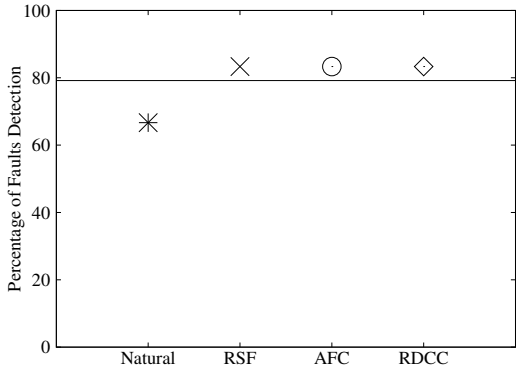
(b) Scientific Calculator



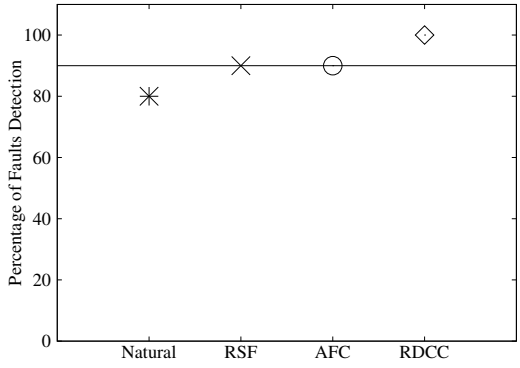
(c) Painter



(d) News A



(e) Sparrow



(f) POAS

Figure 5.3: Fault Detection after 75% Test Case Execution

different subfigures in Figure 5.3 represent the prioritization techniques and fault detection percentage in Phase 1.3. The horizontal line represents the average fault detection value of different prioritization techniques after 75% test case execution.

In the third quartile, selected prominent test case prioritization schemes performs almost similar result (like Figure 5.3.(a, b, d)), because all of those prioritization approaches try to find early faults. However in large dataset like POAS, proposed RDCC technique outperforms other prioritization approaches in terms of detecting faults after 75% test case execution (Figure 5.3.(f)).

5.3.2 Percentage of Test Case Execution to Detect Different Amount of Faults

Maximum fault detection in minimum test case execution is one of the major goals of test case prioritization. The percentage of test case execution results can measure the efficiency of prioritization approach in terms of early fault detection by detecting certain amount of faults. The lower the number of test case execution, is the higher the efficiency of prioritization technique. This measuring approach is experimented into 4 different phases of fault detection percentages named as 25%, 50%, 75% and 100%. The whole test cases are divided into those 4 different quarters to measure the prioritization schemes' efficiency in step by step. The prioritization approach which can execute minimum number of test cases to detect those quarters of faults is the most efficient approach among all others. The percentage results of test case execution to detect different quarters of faults are listed and described below.

1. Phase 2.1: Percentage of Test Case Execution to Detect 25% Faults

2. Phase 2.2: Percentage of Test Case Execution to Detect 50% Faults

3. Phase 2.3: Percentage of Test Case Execution to Detect 75% Faults

4. Phase 2.4: Percentage of Test Case Execution to Detect 100% Faults

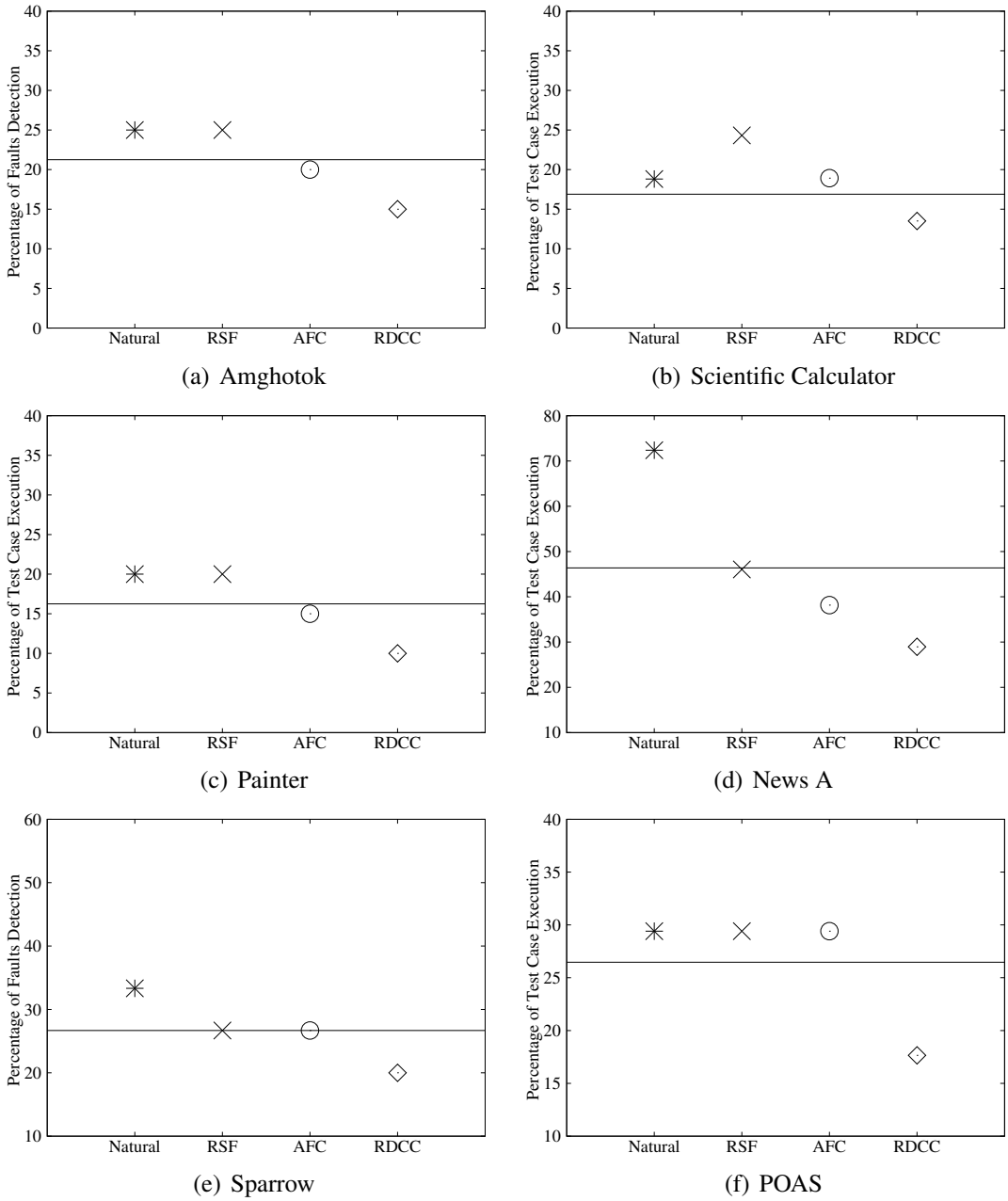
Table 5.10: Percentage of Test Case Execution to Detect 25% Faults

Test Case Prioritization Techniques				
Dataset	Natural Order	RSF [7]	AFC [8]	RDCC
Amghotok	25.00	25.00	20.00	15.00
Scientific Calculator	18.81	24.32	18.92	13.51
Painter	20.00	20.00	15.00	10.00
News A	72.37	46.05	38.16	28.95
Sparrow	33.33	26.67	26.67	20.00
POAS	29.41	29.41	29.41	17.65
Average	31.82	28.58	24.69	17.52

Phase 2.1: Percentage of Test Case Execution to Detect 25% Faults

In Phase 2.1, the execution numbers of test cases are calculated for detecting 25% software faults. The percentage of test case execution for different prioritization techniques are presented in Table 5.10. The average percentage results are also presented in that table to compare the outcome for 25% fault detection. According to Table 5.10, the average of test case execution percentages are 31.82%, 28.58%, 24.69% and 17.52% for natural order, requirements [7], code based [8] and RDCC approaches respectively. This table concludes that, on average the proposed RDCC technique detects 25% faults by executing minimum percentage of test cases.

Figure 5.4 shows the graphical representation of test case execution percentage to detect 25% faults. The comparison results of prioritization techniques on six different datasets are shown by the six subfigures of Figure 5.4. The X and Y axis of different subfigures



(a) Amghotok (b) Scientific Calculator

(c) Painter (d) News A

(e) Sparrow (f) POAS

Figure 5.4: Percentage of Test Case Execution to Detect 25% of Faults

Table 5.11: Percentage of Test Case Execution to Detect 50% Faults

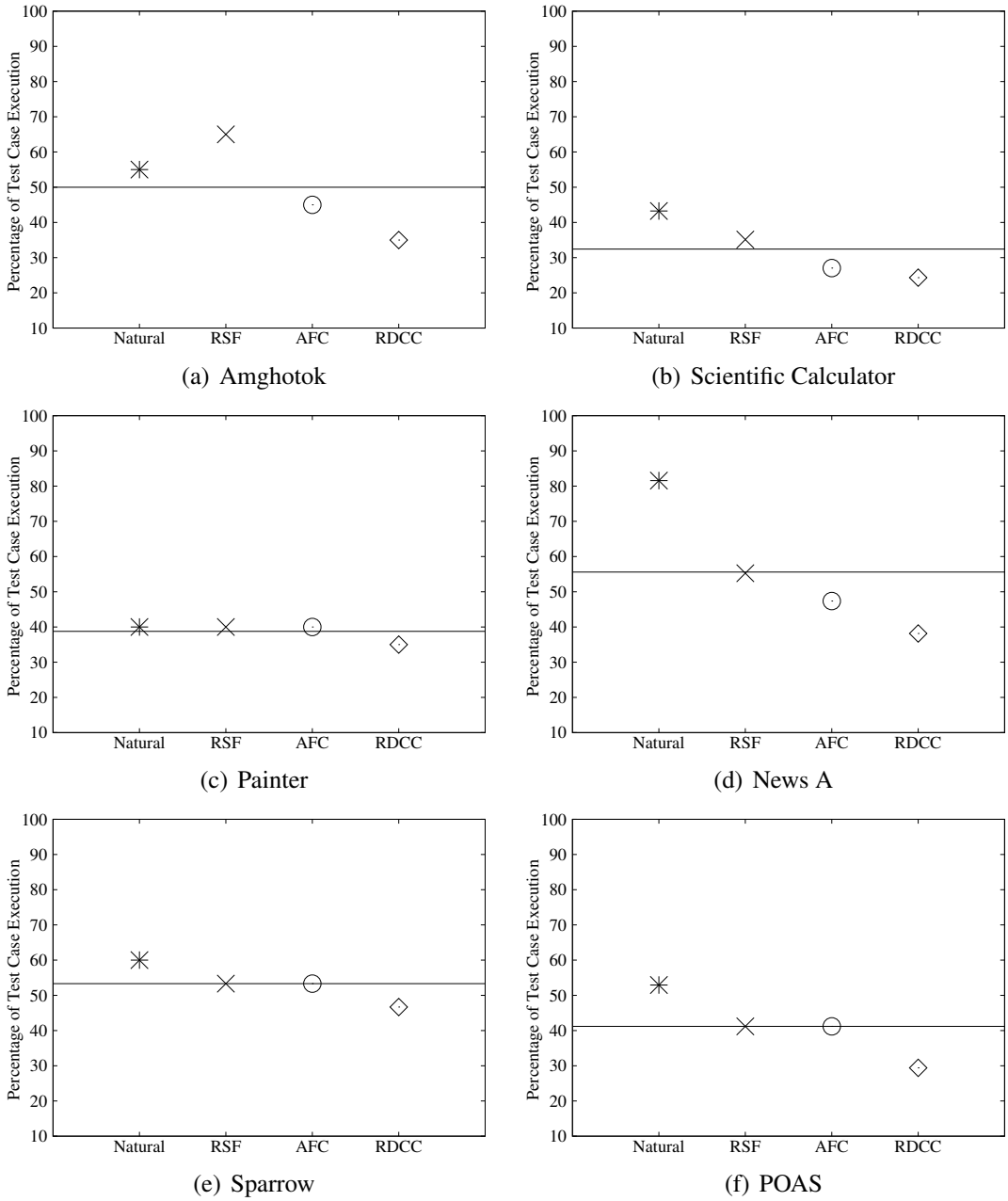
Test Case Prioritization Techniques				
Dataset	Natural Order	RSF [7]	AFC [8]	RDCC
Amghotok	55.00	65.00	45.00	35.00
Scientific Calculator	43.24	35.14	27.03	24.32
Painter	40.00	40.00	40.00	35.00
News A	81.58	55.26	47.37	38.16
Sparrow	60.00	53.33	53.33	46.67
POAS	52.94	41.18	41.18	29.41
Average	55.46	44.98	42.32	38.09

in Figure 5.4 denote the prioritization techniques and test case execution percentages in Phase 2.1. The horizontal line represents the average test case execution value of different prioritization techniques to detect 25% faults. According to Figure 5.4 proposed RDCC approach performs better in terms of test case execution to detect 25% faults for all datasets.

Phase 2.2: Percentage of Test Case Execution to Detect 50% Faults

To detect half of the total faults, the value of test case execution are experimented in Phase 2.2. Table 5.11 presents the test case execution percentage values of different prioritization schemes to detect 50% software faults in datasets. According to Table 5.11, the average test case execution percentages for natural order, requirements [7], code based [8] and RDCC approaches are 55.96 %, 45.75%, 42.55 % and 39.83% respectively in Phase 2.2. That means for detecting 50% faults, proposed RDCC approach executes minimum numbers of test cases on average.

The graphical representation of test case execution percentage to detect 50% faults on different datasets are shown at Figure 5.5. The comparison results of prioritization techniques on six different datasets are also shown by the six subfigures of Figure 5.5. The X



(a) Amghotok (b) Scientific Calculator

(c) Painter (d) News A

(e) Sparrow (f) POAS

Figure 5.5: Percentage of Test Case Execution to Detect 50% of Faults

Table 5.12: Percentage of Test Case Execution to Detect 75% Faults

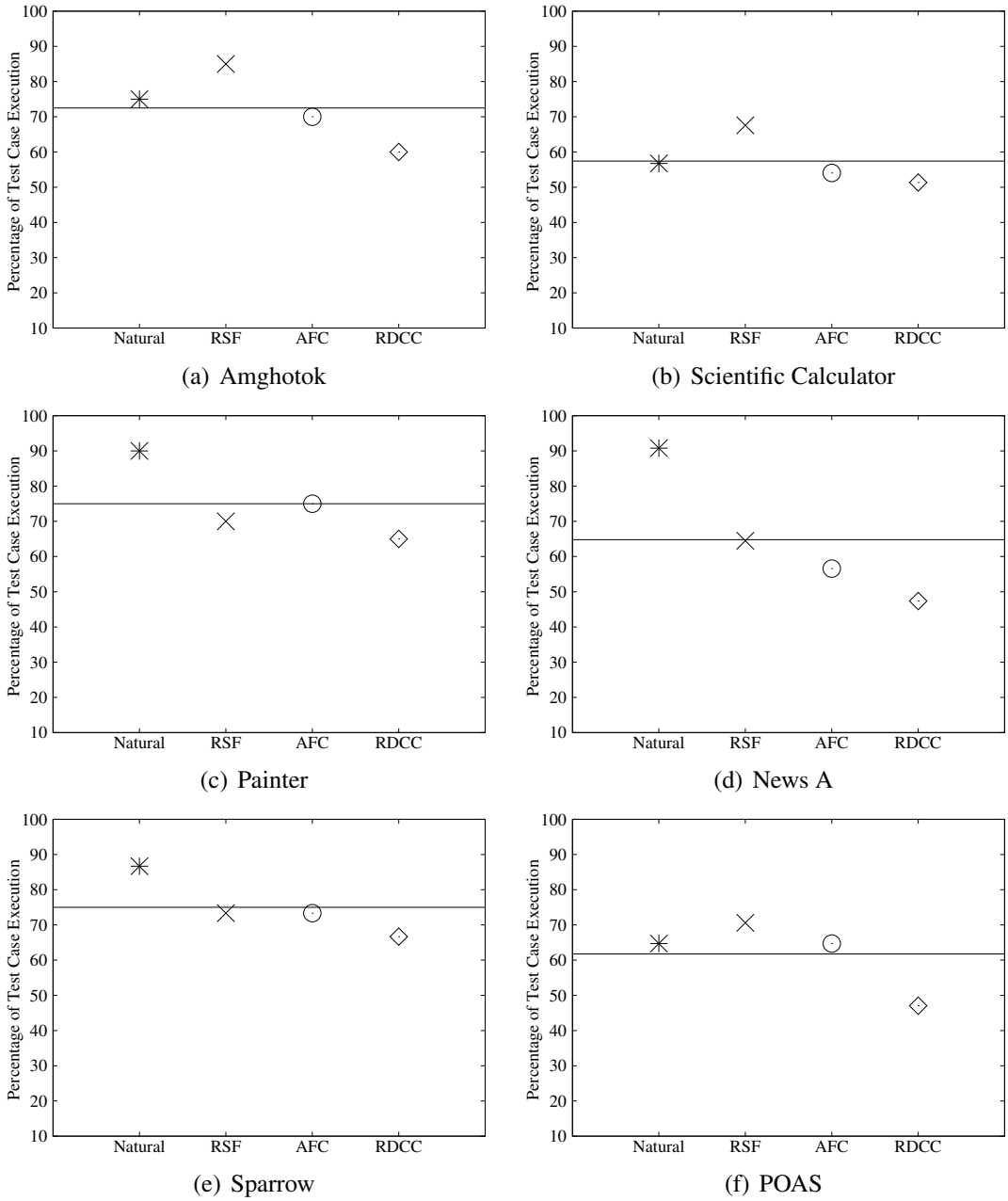
Test Case Prioritization Techniques				
Dataset	Natural Order	RSF [7]	AFC [8]	RDCC
Amghotok	75.00	85.00	70.00	60.00
Scientific Calculator	56.76	67.57	54.05	51.35
Painter	90.00	70.00	75.00	65.00
News A	90.79	64.47	56.58	47.37
Sparrow	86.67	73.33	73.33	66.67
POAS	64.71	70.59	64.71	47.06
Average	77.32	70.72	65.61	56.24

and Y axis of different subfigures in Figure 5.5 represent the prioritization techniques and test case execution percentages to detect 50% faults. The horizontal lines denote the average test case execution value of different prioritization techniques in Phase 2.2. According to Figure 5.5, proposed RDCC produces lowest execution value for all datasets, that means RDCC performs the most efficient result in terms of 50% fault detection.

Phase 2.3: Percentage of Test Case Execution to Detect 75% Faults

In the Phase 2.3, the execution numbers of test cases are calculated for detecting two third software faults. Table 5.12 presents the percentage of test case execution for different prioritization techniques to measure early fault detection. The average percentage results are also included in that table to compare the outcome for 75% fault detection. According to Table 5.12, the average of test case execution percentages for natural order, requirements [7], code based [8] and RDCC approaches are 77.32%, 70.72%, 65.61% and 58.46% respectively. That means the RDCC technique detects 75% faults by executing minimum test cases on average.

Figure 5.6 shows the graphical representation of test case execution percentage results



(a) Amghotok (b) Scientific Calculator

(c) Painter (d) News A

(e) Sparrow (f) POAS

Figure 5.6: Percentage of Test Case Execution to Detect 75% of Faults

Table 5.13: Percentage of Test Case Execution to Detect 100% Faults

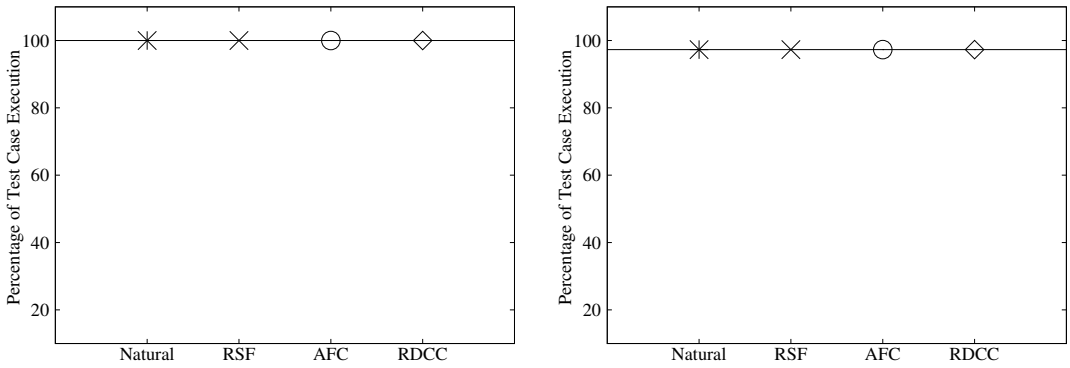
Test Case Prioritization Techniques				
Dataset	Natural Order	RSF [7]	AFC [8]	RDCC
Amghotok	100.00	100.00	100.00	100.00
Scientific Calculator	97.30	97.30	97.30	97.30
Painter	100.00	85.00	90.00	85.00
News A	100.00	100.00	100.00	100.00
Sparrow	100.00	100.00	100.00	86.67
POAS	94.12	94.12	88.24	70.59
Average	97.74	96.07	95.92	89.93

to detect 75% faults. The comparison results of different prioritization techniques on six datasets are shown by the six subfigures of Figure 5.6. The X and Y axis of different subfigures in Figure 5.6 denote the prioritization techniques and test case execution percentages to detect 75% faults. The horizontal line represents the average test case execution value of different prioritization techniques in Phase 2.3. According to Figure 5.6, RDCC executes minimum percentage of test cases for all test datasets in terms of detecting 75% faults.

Phase 2.4: Percentage of Test Case Execution to Detect 100% Faults

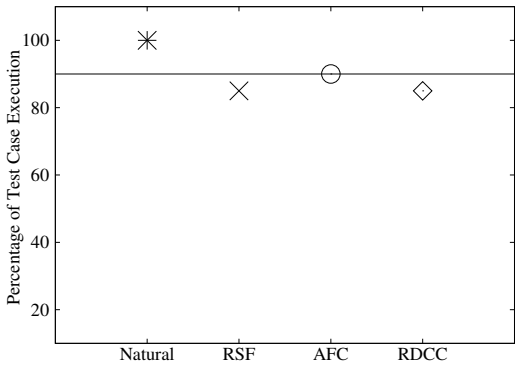
To detect total faults in a software, the value of test case execution are experimented in Phase 2.4. Table 5.13 presents the individual and average test case execution percentage values of different prioritization schemes to detect all software faults in every datasets. According to Table 5.13, the respective average of test case execution percentages for natural order, requirements [7], code based [8] and proposed RDCC approaches are 97.74%, 96.07%, 95.92% and 89.93%. That means the proposed RDCC technique needs minimum test case execution value for all fault detection on average.

The graphical representation of test case execution percentage to detect 100% faults

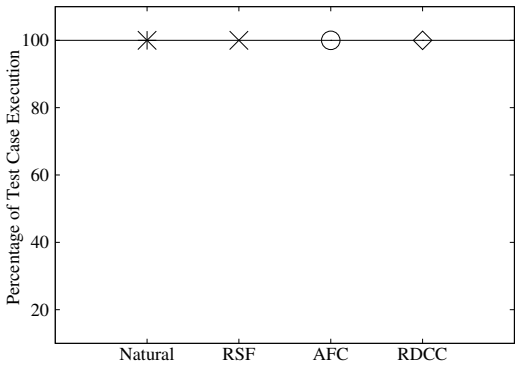


(a) Amghotok

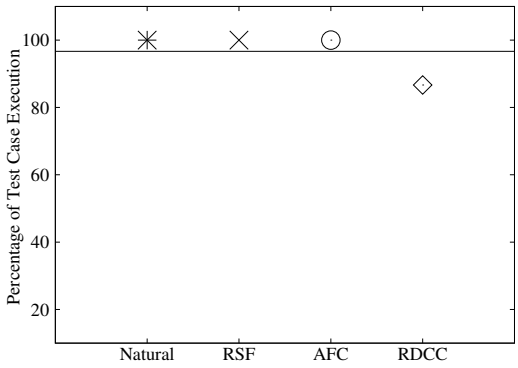
(b) Scientific Calculator



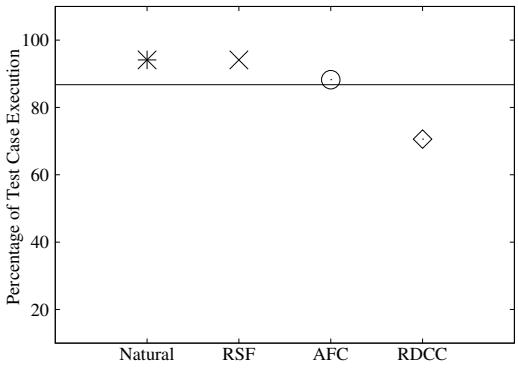
(c) Painter



(d) News A



(e) Sparrow



(f) POAS

Figure 5.7: Percentage of Test Case Execution to Detect 100% Faults

on different datasets are shown at Figure 5.7. The X and Y axis of different subfigures in Figure 5.5 represent the prioritization techniques and test case execution percentages to detect all software faults in every dataset. The horizontal lines denote the average test case execution value of different prioritization techniques in Phase 2.4.

For detecting 100% faults, prominent test case prioritization approaches may perform similar, because some of faults may not detect earlier based on pre-execution knowledge (for example Figure 5.7.(a, b, d)). However in large dataset, proposed RDCC approach performs better than any other prioritization schemes which is shown by Figure 5.7.(f).

5.3.3 Average Percentage of Fault Detection and Test Case Execution

The efficiency of a test case prioritization technique depends on how quickly the faults are detected by this approach. As a measure of how rapidly a prioritized test suite detects faults, Average of the Percentage of Fault Detection (APFD) and Average Percentage of Test Case Execution (APTCE) are used during the prioritization of the test suite. The higher APFD value means the faster fault detection rates, where the lower APTCE value denotes the better execution lists for test case prioritization. Those prioritization measurement processes are described with the implementation result below.

Average of the Percentage of Fault Detection (APFD)

APDF is calculated by the average value of fault detection from different datasets (presented in Section 5.3). Six different datasets (presented in Section 5.1.2) are used to calculate those values. Table 5.14 presents the average percentage of fault detection lists for different comparing test case prioritization approaches. According to Table 5.14, the respective APFD values of RDCC technique are 33.69%, 59.76%, 84.40% and 100.00% for

Table 5.14: Average Percentage of Fault Detection for Result Comparison

Test Case Prioritization Techniques				
Test Case Execution	Natural Order	RSF [7]	AFC [8]	RDCC
0%	0.00	0.00	0.00	0.00
25%	19.80	22.58	25.63	34.74
50%	41.47	52.76	53.85	63.13
75%	68.00	79.09	83.25	87.00
100%	100.00	100.00	100.00	100.00

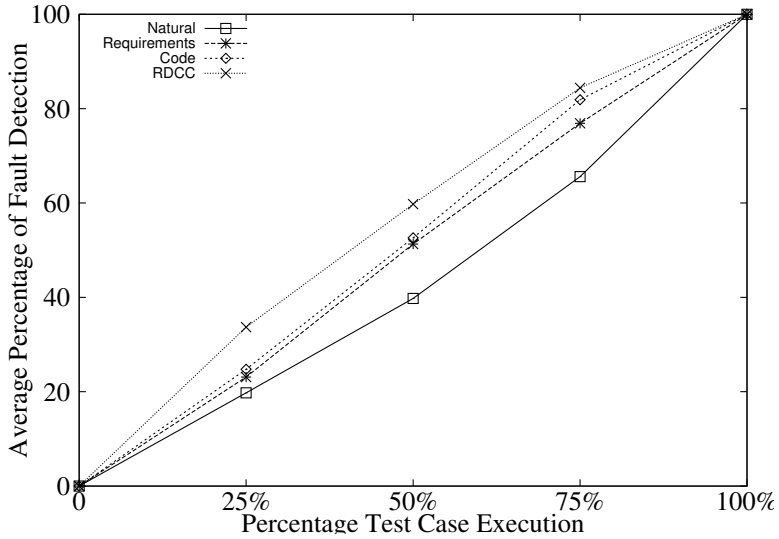


Figure 5.8: Average Percentage of Fault Detection for Result Comparison

one fourth, half, two third and whole test suite execution. Those values are the highest among all other prioritization techniques named natural order, RSF [7] and AFC [8].

The graphical representation of APFD values are shown by Figure 5.8 where the various percentage of test case execution (named as 25%, 50%, 75%) and average percentage of fault detection denote the X and Y axis respectively. According to Figure 5.8 proposed RDCC scheme generates 33.06%, 20.18% and 15.08% better results than natural order, RSF [7] and AFC [8] approach respectively in terms of average fault detection. That means on average RDCC approach is 13.414% more efficient than any other test case prioritization techniques which are implemented for result comparison.

Table 5.15: Average Percentage of Test Case Execution for Result Comparison

Test Case Prioritization Techniques				
Fault Detection	Natural Order	RSF [7]	AFC [8]	RDCC
0%	0.00	0.00	0.00	0.00
25%	31.82	28.58	24.69	17.52
50%	55.46	44.98	42.32	38.09
75%	77.32	70.72	65.61	58.46
100%	97.74	96.07	95.92	89.93

Average Percentage of Test Case Execution (APTCE)

To calculate APTCE, all the numbers of test case execution for detecting various scaling faults are averaged. The average APTCE values of all datasets are presented in Table 5.15. According to Table 5.15, the APTCE values of RDCC framework are 17.49%, 39.83%, 60.74%, and 93.79% for one fourth, half, two third and total fault detection respectively.

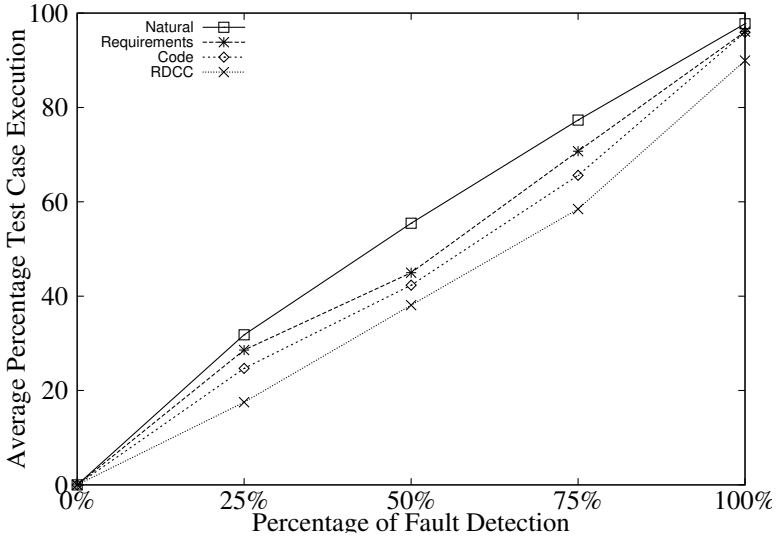


Figure 5.9: Average Percentage of Test Case Execution

Those values are the lowest among all other prioritization techniques named natural order, RSF [7] and AFC [8] schemes. Since lower APTCE value leads better prioritization approach, RDCC framework provides better result than any other prioritization schemes.

The cumulative average values for different scale of fault detection are shown in Figure 5.9. The X and Y axis of Figure 5.9, represents the different percentage of fault detection and average percentage of test cases respectively. According to Figure 5.8 proposed RDCC scheme generates 42.04%, 27.25%, and 17.74% better results than natural order, RSF [7] and AFC [8] approach respectively in terms of average percentage of test case execution. Hence, on average RDCC approach is 13.893% more efficient in terms of test case execution than any other prioritization schemes which are implemented for result comparison.

5.4 Result Analysis and Discussion

The experimental results are generated from six different datasets (presented in subsection 5.1.2) and compared to four different test case prioritization schemes named as RDCC approach (Chapter 4), RSF [7], AFC [8] and natural order. Section 5.3 shows the experimental results of those prioritization schemes for two different performance metrics named as percentage of test case execution (presented in subsection 5.3.1) and percentage of fault detection (presented in subsection 5.3.2). Both of matrices are analyzed by different phases of experiments (like 25%, 50% test case execution) for different datasets.

Subsection 5.3.1 and 5.3.2 present the detail explanation of how the collaborative information from SDLC performs better than any other test case prioritization schemes for individual dataset in terms of percentage of test case execution and fault detection respectively. Since minimization of test case execution can reduce testing time and cost, detecting a specific percentage of faults, prioritization approaches should need minimum numbers of test case execution. RDCC scheme executes minimum number of test cases to detect same percentage of faults and performs better than any other prioritization schemes.

Above analysis and discussions denote that proposed RDCC approach outperforms in terms of average fault detection (subsection 5.3.2) and test case execution (subsection

5.3.1). However after executing specific percentage of test cases, it performs similar results with AFC or SRF (for example Figure 5.1 (e) and Figure 5.3 (e)). It may happen in small datasets, because collaborative information from every phases of SDLC may not generate extra supporting issues for test case prioritization (such as incorporating requirements and code, requirements test cases mapping etc.) in those applications. In those cases, proposed prioritization technique may perform similar results to requirements and source code analysis only. For example, in Phase 1.1, percentage of fault detection after 25% of test case execution RDCC and AFC approaches produce similar results for dataset named Sparrow. However most of the real life applications are usually big.

Since early fault detection is one of the major goals of test case prioritization, RDCC approach performs 61.89%, 24.93%, 21.82% and 7.39% faster on average of all other prioritization schemes for 25%, 50%, 75% and 100% fault detection. Those percentages present a descending order of performance, because RDCC approach already detects faults in early phases. Since majority faults are detected in early phases of execution, there should be no vast significance in the later phases of fault detection. That means collaborative information from different phases of SDLC can increase the efficiency of test case prioritization technique by detecting early faults. Because early fault detection reduces the time and cost in testing phases, which can positively effects on whole software budget.

In a nutshell, RDCC outperforms for all dataset considering average fault detection and percentage of test case execution, specially in large datatset (for example, dataset named POAS). This scheme performs 13.414% and 13.893% better results on average than any other implemented test case prioritization approaches in terms of fault detection and test case execution respectively. Finally, it can be concluded that prioritization approach using the collaborative information from different phases of SDLC can reduce time and cost in testing phase by increasing the fault detection and minimizing the test case execution rates.

5.5 Summary

This chapter presents the performance analysis of proposed RDCC framework along with its implementation scheme. The environmental setup and experimental details of the test application are also provided so that others can simulate similar test environments. The experimental results have been compared to natural order, requirements and source code based test case prioritization schemes in terms of average percentage of test case execution and fault detection. It has been shown that RDCC performs best results among other comparative prioritization schemes. The problem of incorporating all phases of software development life cycle is also removed in proposed RDCC framework. The concluding remarks and future directions anbout test case prioritization research are presented in the very next chapter.

Chapter 6

Conclusion and Suggestion for Future Work

This chapter presents the summary of RDCC (the proposed test case prioritization framework) with its achievements. RDCC framework takes software requirements specification, design diagrams, source code and test cases as input and provides a prioritized order of test cases using their collaborative information as output. These information are collaborated to achieve the prioritization goals such as early fault detection and minimum test case execution. It has been analyzed that proposed prioritization approach using collaborative information outperforms any other prioritization schemes using individual information of SDLC. Finally, this chapter is concluded by providing an insight of future research directions regarding test case prioritization.

6.1 RDCC: The Proposed Framework

This thesis proposes RDCC - an effective software test case prioritization using requirements, design diagrams and source code. Test case prioritization schemes can reduce time and cost in testing phase by early fault detection and minimum number of test case execution, which positively effects on whole software budget. Determining the priority of test cases, user requirements and design should be reflected as well as code execution information, because the view points and prioritized modules may vary from designers, requirements engineers and coders. Requirement engineers may prioritize those modules which are absent in designer and coder prioritized modules. This event may occur at every phase in SDLC. Study of related work showed that existing prominent prioritization schemes are usually developed based on the relationship between test cases and individual

phases of SDLC (either requirements or source code). Those schemes may miss the total view points about a software, because, without considering the relationship between all phases of SDLC, none of the prioritization schemes can work accurately.

In the proposed prioritization scheme, requirements are analyzed based on the textual similarity by filtering the stop words. The remaining terms of every requirements are prioritized using term frequency - inverse document frequency to detect requirements priority. Those priority values are normalized in a specific range for computational simplicity. Design diagrams are extracted as readable XML format to detect the dependencies and connectivities among all modules corresponding to each requirement. Each of the states or modules in diagrams are prioritized based on their number of detected connectivities and dependencies. Source code are prioritized using various software code metrics named as line of code, cyclomatic complexity, number of parameters and weighted methods per class. Classes and functions related to each requirement are used as the idiosyncratic values of source code for prioritization.

Every requirement IDs are uniquely identified as RDCC IDs in this framework. The priority values of RDCC IDs are calculated by using the relationships between requirement IDs and design modules, and requirement IDs and classes. Final RDCC priorities are calculated by the multiplication of those calculated priorities and priority constants. The priority constants may vary from software to software. However, the equal priority is assigned for every phases in this thesis to give similar importance of requirements, design diagrams and source code. If any of those phases is missing, the priority constant of that phase is assigned to 0. The calculated final priority values are used to assign priority of related test cases. If a test case is related to more than one requirement IDs, the cumulative priorities are assigned for that test cases. Finally, the test cases are sorted in descending order for early fault detection.

6.2 Achievements of RDCC

It has been shown experimentally that proposed RDCC scheme has a lower percentage of test case execution and higher percentage of fault detection rate. From the experiment it has been seen that proposed RDCC framework provides 33.06%, 20.18% and 15.08% better results than natural order, RSF [7] and AFC [8] approach respectively in terms of average fault detection. That means collaborative information from different phases of SDLC performs 22.77% more efficient on average than any other prioritization schemes in terms of early fault detection.

Experimental results also show that RDCC framework outperforms in terms of number of test case execution. It generates 42.04%, 27.25%, and 17.74% better results than natural order, RSF [7] and AFC [8] approach respectively in terms of average percentage of test case execution. Hence, prioritization approach by integrating all phases of SDLC, executes 29.01% less test cases for detecting specific percentage of faults.

Both of two measurement metrics denote that prioritization scheme using collaborative information from different phases of SDLC outperforms using individual phase of SDLC (such as requirements or source code) and natural order. Those results infer that proposed RDCC scheme can lead to an efficient prioritization approach by maximizing the fault detection rate and minimizing the percentage of test case execution.

6.3 Future Directions to Test Case Prioritization

The proposed RDCC scheme, presented in this thesis has achieved the intended goals which are early fault detection and minimum percentage of test case execution, but there may have some other research directions from different perspectives to achieve. These research issues are briefly described below.

- The efficiency of test case prioritization schemes are closely related to test case generation process. Hence irrelevant test case can negatively affect on prioritization performance. Appropriate test case generation could be a future research direction to improve the performance of test case prioritization schemes. Hence before prioritizing test cases, test generation processes need to be verified.
- Test smells decrease the efficiency of test cases. Any prominent prioritization schemes may fail, if the test cases contains test smells. For an efficient test case prioritization scheme, test smells need to be resolved, which can be another research direction. After removing the test smells, every prioritization schemes performs as expected.
- Traditional test case prioritization schemes are basically developed based on the relationship between test cases and other software development information such as requirements, source code etc. However, test cases with their own characteristics like testing assert values can be considered for further research direction in test case prioritization family. Because if the assert values are not validated, none of prioritization approaches work properly.

6.4 Concluding Remarks

Test case prioritization approaches reduce software development time and cost by maximizing the percentage of fault detection and minimizing the percentage of test case execution. To achieve an efficient prioritization technique, information from different phases of SDLC which are requirements, design diagrams and source code are needed to be collaborated. Because those phases are uniquely connected to test cases by test case generation, test modules connection and test case execution respectively including individual view points. This hypothesis is justified and proved throughout the research and a test case prioritization framework is proposed using the collaborative information of all those phases. The comparative study of the proposed framework supports the hypothesis.

Bibliography

[1] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[2] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *IEEE International Conference on Software Maintenance (ICSM'99)*, pages 179–188. IEEE, 1999.

[3] Hema Srikanth and Laurie Williams. On the economics of requirements-based test case prioritization. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–3. ACM, 2005.

[4] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, 38(6):1258–1275, 2012.

[5] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 6 edition, 2005.

[6] M.J. Arafeen and Hyunsook Do. Test case prioritization using requirements-based clustering. In *Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pages 312–321. IEEE, March 2013.

[7] Praveen Ranjan Srivastva, Krishan Kumar, and G Raghurama. Test case prioritization based on requirements and risk factors. *ACM SIGSOFT Software Engineering Notes*, 33(4):7, 2008.

[8] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[9] Zhang Zhi-hua, Mu Yong-min, and Tian Ying-ai. Test case prioritization for regression testing based on function call path. In *Fourth International Conference on Computational and Information Sciences (ICCIS)*, pages 1372–1375. IEEE, 2012.

[10] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[11] Nenad Ivezic and Jungyub Woo. Testing interoperability standards–a test case generation methodology. In *Interoperability for Enterprise Software and Applications*, pages 23–30. John Wiley and Sons, Inc., 2013.

[12] IEEE Standard for Software Engineering - 610 . *IEEE Std. 610-1990*, pages 1 –22, 1990.

[13] Ron Patton. *Software Testing (2nd Edition)*. Sams, Indianapolis, IN, USA, 2005.

[14] Luay Ho Tahat, Boris Vaysburg, Bogdan Korel, and Atef J Bader. Requirement-based automated black-box test generation. In *25th Annual International Computer Software and Applications Conference, (COMPSAC)*, pages 489–495. IEEE, 2001.

[15] Md Mahfuzul Islam, Alessandro Marchetto, Angelo Susi, and Giuseppe Scanniello. A multi-objective technique to prioritize test cases based on latent semantic indexing. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 21–30. IEEE, 2012.

[16] J. Karlsson and K. Ryan. A cost-value approach for prioritizing requirements. *IEEE Software*, 14(5):67–74, Sep 1997.

[17] S Haidry and Tim Miller. Using dependency structures for prioritization of functional test suites. *IEEE Transactions on Software Engineering*, 39(2):258–275, 2013.

[18] Murali Krishna Ramanathan, Mehmet Koyuturk, Ananth Grama, and Suresh Jagannathan. Phalanx: a graph-theoretic framework for test case prioritization. In *ACM symposium on Applied computing*, pages 667–673. ACM, 2008.

[19] Hema Srikanth and Laurie Williams. Requirements based test case prioritization. *IEEE Transaction on Software Engineering*, 28(10):21 – 30, 2002.

[20] Dan Hao, Xu Zhao, and Lu Zhang. Adaptive test-case prioritization guided by output inspection. In *36th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 169–179. IEEE, 2013.

[21] Gregory M Kapfhammer and Mary Lou Soffa. Using coverage effectiveness to evaluate test suite prioritizations. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 19–20. ACM, 2007.

[22] Ståle Amland. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3):287–295, 2000.

[23] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[24] Sihan Li, Naiwen Bian, Zhenyu Chen, Dongjiang You, and Yuchen He. A simulation study on some search algorithms for regression test case prioritization. In *10th International Conference on Quality Software (QSIC)*, pages 72–81. IEEE, 2010.

[25] Ian Sommerville. *Software Engineering*. Pearson Addison Wesley, 7 edition, 2004.

[26] Hema Srikanth, Laurie Williams, and Jason Osborne. System test case prioritization of new and regression test cases. In *International Symposium on Empirical Software Engineering*, pages 10–19. IEEE, 2005.

[27] D. Hiemstra. A probabilistic justification for using tf.idf term weighting in information retrieval. *International Journal on Digital Libraries*, 3(2):131–139, 2000.

[28] Orlena CZ Gotel and Anthony CW Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101. IEEE, 1994.

[29] Md Mahfuzul Islam, Alessandro Marchetto, Angelo Susi, Fondazione Bruno Kessler, and Giuseppe Scanniello. Motcp: A tool for the prioritization of test cases based on a sorting genetic algorithm and latent semantic indexing. In *28th International Conference on Software Maintenance (ICSM)*, pages 654–657. IEEE, 2012.

[30] Allen Newell and Herbert A Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

[31] Ruchika Malhotra and Divya Tiwari. Development of a framework for test case prioritization using genetic algorithm. *ACM SIGSOFT Software Engineering Notes*, 38(3):1–6, 2013.

[32] Scott C. Deerwester, Susan T Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[33] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *An International Journal of Empirical Software Engineering*, 10(4):405–435, 2005.

[34] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet physics doklady*, 10:707, 1966.

[35] Alan George and Alex Pothen. An analysis of spectral envelope reduction via quadratic assignment problems. *SIAM Journal on Matrix Analysis and Applications*, 18(3):706–732, 1997.

[36] Mark Harman. Why source code analysis and manipulation will always be important. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 7–19. IEEE Computer Society, 2010.

[37] Árpád Beszédes, Tamás Gergely, Lajos Schrettner, Judit Jász, László Langó, and Tibor Gyimóthy. Code coverage-based regression test selection and prioritization in webkit. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 46–55. IEEE, 2012.

[38] D Jeya Mala and M Ramalakshmi Praba. Critical components identification and verification for effective software test prioritization. In *Third International Conference on Advanced Computing (ICoAC)*, pages 181–186. IEEE, 2011.

[39] Simonetta Balsamo, Paola Inverardi, and Calogero Mangano. An approach to performance evaluation of software architectures. In *Proceedings of the 1st international workshop on Software and performance*, pages 178–190. ACM, 1998.

[40] CharuC. Aggarwal and ChengXiang Zhai. A survey of text clustering algorithms. In *Mining Text Data*, pages 77–128. Springer US, 2012.

[41] Karl E Wiegers. Writing quality requirements. *Software Development*, 7(5):44–48, 1999.

[42] Andreas Hotho, Steffen Staab, and Gerd Stumme. Ontologies improve text document clustering. In *Third International Conference on Data Mining (ICDM)*, pages 541–544. IEEE, 2003.

[43] Satanjeev Banerjee and Ted Pedersen. Extended gloss overlaps as a measure of semantic relatedness. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, volume 3, pages 805–810, 2003.

[44] Steven Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 69–72. Association for Computational Linguistics, 2006.

[45] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In *Model Driven Engineering Languages and Systems*, pages 425–439. Springer, 2006.

[46] Enterprise architect - uml design tools and uml case tools for software development. `http://www.sparxsystems.com/products/ea/`, April 2015.

[47] Cem Kaner and W. Bond. Software engineering metrics: What do they measure and how do we know. In *Proceedings 10th International Software Metrics Symposium*, pages 1–12. IEEE, 2004.

[48] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[49] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[50] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[51] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[52] Jeff Tian. *Software quality engineering: testing, quality assurance, and quantifiable improvement*. John Wiley amd Sons, 2005.

[53] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[54] Java API for WordNet Searching (JAWS). `http://lyle.smu.edu/~tspell/jaws/`. Last Accessed: 2015-04-20.

[55] Eclipse metrics plugin. `http://sourceforge.net/projects/metrics/`. Last Accessed: 2015-04-24.

[56] News-A: An Online News Portal. `https://github.com/iitprojectlab/News-A-master`. Last Accessed: 2015-04-25.

[57] jsoup: Java HTML Parser. `http://jsoup.org/`. Last Accessed: 2015-04-24.

[58] Scientific calculator. `https://github.com/iitprojectlab/Scientific\Calculator`. Last Accessed: 2015-04-27.

[59] Sparrow: File Reading Software. `https://github.com/iitprojectlab/Sparrow`. Last Accessed: 2015-04-24.

[60] Thierry Dutoit, Vincent Pagel, Nicolas Pierret, François Bataille, and Olivier Van der Vrecken. The mbrola project: Towards a set of high quality speech synthesizers free of use for non commercial purposes. In *Fourth International Conference on Spoken Language, ICSLP*, volume 3, pages 1393–1396. IEEE, 1996.

[61] Amghotok: A platform of matchmaking. `https://github.com/iitprojectlab/Amghotok1`. Last Accessed: 2015-04-28.

[62] Painter: A canvas for painting freely. `https://github.com/iitprojectlab/Painter`. Last Accessed: 2015-04-27.

[63] POAS: Program Office Automation Software. `https://github.com/iitprojectlab/POAS`. Last Accessed: 2015-04-29.

Appendix A

News-A: An Online News Portal [56]

Table A.1 presents the requirements of News-A dataset. List of test cases with their execution results are presented in Table A.2 where result 0 and 1 denote the fail and success of test case execution. Number of fails are the numbers of detected faults.

Table A.1: List of Requirements of News-A Dataset

ID	Requirement Description
1	The app will retrieve and display all headlines of the Daily Prothom Alo
2	The app will retrieve and display all headlines of the DailyStar
3	The app will retrieve and display all headlines of the Bangladesh Today
4	The app will retrieve and display all headlines of the BDNews24
5	The app will retrieve and display all headlines of the Daily Sun
6	The app will retrieve and display all headlines of the New Age
7	The app will fetch and all news catagorized by sports
8	The app will fetch and show all news catagorized by politics
9	The app will fetch and show all news catagorized by entertainment
10	The app will fetch and show all news catagorized by business
11	The app will show the news in details based on the given headline including author and reviews
12	The detail news view will show image of the news in every image format with their descriptive date and time
13	The app will take not more than 30 seconds to display a single news
14	The app will support both Bangla and English language

Table A.2: List of Test Cases of News-A Dataset

Test Case ID	**Test Cases**	**Result**	**Requirement ID**
1	Check if the app display bangla news of the Daily Prothom Alo	0	14
2	Check average time to display a news from The The Daily Prothom Alo less than 30 seconds	0	13
3	Check if the app display all the headlines of the daily prothom alo catagorized by sports	1	1,7
4	Check if the app display all the headlines of the daily prothom alo catagorized by politics	1	1,8

Table A.2: List of Test Cases of News-A Dataset

Test Case ID	**Test Cases**	**Result**	**Requirement ID**
5	Check if the app display all the headlines of the daily prothom alo catagorized by entertainment	1	1,9
6	Check if the app display all the headlines of the daily prothom alo catagorized by business	1	1,10
7	Check if the app display all the headlines of the DailyStar catagorized by sports	1	2,7
8	Check if the app display all the headlines of the DailyStar catagorized by politics	1	2,8
9	Check if the app display all the headlines of the DailyStar catagorized by entertainment	1	2,9
10	Check if the app display all the headlines of the DailyStar catagorized by business	1	2,10
11	Check if the app display all the headlines of the Bangladesh Today catagorized by sports	1	3,7
12	Check if the app display all the headlines of the Bangladesh Today catagorized by politics	1	3,8
13	Check if the app display all the headlines of the Bangladesh Today catagorized by entertainment	1	3,9
14	Check if the app display all the headlines of the Bangladesh Today catagorized by business	1	3,10
15	Check if the app display all the headlines of the BDNews24 catagorized by sports	1	4,7
16	Check if the app display all the headlines of the BDNews24 catagorized by politics	1	4,8
17	Check if the app display all the headlines of the BDNews24 catagorized by entertainment	1	4,9
18	Check if the app display all the headlines of the BDNews24 catagorized by business	1	4,10
19	Check if the app display all the headlines of the Daily Sun catagorized by sports	1	5,7
20	Check if the app display all the headlines of the Daily Sun catagorized by politics	1	5,8
21	Check if the app display all the headlines of the Daily Sun catagorized by entertainment	1	5,9
22	Check if the app display all the headlines of the Daily Sun catagorized by business	1	5,10

Table A.2: List of Test Cases of News-A Dataset

Test Case ID	Test Cases	Result	Requirement ID
23	Check if the app display all the headlines of the New Age catagorized by sports	1	6,7
24	Check if the app display all the headlines of the New Age catagorized by politics	1	6,8
25	Check if the app display all the headlines of the New Age catagorized by entertainment	1	6,9
26	Check if the app display all the headlines of the New Age catagorized by business	1	6,10
27	Check if the app display a news in details of the daily prothom alo catagorized by sports	1	1,7,11
28	Check if the app display a news in details of the daily prothom alo catagorized by politics	1	1,8,11
29	Check if the app display a news in details of the daily prothom alo catagorized by entertainment	1	1,9,11
30	Check if the app display a news in details of the daily prothom alo catagorized by business	1	1,10,11
31	Check if the app display a news in details of the DailyStar catagorized by sports	1	2,7,11
32	Check if the app display a news in details of the DailyStar catagorized by politics	1	2,8,11
33	Check if the app display a news in details of the DailyStar catagorized by entertainment	1	2,9,11
34	Check if the app display a news in details of the DailyStar catagorized by business	1	2,10,11
35	Check if the app display a news in details of the Bangladesh Today catagorized by sports	1	3,7,11
36	Check if the app display a news in details of the Bangladesh Today catagorized by politics	1	3,8,11
37	Check if the app display a news in details of the Bangladesh Today catagorized by entertainment	1	3,9,11
38	Check if the app display a news in details of the Bangladesh Today catagorized by business	1	3,10,11
39	Check if the app display a news in details of the BDNews24 catagorized by sports	1	4,7,11
40	Check if the app display a news in details of the BDNews24 catagorized by politics	1	4,8,11

Table A.2: List of Test Cases of News-A Dataset

Test Case ID	Test Cases	Result	Requirement ID
41	Check if the app display a news in details of the BDNews24 catagorized by entertainment	1	4,9,11
42	Check if the app display a news in details of the BDNews24 catagorized by business	1	4,10,11
43	Check if the app display a news in details of the Daily Sun catagorized by sports	1	5,7,11
44	Check if the app display a news in details of the Daily Sun catagorized by politics	1	5,8,11
45	Check if the app display a news in details of the Daily Sun catagorized by entertainment	1	5,9,11
46	Check if the app display a news in details of the Daily Sun catagorized by business	1	5,10,11
47	Check if the app display a news in details of the New Age catagorized by sports	1	6,7,11
48	Check if the app display a news in details of the New Age catagorized by politics	1	6,8,11
49	Check if the app display a news in details of the New Age catagorized by entertainment	1	6,9,11
50	Check if the app display a news in details of the New Age catagorized by business	1	6,10,11
51	Check if the app display images of a news of the daily prothom alo catagorized by sports	0	11,12, 7
52	Check if the app display images of a news of the daily prothom alo catagorized by politics	0	11,12, 8
53	Check if the app display images of a news of the daily prothom alo catagorized by entertainment	0	11,12, 9
54	Check if the app display images of a news of the daily prothom alo catagorized by business	0	11,12, 10
55	Check if the app display images of a news of the DailyStar catagorized by sports	0	11,12, 7
56	Check if the app display images of a news of the DailyStar catagorized by politics	0	11,12, 8
57	Check if the app display images of a news of the DailyStar catagorized by entertainment	0	11,12, 9
58	Check if the app display images of a news of the DailyStar catagorized by business	0	11,12, 10

Table A.2: List of Test Cases of News-A Dataset

Test Case ID	Test Cases	Result	Requirement ID
59	Check if the app display images of a news of the Bangladesh Today catagorized by sports	0	11,12, 7
60	Check if the app display images of a news of the Bangladesh Today catagorized by politics	0	11,12, 8
61	Check if the app display images of a news of the Bangladesh Today catagorized by entertainment	0	11,12, 9
62	Check if the app display images of a news of the Bangladesh Today catagorized by business	0	11,12, 10
63	Check if the app display images of a news of the BDNews24 catagorized by sports	0	11,12, 7
64	Check if the app display images of a news of the BDNews24 catagorized by politics	0	11,12, 8
65	Check if the app display images of a news of the BDNews24 catagorized by entertainment	0	11,12, 9
66	Check if the app display images of a news of the BDNews24 catagorized by business	0	11,12, 10
67	Check if the app display images of a news of the Daily Sun catagorized by sports	0	11,12, 7
68	Check if the app display images of a news of the Daily Sun catagorized by politics	0	11,12, 8
69	Check if the app display images of a news of the Daily Sun catagorized by entertainment	0	11,12, 9
70	Check if the app display images of a news of the Daily Sun catagorized by business	0	11,12, 10
71	Check if the app display images of a news of the New Age catagorized by sports	0	11,12, 7
72	Check if the app display images of a news of the New Age catagorized by politics	0	11,12, 8
73	Check if the app display images of a news of the New Age catagorized by entertainment	0	11,12, 9
74	Check if the app display images of a news of the New Age catagorized by business	0	11,12, 10
75	Check if the app display headlines in Bangla version	0	12,13,14
76	Check if the app display all news in Bangla version	0	12,13,14

Appendix B

Scientific Calculator [58]

Table B.1 presents the requirements of Scientific Calculator. List of test cases with their execution results are presented in Table B.2 where result 0 and 1 denote the fail and success of test case execution. Number of fails are the numbers of detected faults.

Table B.1: List of Requirements of Scientific Calculator

ID	Requirement Description
1	In any situation the calculator has to produce a correct result defined by the well-known arithmetic rules
2	On encountering a division by 0 the display should read ”Infinity” and typing the key Clear should reset the calculator
3	On calculating the square root value of a negative operand the display should read ”NaN”
4	On erroneous operand or operation keys the display should read Reset (Clear) to continue as appropriate
5	On calculating arcsine, arccosine value input should be within -1 to 1 otherwise the display should read NaN
6	On calculating Ln and Log10 value must be greater than 0 otherwise the display should read -Infinity
7	This application will run and close properly
8	In this application all keys are functional.

Table B.2: List of Test Cases of Scientific Calculator

Test Case ID	Test Cases	Result	Requirement ID
1	Check if the calculator window maximize to certain window size	0	7
2	Check if the calculator closes when the close button is pressed.	1	8
3	Check if the calculator allows copy and paste functionality.	0	7
4	Check if the calculator has any specific preferences.	0	7
5	Check if all the numbers are working (0 to 9)	1	8
6	Check if the arithmetic keys (+, -, *, %, /) are working.	1	8
7	Check if the user can reset all by using Clear and and delete one by one Deletkey.	1	8

Table B.2: List of Test Cases of Scientific Calculator

Test Case ID	Test Cases	Result	Requirement ID
8	Check if the equal key is working.	1	8
9	Check if the power, exp and square root key is working.	1	8
10	Check if the 1/x, Ln and Log10 key is working.	1	8
11	Check if the sin, asin, cos, acos ,tan, atan key is working.	1	8
12	Check the addition of two integer numbers.	1	1,4
13	Check the addition of two negative numbers.	0	1,4
14	Check the addition of one positive and negative number.	0	1,4
15	Check the subtraction of two integer numbers.	1	1,4
16	Check the subtraction of two negative numbers.	0	1,4
17	Check the subtraction of one negative and positive number.	0	1,4
18	Check the multiplication of two integer numbers.	1	1,4
19	Check the multiplication of two negative numbers.	0	1,4
20	Check the multiplication of one negative and positive number.	0	1,4
21	Check the division of two integer numbers.	1	1,4
22	Check the division of two negative numbers.	0	1,4
23	Check the division of one positive number and integer number.	1	1,4
24	Check the division of a number by zero.	1	2,4
25	Check the division of a number by negative number.	0	1,4
26	Check the division of zero by any number.	1	1,4
27	Check if the 1/x is operational and works as expected.	1	2,4
28	Check if the Log10 key is operational and works as expected.	1	6,4
29	Check if the Ln key is operational and works as expected.	1	6,4
30	Check if the asin is operational and works as expected.	1	5,4
31	Check if the acos is operational and works as expected.	1	5,4
32	Check if the atan is operational and works as expected.	1	1,4
33	Check if the sin is operational and works as expected.	1	1,4
34	Check if the cos is operational and works as expected.	1	1,4
35	Check if the tan is operational and works as expected.	1	1,4
36	Check if the atan key for 90 degrees result with infinity.	0	1,4
37	Check if the sqrt key operational and result with infinity for negative value	1	3

Appendix C

Sparrow: File Reading Software [59]

Table C.1 presents the requirements of Sparrow. List of test cases with their execution results are presented in Table C.2 where result 0 and 1 denote the fail and success of test case execution. Number of fails are the numbers of detected faults.

Table C.1: List of Requirements of Sparrow

ID	**Requirement Description**
1	Sparrow has a well shaped user interface to use
2	User interface must be functional properly
3	Sparrow can upload any formatted and non formated file
4	It must be capable to extract formatted text file like pdf, doc etc.
5	It can process direct user input from input text field
6	Sparrow can generate understandable virtual voice to read in every input and every scenario
7	Voice duration can be changeable
8	Volume must be manually adjustable
9	Sparrow must have a manual pause button to pause and voice
10	Sparrow must have a resume option to play the voice
11	program must have a restart button for replay any voice

Table C.2: List of Test Cases of Sparrow

Test Case ID	**Test Cases**	**Result**	**Requirement ID**
1	Check if the sparrow application window maximize and resize to certain window size	1	1
2	Check if the sparrow closes when the close button is pressed.	1	1
3	check the user interface button functionality	1	2
4	Check file upload button, input text button, exit button etc for extected output	0	1, 2
5	check the file uploded activities like pdf, doc files. Not any image file	0	3
6	check the text extraction from formated text file	1	4
7	check the text extraction from unformated text file	1	4
8	check the direct input from user option	1	5
9	Check the virtual voice for understanding in all formate	0	6
10	Check the voice duration	1	7
11	Check the volume adjustability	1	8
12	Check the pause button functionality	0	9
13	Check the resume button functionality	0	10
14	Check the stop button functionality	1	9, 10
15	Check the restart option	0	11

Appendix D

Amghotok: A Platform of Matchmaking [61]

Table D.1 presents the requirements of Amghotok. List of test cases with their execution results are presented in Table D.2 where result 0 and 1 denote the fail and success of test case execution. Number of fails are the numbers of detected faults.

Table D.1: List of Requirements of Amghotok

ID	Requirement Description
1	Accessible via smartphone synchronized with Google account.
2	Allow user to create own profile and spouses profile.
3	Allow user to update any information with or without internet.
4	Save the user information into local database and in web server.
5	Allow registered user to contact with any profiles.
6	Dont allow visitor to contact anyone but have privilege to search on specific constraints.
7	Have privilege to search on any information for registered user.
8	Get all featured and latest update from web server to local database and to show the user.
9	Have the privacy about email, phone number, picture or any information that user wants.
10	Create local database to store last seen page information, update information and profile information.
11	Have the privilege to see the last visited page.
12	All forms must be verified.
13	All UI must working properly.

Table D.2: List of Test Cases of Amghotok

Test Case ID	**Test Cases**	**Result**	**Requirement ID**
1	Check it is accessible via smartphone synchronized with Google account	0	1
2	Check it allows user to create own profile and spouses profile	1	2
3	Check it allows user to update any information with or without internet	0	3
4	Check it saves the user information into local database and in web server	1	4
5	Check it allows registered user to contact with any profiles	0	5
6	Check it doesnt allow visitor to contact anyone but have privilege to search on specific constraints	0	6
7	Check there isany privilege to search on any information for registered user	1	7
8	Check system can get all featured and latest update from web server to local database and to show the user	1	8
9	Check there is any privacy about email, phone number, picture or any information that user wants	0	9
10	Check system can create local database to store last seen page information, update information and profile information	1	10
11	Check there is any privilege to see the last visited page	0	11
12	Check if user can upload photos from device	0	2,3
13	Leave the username blank and click submit in user profile	0	12
14	Check birthday is given in proper format in user profile	1	12
15	Check height is given in number in user profile	0	12
16	Check weight is given in number in user profile	0	12
17	Check height is given in number in partners profile	0	12
18	Check weight is given in number in user profile	0	12
19	Check Up navigation is working properly	1	13
20	Check all dropdown input field is working properly	1	13

Appendix E

Painter: A Canvas for Painting Freely [62]

Table E.1 presents the requirements of Painter. List of test cases with their execution results are presented in Table E.2 where result 0 and 1 denote the fail and success of test case execution. Number of fails are the numbers of detected faults.

Table E.1: List of Requirements of Painter

ID	Requirement Description
1	Drawing canvas must be well shaped and resizable as user need
2	Drawing image using pencil and different types of shape.
3	The Painting editor will be combined with some painting tools. Such as: Pencil, Eraser, Color picker, brushes.
4	There will be a huge collections of different types of shapes to enhance ones drawing ability.
5	The editor will provide some operations such as cut, copy , paste, delete & upload image & files.
6	The editor will provide simple image manipulation such as crop, rotate, resize.
7	Image can be uploaded to edit contrast and add text.
8	The menu Color contains the menuItem Edit color by which user can select any type of colors using color chooser for drawing
9	Erase and smudge an image using eraser and smudge tool.
10	Image can be rotate in ninty degree angle
11	Slider will be used for changing the size of pencil and eraser.
12	Mnemonics and Tool Tip Text are used to make user friendly
13	Extra feature will be work with image animations
14	Save paint work in any format such as gif, png, jpg, bmp, jpeg.
15	There will be a user guideline that will help the user to use the editor perfectly.

Table E.2: List of Test Cases of Painter

Test Case ID	**Test Cases**	**Result**	**Requirement ID**
1	Check if the painter window maximize to certain window size	0	1
2	Check if the painter closes when the close button is pressed.	1	1
3	Check if the pencil can draw	1	2, 3
4	check if the eraser can erase the paint properly	0	3, 9
5	Check the brush size is resizable	1	3, 4
6	check if the cut and copy option for shapes works properly	1	5
7	check the past option in any position in that canvas	0	5
8	check the rotate option	0	6, 10
9	check if the canvas is resizable for any ratio	1	6
10	check the image upload system for painting canvas	1	7
11	check the color bucket for every color in background and foreground	1	8
12	check the image rotation for every angle	0	10
13	check the slider activities	1	11
14	check if the tooltip text is works properly or not	1	12
15	check the image animation section is working for every format of image	1	13
16	check the save option for all image option such as gif, png, jpeg, jpg, bmp etc.	1	14
17	check the user guideline is well written	1	15
18	check the polygon size and activities	0	4
19	check the shape over written option to overwrite any shapes	0	4, 3, 11
20	check the color contrast and blare color option	0	8

Appendix F

POAS: Program Office Automation Software [63]

Table F.1 presents the requirements of POAS. List of test cases with their execution results are presented in Table E.2 where result 0 and 1 denote the fail and success of test case execution. Number of fails are the numbers of detected faults.

Table F.1: List of Requirements of POAS

ID	Requirement Description
1	The Accounting Software must keep track of the accounting records it is expected to generate and must clearly signal if data associated to any accounting record is missing or incomplete
2	The system must store and manage accounting records and accounting data into ledger.
3	The system will allow only permitted users to access through individual login account. They can insert, update and delete data.
4	All entries within a transaction must have the same transaction ID. All the dates and times will be saved automatically.
5	Current status of any field can be shown
6	The administrators can search on any fields.
7	The system will automatically send task forwarding e-mail to the users.
8	The system must ensure database backups, which are as recent and complete as possible
9	The interface will be designed in a manner which implements functionality that are easily accessible by the user, for example by providing a single-click link or drop down list.
10	Backup files should be in readable file format, using plain text and standard compression formats.
11	The system will provide a basic calculations facility for users
12	The system may allow the user to enter a date through using a GUI calendar and selecting the day

Table F.2: List of Test Cases of POAS

Test Case ID	**Test Cases**	**Result**	**Requirement ID**
1	Can the accounting software keep track of the accounting records?	0	1
2	Can clearly signal if data associated to any accounting record is missing or incomplete?	1	1
3	Can system store and manage accounting records and accounting data into ledger?	0	2 1 6
4	Do the system only allow permitted users?	1	3
5	Can users insert, update and delete data?	0	3 4 2 1
6	Can users generate reports?	0	7 4 10
7	Each transaction must have a transaction ID.	1	4
8	Is the dates and times save automatically?	1	4
9	Is current status of any field can be shown?	0	5 4
10	Can the administrators search on any fields?	0	6 4
11	Shall the system automatically send e-mail to the user when a task is forwarded?	0	7 4 10
12	Can the system ensure database backups, which are as recent and complete as possible?	0	7 4 10
13	Is the interface be designed in a manner which implements functionality that are easily accessible by the user, for example by providing a single-click link or drop down list?	1	9
14	Are the backup files in readable file format, using plain text and standard compression formats?	0	10 4 8
15	Is the system provide a calculator facility to allow the user to make basic calculations?	1	11
16	Is the system allow the user to enter a date through using a GUI calendar and selecting the day?	0	12 1 8 4
17	Are all entries within a transaction have the same transaction ID?	1	4