Automated Techniques to Detect Faults Early in Large Software Applications

This Thesis is
presented to the
school of computer science & software engineering
for the degree of
doctor of philosophy
of
the university of western australia

THE UNIVERSITY OF
WESTERN AUSTRALIA

By
Deepak Garg
July 2014

This thesis is dedicated to my parents,
Prof. A.K. Verma and Prof. A. Srividya for
their love, endless support and encouragement.

Abstract

Modern software applications are very complex and they need frequent changes as per the changes in user requirements. These applications are developed using the combination of various different programming languages. They consist of a multi-tiered application structure and have often a database as an essential tier. The correct functioning of a multi-tiered application is dependent on its interaction between different tiers and its database connectivity and storage. Most of these applications rely on a database server for serving client requests. Any changes in the database result in erroneous client interactions and may bring down the entire software application.

Software applications grow in size due to the change in user requirements and need a large number of test cases. Hence, it becomes very difficult to execute a large number of test cases within a specified period of time. Researchers have suggested many Test Case Prioritisation (TCP) techniques and selection techniques for reducing testing time. A TCP technique reorders test cases to achieve early fault detection. The selection techniques help in selecting the subset of test cases from the large set of test cases that are required for the testing of particular functionalities. Most TCP techniques have been developed for efficient testing of source code of an application. However, most prioritisation techniques are unsuitable for automatically prioritising test suites when the software applications are written using a combination of various different programming languages.

The manual prioritisation of large set of test cases is very time consuming. We suggest automated prioritisation approaches to detect faults early in multi-tier software applications. The new two level prioritisation approach automatically prioritises test cases of multi-tier software applications. This approach is based on Functional Dependency Graphs (FDG) and Inter-procedural Control Flow Graphs (ICG). This approach selects modified functionalities using FDG and automatically

executes test cases on the basis of the impact of modified source code using ICG. We have suggested several new prioritisation strategies and analysed whether these prioritisation strategies improve the rate of fault detection. We propose a new test suite prioritisation model that detects modified functionalities, selects test cases related to those functionalities and reschedules them to detect faults early in test suite execution.

Every testing cycle of a software application results in new test cases being introduced in a test suite. Many test cases from previous testing cycles become unstable or unusable due to the removal/addition of the new functionalities. The execution of a large number of unusable test cases results in less test coverage and higher test execution time. The lower test coverage is due to the coverage of the non-existent code statements. The higher test execution time is due to the execution of unused and broken test cases. In this thesis, we propose a new bipartite graph approach to automatically map the test cases with program source code and eliminate the subset of test cases that is not relevant for the testing of the current version of a multi-tier application. The suggested approach helps in executing a minimal set of test cases that is required to cover more code statements.

There are very few proposals in the literature for prioritisation of test cases that can detect faults in the database early. The existing techniques are unsuitable for automated early detection of faults due to modifications in databases. We present a new technique to detect database schematic changes in a multi-tiered software application and suggest a new technique to detect the faults early that are related to database schematic changes. Our approach is based on an improved technique for the selection and prioritisation of test cases. We use the selection technique to select database related test cases of a multi-tiered software application using bipartite graphs. After selecting these test cases, we reorder them using our new prioritisation strategies. Our new approach results in early detection of faults related to database schematic changes and reduction of test suite execution time. Using the suggested approach, software engineers will be able to detect mod-

ifications in the database and execute only those test cases that are related to the database schema and its modifications.

We have suggested techniques using Functional Dependency Graph (FDG) and Inter-procedural Control Flow Graphs (ICG) to detect faults due to the modifications in the database. The modifications in database schema may result in faults and may end up in the non-functionality of a software application. To test for the modifications in the database schema of a software application, software engineers normally execute all the test cases. We propose a new automated TCP technique for software applications that automatically identifies modifications in database, prioritises test cases related to these modifications and executes them to detect faults early.

Due to the large number of test cases, it becomes very difficult to execute a large set of test cases on a single machine. To overcome this issue, we present a new approach for automatically prioritising and distributing test cases among multiple machines. Our approach is based on the FDG of a software application. We partition the test suite into test sets according to the functionalities and associate the test sets with each module of the FDG. The high priority modules and their associated test sets are then distributed evenly among the available machines. Moreover, we further prioritise the test cases by using ICGs within individual functional modules. Our suggested approach reduces the test suite execution time and helps in detecting the faults early. We demonstrate the effectiveness of our technique through an experimental study of a software application and measure the performance of our technique by using the well known APFD (Average Percentage of Faults Detected) metric.

The software applications that are built on a multi-tier architecture are very large in size and grow in time due to the enhancement of new features. To ease expenses and reduce the amount of time required, many companies follow distributed software development. The applications developed using distributed software development, require tremendous

amount of testing as these applications are developed by a team of developers deployed at different locations. Hence, it becomes very difficult to detect faults. To make this testing process easier, more efficient and to reduce software testing costs, the software companies allow their users to raise issues in a centralised bug tracking system. This bug tracking system is accessible to all the users and software development teams. Due to large number of test cases, it becomes very difficult for the software engineers to execute all these test cases and detect issues raised by end-users. We suggest a new technique to detect faults early in distributed software development environment. We suggest this new technique by introducing a new metric that identifies the severity of a fault and calculates the severity of faults in some portion of a program source code. This metric helps in prioritising the test cases based on the score obtained from this metric. Our new suggested approach detects faults earlier in a test suite execution process.

In this thesis, we have suggested many new techniques and algorithmic approaches to detect faults earlier in multi-tier software applications. The early detection of faults helps in reducing testing costs and minimises the time required to test a particular software application.

Preface

Some of the research presented in this thesis has already been published.

Multi-tier software applications are very large and require frequent changes in the source code to meet the user requirements. The modifications in program source code and database may give rise to faults. To identify the faults due to the modifications in these software applications, software engineers execute all the test cases, which is very time consuming and increase the software testing and maintenance costs. Existing research is based on the static analysis of code and is dependent of the program code of that programming language. This thesis focuses on fault detection in multi-tier applications developed using combination of different programming languages. It also suggests new techniques to detect faults early due to modifications in database.

This thesis work has been carried out from November 2010 to January 2014 at the School of Computer Science and Software Engineering. This thesis consists of nine chapters. Six chapters contains papers that are accepted by or intended for international journal or conference proceedings. These chapters covers the report of our new selection and prioritisation approaches of test cases to detect faults in multi-tier software applications.

Acknowledgements

My PhD research has been a rewarding journey and many people made its completion possible. I must extend my whole hearted gratitude and respect to my principal advisor Professor Amitava Datta. My research would not have progressed without his help, support and encouragement throughout my candidature. It has been an honour to study under his supervision. I also would like to thank my co-advisor A/Prof. Tim French for his reviews and comments on my research papers. The meetings with Tim were helpful in developing new ideas.

I am grateful to the anonymous referees of the papers whose comments improved my research papers that in turn improved my thesis chapters.

I am grateful to the researchers, computer systems administrators, and the secretaries in the school of computer science and software engineering, for assisting me in different ways. Ryan McConigley, Ashley Chew and Laurie McKeig who, despite heavy demands from the school, found the time to resolve my technical problems and fix my broken computers. I am thankful to Ms. Yvette Harrap for her assistance in resolving the administrative issues.

My research has been fully funded by the SIRF, UIS, Ad-Hoc Safety Net Top-Up, UWA travel grant for attending ICST 2012 (IEEE International Conference on Software Testing, Verification and Validation) conference in Montreal, Canada and UWA CSSE Adhoc Top-Up scholarships from The University of Western Australia, IEEE Richard E. Merwin Scholarship, and the ACM travel grant for attending ISSTA

2013 (International Symposium on Software Testing and Analysis) conference in Lugano, Switzerland.

Finally and most importantly, I dedicate this thesis to my parents. I would like to express my heart-felt gratitude to Prof. Ajit Kumar Verma and Prof. A. Srividya for their guidance and support. Words cannot describe how grateful and appreciative I am for their retentless dedication, support, care and love.

Publications

- D.Garg and A.Datta, "A Novel Metric for Prioritising Test Cases in Distributed Software Testing and Development,". (to be submitted)

- D. Garg, A. Datta, and T. French, "A Novel Bipartite Graph Approach for Selection and Prioritisation of Test Cases," ACM SIGSOFT Softw. Eng. Notes, 38(6):1-6, November 2013.

- D. Garg and A. Datta, Early detection of faults related to database schematic changes. In Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation, JAMAICA 2013, pages 1 - 6, New York, NY, USA, 2013. ACM.

- D. Garg and A. Datta. Parallel execution of prioritized test cases for regression testing of web applications. In Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135, ACSC '13, pages 61 - 68, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc.

- D. Garg, A. Datta, and T. French. A two-level prioritization approach for regression testing of web applications. In 19th Asia-Pacific Software Engineering Conference (APSEC), volume 2, pages 150 - 153, 2012.

- D. Garg, A. Datta and T. French, New Test Case Prioritization Strategies for Regression Testing of Web Applications," Interna-

tional Journal of Systems Assurance Engineering and Management (Springer), 3(4):300 - 309, 2012.

- D. Garg, A. Datta, Test Case Prioritization Due to Database Changes in Web Applications," In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pages 726 - 730, April 2012.

Contribution to Candidate to Published Papers

My contribution in all the papers was 85%. I developed and implemented the techniques, performed execution and wrote the papers. My supervisor, Professor Amitava Datta and Assistant Professor Tim French, reviewed the papers and provided useful feedback for improvement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software testing is as old as computer programming [93]. Computer programs have increased in size and complexity since their early days in 1960s and the need for eliminating defects from them in a systematic manner received greater attention. Thus, in the 1970s, a new field of research called testing theory came into existence. In 1977, McCall et. al first described the concepts and definitions of software quality [64] [93]. In 1979, Myers, defined testing as "the process of executing a program with the intent of finding errors" [5]. This definition made implementation of software to detect fault a primary goal. The main aim of testing is to detect faults in software applications.

Software testing plays an important role in achieving a standard of quality. The importance of the safety of software in industries like aerospace, aeronautics, and medicine goes without saying but even in non-safety critical industries like telecommunications, the importance of an uninterrupted service is important from a customer satisfaction and retention perspective [69]. Software testing is useful in allowing developers of software prove that a system meets all of the objectives and system requirements that are specified [5] [26]. The criticality of software testing is another important aspect. Testing is critical because failure may be very costly as sometimes faults in software applications may result in losses to organisations and businesses that use software

applications extensively. Most such applications are very complex and require frequent changes in their functionalities for supporting diverse user requirements. Therefore testing of software applications is particularly challenging and failure of such applications may be very costly.

Moreover, many software applications must have extremely high availability and software testing of such applications must be efficient. It is not possible to test complex applications offline as the time for testing should be minimal and an aplication should be up and running as soon as possible. Many software applications usually provide both a static and a dynamic interface [50] [85]. Static interfaces present the same content to all the users and dynamic interfaces present content depending upon the user input. Dynamic run-time interactions make the testing of these software applications a challenging task [66].

There are numerous examples of the problematic impact of interrupted service on industries [69]. Recently in 2011, a software fault in Google mail application started erasing emails of its Gmail users over the weekend and two percent of Gmail users lost access to their email [100]. In 2010, a software fault in an iSoft application(Australia′s biggest health technology company) used at Gisborne Hospital resulted in switching of patient's details; this incident led to the extensive remedial work on the iSoft product [45]. Also in 2010, a software fault in a banking application left millions of German debit and credit card holders unable to withdraw money or make payments in shops and many of them were left without having access to cash [15].

Conformance testing is used to determine whether a product or system behaves according to specifications. A program written in any programming language like C, C++ or Java requires frequent changes due to the change in user requirements. The main aim of testing of the program is to detect faults due to the modifications in the required services/-functionalities according to specifications. However, it takes long time to detect faults in multi-tier applications as they follow multi layered architecture and are very large in size. Due to the large size of these

applications, it has large number of test cases. The running of a large number of test cases requires a long time.

Modern software applications are usually multi-tier, consisting of modules written in different programming languages as well as back-end databases. These multi-tier applications undergo rapid development cycles due to the presence of bugs and changing client and user requirements. Regression testing is a widely used testing technique for rapid testing of multi-tier applications. For every new version of a software application released, regression testing is required to test the compatibility of the new features with the previously tested functionalities. Regression testing is a very expensive testing activity and used to validate a modified software according to its specifications and detect whether the previously tested functionalities have not introduced any new faults [80]. The main aim of regression testing of a multi-tier application is to uncover new bugs as early as possible after changes in the code. A test suite used for regression testing consists of test cases that are required to test a multi-tier software application. New test cases in the test suite are generated depending on the changes in the client and user requirements and subsequent changes in the code.

Usually multi-tier software applications require more testing and maintenance than any other software systems as new functionalities are introduced depending on user requirements [21]. The testing of these applications differs considerably from traditional software systems due to many technologies used for developing different modules. All these components are handled by software development teams located at various different locations. The end-users from across the world are connected through the multi-tier application client interface to the internet. The user submits a request to the web server through the client. The web server then requests the application objects and the application objects synchronise with the database server. The application server then synchronises with the multi-tier application for processing of the request. The failure of a request results into fault. There are chances of faults in any tier of these applications.

Since modern software applications typically have a rapid turn around time, it becomes very difficult to execute all the test cases within a specified amount of time. The cost of re-running all test cases may be expensive and not always useful as sometimes only selected functionalities need to be tested. Hence, test cases are usually prioritised during testing in order to discover the likely vulnerable parts of the code early so that developers have more time to identify and debug the faults. The serial execution of a test suite on a single machine might take many hours depending on the size of an application, the machine where the test suite is run and its workload [53].

This thesis presents a new framework to detect faults early in multi-tier software applications. This framework comprises new techniques that are based on the prioritisation and selection techniques of test cases. These techniques can be used on any software application developed using multi-tier architecture. This chapter describes the research background, aims and the thesis outline.

1.1 Research background

There are several different approaches to software testing. Dynamic testing involves the execution of the program or some part of the program in order to detect faults. Dynamic testing can be further broken down into various sub-types (Black, White, and Gray Box Testing). Regression testing is a testing process which is applied after a program gets modified. The purpose is to check whether the program performs according to specifications [55].

A test case is a complete set of test inputs, execution conditions, and expected outputs developed for a particular objective to validate some specific functionality in the application under test [57]. A test suite is a collection of test cases to test a software application. An effective test suite is required to test an application. A test suite is effective if it can detect a significant percentage of bugs in a software.

Regression testing attempts to validate a modified software and ensures that no new errors are introduced due to recent modifications. The most important issue in regression testing is to find a cost-effective approach in terms of the testing effort and time required for testing. In regression testing, tests are selected, prioritised, and executed from the existing pool of test cases to ensure that bugs are minimized in the new version of the software. Regression testing is an expensive process and accounts for a predominant portion of testing effort in the industry.

Test cases are prioritised during the software testing in an order to exercise the likely vulnerable part of the code early so that developers will get more time to debug and fix the faults if any of them are found. For example, if the faults are revealed towards the end of a test cycle, the developers may not have enough time to fix the faults; moreover, the software test team may not be able to validate the fixes in a very short time [93]. As the suite of test cases grows from version to version, test case selection and TCP for regression testing becomes an increasingly important task [7]. Hence prioritisation techniques aim to reduce the software testing costs. These costs include the time required by software engineers to test the entire software application and the designing, maintaining, and executing the various test cases [93].

In the development phase, regression testing begins after detecting and correcting the errors in the program. Leung et al. described two types of regression testing based on whether specifications have changed [56]. *Corrective* regression testing applies to the unmodified specifications and test cases can be reused in this case. *Progressive* regression testing applies when specifications are modified and new tests are needed to be designed. In general, corrective regression testing should be an easier process than progressive regression testing because more test cases can be reused. Testing at unit, integration or system level reveals different types of failures but regression testing is considered as a sub phase of unit, integration, and system-level testing.

1.1.1 Software faults

Ploski et al. described that software fault refers to design faults introduced into software during any phase of software development, such as specification, programming, or installation [74] [1]. Gray classified software faults as transient faults and non-transient faults [35]. Transient faults are also known as Heisenbugs that cannot be simply reproduced by repeated execution and non-transient faults are also known as Bohrbugs, they are like Bohr atom, and are very solid and can be easily reproducible. The following are the different categories of faults considered in the literature.

1. **Data store faults:** Faults that are used to manipulate the data in any kind of data store.
2. **Logical faults:** Faults that manipulate the business logic and control flow e.g., if the user inputs the same string for the *password* and *confirm password* fields, still the application displays the error message *Password and Confirm Password fields don't match.*
3. **Form faults:** Faults that control, modifies and displays name value pairs in forms.
4. **Appearance faults:** Faults that control the way in which a web page is displayed.
5. **Link Faults:** Faults that change the page pointed by an URL.

1.1.2 Test coverage

Test coverage is a measure of the proportion of program executed by a test suite [43]. Test coverage is expressed in percentage. This allows the collection of information about the parts of the program that are actually executed while running the test suite. From the late 1970 to early 1980, test coverage was done on large scale projects [73]. Test coverage measurement provides feedback to software developers which

helps them to increase the coverage by adding more test cases but the test cases are not replicated [61]. This increased coverage will result in lower error removal costs and less errors in the software product. We can increase the test coverage by prioritising the test cases in an order that the test cases that cover most part of the program are to be executed first.

Test coverage in software is measured in terms of structural or data-flow units. These units can be statements, branches etc as described below:

1. **Statement coverage:** The fraction of the number of statements that have been executed by the test data.
2. **Branch (or decision) coverage:** The fraction of the number of branches that have been executed by the test data.
3. **C-use coverage:** The fraction of the number of computation uses (c-use) that has been covered by one c-use path during testing. A c-use path is a path through a program from each point where the value of a variable is modified to each c-use.
4. **P-use coverage:** The fraction of the number of p-uses that have been covered by one p-use path during testing. A p-use path is a path from each point where the value of a variable is modified to each p-use, a use in a predicate or decision.

1.1.3 Overview of multi-tier software applications

A complex and multi-tier, heterogeneous architecture may include web servers, application servers, database servers and client interpreters and testing must handle all of these components. The testing of these applications differs considerably from traditional software systems due to many different technologies used for developing the different modules.

The maintenance of multi-tier applications is more costly due to the complexity of the applications and their frequent updates [102], and to test the updates to check whether these updates have not affected

any of the already working functionalities. Software engineers need to execute all the test cases to validate the software according to its specifications. Every regression testing cycle results in new test cases due to the development of the new functionalities and enhancement of the existing functionalities. Due to the rapid turn around time in testing of multi-tier applications, sometimes it becomes difficult to execute all the test cases in a specified amount of time. Hence, efficient testing needs to prioritise the test cases to detect the faults early in a test suite execution.

Most of the modern widely deployed multi-tier applications follow a three-tiered application structure, which is composed of a front-end web server, an application server and a back-end database server. The end users/customers are connected to internet and software application. The user submits the request to the web server; the web server requests to the application objects and the application objects synchronises with the database server and the application server synchronises with the banking application and legacy application for the processing of the transaction. Any modifications in the back-end server result in faults. The back-end server faults mainly relate to database-specific faults. These faults include problems related to the schema of the database, incorrect storage of values in the database and problems related to the execution of *SQL* commands [14].

1.1.4 Distributed development and testing of multi-tier software applications

Distributed development and testing of applications help in reducing software development costs. Multi-tier applications are also considered as distributed software applications as these applications are typically composed of communicating components hosted over distributed and heterogeneous platforms [20]. It is very important to ensure the quality and reliability of these applications as many are used by a large number of users running large businesses. Businesses are spread over

different locations and distributed applications are used to manage such large businesses. These applications are very complex and require frequent changes in their functionalities for supporting diverse user requirements [63]. Many software applications must have extremely high availability. Therefore software testing of these software applications is particularly challenging and failure may be very costly.

Rothermel et al. suggested a technique to prioritise test cases when the software is written in a single language [83] [80]. Their technique constructs control graphs and uses those graphs to select the test cases related to the modified version of that software. As distributed software applications are usually written using multiple languages, it is difficult to directly apply their technique on this kind of applications.

1.1.5 Overview of testing of multi-tier applications

We will mainly consider web applications for illustrating multi-tier software applications. This is due to the fact that web applications are ubiquitous and they typically use a complex and multi-tier, heterogeneous architecture including web servers, application servers, database servers and client interpreters and testing must handle all these components [63]. The end-users are connected through the web application client interface to the internet. The user submits a request to the web server through the client. The web server then requests the application objects and the application objects synchronise with the database server. The application server then synchronises with the web application for processing of the transaction.

Web applications are very complex, ever revolving and rapidly updated software systems. The testing of web applications is very challenging and critical. It is challenging because traditional tools and methods are not always sufficient due to the heterogeneous nature of web applications. Testing of web applications is very critical because failure may be very costly. e.g. A failure in Amazon.com in 1998 brought down the

site for several hours and resulted in an estimated loss of $400,000 [98]. Some of the web applications must have extremely high availability, not just 24/7, but 24/7/365. There are many competitors in the same business. The only thing that distinguishes the winners from losers in the domain of web applications is the ability to provide a unique and satisfying web experience that has round the clock availability.

An effective testing approach must handle all the components of a web application. The major problem in testing is at unit testing level where a unique definition of scope of a unit test cannot be provided due to the heterogeneous architecture used in these applications [16]. As web applications have rapid development and testing cycles, many of the testing methods proposed in the literature cannot be used in a rapid testing environment as they are very time consuming.

As customer functionality requirements increase, web applications grow in size. Functional test cases are designed to test the functionalities of web applications according to the user specifications. For every new version of a web application released, regression testing is required to test the compatibility of the new features with the previous already tested functionalities. New test cases are generated for performing regression testing. As the test cases grow in number due to the growth of applications, it becomes very difficult to execute all the test cases within a specified amount of time.

1.2 Research aims and thesis outline

Our aim in this thesis is to propose novel prioritisation frameworks testing for multi-tier applications. Though our primary aim is to prioritize the existing test cases in a test suite, we also investigate the selection of test cases, as prioritization is not sufficient for reducing the testing effort of large multi-tier applications.

Our first proposal is a new framework to detect faults early in multi-tier applications. Our framework combines selection and prioritisation techniques of test cases. We detect the modifications in source code using existing techniques that use Inter-Procedural Control Graph (ICG). Further, one of our novel contributions is the detection of the modified functionalities in a software application using Functional Dependency Graphs (FDG). FDG is based on the functional dependency analysis of a software application. We select the test cases that are required to test the modified parts of a program source code and executes only those test cases. We propose extensive prioritization strategies for this approach.

We also propose a novel selection strategy for selecting only the relevant test cases using an approach based on bipartite graphs. This approach has not been explored in the literature and is based on a matching of the test cases and source code components. As most test suites collect a large number of test cases over time, it is important to selct only those test cases that are relevant for the current version of the software application. We show that our proposed approach significantly reduces the testing effort.

One of the main contributions of this thesis is the focus on testing the database tier of a multi-tier application. The testing of the application code related to database changes has not been explored in the literature, although it is very common that a multi-tier application can become unavailable quite often due to the failure of the database back-end. We suggest different techniques for selection and prioritisation of test cases that are related to database changes and reflected in program source code. These changes could be due to dabase schematic changes or bugs in the program source code that prevent correct access to the database.

Since rgression testing of large multi-tier applications is time consuming, it is desirable that the test suite can be partitioned into separate independent test suites. The advantage of this approach is that these smaller test suites can then be executed on multiple machines and over-

all thoroughput of testing can be improved considerably. We suggest an approach to partition a test suite and simultaneously executing these subset of test cases on multiple machines.

Finally, we suggest a new metric and prioritisation techniques to detect faults early in software applications that are developed using distributed software engineering environments. We suggest a new approach to detect faults early when the software development teams are geographically distributed.

1.2.1 Thesis outline

This thesis consists of nine chapters that are organized as follows.

Chapter 2 discusses state-of-the-art work related to regression testing. This chapter describes the work related to prioritisation of test cases, selection of test cases and metric for measuring fault detection rates of test suites. Besides that, an extensive review of generation of test cases and execution of test cases is also given. This chapter also describes the research methodology employed in relation to the state of art. The explanation includes the framework, heuristics and validation process.

Chapter 3 describes a new approach to detect faults due to the modifications in source code. In this chapter, the new two-level approach is described using functional dependency graphs and inter-procedural control flow graphs. Furthermore, this chapter describes the new prioritisation strategies and includes its experimental evaluation. Each process in this stage is comprehensively described and a brief example of each process implementation is given.

Chapter 4 describes a new selection approach using bipartite graphs. This technique is used to select the subset of test cases related to modifications in source code using bipartite graphs. Based on the results, a comprehensive discussion on why the new suggested technique and framework is able to detect faults early is then presented.

Chapter 5 describes a new technique to select the subset of test cases related to modifications in database. This technique is based on bipartite graphs. This chapter further describes the mapping between source code and database. It also describes the mapping between source code and test cases. The selected test cases are prioritised using new prioritisation approaches.

Chapter 6 describes a new approach to detect faults related to modifications in database. Several new prioritisation techniques are proposed in this chapter. The techniques proposed in this chapter are based on functional dependency graphs.

Chapter 7 describes a new approach to execute a large set of test cases in parallel and simultaneously on different machines. An experimental implementation of this approach is also presented.

Chapter 8 describes a new metric to prioritise the test cases when the software applications are developed using distributed software development and testing environments. This approach introduces a new metric to reorder the test cases on the basis of the score obtained from the new metric. This approach is beneficial when the software applications are developed using distributed software development environment and faults are raised by users from different locations.

Chapter 9 summarises the conclusions and proposes future work.

Chapter 2

Background Study Related to Selection and Prioritisation Techniques of Test Cases

2.1 Introduction

We discuss the backgorund on regression testing and the state-of-the-art work on regression testing in this chapter. Since regression testing is an expensive process, selection and prioritization of test cases are the most important research issues for improving the time for regression testing. This chapter describes various prioritization, selection and minimisation techniques for test cases. This chapter also reviews the known prioritization techniques that are specifically for testing of web applications. We have compared our proposed techniques with some these state-of-the-art techniques in Chapters 3 - 8.

We also review the various metrics for evaluating the effectiveness of prioritisation techniques. We have used these metrics to compare our proposed prioritization techniques with the available state-of-the-art techniques.This chapter further describes a brief overview of our research work in comparison to the state-of-the-art.

2.2 Regression testing

Regression testing is widely used for testing a modified software application to check whether it behaves as per specifications. Due to frequent addition of new requirements and functionalities, the software engineers perform regression testing after every change to check whether the newly introduced code affects the previous working code. It attempts to validate modified software and ensure that no new errors are introduced.

Regression testing is very important, but it's also very expensive. In the current situation, regression testing accounts for one-third of the total cost of a software system [55] [40]. Therefore the most important issue in regression testing is to find a cost-effective way.

2.3 Generation of test cases

Xu et al. suggested the automatic generation of test cases by using high-level Petri nets as finite state test models [99]. The generated test cases can be executed with the selenium test tool.

Törsel et al. and Tung et al. suggested the generation of test cases using UML diagrams for software applications [94] [92]. They described a fully automated approach to generate test cases for functional testing.

2.4 Why prioritise test cases

Rothermel et al. defined the problem of test suite prioritisation in [80]. Given T as a test suite, P is the set of all test suites that are the prioritised orderings of T obtained by permuting the tests of T and F is a function obtained from P to the reals, the problem is to find a permutation, $T' \in P$ such that $(\forall T")(T" \in P)[f(T') \geq f(T")]$.

TCP produces a single sequence of ordered test cases, while the need and opportunity for producing smaller, concurrent test execution sequences have been ignored. Furthermore, a major concern from software developers is that test engineers report critical defects (e.g., defects related to system crash) toward the end of test cycles. This does not give them enough time to fix those defects. In addition, blocking defects need to be fixed earlier in the test cycles in order to execute the blocked test cases.

2.4.1 Why prioritising of test cases is required in case of multi-tier software applications

Most multi-tier applications follow the agile/iterative development process. The new requirements are added very frequently in these applications.

- The customers/end users are always in need of the new functionalities for their routine activities.
- The client is not sure about the requirements, and suggests new features/functionalities very frequently.
- In case of everyday nightly build in some organisations, as the test cases grow at a fast rate (due to addition of new functionalities), sometimes it is not possible to execute all the test cases, so to complete the execution of the important parts of the test suite in a speedy manner, we need to prioritise the test cases.

In industries, prioritisation is very helpful in case of routine nightly builds or the fortnightly builds, there is a need to do the regression test to check out the faults that arise in the already working existing functionalities by changing the features of the existing functionalities or by adding the new functionalities. Sometimes, it takes days to completely execute all the test cases or the execution of test cases cannot be completed within the specified time, the test cases are prioritised according

to the priority of the functionalities. This helps the industries in early detection of faults.

Some key issues related to prioritisation of test cases

Rothermel et al. described techniques for dealing with test execution information to prioritise test cases for regression testing [80] :

- Total coverage of code components.
- Coverage of code components that are not covered in the previous test execution.
- Ability to find faults in the code components covered by functionalities.

They analysed many test suites using these prioritisation techniques and came to the conclusion that every prioritisation technique improved the rate of fault detection of the test suites. In order to reduce the cost of regression testing, software engineers prioritise the test cases in an order such that important test cases are executed earlier in the regression testing process.

2.5 Existing prioritisation techniques

Rothermel et al. first defined the problem of test suite prioritisation [80]. Prioritisation techniques attempt to increase the effectiveness in achieving some specified performance level in software testing. Their technique constructs control flow graphs and uses those graphs to select the test cases related to a modified version of that software application. Rothermel et al. suggested various prioritisation techniques to reorder the test cases such as random, optimal, stmt-total.

There are many regression testing prioritisation techniques available in the literature for prioritising C [24] [22] and Java programs [41] [19] [46].

We discuss below the main prioritisation techniques for software testing that have been proposed in the literature.

Zhang et al. proposes a model that unify the total strategy and additional strategy [101]. The total and additional prioritisation strategies prioritise the test cases based on total numbers of elements covered per test, and the number of additional (not-yet-covered) elements covered per test. Their experimental results have performed better than total and additional strategies.

Di Nardo et al. presents an industrial case study of coverage-based prioritisation techniques on a real complex system with real regression faults [18]. The main aim of their study is to evaluate and compare coverage based prioritisation techniques with and without the use of modification information. The results indicated that the techniques that are based on additional coverage with finer grained coverage criteria enhance fault detection rates.

Arafeen et al. investigated whether clustering test cases incorporating traditional code analysis can improve the effectiveness of test case prioritisation techniques [3]. Their approach is based on a textmining technique using cluster relevant requirements that contain code complexity for each cluster and creates a set of reordered test cases using the requirements priority. Their results indicated that prioritised test cases using requirements-based clustering approaches help in detecting the faults early.

Praphamontripong et al. suggested a solution to the problem of performing integration testing of web applications using mutation analysis [75]. Wu et al. suggested an analysis model for testing of web applications that ensures different levels of quality control of web applications under different situations [98]. Halfond et al. presented a technique for automatically discovering web application interfaces based on a static analysis algorithm [39]. Alshahwan et al. proposed a framework for collection of testability measures during automated testing process of web applications [2]. Mansour et al. proposed a technique

for white box testing of web applications that are developed in .NET environment [63].

Lucca et al. proposed a new unit testing level approach for web applications [16]. Lucca et al. were also one of the first to discuss the challenges of testing web applications as opposed to testing of traditional software [60]. Guo et al. classified the possible faults in web applications [36]. Bruns et al. proposed the acceptance testing process of web applications using the Selenium tool [8]. Ricca et al. investigated the various reported faults in web application testing and abstracted them into a fault model [77]. Ricca et al. proposed a technique for testing real world web applications for automation support for verification and validation activities [78]. Tonella et al. proposed a dynamic analysis technique for testing of web applications [91]. Choudhary et al. suggested a new tool for cross-browser web application testing [13]. Artzi et al. suggested a framework for feedback-directed automated test generation for web applications written in Javascript [4].

Existing prioritisation techniques related to software applications prioritise the test cases when the test cases are a part of a single test suite as there is only one processing queue that selects the test cases to run. Qu et al. describes the prioritisation problem using parallel scenario [76]. They presented their results for a standalone application that was installed on one machine. They tested the Microsoft PowerPoint application as their target application. Their approach was not designed to test the application according to its functional requirements. Hence their approach is not applicable for regression testing of complex software applications.

Haftmann et al. suggested a technique for parallel execution of test cases but their technique involves the testing of only the databases [37].

Chakraborty et al. suggested an approach for parallel execution of test cases by collecting the data from manual test processes and they assumed that manual test execution time and automated test execution time are equal [12]. In case of complex software applications, manual

test execution time and automated test execution time may vary. It is not possible to test the functional requirements with the approach suggested by Chakraborty et al. as they do not consider the functional specifications while extracting the dependency graph.

Haidry et al. presented a family of TCP techniques that prioritises the test suite using dependency information [38]. They proposed a set of new techniques for functional test case prioritisation that are based on the dependencies between test cases. These techniques prioritise tests based on the dependency structure of the test suite. The results indicated that the test suites prioritised by their techniques performed better than random and untreated test suites. Their techniques achieved better APFD than the related techniques in state of art.

Malz et al. suggested a prioritisation approach that collects and evaluate the available information from different test and development tools that were already used and suggested a prioritisation order of the test cases due to that evaluation [62]. Software agents represent software modules and test cases. They have knowledge defined with fuzzy logic. They collected information from many sources and suggested a prioritisation approach using that input information. But our new suggested approaches are not dependent on the information sources. It becomes very difficult to collect data from the different information sources in distributed software applications. Our techniques suggested in this thesis collects the faults raised in the previous test executions and prioritise the test cases based on the severity of those faults.

Rogstad et al. suggested a practical approach for regression testing of database applications [79]. Their technique automatically identifies all the potential faults and prioritises the inspections for early fault detection. They used dynamically generated database triggers to capture data manipulations in the database during test execution of the system under test. They identified the faults by following consecutive executions on different versions of the systems. Their prioritisation approach increased the likelihood of early fault detection. But our approaches

identifies the database changes and executes those tests early in test suite execution. Rogstad et al. used legacy database applications to implement their approach. Their technique is based on the data of an application. However, it is possible to use their approach after our suggested approach as their technique mainly targets database intensive batch jobs.

Khalilian et al. proposed a new prioritisation equation with variable coefficients gained according to the available historical performance data, which acts as a feedback from the previous test sessions [49]. It is not clear from the paper whether we can apply their technique on the distributed software applications.

Carlson et al. suggested a new clustering approach to improve TCP [10]. They implemented new prioritisation techniques that incorporate a clustering approach. Their suggested technique is not specifically designed to detect faults related to database schematic changes. However, we use their approach to evaluate whether we can detect faults related to database schematic changes early. To implement the approach suggested by Carlson et al., we select only database related test cases and group these test cases into clusters and implement it using their suggested prioritisation techniques.

Panigrahi et al. proposed a regression test case prioritisation technique for object-oriented programs [71] [70]. They constructed an intermediate graph model of a program from its source code. The model is updated according to the modifications in the programs. Their model represents the control and data dependencies. The results show that their technique detect faults early as compared to the other related approaches.

A Pair-wise timeaware Test Case Prioritisation (PTCP) technique has been proposed in [72]. PTCP determines whether the test case is effective on the basis of total number of faults that are present in a software. It also considers the number of detected faults and the execution time

of different test cases. It selects the test cases that detect more faults in less time.

Walcott presented a TCP technique that uses a genetic algorithm to reorder test suites [96]. They presented a new ordering of test cases that executes in a given time limit. Their presented technique detects faults based on the derived coverage information. The technique uses both testing time and potential fault detection information to reorder a test suite.

Mei et al. proposed an approach to prioritising test cases that operates on Java programs tested under the JUnit framework in the absence of coverage information [65]. Their technique JUPTA (**JU**nit test case **P**rioritisation **T**echniques operating in the **A**bsence of coverage information) prioritises test cases based on coverage information estimated from static structures rather than gathered through instrumentation and execution. Their results showed that the test suites constructed by their technique detect faults early as compared to the random and untreated test orders.

Kayes et al. presented a new metric for assessing rate of fault dependency detection [48]. They also suggested an algorithm to prioritise test cases based on that metric. They have shown the effectiveness of their approach by comparing it with non-prioritised test cases.

Qu et al. describes the prioritisation problem using parallel scenario [76]. They presented their results for a standalone application that was installed on one machine. They tested the Microsoft PowerPoint application as their target application. Their approach was not designed to test the application according to its functional requirements. Hence their approach is not applicable for regression testing of complex software applications.

Li et al. applied various meta-heuristics for TCP [59]. The additional greedy and 2-optimal algorithms showed the best performance. They considered the Siemens suite programs and the program space, and

evaluated each technique. The results showed that the additional greedy algorithm is the most efficient in general.

Mirarab et al. suggested a new prioritisation approach which incorporates various kind of information into one single model [67]. Their approach is based on probability theory and utilises Bayesian Networks (BN) to incorporate source code changes, software fault-proneness, and test coverage data into a unified model. The results indicate that their approach detect faults early.

Tonella proposed a test case prioritisation using user knowledge through a machine learning algorithm, Case-Based Ranking (CBR) [90]. CBR evokes relative priority information from the user, in the form of pairwise test case comparisons.

A new approach to generate test cases from UML 2.0 activity diagrams and prioritising those test cases is proposed in [27]. This technique uses model information that is enclosed in the activity diagrams. They proposed a method based on coverage of all transitions in the activity diagram. They also selected test data based on the analysis of the branch conditions at the decision node in activity diagrams.

There are many prioritisation techniques available for testing of software applications in literature but none of these papers suggested the automated approach for testing of multi-tier applications. It is possible to automate the approach proposed by Rothermel et al. when the software is written in a single language. This can be done through a static analysis of the entire software and identifying the function modules and their connectivity in terms of the calling sequences. However, when the software consists of different modules written in different languages, it is not possible to perform static analysis of the entire program as static analysis tools are based on the grammar of a particular language. It may be possible to manually construct a control flow graph by checking the connectivity between different modules, but this process is extremely time consuming.

Moreover, software applications are prone to rapid changes and such connectivities need to be established manually every time a module is changed. This is not possible when rapid turn around in testing is required. Hence, it is difficult to base control flow graphs for software applications on the underlying code. Our proposed two-level approach generalises the approach proposed by Rothermel et al. at a functional level [83] [80]. Moreover, this approach also generalises the approach proposed by Sampath et al. [85]. Since all user sessions are sub-graphs of the FDG of a software application, our approach will be able to prioritise test cases for all possible user sessions.

We have categorised some of the prioritisation techniques as follows:

- **Software fault diagnosis**

  Recently in 2011, Alberto et al. suggested a diagnostic prioritisation technique that maximises the improvement of the diagnostic information for every test case [34]. The suggested technique helps in reducing the loss of diagnostic quality to a minimum in the prioritised test suite. Using this technique, tests are selected dynamically on the basis of the actual pass/fail results of earlier executed tests, resulting in a higher diagnostic performance per test.

  They considered cost as a combination of testing cost and debugging cost. Using this new diagnostic prioritisation technique, the tests are selected dynamically on the basis of the actual pass/fail results of the earlier executed tests and this technique results in a higher diagnostic performance per test.

  This technique is based on the online "greedy" prioritisation approach that considers the observed test outcomes to determine the next possible test case. They consider Bayesian approach to obtain the rankings of test cases. The implementation of this prioritisation technique resulted in a slightly reduced APFD. They compared this new technique to the existing prioritisation techniques with respect to fault localisation and failure detection per-

formance and this new TCP approach has shown reductions of up to 60% of overall combined cost of testing and debugging.

- **Execution strategies**

  Baresi et al. prioritised test cases by defining the priority among the tests and different types of execution strategies [6]. Priorities always aim at maximising the effectiveness of tests by postponing the execution of test cases that are less likely to reveal faults. Popular priority schemas are based on [6]:

  - **History-based priority schemas** assign lower priority to the test cases that were recently executed. By doing so all test cases will be re-executed in the long run. This technique works well for frequent releases, e.g., overnight regression testing.
  - **Priority schemas** aim to detect faults to raise the priority of tests that revealed faults in the most recent versions.
  - **Structural priority schemas** give priority to the test cases that use elements that were not recently executed or the test cases that result in high coverage.

- **Black box tests**

  Srikanth et al. suggested a cost effective TCP technique that improves quality of software by considering defect severity [88]. They suggested to improve the fault detection rate of severe faults during the testing of new code and regression testing of existing code . They called this new approach as PORT (Prioritisation of Requirements for Test). PORT prioritises black box tests at the system level when information between requirements, test case, and test failures is maintained by the software development team.

  The PORT algorithm prioritises system level test cases by considering four prioritisation factors: *Customer-assigned priority* measures the significance of a requirement to the customer. The cus-

tomer assigns some value for each requirement that ranges from 1 to 10 where 10 is the highest customer priority requirement. *Requirements volatility (RV)* is based on the time when a requirement has been modified during the development cycle. *Implementation complexity (IC)* tells about the difficulty in measuring the implementation of requirement. *Fault proneness of requirements (FP)* provides the information to the development team to identify the requirements that had customer-reported failures.

The results indicate that PORT prioritisation at the system level improves the fault detection rate of severe faults. Customer priority was identified as the most important prioritisation factor resulting in the improved rate of fault detection.

- **Function coverage**

  Elbaum et al. proposed a cost effective prioritisation technique that improves fault detection rates [25]. They applied different prioritisation techniques to different programs and identified a technique that is cost effective for early detection of faults. Their method is based on total function coverage prioritisation, which orders the test cases according to the number of functions they cover. If multiple test cases cover the same number of functions, then this technique orders them randomly. They found that the performance of this technique varied according to the program attributes, change attributes and test suite characteristics.

- **Coverage of statements**

  Srivastava et al. presented a technique for prioritisation of test cases on the basis of coverage of statements, using change information and feedback approach [89]. This technique is used for regression testing during development and is different from other techniques because of the calculation of flow graphs and coverage using binaries. While Srivastava and co-workers do not justify their use of binaries in their testing process, they may have used this form of testing to identify the vulnerabilities in the pre-release

software, which is usually shipped in binary format [58]. They described the applicability of this technique to several large software systems at Microsoft. Their test prioritisation algorithm is based on statement-level differences between the compiled binary codes of the previous versions of a system and its current version for testing. The prioritisation algorithm tries to pick a test from the remaining tests that will cover the maximum number of the remaining impacted statements, new or modified.

- **Minimisation techniques**

  Wong et al. prioritised test cases using the criterion of extending cost per additional coverage [97]. They proposed a technique that is a combination of minimisation technique and a prioritisation technique to determine which regression tests should be re-run. They first find the minimal subset in terms of the number of test cases that preserves the same test coverage as the original test set. Then they sort the test cases in order of increasing cost per increasing coverage and then select the top n test cases for revalidation. The results suggest that this technique can provide software engineers with cost-effective alternatives to help conduct quick regression testing under budget constraints and time pressure.

Prioritisation techniques specifically for web application testing

There are many prioritisation techniques available in literature for software applications but very few of these techniques are suitable for web application testing [85].

Though prioritisation of test cases is very important for supporting rapid development and testing cycles of web applications, there are very few available techniques for prioritisation specifically developed

for web applications. Several researchers have recently proposed testing approaches for web applications. We present a quick overview below.

1. **Combined model for prioritisation of test cases for GUI and web applications**

   Bryce et al. suggested a combined model for prioritisation of test cases for both GUI (Graphical User Interface) and web applications [9]. They suggested some common characteristics that helped them in developing a single model for testing event-driven systems. Web applications exhibit many characteristics of GUI, both in terms of traditional and distributed applications. The absence of a combined model for GUI and web applications has prevented the development of common testing techniques and algorithms that may be used to test both of these two classes of applications.

   All event driven software take sequences of events (e.g., messages, mouse-clicks) as input and change their state and produce an output (e.g., events, system calls, text messages). Some examples of event driven software are web applications, GUI, network protocols, device drivers, and embedded software.

   The main emphasis in this technique is to cover a maximum number of criteria or elements like windows and parameters in the selected test cases. The model also presented the development of a common set of metrics that may be used to evaluate the test results of this type of applications. They developed and empirically evaluated several prioritisation criteria and applied them to their four stand-alone GUIs and three web-based applications using seeded faults.

   The proposed prioritisation techniques achieve different rates of fault detection for separate levels of test suite execution for different applications. The drawbacks of this approach is that they have assumed a uniform cost of running (processor time) and monitoring (human time) for every test case and secondly they assumed

that each fault contributes to the overall cost. In practice, the scenarios of having uniform costs for running and monitoring every test case are simplistic.

2. **Prioritising user session based test cases for web application testing**

Sampath et al. suggested a test suite prioritisation technique for web applications using user session based test cases [85]. They considered test lengths, frequency of appearance of request sequences, systematic coverage of parameter values and their interactions. It is not clear whether this technique can be used to detect faults related to database schematic changes early. If there are changes in the database, it is difficult to test web applications with respect to the changes in the database within a very short period of time. Any modifications in the database schema give rise to many blocker and critical faults. Blocker faults prevent the application under test to get loaded or to proceed for testing. Critical faults affect the major functionality of the application and need to be fixed immediately. If these faults are identified at the end of test suite execution, it gives very little time for the software developers to fix these faults.

They considered the frequency of user requests and interaction of parameter values in the requests. Their test cases were designed according to user sessions based on a series of HTTP requests consisting of base requests and name-value pairs. A base request is a HTTP request to access both static and dynamic content in web pages. They used the entire test suite for execution but they ordered the test cases on the basis of user session requests to detect the faults early in test suite execution.

2.5.1 Minimisation of test suite and test cases

A large number of test cases are usually added due to the coverage of program requirements which results in redundancy of test cases and wastage of the additional resources [86] [47] . The aim of test suite reduction is to produce an optimised test suite that is smaller in size than the original test suite but satisfies all the requirements of the original test suite. Test cases are also minimised to locate faults faster and to reduce the execution time [54].

2.5.2 Selection of test cases

Selection of test cases reduces the amount of time required to retest a modified application by selecting a subset of the test cases from an existing test suite [81]. Some regression test selection techniques select tests on the basis of the collected information from program specifications and some select tests on the basis of the information about the program and the modified version.

Rothermel et al. suggested a regression test selection technique for object oriented software like C++ [83] [80] and Harrold et al. suggested a test selection technique for software written in Java [41]. The technique suggested by Rothermel et al. constructs control flow graphs for software written in an object oriented language like C++, and the graph is used to select test cases for modified code from the original test suite [83]. The test suites in this technique cover all components and reveal faults in them. This technique includes both intra-procedural and inter-procedural control flow graphs.

Rothermel et al. reduced the amount of time required to retest a modified software program by selecting a subset of test cases from an existing test suite [81] [82]. Some regression test selection techniques select tests on the basis of the collected information from program specifications and some select tests on the basis of the information about the

program code and the modified version. The effectiveness of the test case selection strategy comprise three different aspects of software testing: effectiveness of fault detection, cost of fault detection and classes of faults detected. Test oracle is the method used for checking whether the system under test has behaved correctly on a particular execution.

Nanda et al. suggested a regression *test selection technique* (RTS) that performs accurate test selection in the presence of changes to configuration files and database files [68]. Their suggested technique computes code-differencing components to compute differences between two versions of a property file and a database file. They compared these two models to identify the affected model entities. The traceability analysis associates the test cases with the model entities to represent code, property files or databases. The selected tests cover the affected model entities. They selected the entire files and database related to the modifications. But the changes in database also affects code components. Our techniques will identify the code components affected by the changes in database. Our techniques will select only the affected columns/tables from the database and executes those test cases early in test suite execution.

2.6 Functionality and Functional modules

A functionality refers to user actions such as keyboard and mouse events required to navigate through software applications [86] [17]. UML sequence diagrams provide information about functionality and the interaction among different objects in a software application [11]. Functionality is defined in the specification documents and is provided by the user interface [42].

A functional module is a collection of different functions (at the code level) to enable the functionality to behave according to the specifications. Any modification in the functionality of a software application affects other working functionalities. To test for faults that arise in the

already tested functionalities due to changes in other functionalities, there is a need to perform regression testing. Regression testing is helpful in performing the compatibility analysis of new features with the previous ones and can be performed for any kind of program including web based applications or system applications.

2.7 Evaluation metrics

Rothermel et al. presented the APFD (Average Percentage of Faults Detected) metric for measuring fault detection rates of test suites in a given order [80]. APFD provides us with a value between 0 and 100 that indicates the detection of faults. A value closer to 100 implies faster (better) fault detection rates.

APFD can be calculated using the following formula:

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + ... + TF_{\mathrm{n}}}{mn} + \frac{1}{2n} \qquad (2.1)$$

TF_i is the position of first test in T that exposes fault i
n= no. of faults
m= no. of test cases

Informally, APFD measures the area under the curve that is plotted by the percentage of faults detected by prioritised test case order and the test suite fraction.

We can modify APFD on the basis of types of faults. We compared our new prioritisation approach with the APFD metric suggested in the literature. The APFD metric is able to identify which prioritisation ordering of test cases detect faults early.

Elbaum et al. presented a new metric for assessing the rate of fault detection of prioritised test cases [23]. This metric uses varying test case and fault costs. They named this new metric as "cost-cognizant" metric $APFD_{\mathrm{c}}$.

$APFD_c$ can be calculated using the following formula:

$$APFD_c = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TFi}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{2.2}$$

2.8 Motivation

Many strategies have been proposed for prioritising C [24] [22] and Java programs [41] [19] [46]. Sampath et al. suggested prioritisation techniques for using user-session based test cases [85]. Korel et al. presented an analytical framework for evaluation of the test prioritisation methods [51]. They suggested this method utilising the state-based model of the system under test.

It is possible to automate the approach proposed by Rothermel et al. when the software is written in some particular language as their technique is dependent upon the programming language [81] [80]. This can be done through a static analysis of the entire software and identifying the function modules and their connectivity in terms of the calling sequences. However, when a software consists of different modules written in different languages, it is not possible to do a static analysis of the entire program as static analysis tools are based on the grammar of a particular language. It may be possible to manually construct a control flow graph by checking the connectivity between the different modules, but this process is extremely time consuming. Moreover, multi tier applications are prone to rapid changes and such connectivities need to be established manually every time a module is changed. This is not possible when rapid turn around in testing is very important. Hence, any control flow graph for multi tier applications cannot be based on the underlying code.

Our new suggested approaches constructs control flow graphs based on the functional dependency of the different modules in a software application. Furthermore, test suites can be constructed to cover functional

modules. Hence our approach will generalise the approach proposed by Rothermel [81] [80]. This approach generalises the approach proposed by Sampath et al. [85]. Since all user sessions are sub-graphs of the FDG of a software application, our approach will be able to prioritise test cases for all possible user sessions. Our approach is useful to detect faults in software applications when different application modules are developed using different programming languages.

Sampath et al. suggested prioritisation techniques using user-session based test cases [85]. It is not possible to detect the faults due to database modifications earlier by applying this technique. If there are modifications in database, it is difficult to test software applications with respect to the modifications in a short period of time.

The functional modules in multi-tier applications like software applications are written in many different programming languages like JavaScript, AJAX an acronym for "Asynchronous JavaScript and XML [95], PHP, ASP, JSP, Java servlets, and HTML. A software application may also use other external application interfaces for database connectivity and these interfaces may be written in other languages. One of our main aims is to suggest a new framework to automate the process of testing of complex software applications as a manual testing process is too slow for supporting the rapid turn around time. Though our approach is motivated by the approach proposed by Rothermel [81] [80].

None of the papers in the literature, suggested fully automated prioritisation approach to detect faults due to database changes early in regression testing cycles for software applications. The modification of key attributes in the tables in a database and their interaction with various code components result in many runtime faults.

None of these papers suggested an approach to select and prioritise the test cases related to modified source code. This thesis suggests a new approach to select the test cases using a bipartite graph and then prioritise these selected test cases. This new technique helps in early detection of faults related to modified source code. This novel approach

is based on bipartite graphs for minimising the time needed for test case execution.

None of these papers suggested fully automated prioritisation approach to detect faults due to database changes early in regression testing cycles for software applications. The modification of key attributes in the tables in a database and their interaction with various code components result in many runtime faults. Our suggested approach captures the entire current database schema and the modified database schema of a software application. Our approach automatically identifies the new tables/key attributes in a database and automatically detects the changed data-types. It also detects the key attributes that are present in the code components but are missing in the database. The test cases related to modifications in the database are prioritised in the test suite execution. Our proposed approach will automatically reschedule test cases in order to identify the faults related to database changes earlier in the test suite execution.

None of these papers in the literature suggested an automated approach to detect faults due to database schematic changes early in regression testing cycles for multi-tiered applications. This thesis presents a new automated approach for selecting and prioritising the test cases related to database schematic changes. This approach extract the test cases related to database from the entire test suite by matching the key attributes used in the database with the source code of the software application and then these key attributes are matched with the test suite. The test cases are prioritised on the basis of modifications of tables/key attributes in a database and also by counting the appearance of key attributes in a database. The test cases are further prioritised on the basis of the occurrence of these key attributes in the source code of a software application. This approach helps in identifying the key attributes that are present in the database but absent in the source code and vice versa.

None of these papers suggested an automated parallel prioritisation approach to test software applications. Existing prioritisation techniques related to software applications prioritise the test cases when the test cases are a part of a single test suite as there is only one processing queue that selects the test cases to run. This thesis presents a new technique to execute the tests on different machines in parallel. Even though prioritisation mitigates some of the drawbacks of executing a complete set of test cases, execution of test cases on a single machine may not achieve the rapid testing criterion of large software applications. Also, most organisations can afford to deploy multiple machines for testing. Hence, a possible way of expediting the speed of regression testing is to run disjoint parts of the same test suite on multiple machines.

In this thesis, we investigate the parallel execution of prioritised test cases. Even though prioritisation mitigates some of the drawbacks of executing a complete set of test cases, execution of test cases on a single machine may not achieve the rapid testing criterion of large software applications. Also, most organisations can afford to deploy multiple machines for testing. Hence, a possible way of expediting the speed of regression testing is to run disjoint parts of the same test suite on multiple machines. In this thesis, we investigate the parallel execution of prioritised test cases. We first construct a *functionality dependency graph* (FDG) of the entire software application from its UML specification. A node in the FDG is a unique functionality and a directed edge from node m to node n indicates the functional dependency of node n on node m. This technique partitions the entire test suite into test sets such that each test set is associated with a unique functional module in the FDG. This thesis proposed a new automated approach for execution of prioritised test suite by distributing it into several test sets. The execution of parallel prioritised test sets reduces the execution time and detects faults early. This technique involves two major challenges: partitioning and ordering. In partitioning a test suite, we have to decide which test cases need to be executed on which machine

and ordering is used to decide in which order the subsets of the test cases are executed in each machine.

While there are many prioritisation techniques available for software testing, very few are available for distributed software testing and development. None of these papers suggested an approach to prioritise test cases for distributed applications to resolve faults early. This thesis suggests a new approach to verify the already resolved faults as the resolved faults in the previous version of the software may give rise to new faults or they may appear again in the new version of the same software. This technique calculates the severity of such faults using our suggested metric. It prioritises the test cases according to the scores obtained from this metric and executes these test cases early to detect the faults.

While there are many possible goals of prioritisation, this thesis focuses on the goal of reduction of test suite execution time and early detection of faults. As prioritisation techniques available for software testing, very few are available for testing of multi-tier applications. This thesis suggests new techniques to detect faults early in multi-tier applications.

2.9 Contributions

The main contributions of this thesis is a new automated framework to detect faults in software applications. The new framework uses the static analysis as well as the dynamic analysis techniques. This framework is different from the current approaches as the new framework suggests the automated techniques to detect faults in multi-tier software applications but the current approaches detect faults using static analysis and are dependent on the syntax and semantics of a language. This work also introduces the new techniques to detect faults early related to the modifications in the database of a multi-tier software application or in the source code that is connected to database. The

techniques suggested in this thesis detects modified source code in a software application and test that particular source code.

This thesis expand upon the prioritisation of test cases for testing of software applications. This provides a new technique using two-level prioritisation approach for prioritisation of software applications based on the work of Rothermel et al. [24]. This new technique detects the new/modified functional modules in software applications. This technique prioritises the test cases using FDG *Functional Dependency Graphs* and CFG *Control Flow Graphs* and suggest new strategies to prioritise test cases related to new/modified functionalities. This technique helps the software engineers to detect more faults earlier in test suite execution.

The modifications in a program may result in instability of existing test cases. This is due to the removal of the components. The test cases that point to the removed components are no longer required to test the new version of a software application. This thesis suggests a novel approach based on a bipartite graph for minimising the test suite execution time. In this thesis, we present a new approach for detecting accuracy and redundancy in test cases using a bipartite graph. Any program consists of two kinds of code components: system-defined components that are defined from the libraries of that programming language and the other are user-defined components, that are defined by the programmer. We extract the user-defined components from a program. We match the test cases with the user-defined components. The bipartite graph is used to perform the mapping between the test cases and the components of a program. We select the minimal set of test cases using this bipartite graph. It helps in determining the redundant test cases and also helps in selecting the test cases that are required for the testing of current version of a software application.

This thesis introduces a new technique that focus mainly on the database-specific faults (i.e., *schema faults*). This describes a new automated approach for early detection of faults related to database schematic

changes using automated mapping techniques between the source code and test cases. This technique selects test cases related to database from the entire test suite. This technique uses the new proposed prioritisation strategies to prioritise the test cases related to a back-end database.

This thesis provides a framework to automatically detect modifications in the database and identify the relationship between the database tables and code components. This new framework prioritise the test cases related to new/modified key attributes/tables in a database.

This thesis suggests an approach that captures the entire current database schema and the modified database schema of a software application. This approach automatically identifies the new tables/key attributes in a database and automatically detects the changed data-types. It also detects the key attributes that are present in the code components but are missing in the database. The test cases related to modifications in the database are prioritised in the test suite execution. Our proposed approach will automatically reschedule test cases in order to identify the faults related to database changes earlier in the test suite execution.

This thesis investigate the parallel execution of prioritised test cases. This technique uses *functionality dependency graph* (FDG). We identify a subgraph S of the FDG for prioritisation. This subgraph consists of the nodes that have been modified after the previous regression testing cycle and nodes that are dependent on these modified nodes. We assign priorities to the nodes of S. The test sets are executed according to these priorities when a single machine is used for testing. For multiple machines, we sort the nodes of S in priority order and allocate nodes from different priority groups approximately evenly among the available machines. A node in the FDG is a unique functionality and a directed edge from node m to node n indicates the functional dependency of node n on node m. The test set is associated with a unique functional module in the FDG and then identified a subgraph of the FDG for prioritisation. This subgraph consists of the nodes that have been modified after the

previous regression testing cycle and nodes that are dependent on these modified nodes. We further prioritise the execution of test sets in each machine by using code level *control flow graphs* (CFG). We execute the test sets allocated to each machine simultaneously in parallel. Finally, we collect the test results in a single machine. The test sets are executed according to these prioritises when a single machine is used for testing. The test sets are allocated to each machine and executed simultaneously in parallel. The suggested approach is quite general and can be used if it is possible to construct a FDG for a software system.

This thesis presents a new approach for testing of distributed software applications. This new approach partitions the entire software application into different partitions and uses a new metric to calculate the severity of the faults according to one partition slice. The score obtained from this new metric is used to prioritise the test cases in distributed software applications.

2.10 Research design

In order to reduce the software testing cost and to improve the effectiveness of regression testing, various approaches like test case prioritisation, test selection and test suite minimisation have been proposed in the literature. There are no techniques to automatically prioritise and select the test cases for multi-tier applications. We suggest a new functionality dependency analysis approach for test case prioritisation of software applications.

The *functionality dependency graph* is used to describe the relationships between functionalities in software applications. Firstly, we will discuss the functionality dependency graph with a modified functionality i.e., when a functional module or a node in the graph has been modified. A functional module is a collection of different functions (at the code level) to enable the functionality behaves according to the specifications. We say that a functional module f_i *invokes* a functional module f_j

(denoted by a directed arrow from node f_i to f_j) if functionality f_j may follow functionality f_i during a user session, however, functionality f_j cannot be accessed by a user without first accessing functionality f_i. The functionality dependency graph captures all possible invocation of functionalities by a user, however, a single user session may only access a sub-graph of the functionality dependency graph. Each functional module in a dependency graph is represented by a single node. Note that a functional module may consist of several functions or methods at the code level. However, different functional modules may be written in different languages.

To perform regression testing, we will propose a two level approach using functionality dependency graph and new/existing prioritisation techniques. When some of the functionality in a complex software application is modified, our technique will consist of the following two steps:

1. We will first identify the sub-graph of the dependency graph that is affected by the modifications. In other words, we will identify the functional modules that need to be tested in the form of a sub-graph. This is the selection step for prioritisation of test cases.

2. We will next test the functional modules in the selected sub-graph using new or existing prioritisation techniques.

A software application consists of many functional modules. We assume that the specifications of functionality F has been changed. Now we have two different cases:

- Functional module F invokes some functional modules.

- Functional module F is invoked by some functional modules.

If the functional module F invokes other functional modules that have not changed after the previous testing cycles, there is no need to test these modules. Moreover, these functional modules (that are invoked by F) should be invoked correctly from F. Hence, these functional modules are locked as per specifications.

But the functional modules that are directly or indirectly invoking functional module F may get affected. These functional modules get affected due to the following reasons:

- If the functional module F returns a different functionality, then it will affect the functional modules that are invoking functional module F.

- If the internal source code is changed in terms of computation or functionality changes but it is still providing the same functionality to the modules that are invoking it, then it may or may not affect the other functional modules.

Our technique will automatically construct functionality dependency graphs by identifying the functionalities in software application as per specifications.

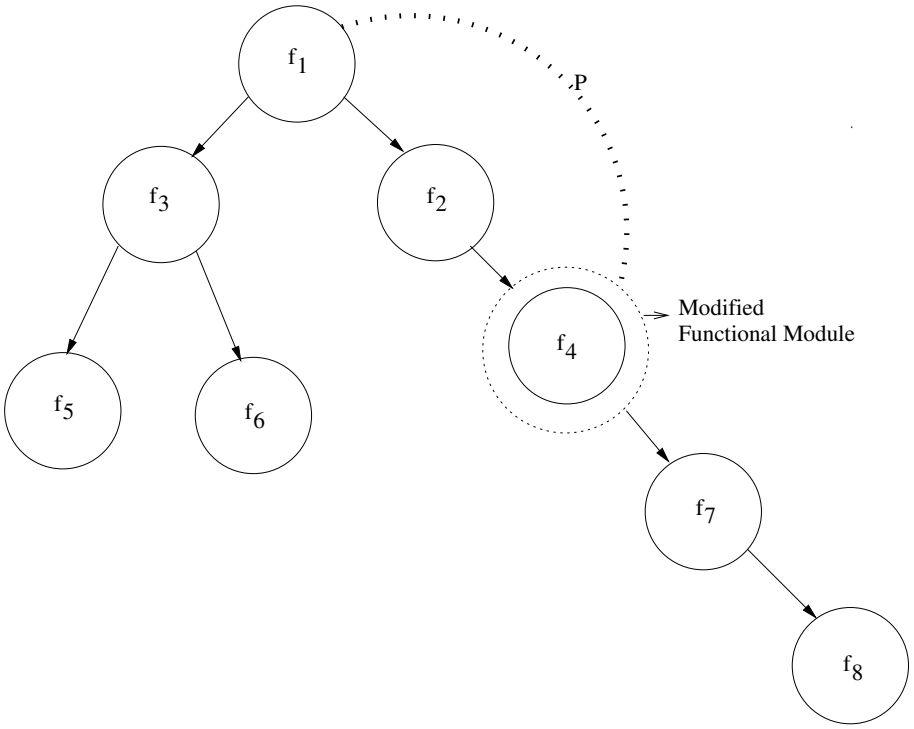

Figure 2.1: Functionality dependency graph

We illustrate our proposed technique using the dependency graph in Figure 2.1. We assume that functional module f_4 has been modified in a regression testing scenario.

We need to investigate which functional modules in the dependency graph get affected due to modifications in f_4.

1. The functional modules along the path P from the root of the dependency graph to f_4 are directly affected. These functional modules are directly or indirectly invoking f_4. The return values from f_4 may detect new faults in these functional modules. For example, the functional modules f_1 and f_2 in Figure 2.1 are affected in this way.

2. The functional modules that are invoked from the functional modules along P may also be affected. These modules include f_3, f_5 and f_6.

We need to do the dependency analysis of functionalities for determining this kind of dependencies (for case 2 above). After doing this analysis, we will consider only those functional modules that are affected by changes in functionality in module f_4 and get a sub-graph like the following Figure 2.2.

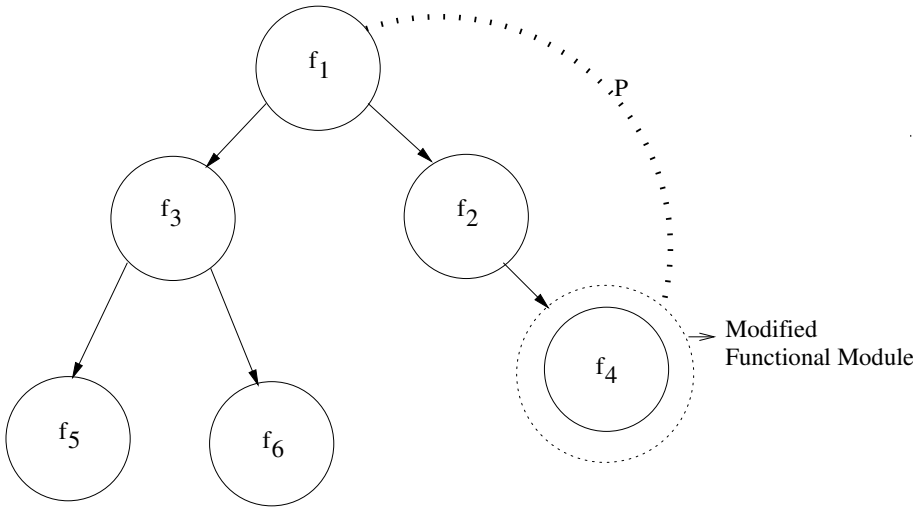

Figure 2.2: Optimised functionality dependency graph

Assuming that f_3, f_5 and f_6 are affected due to modifications in f_4, we can isolate the sub-graph (as shown in Figure 2.2) that needs to be covered during regression testing. Functional modules f_7 and f_8 are not connected to any node outside the sub-graph rooted at f_7. Note that the functional modules that are invoked by f_4 (either directly or indirectly) need not be tested again as this invocation cannot detect new faults in these functional modules.

The case when a new functional module is introduced in the functional dependency graph can be handled in a similar way. Firstly, we will discover the new functional module and then we will follow the same 2 cases as discussed above to detect the functional modules that are affected by integrating this new functional module.

We will develop a pipeline for isolating the affected sub-graphs in a dependency graph during regression testing. We will first generate the functionality dependency graph for the functionalities as per specifications. At every stage of regression testing, we will identify the new/-modified functional modules and identify the sub-graph that is affected by the modifications. We will perform dependency analysis to identify the functional modules that are indirectly affected by new/modified functional modules. After identifying the affected functional modules, we will generate a new sub-graph by removing the non-dependent functional modules.

The new sub-graph generated by removing non-dependent functional modules will help in reducing execution time of test cases as we will execute only the selected test cases. These selected test cases will only test those functionalities that are impacted by the modified functional modules. We will apply new/existing prioritisation techniques on the generated sub-graph to detect faults early in a test suite execution.

2.11 Research stages

There are four stages of developing a new solution for automated detection of faults.

Stage 1 Detection of faults due to modifications in source code.

A set of new techniques are suggested to detect the modified source code and to identify the impact of these modifications in a source code.

In this stage, there is a further investigation about the selection of minimal subset of test cases that are required to test a particular source code. New approaches are suggested to detect the modified source code. Suggestion of the new selection and prioritisation approaches to test the modified source code using bipartite graphs.

Stage 2 Detection of faults due to modifications in a database.

A set of new techniques are suggested to detect the new/modified tables in a database. New techniques are suggested for selection and prioritisation of test cases related to database modifications.

Stage 3 Parallel execution of test cases.

In this stage, new techniques are suggested to distribute a set of test cases among different machines and execute all these test cases in parallel. These techniques helps in early detection of faults by executing test cases in a minimum time period.

Stage 4 Novel metric for prioritisation of test cases in distributed software development and testing.

New metric is suggested to prioritise the test cases when the software is developed using distributed software development process. This technique is used to detect faults earlier in large software applications.

Chapter 3

Fault detection due to modifications in source code

We can see from our review of the existing literature that there are very few systematic studies of prioritisation of test cases for complex software applications consisting of many different functionalities. Different functionalities in large multi-tier applications are usually written using multiple languages. The techniques for prioritisation of test cases that have been proposed in the literature are applicable mainly for software applications written in one programming language.

Most of the proposed techniques cover whole functionality during regression testing. Software applications usually have a rapid turn-around time in terms of development and testing. Hence there is a need to prioritise test cases for regression testing of large software applications. We will focus on these aspects of regression testing of large software applications in this chapter.

In this chapter, we suggest a new automated framework to detect faults in the large multi-tier applications. We suggest this new framework using Functional Dependency Graph (FDG) and Interprocedural Control Flow Graphs (ICG). FDG is based on the functional specifications of a software application and ICG is based on the underlying program code.

We identify the modifications in source code using ICG and FDG. We select test cases that are related to the modified source code and prioritise them. We prioritise the test cases using our new prioritisation strategies. We compare these prioritisation strategies with the strategies that are described in Chapter 2 and evaluate them on the basis of the results obtained from the comparison of these strategies. This prioritisation framework allows to detect faults early due to modifications in source code.

3.1 Functionality dependency graph

Functionality refers to user actions such as keyboard and mouse events required to navigate through software applications [86] [17]. UML sequence diagrams provide information about functionality and the interaction among different objects in a software application [11]. Functionality is defined in the specification documents and is provided by the user interface [42]. A functional module is a collection of different functions (at the code level) to enable the functionality to behave according to the specifications.

A Functionality dependency graph *(FDG)* [103] [87] [28] is a directed graph that is used to describe the relationship between functionalities in software applications.

For an FDG $G = \{V,E\}$, the node set V is the set of functionalities in a software application and E is the set of directed edges that represent the dependency relationship among the functionalities. A directed edge from $m \in V$ to $n \in V$ indicates the functional dependency of the module n on module m. A test suite consists of test cases that are required to test a software application. The test suite is called *bookstore* and contains various test sets. These test sets refer to the functional modules in the FDG. The FDG provides us with a calling relationship among the functionalities in any application.

A possible approach of constructing the FDG is from the UML sequence diagram of the software application. This can be done by identifying the functionalities in a software application from the UML Sequence diagram of that software application. This can be used to capture various functional requirements and interactions between these requirements using UML. But depending upon the technologies, other methods of generation could be applied.

3.2 Prioritisation framework

In order to reduce the software testing cost and to improve the effectiveness of regression testing, various approaches like test case prioritisation, test selection and test suite minimisation have been proposed. We use a functionality dependency analysis approach for test case prioritisation of software applications.

To perform regression testing, we propose a two level approach. We use a FDG that captures the functional dependencies among the functional modules in a software application. We use an *inter-procedural control graph* (ICG) for each of the functional modules in the FDG. We construct ICGs within each functional module of the FDG using the technique proposed by Rothermel et al. [83] [80].

An example FDG in Figure 3.1 shows functional modules in a software application. Each functional module in an FDG contains functional sub-modules. We have generated the ICG for each functional module in the FDG. We have partitioned the entire test suite into different test sets related to the functionality. Each functional module in the FDG is tied to a test set and the sub-modules in the corresponding ICG are tied to the test cases contained in the test set. The FDG is used to prioritise the test sets and the ICG within a functional module of the FDG is used to prioritise the test cases within that test set.

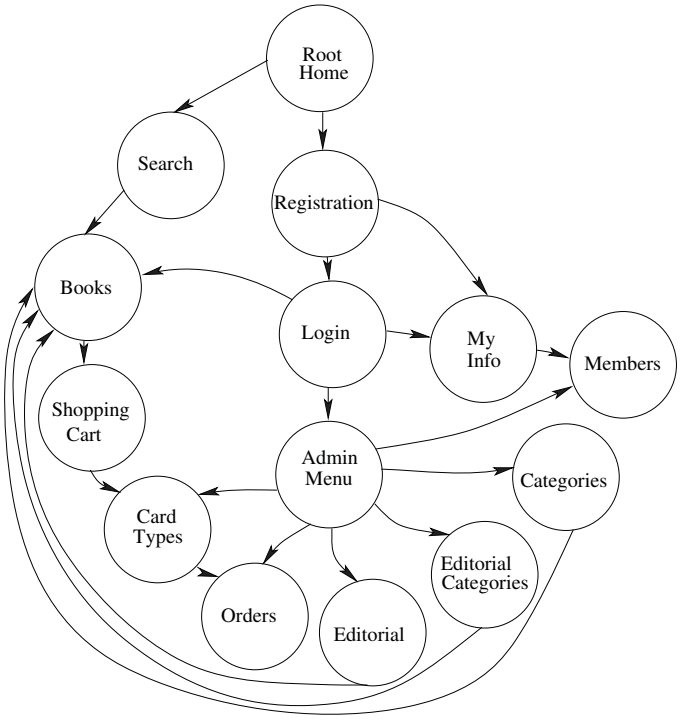

Figure 3.1: FDG for *Online bookstore* application

Suppose an application A has m functionalities $f_1, f_2, \ldots, f_m$. Hence the FDG for this application has m nodes and these functionalities have m corresponding test sets denoted by $TA = \{s_1, s_2, \ldots, s_m\}$. We assume for simplicity that each functionality has n sub-modules at the code level, though the number of sub-modules in a functional module will in general vary. Hence each test set consists of n test cases, each test case covers a functional sub-module (positive and negative) in the corresponding ICG and the set of test cases corresponding to the test set $s_i, 1 \leq i \leq m$, are denoted by $TS_i = \{t_{i1}, t_{i2}, \ldots, t_{in}\}$. We identify the test case $t_{ij}, 1 \leq j \leq n$, with the j-th functional sub-module for the ICG of $f_i \in FDG$.

In the next section, we will discuss the FDG with modified functionality i.e., when a functional module or a node in FDG has been modified. We say that a functional module f_i *invokes* a functional module f_j (denoted by a directed arrow from node f_i to f_j) if functionality f_j may follow

functionality f_i during a user session. However, functionality f_j cannot be accessed by a user without first accessing functionality f_i. The FDG captures all possible invocation of functionalities by a user, however, a single user session may only access a subgraph of the FDG.

3.3 Prioritisation heuristics

A software application consists of many functional modules. We identify the new/modified functionalities and separate them as new/modified functionalities in the FDG (Figure 3.2). We first discuss the prioritisation of the test sets for the FDG. We will next discuss the prioritisation of the test cases for the ICG within each module of the FDG.

3.3.1 Prioritisation of test sets for FDG

Our technique for the prioritisation of the test sets for the FDG consists of the following two steps:

- Identification of the nodes of the FDG that are affected by the modifications. Here we identify the nodes (or modules) of the FDG that need to be tested in the form of a subgraph of the FDG. This is the selection step.
- Prioritisation of the test sets related to the nodes of the selected subgraph of the FDG. This is the prioritisation step.

We assume that the specifications of a node $f \in FDG$ have changed. f is connected to the other nodes in the FDG in two different ways:

- f invokes other nodes in the FDG.
- f is invoked by other nodes in the FDG.

We make two assumptions on the faults in the nodes of the FDG:

- New/modified nodes may introduce new faults in the software application.

- Nodes that are invoked by a new/modified node and that have not changed after the previous testing cycle are assumed to be free of faults.

We note that both of these assumptions are realistic and reasonable, as the chance of introduction of new faults is the highest in new or modified nodes of the FDG. Also, nodes that have not changed, but are invoked by new or modified nodes remain fault free after the previous cycle of testing. For example, the node "Admin Menu" in Figure 3.2 is invoked by the modified node "Login". We assume that "Admin Menu" will be invoked correctly from "Login" even when the specification of "Login" has changed. Moreover, "Admin Menu" was fault-free after the previous testing cycle and according to our assumption, it remains fault-free for the purpose of the current testing cycle. Hence, we assign highest priorities to the test sets for the new/modified nodes and least priorities for the test sets that are invoked by the new/modified nodes. The other nodes are assigned intermediate priorities as discussed below.

We say a node $f_i \in FDG$ *directly* invokes a node $f_j \in FDG$ if there is a directed edge from f_i to f_j. Similarly, a node $f_i \in FDG$ *indirectly* invokes a node $f_j \in FDG$ if there is a directed path from f_i to f_j. When a node $f_i \in FDG$ is new or modified, the nodes that directly or indirectly invokes f_i may or may not be affected for the following reasons:

- If f_i returns a different functionality, then it will affect the nodes that invoke f_i.

- If the source code of f_i is changed in terms of computation or functionality changes, but f_i is still providing the same functionality to the nodes that are invoking it, then it may or may not affect the invoking nodes.

We illustrate our proposed technique using a FDG (Figure 3.2). We assume that the functional modules with dotted circles have been modified.

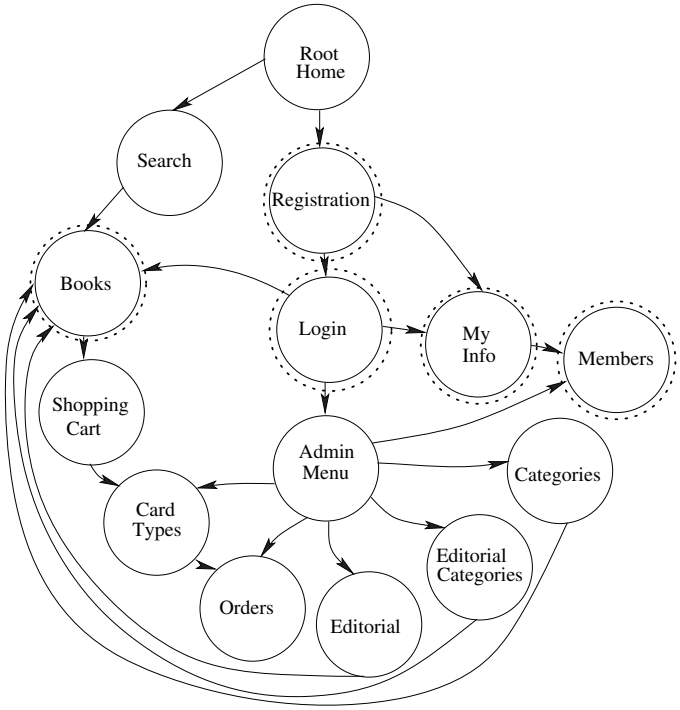

Figure 3.2: The functionality graph with modified functionalities

We assign priorities to the nodes of the FDG in the following way:

1. A newly introduced node in the FDG is given the highest priority. If there are multiple newly introduced nodes, they are assigned the highest priorities in an arbitrary order.

2. The modified nodes in the FDG are given the next lower priorities. The details of these priority assignments vary depending on the prioritisation scheme used as we explain in Section 3.4.

3. The next lower priorities are assigned to the nodes that directly or indirectly invoke the modified or newly introduced nodes. A higher priority among these nodes is assigned to a node that is closer (in terms of path length) to a newly introduced or modified node.

4. All other nodes (except the nodes that are invoked by the modified or newly introduced nodes) are assigned the next lower priority in an arbitrary order.

5. The nodes that are invoked either directly or indirectly by the modified or newly introduced nodes are assigned the least priorities. These nodes may not even be tested in a regression testing scenario. Hence these nodes are not part of the *selected subgraph* for testing.

3.3.2 Prioritisation of test cases for ICGs

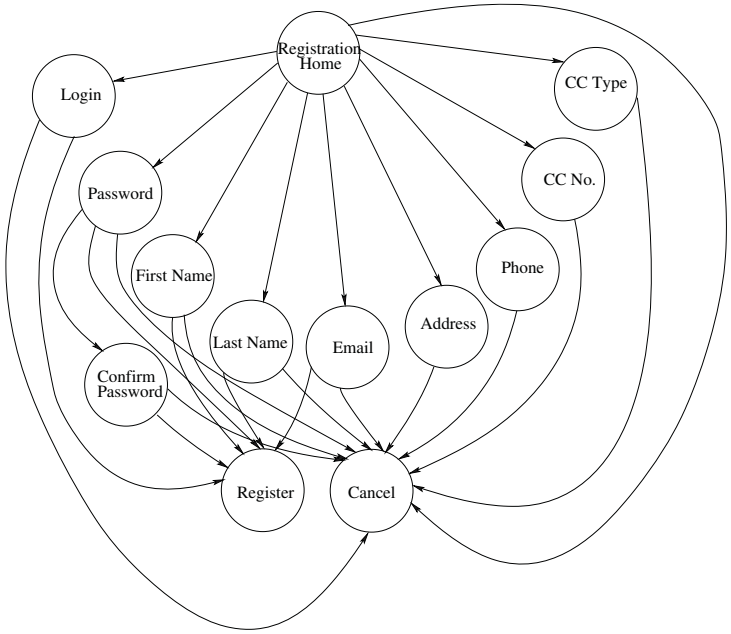

Figure 3.3: Inter-procedural Control Flow Graph (ICG) for the *Registration* node in FDG of *Online bookstore* application

This prioritisation deals with the inner level prioritisation of test cases for the ICGs for each of the nodes in the FDG within our two-level prioritisation approach. As explained before, the inner prioritisation of test cases is based on the inter-procedural control flow graphs (ICG). We show the modified functional sub-modules for an ICG of a node

in the FDG for the *Online bookstore* application in Figure 3.3 and 3.4. The prioritisation strategy of the test cases for each ICG of the selected nodes of the FDG is same.

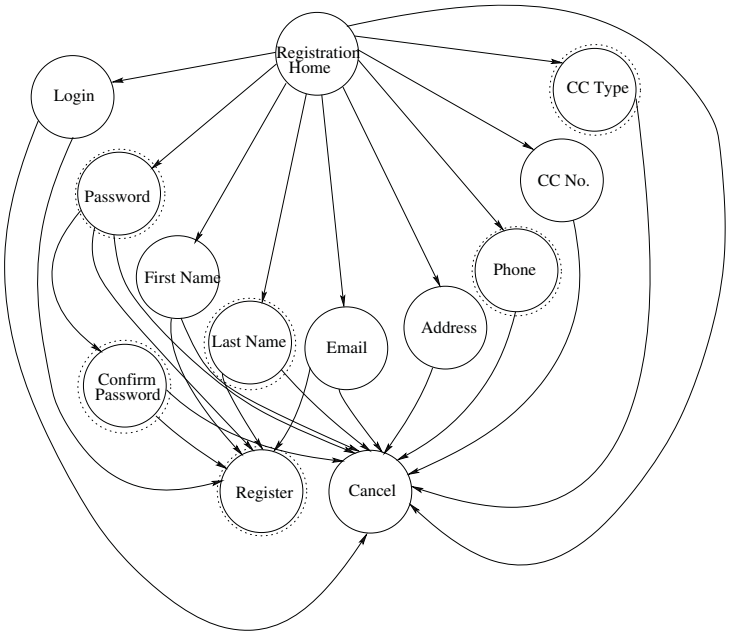

Figure 3.4: Control flow graph - Registration (Modified functional sub-module)

Assume that two new or modified functional sub-modules (nodes in the ICG) e_p and e_q for a node $f_k \in FDG$ have the associated test cases t_{kp} and t_{kq}. The *degree of modification* of a node in the ICG is determined by the changes in the lines of code. If node $e_p \in ICG$ has more changes than another node e_q, we say that e_p has a higher degree of modification. The prioritisation scheme for the sub-modules in an ICG is as follows:

- A new node in an ICG is assigned the highest priority. If there are multiple sub-modules, they are assigned the highest priorities in an arbitrary order.
- The next lower priorities are assigned to the nodes in decreasing order of degree of modifications. If two nodes have the same degree of modification, the one closer to the root of the ICG is assigned a higher priority.

- The next lower priorities are assigned to the nodes that directly or indirectly invoke the new or modified nodes, with higher priorities assigned to nodes that are closer to the new or modified nodes.
- All other nodes are assigned the least priorities in an arbitrary order.

3.4 Prioritisation strategies

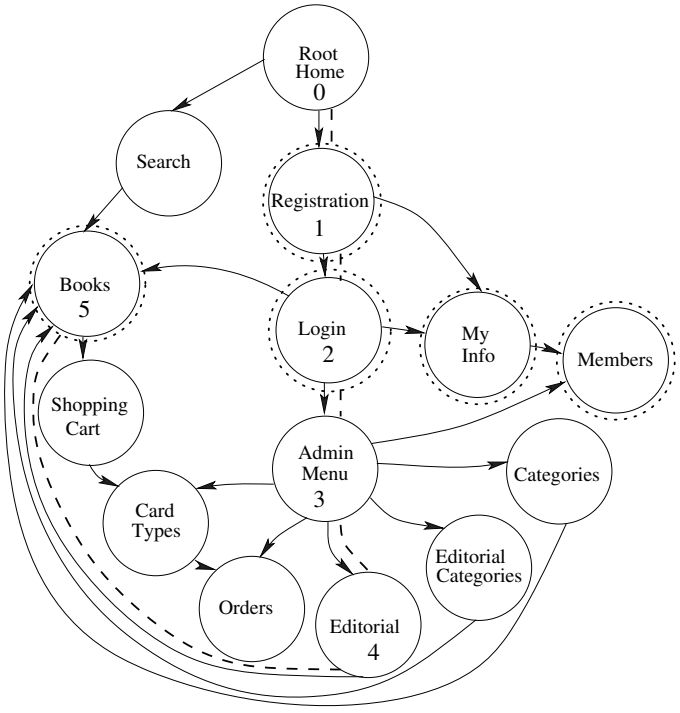

Figure 3.5: FDG showing longest path

Rothermel et al. defined the problem of test suite prioritisation in [80]. Given T as a test suite, P is the set of all test suites that are the prioritised orderings of T obtained by permuting the tests of T and F is a function obtained from P to the reals, the problem is to find a permutation, $T' \in P$ such that $(\forall T")(T" \in P)[f(T') \geq f(T")]$.

Prioritisation can be based on any criteria (fault detection, code coverage and others [89] [19] [84] [24] [85] [41]). In this section, we introduce different prioritisation strategies for prioritising test sets that contain test cases for specific functionalities in a software application. These functionalities are the nodes in the FDG. The dashed path (Figure 3.5) shows the longest route from the root.

The suggested strategies are based on the possible routes in the FDG to detect faults early.

- **Position from root *longest route* high to low** - This will identify the longest route of the modified nodes from the root in the FDG. The test cases related to nodes that have the longest routes are executed first.
- **Position from root *longest route* low to high** - This will identify the longest route of the modified nodes from the root in the FDG. The test cases related to nodes that have the longest routes are executed last.
- **Position from root *shortest route* high to low** - This will identify the shortest route of the modified nodes from the root in the FDG. The test cases related to nodes that have the shortest routes are executed first.
- **Position from root *shortest route* low to high** - This will identify the shortest route of the modified nodes from the root in the FDG. The test cases related to nodes that have the shortest routes are executed last.
- **No. of invoking functionalities *max to min*** - The test cases related to the modified nodes in the FDG that invoke more functional modules are executed first.
- **No. of invoking functionalities *min to max*** - The test cases related to the modified nodes that invoke more functional modules are executed last.

- **No. of changes *max to min*** - The changes are identified as per the ICGs of the nodes of the FDG. The test cases related to nodes that have more changes/modifications are executed first.
- **No. of changes *min to max*** - The changes are identified as per the ICGs of the nodes of the FDG. The test cases related to nodes that have more changes/modifications are executed last.
- **Random** - Randomly permute the order of tests.

3.5 Experimental evaluation

In this section, we detect the different number of faults with various prioritisation strategies and evaluate which prioritisation strategy detects faults early in test suite execution. We selected an application *Online bookstore* for our experiments. The *Online bookstore* application is an online shopping portal for buying books. This application uses ASP for its frontend and MySQL for its backend connectivity. The application allows the users to search for the books by different keywords, add to the shopping cart and proceed to the orders. We seeded faults in the *Online bookstore* application.

Experimental methodology: We used C# to implement this approach. We generated 130 test cases of the *Online bookstore* application and converted them to C# test scripts readable by the Selenium test tool for automatic test suite execution. We categorised these test cases into different functionalities using the FDG of the software application and sliced the entire test suite into test sets. Each test set in a test suite is composed of test cases related to that specific functionality. We generated the test cases by following the activities of each module. The test cases were converted into a file format supported by the various automatic test case execution tools like Selenium [92]. The generated test cases were assumed to be non-redundant and generated as per the functional specifications.

We prioritised the test sets using our prioritisation strategies and the test cases within the test set were prioritised using the ICGs. After prioritising the test cases related to new/modified functionalities, we divided the entire test suite execution into two parts, one part deals with the test cases related to new/modified functionalities and these test cases are prioritised using our two-level prioritisation approach and executed as a front-end activity in test suite execution. The remaining test cases are executed as a backend test suite execution.

Evaluation Metrics: Rothermel et al. presented a metric for measuring fault detection rates of test suites in a given order [81] [80]. This metric is called APFD (Average Percentage of Faults Detected). APFD values range from 0 to 1; higher numbers imply faster (better) fault detection rates [23]. We have used a variation of the existing APFD metric for measuring fault detection rate. This form of the APFD metric helps us to calculate the results for incremental test suite execution.

The original metric used by Rothermel et al. is as follows [80]:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_{\mathrm{m}}}{mn} + \frac{1}{2n}$$

We have rewritten this metric in the following form for the incremental computation of APFD:

$$APFD = \frac{(n - TF_1) + (n - TF_2).. + (n - TF_{\mathrm{m}})}{mn} + \frac{1}{2n}$$

TF_i is the position of first test in T that exposes fault i
n= no. of test cases
m= no. of faults

Informally, APFD measures the area under the curve that is plotted by the percentage of faults detected by prioritised test case order and the test suite fraction. To perform the experimental evaluation, we randomly seeded 20 faults in various modified functionalities of the *Online bookstore* application. Three different kinds of faults were seeded in the application:

- Logical Faults
- Form Faults
- Appearance Faults

We seeded faults in the various modified functionalities like *Registration*, *Members*, *MyInfo*, *Login* and *Books* and assumed that the faults behave like real faults.

Threats to Validity: We manually seeded the faults in the software application. The faults may not be evenly distributed among the functionalities. Although they are considered to be faults of equal severity, faults with different severity levels may vary the results. The test execution may differ depending on hardware. The functional test case execution time may differ due to the varying lengths of test cases.

3.6 Results and analysis

Table 3.1 shows the prioritised sequences of test sets using various prioritisation strategies. In Table 3.1, B refers to test cases related to functional module *Books*, M refers to test cases related to functional module *Members*, MI refers to test cases related to functional module *My Info*, L refers to test cases related to functional module *Login* and R refers to test cases related to functional module *Registration*.

Table 3.2 shows the results obtained using different prioritisation strategies by prioritising test cases using the FDG and ICGs. The table shows the APFD results for the corresponding percentage of test suite run. Table 3.2 shows results in 10% increments. The suggested prioritisation strategies deliver higher APFD results in the first 10% of test suite execution. *Position from root longest route high to low* shows the lowest APFD results in the first 10% of test suite execution. The experimental results indicate that *Position from route shortest route low to high* detects the maximum number of faults in the first 10% of test suite execution. This is not surprising, as this strategy detects faults in the

Table 3.1: Test case prioritisation strategies

Modified functionality	**Position from root *longest route* high to low**	**Position from root *longest route* low to high**	**Position from root *shortest route* high to low**	**Position from root *shortest route* low to high**	**No. of invoking functionalities *max to min***	**No. of invoking functionalities *min to max***	**No. of changes *max to min***	**No. of changes *min to max***	**Random**
Reg	1	1	1	1	11	11	7	7	1
Login	2	2	2	2	10	10	1	1	2
My-Info	3	3	2	2	1	1	2	2	3
Books	5	5	2	2	3	3	8	8	4
Members	4	4	3	3	0	0	2	2	5
Prioritised Sequence	• B • M • MI • L • R	• R • L • MI • M • B	• M • B • MI • L • R	• R • L • MI • B • M	• R • L • B • MI • M	• M • MI • B • L • R	• R • B • M • L • MI	• MI • L • M • B • R	• R • L • MI • B • M

Table 3.2: Results from prioritisation strategies

%age of test suite run	**Position from root *longest route* high to low**	**Position from root *longest route* low to high**	**Position from root *shortest route* high to low**	**Position from root *shortest route* low to high**	**No. of invoking functionalities *max to min***	**No. of invoking functionalities *min to max***	**No. of changes *max to min***	**No. of changes *min to max***	**Random**
10%	49.84	58.57	58.57	**90.45**	77.14	67.57	**78.07**	66.68	36.49
20%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	40.76
30%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	40.76
40%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	40.76
50%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	46.30
60%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	46.30
70%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	55.64
80%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	56.95
90%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	56.95
100%	91.95	94.11	91.88	**94.49**	93.49	92.26	**94.49**	91.38	57.03

functional modules that are closer to the root of the FDG and hence this is a good strategy for detecting faults early.

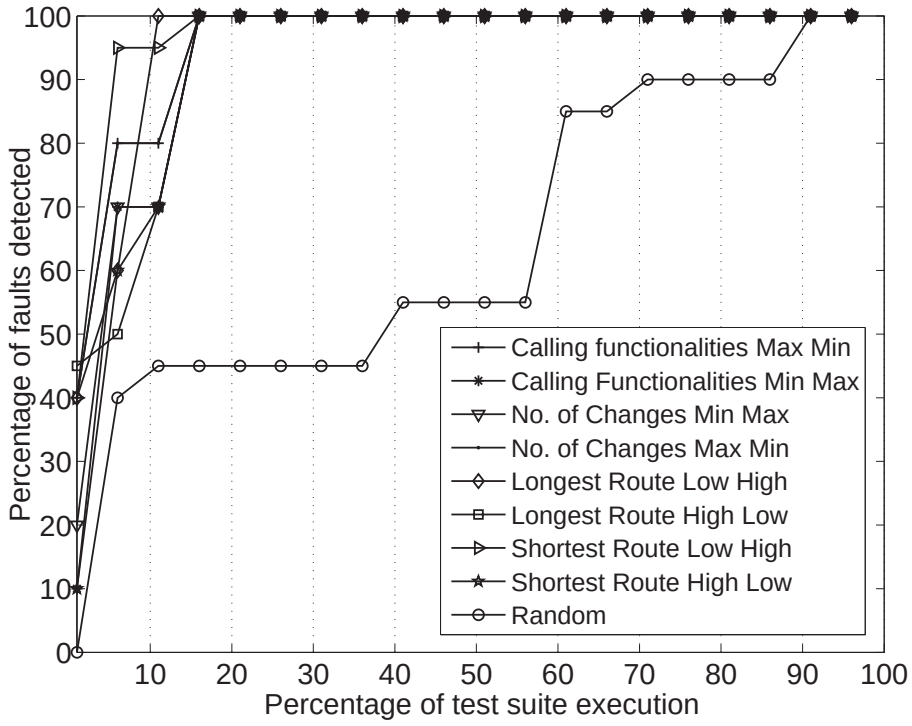

Figure 3.6: Faults detected using various prioritisation strategies

Figure 3.6 shows the faults detected using various prioritisation strategies. *Random* ordering of test suite execution shows the lowest APFD results as compared to all our prioritisation strategies. *Random* ordering shows the APFD result of 57.03 even after 100% of test suite execution. Our prioritisation strategies are able to detect close to 100% faults in the first 20% of test suite execution. *No. of changes max to min* and *Position from route shortest route low to high* show the highest APFD of 94.49.

3.7 Conclusions and future work

We conclude that modified functionalities have more chances of faults. The already tested functionalities in the previous regression test cycles are locked in as per specifications. If the new/modified functionalities are receiving the correct values and parameters from the already tested functionalities, then we may not need to test those functionalities again. We suggested this approach using FDG and ICG graphs as the different modules are written using different languages in software applications. ICG graphs are constructed using the particular language used for that module. We suggested nine different prioritisation strategies for prioritising functional test cases. Of these prioritisation strategies, if we prioritise the test cases based on the number of changes in different components of software applications, the components that have the maximum number of changes/modifications have more chances of faults. But on the other hand, if we follow the shortest paths from the root for the modified functionalities and execute the tests accordingly, they are able to detect most of the faults in the first 10% of test suite execution.

We validated the results using various different test combinations. In the future, we will validate our results on two more software applications. We will consider real faults with different cost levels. As we have shown, our first 20% test suite execution detects most of the faults. In the future, we will suggest a technique that will select the test cases related only to the modified functionalities in software applications and execute only those test cases that will provide maximum fault detection. This may help to reduce the total test execution time as we need to execute only a subset of test cases.

Chapter 4

A bipartite graph approach for selection of test cases

We discussed in Chapter 3 various prioritisation strategies to detect faults early due to modifications in program source code. But using the prioritisation techniques described in Chapter 3, we still need to execute all the test cases. The execution of all test cases requires a tremendous amount of time and it is important to select only the relevant test cases for a rapid testing in a regression testing cycle.

In this chapter, we describe a new technique for the selection and prioritisation of test cases for large multi-tier applications. Such applications undergo changes rapidly due to changes in specifications, requirements and reported bugs. New test cases are generated over time to test the modifications in source code due to these changes. As the number of test cases grows due to modifications in the source code, it becomes difficult to execute all the test cases in every regression testing cycle. The execution of obsolete test cases results in less test coverage and longer test suite execution time, which is unacceptable for improving the availability of these applications.

In this chapter, we suggest a new approach using bipartite graphs for selection of test cases. We divide the entire software application into

different partitions. We map the test cases and extracted user-defined source code components using bipartite graphs. We select the subset of test cases that corresponds to the modified source code. After selecting the required test cases, we prioritise the selected test cases using new prioritisation strategies. We further compare the new suggested strategies with the strategies that are discussed in Chapter 2.

4.1 Our approach

Our approach uses the program source code and the test cases. We extract the user-defined components *(as explained in Section 4.1.2)* and then match these components with the test cases *(as explained in Section 4.1.3)*. We use the program source code and the test cases to create two different sets. We construct a bipartite graph from these two sets. The test cases that match the code components are extracted in a different test set as a part of an active test process. This test process is considered as the *front-end test suite execution process*, a part of the priority test execution process. We describe our approach in detail in the next few sections.

To evaluate our approach, we consider two different applications. We use *Online bookstore* [1] to describe our approach in this thesis due to the simplicity of this application. The *Online bookstore* application is an online shopping portal for buying books and is written using C#. This application uses ASPX for its front-end and MySQL for its back-end storage. The application allows users to search for books by typing different keywords, add to the shopping cart and proceed to orders.

The other application that we used is *Moodle* [2]. This application uses PHP and MySQL for back-end storage. This application has a huge database comprising more than 200 tables. It allows users to submit

[1] available freely at `www.gotocode.com`
[2] available freely at `www.moodle.org`

their assignments and grades and to enroll in courses, in addition to many other services associated with a learning management system.

4.1.1 Division of software application into partitions

We divide the entire application into different partitions. Each partition corresponded to a web-page. We extract the user-defined components of each partition as explained in Section 4.1.2. The extracted user-defined components are a subset of the x_1 to x_n nodes.

The test suite is composed of different test cases. We divided the test suite into different test sets and each test set corresponds to the group of test cases that are required to test that partition. We divided these software applications into different partitions by considering the .aspx code of a web-page. This file further contains two different files called .cs (C#) file and designer.cs file. This partition also contains information about the data layer and meta layer related to that particular web page. An example is shown in Figure 4.1.

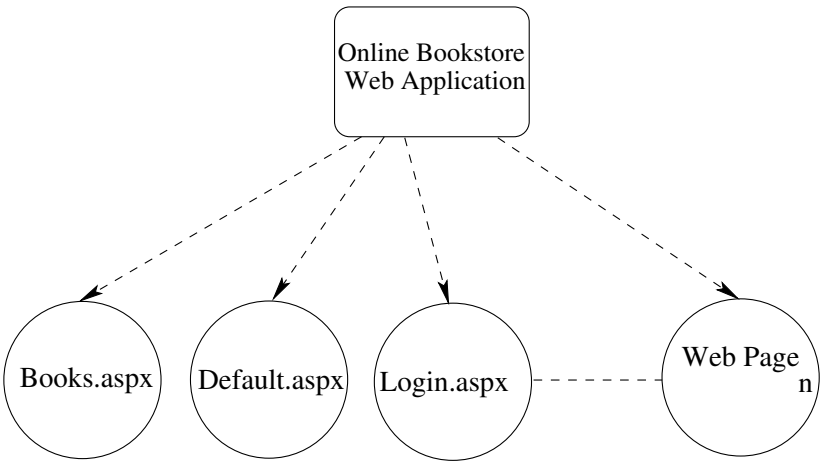

Figure 4.1: Partitioned Software Application - *Online bookstore* application

4.1.2 Extraction of user-defined code components

We extracted user-defined components from the program source code to simplify our matching process. The program source code contains language keywords and user-defined variables. We identify the user-defined variables and store them in a file called the *Captured Strings File* or *CSF*.

To begin with the extraction process of user-defined components, we have a created a text file that contains the system-defined components related to the programming language. These system-defined components are like *int*, *void* and all other system-defined components. We match this text file with our program to determine the user-defined components. These components are not related to the programming language keywords. These components are associated with the name of the classes or variables that are used to define the language parameters. We show this in an example below:

```
void Login_login_Click(Object Src, EventArgs E)
{
    if (Login_logged)
    {

        // Login Logout begin

// Login OnLogout Event begin
// Login OnLogout Event end
Login_logged = false;
        Session[``UserID"] = 0;
        Session[``UserRights"] = 0;
        Login_Show();
        // Login Logout end
    }
     else
     {
```

```
// Login Login begin
int iPassed =
Convert.ToInt32(Utility.Dlookup
(``members", ``count(*)",
``member_login ='"
+ Login_name.Text + ```
and member_password='"
+ CCUtility.Quote
(Login_password.Text) + ```'"));
if (iPassed > 0)
```

We extracted the user-defined components from the application program. In the above example, the user-defined components are *Login_login_Click*, *Login_logged*, *UserID*, *UserRights*, *Login_Show*, *iPassed*, *members*, *member_login*, *Login_name*, *member_password* and *Login_password* and the system-defined components are *void*, *Object*, *Src*, *EventArgs*, *if*, *else*, *false*, *Session*, *int*, *Convert*, *ToInt32*, *Utility*, *Dlookup*, *count*, *CCUtility* and *Quote*.

We have extracted the user-defined components in the test cases as shown below:

```
browser.Click
(``//a[@id='Header_Menu_Field1']/img");
browser.Type(``Login_name", "guest");
browser.Type(``Login_password", "guest");
browser.Click(``Login_login");
browser.Click
(``//a[@id='Header_Menu_Reg']/img");
browser.Click(``Reg_cancel");
Global.ShutdownServer(browser);
```

Multiple test cases are required to test some portions of source code. Every test case contains system-defined components that are required to execute that test case. We eliminate those system-defined components

to extract the user-defined components. In the above test case snippet that is used to test *registration.aspx* page, the user-defined components are *Header_Menu_Field*
1']/img, *Login_name*, *guest*, *Login_password*, *Login_login*, *Header_Menu_Reg* and *Reg_cancel* and the system-defined components are *browser*, *click*, *type*, *Global* and *ShutdownServer*. The term *browser* represents the test automation tool. We store the user-defined components extracted from the test cases in a separate text file. This text file is further used to represent nodes as explained in Section 4.1.3.

4.1.3 Mapping between source code and test suite

We frame this problem as a matching problem in a bipartite graph. Figure 4.2 shows two types of nodes. The nodes x_1 to x_n contain information about the source code components or *CSF*. In our case, we use *Online bookstore* application to explain our approach. The source of this application is divided into different files/web-pages like *login.aspx* also includes *login.cs*, *books.aspx* also contains *books.cs*. The *login.aspx* file represents the front-end view to the end-user and *login.cs* contains the source code written in C# language related to that particular web-page. The *login.aspx* and *login.cs* together build a login web-page. Each such node contains user-defined components that are extracted in Section 4.1.2 and one node contains user-defined components related to one web-page.

The nodes y_1 to y_n contain information about the test cases. The test suite is divided into test sets. The test set contains test cases belonging to a particular web page. Each node in y_1 to y_n contains user-defined components related to test cases including the test cases from the previous versions of that software application. Each node corresponds to one test set.

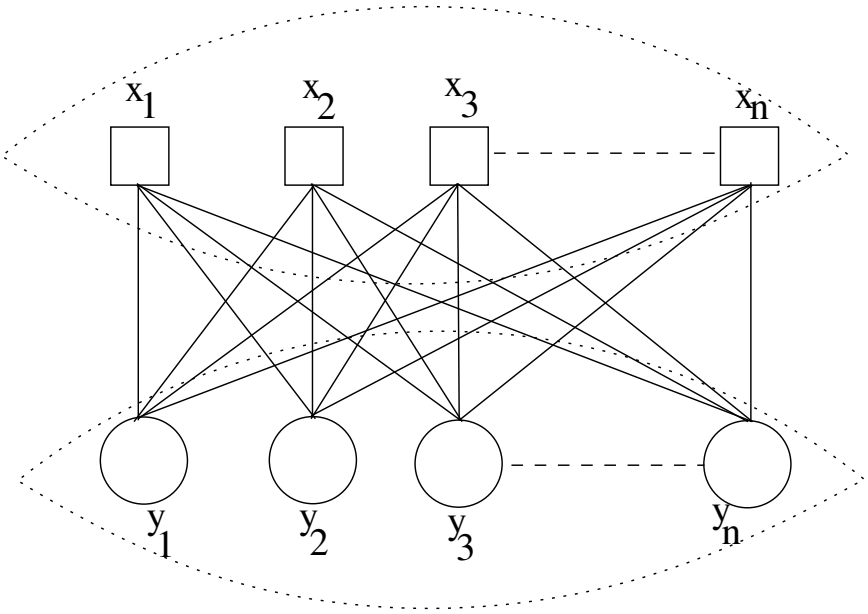

Figure 4.2: Mapping between source code and test cases

Each node that represents source code in Figure 4.2 is matched with one or more nodes that represent test sets. This matching associates each source code component with one or more test cases in the test suite. The test cases associated with a source code are used to test the correct functioning of the code associated with that program. This matching information helps in identifying the test cases that are required to test the source code.

4.1.4 Selection of test cases

In our approach, we consider two different cases:

- Case1: The mapping of the test cases to the source code is already known and we select the subset of test cases that relates to the source code.
- Case2: The mapping between source code and the test cases is unknown. In that case, we match the nodes x_1 to x_n with the

nodes y_1 to y_n using a maximal matching algorithm to identify the test cases that match the source code.

We select the test cases that correspond to the program source code. We considered bipartite graph G=(V,E). A matching M is said to be maximal if M is not properly contained in any other matching. Figure 4.2 shows the graph that contains two kind of nodes: source code and the test cases that are required to test the source code. The nodes x_1 to x_n are matched to the nodes y_1 to y_n.

We use the Hopcroft-Karp algorithm [44] to find the matching combinations. The graph that contains information about nodes with x is denoted as xx and the graph with nodes y is denoted as yy. The following steps are required to perform the matching operation:

1. We use breadth-first search to partition the vertices of edges into layers. The vertices in xx are used at the starting of the search and form the first layer of the partition.

2. At the first level of search, only the unmatched edges may be traversed. The search terminates when the first layer k where one or more free vertices in yy are reached.

3. All the remaining vertices in yy are collected into a different set f.

4. Repeat the steps until the vertices in xx form the pairs with vertices in yy.

5. Discard the nodes in yy.

We discard the nodes in yy as they are not able to make a match. These nodes are the test cases that are not required for the testing of the software application. We prioritise the selected test cases using our prioritisation approach in Section 4.1.5.

Table 4.1: Results from prioritisation techniques

Applications	**p_mod**	**p_mod_random**	**stmt-total**	**random**
Moodle	83.59	78.59	43.85	73.43
Online bookstore	86.69	33.00	37.85	21.83

4.1.5 Prioritisation of selected test cases

We suggest a new prioritisation strategy *p_mod* for prioritising the selected test cases and the discarded test cases in Section 3.4 are not considered for prioritisation. Our prioritisation strategy is based on the modifications of the source code. This is based on the assumption that the modified or new code that is not tested in the previous testing cycles has more chances of faults. The prioritisation technique is as follows:

1. The test cases that correspond to the newly added source code are assigned the highest priority, as these are new test cases that have not been executed in any of the previous regression testing cycles.
2. The test cases that correspond to the modified source code are considered at the next highest priority level.
3. Any remaining test cases are randomly prioritised at the end.

4.2 Execution of test cases

We use the Selenium test tool for automatic test suite execution. We divided the entire test suite execution of an application into two different test processes: *active test execution process* and *inactive test execution process*. We automatically execute these two different test processes simultaneously on two different machines.

The *active test execution process* contains test cases that are selected and prioritised using our suggested approach. The test process is con-

sidered as a part of the front-end activity in test suite execution. The results from the front-end activity are reported in the active test suite execution report.

The *inactive test execution process* contains discarded test cases from Section 3.4. These remaining test cases are considered as a part of the back-end test suite execution process and are executed as a part of the back-end test suite execution process. The results from the back-end test suite execution process are not reported in the *active test execution report* and are not considered in the test execution report for testing of that particular version of a software application but these results are stored for future reference for understanding the effect of the overall test suite execution.

4.3 Experimental evaluation

In this section, we discuss our experimental set up using our suggested approach. To perform the experimental evaluation, we used two different applications, *Online bookstore* and *Moodle*. *Moodle* is a widely used learning management application and *Online bookstore* is a simple and small application and has already been used in the literature. We have modified some of the functionalities of *Online bookstore* to make them more complex. We used the *Moodle 2.4 stable* version for our experiments. We used the *Moodle 1.7 stable* version to make use of the modified functionalities between these two different versions.

We removed some of the features that are present in the existing version of *Online bookstore* and *Moodle*. The reason for removing these features is to make some of the test cases that are required to test these features no longer required. We select only those test cases with our approach that are required to test the functionalities that are present in that application. We matched the source code components with the test cases for selecting the best possible combination of test cases. We discard all the test cases that are a part of set f (as in Section 4.1.4).

We seeded 10 faults in various functionalities of *Online bookstore* as well as of *Moodle*. These faults were related to the modifications of the source code.

We used C# to implement our proposed approach. We generated 130 functional test cases for the *Online bookstore* and converted them to C# test scripts readable by the Selenium test tool for automatic test suite execution [92]. The generated test cases were assumed to be non-redundant and were generated according to the functional specifications. We generated 260 test cases for *Moodle* using Java. We detect the faults using our suggested selection and prioritisation approach for test cases. After applying our selection technique, we noticed about 30% reduction in the size of test suite as compared to the original test suite. We were able to detect all the seeded faults using our suggested approach. We prioritised the test cases using different prioritisation approaches and used the APFD metric to compare these approaches.

Evaluation Metrics: To evaluate our approach and compare our results with the existing state of the art, we used the APFD metric as described in Section 2.7.

We collected the test execution results and used the APFD metric to determine whether our approach detects faults earlier or faster as compared to other prioritisation approaches.

Threats to Validity: We manually seeded the faults. The faults may not be evenly distributed among all the partitions. The faults severity depends upon the different applications in a real environment. The partitions may vary in size depending upon the size of the web-page. The functional test case execution time may differ due to the varying lengths of test cases.

4.4 Results and analysis

To show the validity of our results, we compared our new two prioritisation approaches with the existing prioritisation approaches in the literature. We used prioritisation techniques suggested in [80] [85] to validate our results. We showed the results using the techniques: *p_mod*, *p_mod_random* and the existing prioritisation approaches: *random* and *stmt-total*.

In *p_mod*, we selected and prioritised the test cases using our suggested selection and prioritisation approach. In *p_mod_random*, we considered our case 1 as described in Section 4.1.4, where the mapping of test cases to the source code is already known and we selected the subset of test cases and randomly executed them. In *random*, we considered the random ordering of test cases [80] [85]. In *stmt-total*, we prioritised on the basis of coverage of statements [80]. We selected the test cases using our suggested approach and used the *stmt-total* approach to prioritise the test cases. For random, we randomly selected the equal number of test cases as we have in other prioritisation approaches. We maintained an equal number of test cases in all the prioritisation strategies for the efficiency of results calculated using the APFD metric.

Table 4.1 shows the results obtained using various prioritisation strategies for Online bookstore and Moodle. In case of Moodle, *p_mod* gives the highest APFD for 100% execution of test cases. *p_mod_random* performs better than *stmt-total* and *stmt-total* gives the lowest APFD among all these prioritisation techniques. *Random* gives the highest APFD.

In case of Online bookstore, *p_mod* gives the highest APFD of for 100% execution of test cases. *Random* gives the worst APFD result. *p_mod* gives the highest APFD result. We have observed that *p_mod_random* and *random* performed better in case of Moodle as compared to Online bookstore.

4.5 Conclusions and future work

We used a bipartite graph to detect the matching between the source code and the test cases. We have proposed a general approach that can be used for regression testing of any software application. Furthermore, our approach is applicable to all kinds of test case formats and is independent of the test tools. We have shown the results of our selection and prioritisation techniques using two different applications.

The *p_mod* approach has delivered good results for both of these applications and has attained the highest APFD. The *p_mod* approach has shown better results as compared to the techniques that are already available in literature. We conclude that new or modified source code that was not tested in the previous testing cycles has more chances of introducing faults. We validated our results using various test combinations. In future, we will validate these results on other complex applications.

Chapter 5

Selection of test cases due to modifications in database

In Chapter 3 and 4, we have suggested techniques to detect faults early due to modifications in source code. In this chapter, we focus on detecting faults due to modifications in databases of multi-tier applications. We discussed in Chapter 2 that multi-tier applications often require modifications in databases due to operational necessities. The modifications in the databases result in the instability of an application and some of the functionalities may not work properly. As discussed in the state of art, the database modifications require execution of all the test cases in a regression testing cycle. The execution of large number of test cases requires tremendous amount of time and provides less test coverage. It provides less test coverage due to the execution of large number of test cases that are not relevant for the testing of current version of the databases and the associated code of a software application.

To overcome this issue, we suggest a new technique for selection and prioritisation of test cases that are relevant for the testing of current databases of a multi-tier software application. The selection and prioritisation of test cases is discussed in this chapter when a mapping between databases keys and the source code of the application can be extracted. We first extract the schema of a database. The extracted

schema is next matched with the source code. The source code and test cases are further mapped using bipartite graphs. The test cases that are related to the database are selected using our techniqu of matching in bipartite graphs. The selected test cases are then prioritised using novel prioritisation approaches. The new suggested techniques are evaluated in comparison with the existing approaches discussed in Chapter 2.

5.1 Our approach

Our approach uses the web application source code, database and the test suite for regression testing. We implemented our approach using the C# language. We first capture the database schema of a web application and the database key attributes are matched with the source code of a web application. The key attributes that are present in the source code of the web application are considered as the *active* key attributes. The source code that uses active key attributes is then matched with the test cases. The test cases that match the active key attributes are extracted in a different test set as a part of a separate test process. This test process is considered as the *front end test suite execution process*, a part of the priority test execution process. We describe our approach in details in the next few sections.

To implement our approach, we consider a web application as a multi-tiered application. We used two different web applications to implement our approach. We used the *Online bookstore* [1] application to describe our approach in this paper due to the simplicity of this application. The other web application that we used is *Moodle* [2]. *Moodle* is an open source community based tool for learning management.

[1] available freely at `www.gotocode.com`
[2] available freely at `www.moodle.org`

5.1.1 Generation of schema diagram

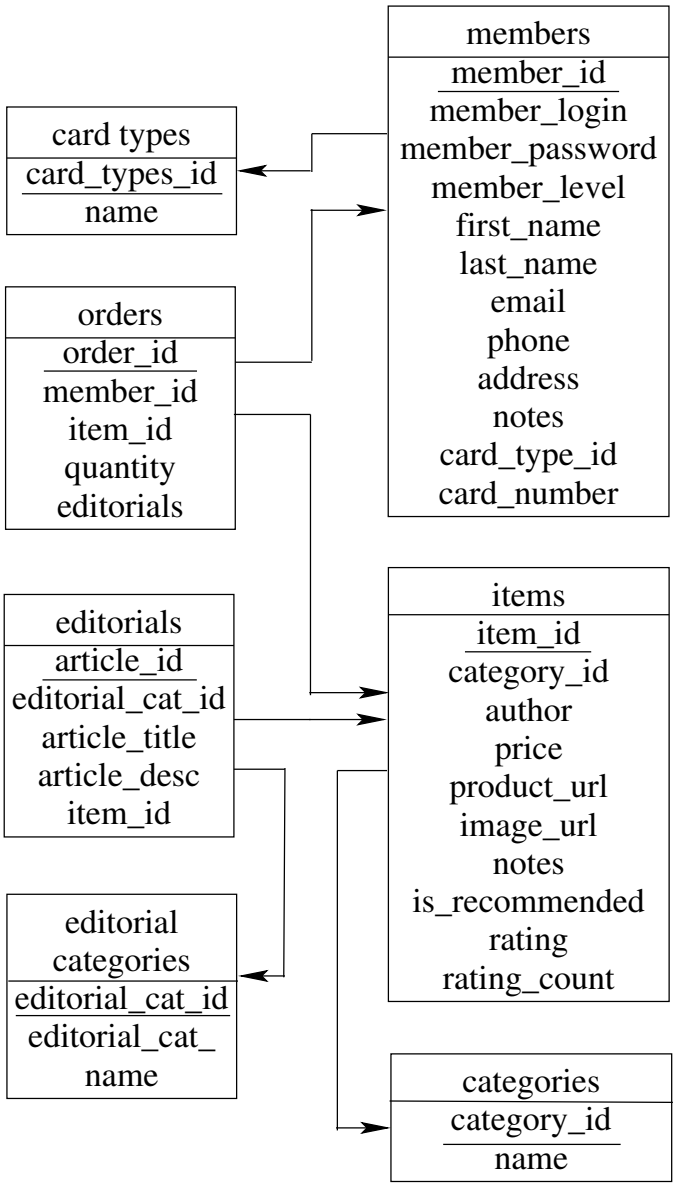

Figure 5.1: Schema Diagram (SD) for the *Online bookStore* application

In this section, we discuss the generation of schema diagram in our context.

A Schema Diagram (SD) describes the various tabular schema in a database. It shows the relationship among various tables in a database and the relationship among different fields in a table. It also provides information about the primary and secondary keys used in the tables. A schema diagram also provides information about the tabular schematic properties like keys, column size, base table name and base column name. We generated SD for all the tables used in the *Online bookStore* application, shown in Figure 5.1 (the primary keys are underlined).

Now we describe the snapshot of the extraction of database schema. We retrieved the database schema for *Online bookStore* application using C# language. The application uses the MS-Access database $Bookstore\_MSAccess$
.mdb. We established an *OleDbConnection* and we established the database connection using database connection strings. We select the tables in a database and retrieved the schema of each individual table in a database using *GetSchemaTable()*. This system command gives us the description of schema of each table in a database. This provides us the information about the column names and the primary keys used in that table. We used *myProperty* to display the property of the various columns. *Iskey* property checks whether a given column is a part of a primary key.

5.1.2 Selection of test cases related to database

Modern applications support various features, including features that are not related to the database and are used to perform operations at the user level. These features may relate to the scrolling of web pages or deal with providing information to the user. We extract only those test cases that are related to the database from the entire test suite of an application. We first extract the database schema of the application and then capture the various key attributes (including primary, secondary and tertiary key attributes) from the database schema. The key attributes are matched with the source code of the application. This

helps us in identifying the key attributes that are missing in the source code but are present in the database of an application.

Extraction of key attributes from the database schema

We extract the database schema. The column names are extracted from the database schema. These column names relate to the primary keys, secondary keys and tertiary keys.

Mapping between database schema and source code

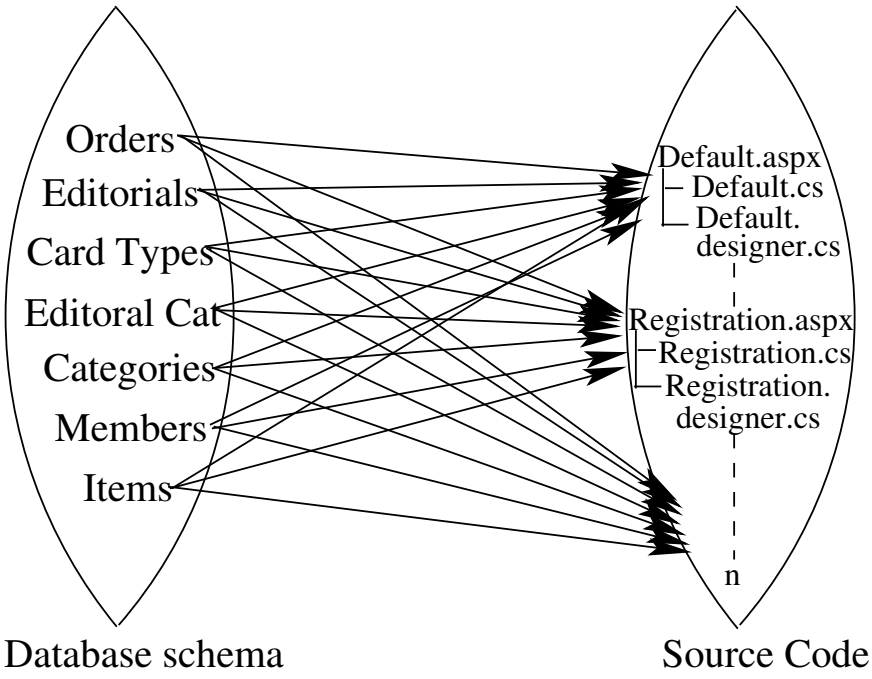

Figure 5.2: Mapping between database schema and source code

Next, the key attributes that are used in the database schema are matched with the source code of the web application. We create two different files to store the key attribute information. The information that relates to the key attributes that are present in the source code of that web application and also present in the database are stored in

a file named *valid*. The key attributes that are present in *valid* are matched with the source code. We select only the test cases that test source code that is matched with the key attributes that are present in *valid*. The information that relates to the key attributes that are not present in the source code but are present in the database of that web application are stored in a separate file called *NMI*, or *Non Matching Information*.

The left column in Figure 5.3 shows the schema of the table that is extracted using our schema extraction process. The right column in Figure 5.3 shows the source code. The key attributes that are present in schema are matched with the source code. The mapping interface for the database schema and source code is shown in Figure 5.2. The database schema on the left hand side in Figure 5.2 displays the list of the names of tables used in the database of that web application. The source code on the right hand side displays the information about the various source code files. The key attributes that are a part of the database schema are matched to the source code. This provides us with the information about the key attributes that are present in the database as well as in the source code of the web application.

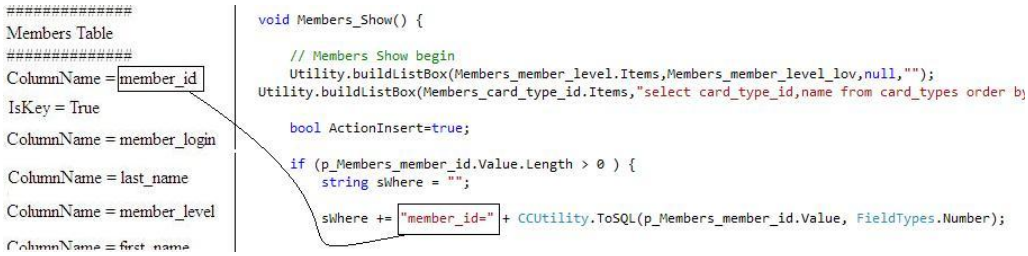

Figure 5.3: Snapshot of mapping between database schema and source code

Mapping between source code and test suite

We assume that the mapping of test cases with source code is already known. This step is used to identify the relevance of the test cases for the testing of source code that is connected to the database. We

capture the database schema using our database schema generation process described above. The mapping interface between the source code and the test suite for the *Online bookStore* application is shown in Figure 5.4. We frame this problem as a matching problem in a bipartite graph. The graph has two types of nodes. The nodes on the left hand side is called Source Code Nodes (SCN) (Figure 5.4). Each such node is a source code file. The nodes on the right hand side are collectively called Test Case Information (TCI) (Figure 5.4) and show the test suite of the web application. Each node in the right hand side is a test case from the test suite.

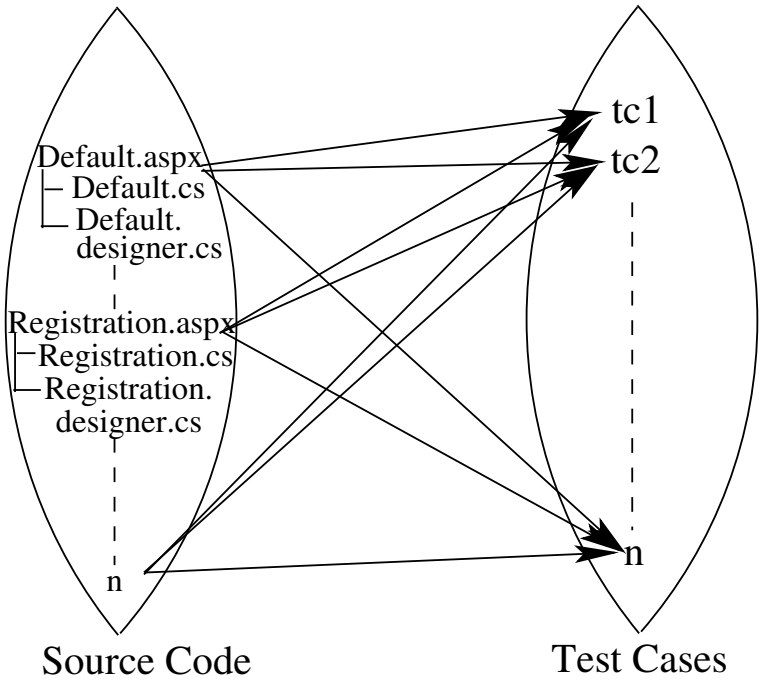

Figure 5.4: Mapping between source code and test cases

Each node in *SCN* is matched with one or more nodes in the *TCI*. We assume that *TCI* has all the test cases that are required to test that web application. This matching associates each source code component related to the database with one or more test cases in the test suite. The test cases associated with a particular source code are used to test the correct functioning of the code associated with that table, where the

code associated with a table invokes any of the key atribute(s) of that table. This matching information helps in identifying whether we have all the test cases that are required to test the database schema. We select the test cases that match with the source code that is connected to the database.

5.1.3 Prioritisation of test cases related to database

We suggest various prioritisation strategies for prioritising the test cases that are selected for the *active test execution process*. These prioritised test cases are executed as a front end test suite execution process. The test cases that are a part of the *inactive test execution process* are not considered for prioritisation. Our prioritisation strategies are based on the modifications as well as the occurrences of the key attributes in the database schema as well as in the source code. This is based on the assumption that if any key attribute appears multiple times in the source code of a web application, there is a higher chance of faults associated with that key attribute. We suggest two new prioritisation techniques. The prioritisation technique *key-new* is based on the new key attributes and is as follows:

1. The test cases that correspond to the key attributes of the newly added tables in the database are assigned the highest priority, as these test cases have not been executed in any of the previous regression testing cycles.

2. The test cases that correspond to the new key attributes in the existing tables in the database are assigned the next highest priority level.

3. The test cases that correspond to the modified key attributes in the existing tables of a database are considered at the next highest priority level.

4. Any remaining test cases are randomly prioritised at the end.

Another prioritisation technique is based on the occurrence of key attributes. The test cases are prioritised on the basis of the occurrence of the key attributes in a database. This prioritisation scheme *key-oc* is as follows:

1. The test cases that correspond to the new/modified primary, secondary and tertiary key attributes that have the highest occurrence in the database schema are assigned priorities in that order, with the primary (resp. tertiary) key attributes assigned the highest (resp. lowest) priority. These test cases are further prioritised on the basis of the occurrence of these key attributes in the source code.

2. Any remaining test cases are randomly prioritised at the end.

5.2 Execution of test cases related to database

We divided the entire test suite execution of a web application into two different test processes. We use the term *active test execution report* to denote the test results that are useful for the testing of database schema and are used by the software developers to debug the faults related to database schema. The remaining test cases are considered as a part of *inactive test execution process* and are executed as a part of the back end test suite execution process. The results obtained from this test execution report are not considered in the test execution report for database schema testing but are considered for the overall testing of the application.

We automatically execute these two different test processes simultaneously on two different machines. We use the Selenium test tool for automatic test suite execution. The first test process deals with the test cases that are selected and prioritised using our new suggested approach and these test cases are considered as a part of the *active test execution*

process. The first test process is considered as a part of the front-end activity in test suite execution. The results from the front-end activity are reported in the active test suite execution report.

The back-end test suite execution process deals with the test cases that do not access the database. The results from the back-end test suite execution process are not reported in the *active test execution report* but these results are stored for future reference for understanding the effect of the overall test suite execution.

5.3 Experimental evaluation

In this section, we discuss our experimental set up using our suggested approach. To perform the experimental evaluation, we used two different applications, *Online bookstore* and *Moodle*. We have modified some of the functionalities of *Online bookstore* to make them more complex. We used *Moodle 2.4 stable* version for our experiments. We used *Moodle 1.7 stable* version to make use of the modified functionalities between these two different versions.

Online bookstore application is an online shopping portal for buying books. This application uses ASPX for its frontend and MySQL for its backend connectivity. The application allows the users to search for books by different keywords, add to the shopping cart and proceed to orders.

Moodle is an abbreviation for Modular Object-Oriented Dynamic Learning Environment. It is a free open-source e-learning software. This application uses PHP and MySQL for this backend connectivity. This application has a huge database comprising more than 200 tables. This application allows the users to submit their assignments and grades and to enrol students in their courses, in addition to many other services associated with a learning management system.

We randomly seeded 10 possible database faults in various functionalities of *Online bookstore*. We seeded 10 possible different faults in the various different functionalities of *Moodle*. The faults were seeded considering various factors like missing key attributes in the source code, missing key attributes in the database, modification of the primary or secondary keys, modification of names of the key attributes in the database and change of the data types or the insertion of new key attributes in the source code.

We used C# to implement our proposed approach. We generated 130 functional test cases for the *Online bookstore*. The generated test cases were assumed to be non-redundant and were generated according to the functional specifications. We generated 260 test cases for *Moodle* using Java. We detect the faults using our suggested selection and prioritisation approach for test cases. We were able to detect all the seeded faults using our suggested approach.

To evaluate our approach and compare our results with the state of the art, we used the APFD metric as described in Section 2.7. We collected the test execution results and used the APFD metric to determine whether our approach detects faults that are related to database earlier and faster compared to the random ordering of the test cases.

Threats to Validity: We manually seeded the faults in the web application. The faults may not be evenly distributed among the functionalities and may not include all the tables in a database. Although they are considered to be faults of equal severity, faults with different severity levels may vary the results. The functional test case execution time may differ due to the varying lengths of test cases.

5.4 Results and analysis

To show the validity of our results, we compared our new two prioritisation approaches with the existing prioritisation approaches in the

literature. We use prioritisation techniques suggested in [80] [85] [10] to validate our results. We show the results using the techniques: *key-new*, *key-oc* and the existing prioritisation approaches: *random*, *stmt-total*, *code coverage*. In *random*, we consider the random ordering of test cases [80] [85]. In *stmt-total*, we prioritise on the basis of coverage of statements [80]. In *code coverage*, we reordered the test cases in each cluster according to the total coverage achieved [10].

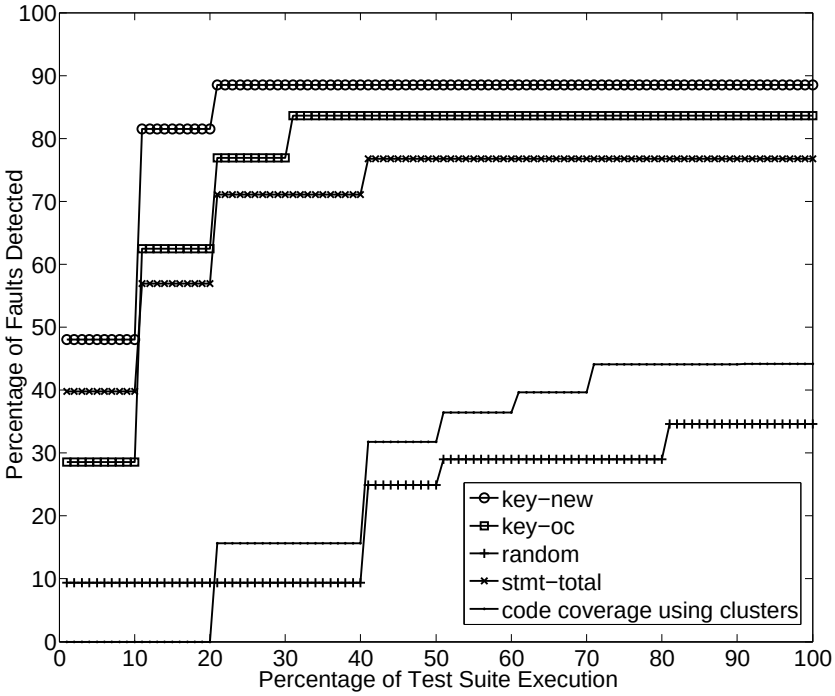

Figure 5.5: Faults detected using various prioritisation techniques *(Online bookStore)*

Figure 5.5 shows the faults detected in *Online bookStore* using five different prioritisation techniques. Our new suggested prioritisation approach *key-new* delivers the highest APFD of 88.53 in 100% test suite execution. More than 40% of the faults were detected in the first 10% test suite execution using *key-new*. The other suggested prioritisation approach *key-oc* has delivered a slightly lower APFD than *key-new*.

It detected less faults in the first 10% of the test suite execution as compared to *key-new*. *key-oc* attains an APFD of 83.66. The *random* approach attains the lowest APFD of 34.69 among all the five prioritisation approaches. *stmt-total* attains an APFD of 76.74. The *code coverage using clusters* approach attains an APFD of 44.18 for 100% test suite execution. Surprisingly, *code coverage using clusters* was not able to detect any faults in the first 20% of test suite execution.

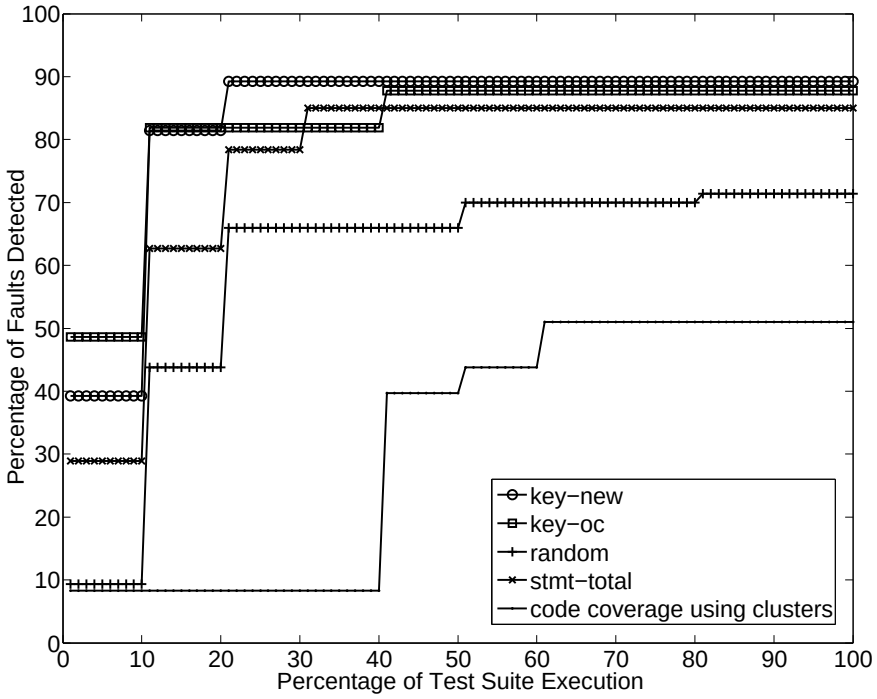

Figure 5.6: Faults detected using various prioritisation techniques *(Moodle)*

Figure 5.6 shows the faults detected in *Moodle* using five different prioritisation approaches. Our new prioritisation technique *key-new* has shown the highest APFD of 89.24 for 100% test suite execution. *key-oc* has attained an APFD of 87.73 for 100% test suite execution. *key-oc* performs better in the first 10% of test suite execution as compared to all the other techniques. *stmt-total* performed slightly worse than *key-*

oc and delivered the APFD of 85.02. The *random* attained the APFD of 71.42. *code coverage using clusters* delivered the lowest APFD of 50.97 and it was able to detect only a few faults in the first 10% of test suite execution.

5.5 Conclusions and future work

We have proposed a general approach that can be used for regression testing of any multi-tiered application with database connectivity, even though our experiments are based on two web applications. Furthermore, our approach is applicable to all kinds of test cases, not only for those based on the Selenium tool. We have shown the results of our selection and prioritisation techniques using two different applications. *Moodle* is a widely used learning management application and *Online bookStore* is a simple and small application and has already been used in the literature. The *key-oc* approach has delivered good results for both of these applications and has attained the highest APFD. We conclude that new or modified keys/tables have more chances of introducing faults. The tables that have already been tested in the previous regression test cycles are locked in as per specifications and introduce less faults. But on the other hand, primary keys also play an important role in database applications and incorrect usage of primary keys results in many faults. We validated our results using various test combinations. In future, we will validate these results on other complex applications. We will also consider real faults with different cost levels instead of seeded faults.

Chapter 6

Prioritisation of test cases to detect faults due to modifications in database

In Chapter 5, we have suggested a technique for the selection and prioritisation of subset of test cases that are required to test the current version of database of a multi-tier application. The approach suggested in Chapter 5 is based on bipartite graphs. The approach is beneficial when it is possible to map the database schema and source code and the further mapping is possible between source code and test cases.

However, this mapping may not be easy to obtain in many situations when diferent languages are used for writing multi-tier applications. Moreover, complex program logic, particularly complex control flow may make this mapping expensive to compute. When it is not possible to obtain the mapping between test cases and source code and/or between source code and database schema, or it is expensive to compute such a mapping, we suggest a new technique using Functional Dependency Graph (FDG). Our technique helps in detecting faults early due to modifications in database when the mapping is unknown. We suggest this technique using FDG and the relationship between schema diagram and FDG. We select the modified tables in a database and extract the

source code that is affected by the modified database. We select the test cases that correspond to the extracted source code. The test cases are prioritised using new prioritisation strategies. The technique suggested in this chapter provides less test coverage as compared to the technique suggested in Chapter 5 as we are prioritising and executing all test cases but in Chapter 5, we execute only a small subset of selected test cases. We compare our results with the existing approaches in literature and the techniques suggested in Chapter 2.

6.1 Generation of schema diagram

A Schema Diagram (SD) describes the various tabular schemas in a database. It shows the relationship among various tables in a database and the relationship among different fields in a table. It also provides information about the primary and secondary keys used in the tables in the database. We generated SD for all the tables used in a software application called *Online bookstore*. We show the SD for the *Online bookstore* in Figure 6.1 (the primary keys are underlined).

If a primary key in one table is modified, that table is marked as a modified table and the related tables that use key attributes from the modified tables will also be marked as modified in the SD.

6.2 Relationship between schema diagram and functionality graph

We extracted FDG as described in 3.1. The SD describes the tables in the database of a software application and the functionality dependency graph (FDG) describes the relationship among the different functionalities. We denote the set of tables in the database as T and the set of functional modules in the FDG as F.

We make two assumptions regarding the use of the database tables from the code components of a node of the FDG.

A1. There is a mapping $\mathcal{M}$: $T \rightarrow F$ that is known at the time of testing. In other words, we can associate a (possibly empty) subset $T_i \subset T$ that is accessed from each functional module $F_i \in F$.

A2. If a table $t_i \in T$ is associated with a functional module $F_j \in F$, the primary key of t_i is accessed from the code components of F_i.

We note that both of these assumptions are quite realistic. First, each table in the database is designed to support some functionality of the software application. Hence, it is possible to associate a table with a functional module at the time of designing the table (Assumption A1). Also, if a table is associated with a functional module, it will be accessed from that functional module through its primary key (Assumption A2). If we modify any table in the SD, we can change its association with the nodes in the FDG, if it is necessary.

6.2.1 Creation of log files

We capture the current database and the modified database of the software application. We extract data related to tables/key attributes from the current and modified databases in different log files. We created three log files: *d_log*, *c_log* and *m_log*. The log file *d_log* contains information about the changes between the current and the modified database schema:

- The information about the new tables in a modified database as compared to the current database.
- The information about the new key attributes in different fields in the modified database as compared to the existing database.
- The information about the modifications of the names of key attributes in all the tables in a modified database as compared with the current database.

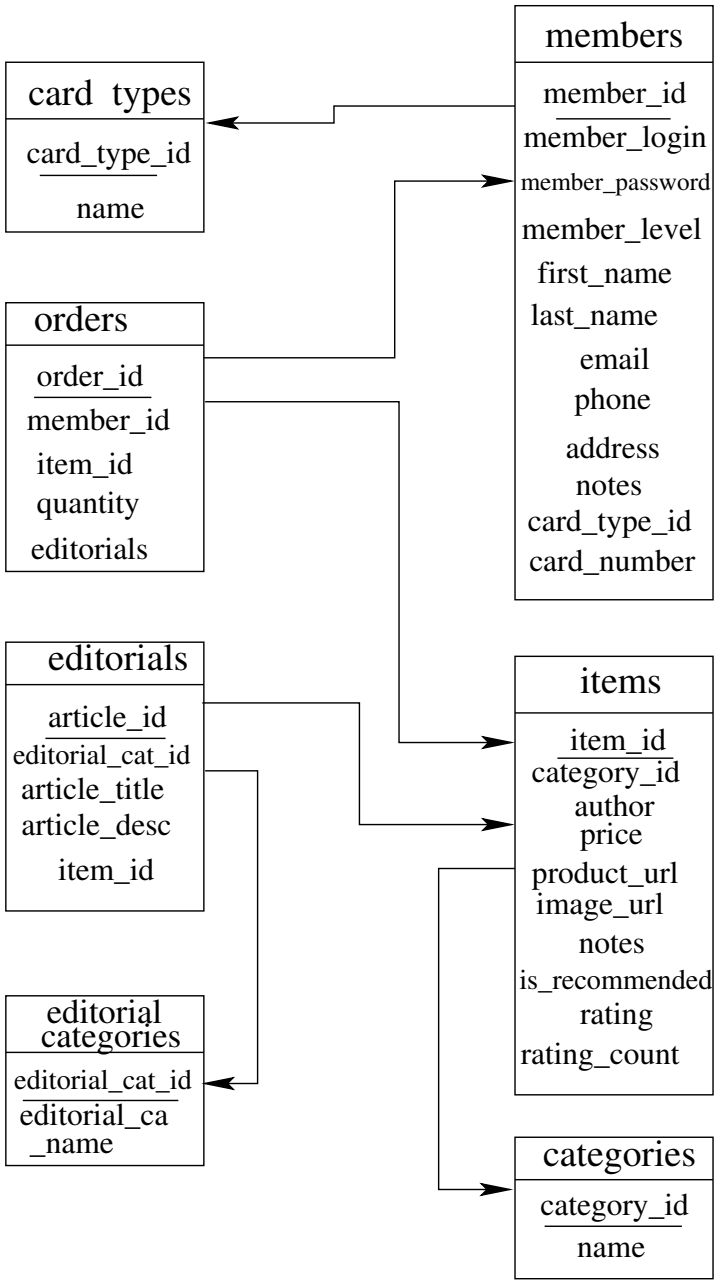

Figure 6.1: Schema Diagram (SD) for the *Online bookstore* application

c_log contains the information about the table schemas in the current database. It also contains information about the various primary keys, secondary keys and tertiary keys used in the tables of the current database. *m_log* contains information about the tables in the modified

database. It also contains information about the primary keys, secondary keys and tertiary keys used in various tables in the modified database.

6.2.2 Log file matching with code components

We match the log file *d_log* with the code components and identify whether all the key attributes are present in the code components. If the key attributes are not matched to the code components, we mark that table as modified in the SD (Figure 6.1) and the functional modules corresponding to the modified tables are marked in the FDG.

We next match the primary keys used in various code components with *m_log*. If any of the primary keys used in code components are not present in the log file *m_log*, those functional modules will be marked as modified in the FDG.

We compare *c_log* and *m_log* for similar key attributes. If similar key attributes are present, we identify the data-types used in the similar key attributes. If similar key attributes with similar names are using different data-types, the key attributes pointing to the functional modules are marked as modified in the FDG.

6.3 Prioritisation engine

Our suggested prioritisation approach reschedules the test cases related to the modified database in a software application early in a test suite execution. Improper use of key attributes of the database in various code components results in many blocker and critical faults and results in exceptions. These faults make the application unstable. As described in the last section, we mark the functional modules as modified in FDG by matching the log files with the code components. We detect the functional modules that are directly connected to the modified tables and

other tables affected by the modified tables in the SD and mark them as modified functional modules in the FDG. We say that a functional module $f_i \in FDG$ invokes a functional module $f_j \in FDG$ *directly* if there is a directed edge from f_i to f_j. f_i invokes f_j *indirectly* if there is a directed path from f_i to f_j.

The test case reordering will be as follows:

- We assign the highest priority to the test cases related to the modified functional modules in the FDG whose modifications are due to the new tables in the SD. This is based on our heuristics that accessing the new tables may give rise to new faults.
- The test cases covering the functional modules that are affected due to new/modified key attributes are assigned the next highest priority.
- The test cases covering the functional modules that are affected due to a mismatch of data-types in the modified tables are assigned the next highest priority.
- The test cases covering the functional modules that invoke the modified functional modules directly or indirectly in the FDG are assigned the next highest priority. Our rationale behind this is that the invoking modules may get affected due to the changes in the functionalities of the modules they are invoking.
- All the remaining test cases are randomly assigned the least priorities.

Let us assume we have modified the key *member_id* to *m_id* in the *members* table and identified the functional modules that are affected due to this change. The test cases related to the modifications in the *members* table will be executed in priority. Suppose *member_id* is also used in the *orders* table as a secondary key. The tables that use key attributes from other tables as secondary or tertiary keys may be affected by the modifications in one table. The test cases related to the functional

modules that are affected due to the table *orders* will be executed after the execution of test cases related to the *members* table.

6.4 Experimental evaluation

We selected the *Online bookstore* application [1] for our experiments. *Online bookstore* is a shopping portal for buying books. *Online bookstore* uses ASP for its frontend and MySQL for its backend connectivity. *Online bookstore* allows the users to search for books using different keywords, add to the shopping cart and proceed to the orders. The *Online bookstore* database is available in MS-Access and uses seven different tables. Each table contains a set of key attributes. We generated 130 test cases from the UML Activity diagrams of the *Online bookstore* application. We generated the test cases by following activities of individual modules. The test cases are converted into a file format that is supported by the automatic test case execution tool Selenium [92]. We randomly seeded 10 faults in the various code components related to the database. We seeded faults in some of the .cs files.

Figure 6.2 shows the modified functional modules (indicated by dotted circles) after detecting the changes in SD. By using our prioritisation engine, we automatically prioritised the test cases. Our approach identified the test cases related to the modified database and marked them as modified functional modules in the FDG. The test cases were reordered using the prioritisation engine. After reordering of the entire test suite, we executed the entire test suite in the prioritised ordering. All the faults seeded due to the database changes were found earlier in the test suite execution. After detecting all the faults, we used the APFD metric as described in Section 2.7 to determine whether our approach detects faults earlier as compared to the random ordering of the test cases.

[1] `https://www.gotocode.com`

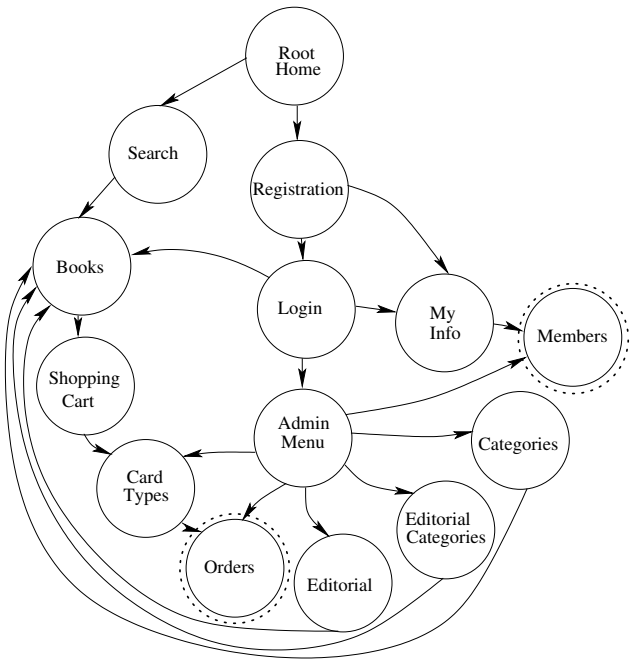

Figure 6.2: FDG with modified functional modules

Threats to Validity. We manually seeded the faults in the software application. We modified many key attributes in different tables in the database and inserted some new tables and key attributes in the existing tables. We inserted faults in various code components related to the new/modified key attributes in the database. Although these faults are considered to be of equal severity, faults with different severity levels may vary the results. The seeded faults may not be equally distributed among all the code components. Test execution times may differ depending on different hardware and due to the varying lengths of test cases.

6.5 Results and analysis

The unordered set of test case execution resulted in the detection of all faults related to database modifications after 55% execution of the test suite. The random ordering shows the APFD results of 70.23%. As

shown in Figure 6.3, the random ordering of test cases detects very few faults related to database modifications in the first 10% of the test suite execution and detected all the faults in 55% of the test suite execution. After prioritising the test cases, our strategy achieves an APFD of 93%. We are able to detect most of the faults in the first 25% of the test suite execution using our new prioritisation strategy and detected 70% of the faults related to database changes in the first 10% of the test suite execution.

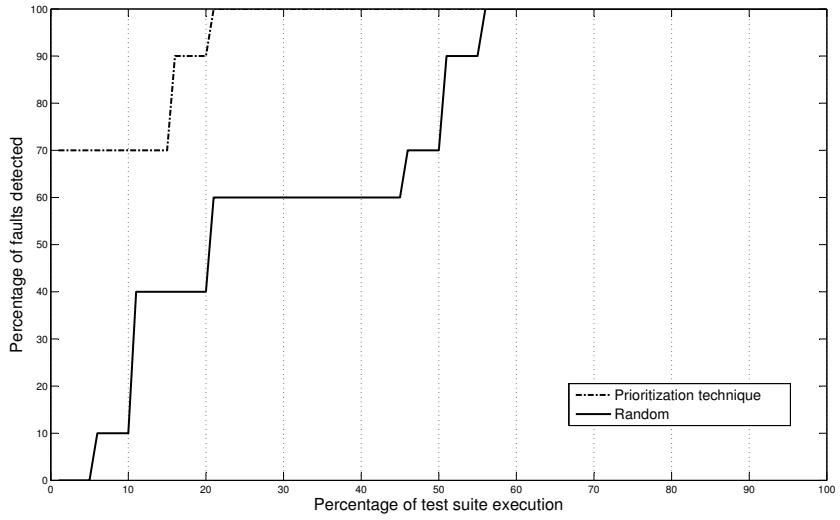

Figure 6.3: Fault detected using various prioritisation approaches

6.6 Conclusions and future work

Faults due to database modifications result in runtime errors in software applications. If the modified tables in a database are not connected properly to the code components, many blocker and critical faults throw runtime exceptions resulting in the failure of the entire software application. Our suggested prioritisation approach automatically reschedules the test cases related to the modified key attributes/tables in a database. Our technique detects most of the faults related to database

changes earlier in test suite execution and will help software developers to fix the faults related to modified database earlier.

We are fully automating this approach and building a new tool. Our approach will automatically detect the modifications in a database in any software application and identify the relationships among the tables in a database and determine the relationships of these tables with the functionalities and code components. It will automatically reschedule the test cases and detect faults due to database modifications early in test suite execution. In future, we will suggest a *regression test selection* (RTS) technique that will select a subset of the test cases related to the modified database from the entire test suite and execute only those test cases to detect faults related to the database modifications.

Chapter 7

Parallel execution of test cases

In the previous chapters, we have suggested techniques to detect faults early due to modifications in source code and database. As multi-tier software applications grow in size due to the change in user requirements, it becomes difficult to execute large number of test cases within a specified period of time. The execution of large number of test cases on one machine takes weeks to execute. To overcome this issue, we suggest a new technique to execute large number of test cases in parallel on multiple machines.

In this chapter, we present a new approach to automatically allocate test cases among multiple machines. We suggest this approach using Functional Dependency Graph (FDG) and Control Flow Graphs (CFG). We partition the test suite into test sets according to the functionalities and associate the test sets with each module of the FDG. The high priority modules and their associated test sets are then distributed evenly among the available machines. Moreover, we further prioritise the test cases by using inter-procedural control flow graphs within individual functional modules. We suggest this approach to reduce the test suite execution time so that the software engineers can detect faults early and within less time.

7.1 Control flow graph

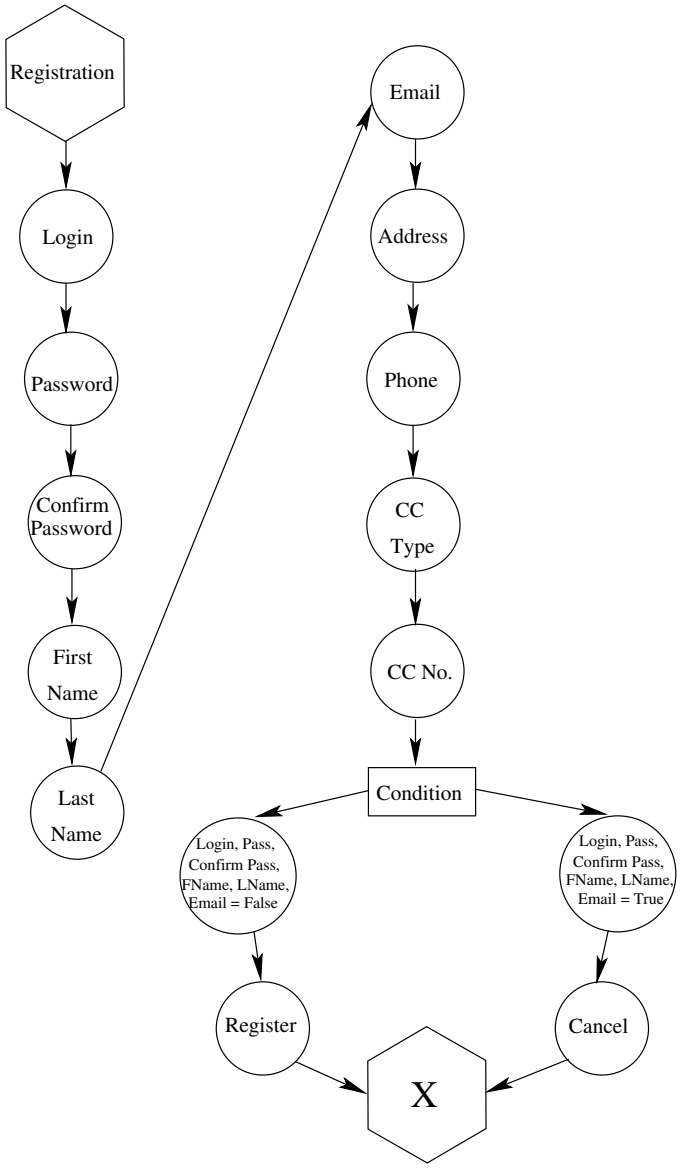

Figure 7.1: Control Flow Graph (CFG) for the *Registration* node in FDG of *Online bookstore* application

We extracted FDG as described in 3.1. We assume that each functionality is composed of a class or a combination of classes in the source code of a software application. We extracted CFG for every functional

module in the FDG of the *Online bookstore*[1] application using the algorithm suggested by Rothermel et al. [83]. Each module in a CFG is either a C# or an ASPX (Active Server Pages Extended) source code module. We show an example CFG in Figure 7.1.

7.2 Our approach

We partition the complete test suite into test sets, each test set is associated with a unique functional module or node in the FDG. We prioritise the test cases within each test set using the CFG of the corresponding functional modules.

7.2.1 Partitioning of test suite

We extract the FDG of the *Online bookstore* application from its functional specifications as described in 3.1. We captured various functional requirements using UML. We captured the interactions between the requirements. We manually extracted the FDG for the *Online bookstore* application from the functional specifications as shown in Figure 7.2 but depending upon the technologies, other methods of generating FDG could be applied. Figure 7.2 shows the various functional modules for this software application.

We generate test cases for all these functional modules and initially randomly store them in a test suite. We partition the entire test suite into test sets, such that each test set is associated with a unique functional module. We automatically identify the test cases that are associated with a particular functional module by reading the test cases and matching the statements. If a test case contains the statements that are required to test some functionality F_i, we store that test case separately in a test set T_i that is associated with F_i. This computation is done in a single machine that we call as the *test server*.

[1] available freely at www.gotocode.com

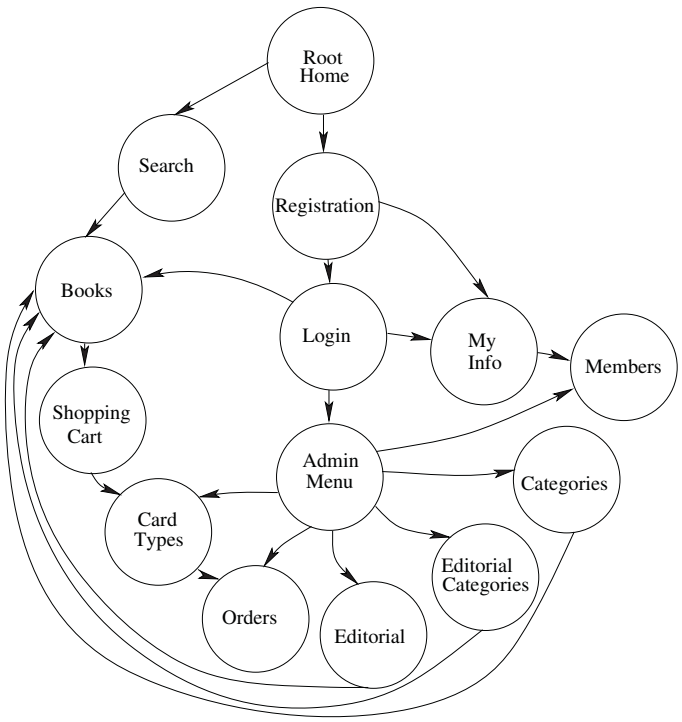

Figure 7.2: Functionality Dependency Graph (FDG) of *Online bookstore* application

7.2.2 Prioritisation of test cases for each functional module

Rothermel et al. defined the problem of test suite prioritisation in [80]. Given T as a test suite, P is the set of all test suites that are the prioritised orderings of T obtained by permuting the tests of T and F is a function obtained from P to the reals, the problem is to find a permutation, $T' \in P$ such that $(\forall T'')(T'' \in P)[f(T') \geq f(T'')]$.

Given a functional module F_i (a node in the FDG), we prioritise the test cases in the corresponding test set T_i according to the CFG of F_i. The modules with the dotted circles in Figure 7.3 refer to the modified source code modules of the *Online bookstore* software application. Each code module in a CFG is associated with a test case or series of test cases. The modified code modules in CFG are executed in priority

compared to the unmodified modules. Our prioritisation strategy is as follows:

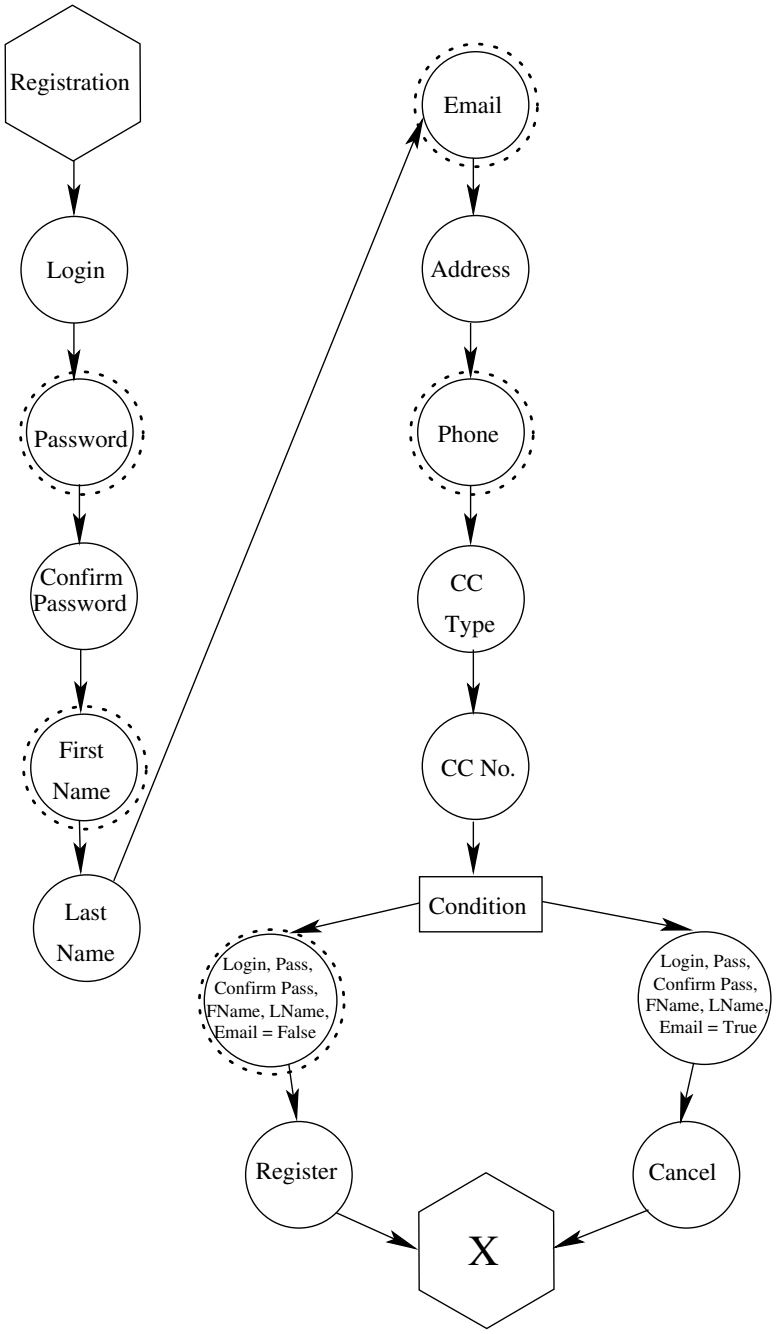

Figure 7.3: Control Flow Graph (CFG) with modified modules of *Registration* functional module of *Online bookstore* application

- The modified modules closer to the root or the class definition are given highest priority in execution of test cases in a test set.
- If two modified modules are at the same level in CFG, the modules having more dependent modules will be given priority in execution.
- We randomly order the test cases belonging to the remaining unmodified modules.

Note that, this prioritisation is done for all the functional modules in the FDG after we generate the test cases in the test suite. As all the functional modules in the FDG may be tested in different regression test cycles, we need to prioritise the test cases for each module using the corresponding CFG. This computation is done in the test server.

7.2.3 Extraction of the affected subgraph from FDG

We recall that a directed edge from node m to node n in the FDG indicates that node n is dependent on node m. We also say node m *invokes* node m. We assign priorities to the nodes of the FDG in the following way:

1. A newly introduced node in the FDG is given the highest priority. If there are multiple newly introduced nodes, they are assigned the highest priorities in an arbitrary order.
2. The modified nodes in the FDG are given the next lower priorities.
3. The next lower priorities are assigned to the nodes that directly or indirectly invoke the modified or newly introduced nodes. A higher priority among these nodes is assigned to a node that is closer (in terms of path length) to a newly introduced or modified node.

4. All other nodes (except the nodes that are invoked by the modified or newly introduced nodes) are assigned the next lower priority in an arbitrary order.

5. The nodes that are invoked either directly or indirectly by the modified or newly introduced nodes are assigned the least priorities. These nodes have been tested in the previous regression test cycles and they are unchanged. Hence we assume that they need not be tested in the current test cycle. Hence these nodes are called *unaffected nodes*. All other nodes are called *affected* nodes.

We use the following observation for extracting the affected subgraph of the FDG.

Lemma 1 *The affected nodes in the FDG form a connected subgraph of the FDG.*

proof 1 *Our prioritisation scheme ensures that every directed path from the root to a leaf of the FDG has all the affected nodes as a connected sub-path. We prove this by contradiction. Consider three consecutive nodes m, n and p on a root to leaf directed path such that m and p are affected but n is not affected. Such a situation will make the affected subgraph disconnected. However, p is affected and n invokes p and hence, our prioritisation scheme ensures that n is also affected, a contradiction.*

We extract the *affected* subgraph S of the FDG by performing a depth-first search. The depth-first search picks up the subgraph containing only the affected nodes, as the search backtracks whenever it encounters an unaffected node. The search returns with the affected subgraph due to Lemma 1. The nodes in S are distributed to the available machines in our parallel prioritisation scheme. This is discussed below. This computation is done in the test server.

7.2.4 Allocating functional modules to machines

The test server allocates the functional modules to the machines participating in the parallel test execution. It is easy to distribute the test sets for the functional modules to different machines if the number of functional modules F is less than or equal to the number of available machines M. Each machine can be allocated the test set of one functional module. However, realistically complex software applications consist of a large number of functional modules and F is usually much greater than M. Hence, we need to allocate multiple functional modules and their associated test sets to each available machine. We construct subsets of functional modules in such a way that each machine is allocated approximately an equal number of functional modules as well as the priorities of the different functional modules in each subset are also approximately equal.

We construct the subsets in the following way. Each node in the selected subgraph S of the FDG has a priority associated with it. We sort the nodes according to these priorities and store in an array A. From the discussion in Section 7.2.3, the nodes can have four different priorities $p_i, 1 \leq i \leq 4$. We denote the number of nodes with priority p_i as $|p_i|$. We allocate $\lceil \frac{|p_i|}{M} \rceil$ $(1 \leq i \leq 4)$ nodes from the priority class p_i to each of the M machines. Though it is possible that the last machine is allocated less than $\frac{|p_i|}{M}$ nodes, nodes from each priority class are approximately evenly distributed among the M participating machines.

7.3 Parallel execution strategy

The functional modules and their associated test sets are allocated to the participating machines by the test server according to the strategy discussed in Section 7.2.4.

Figure 7.4 shows the setup of the entire test process. The main machine acts as a hub and stores all the test case execution data and behaves

like a test server. The other computing nodes execute the allocated test sets in parallel. Each machine stores the test execution results and these results are sent to the test server for constructing the combined test report.

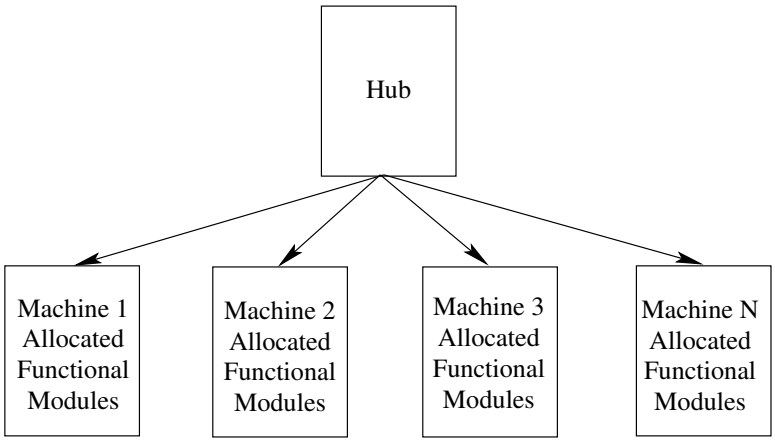

Figure 7.4: Parallel execution of test sets

7.4 Experimental evaluation

In this section, we discuss our experimental set up using our suggested approach.

To perform the experimental evaluation, we extended some of the functionalities of this original application to make it more complex. This application is an online shopping portal for buying books. This application uses ASPX for its frontend and MySQL for its backend connectivity. The application allows the users to search for books by different keywords, add to the shopping cart and proceed to orders.

We randomly seeded 20 faults in various modified functionalities of the *Online bookstore*. The faults were assumed to have similar cost levels. Three different kinds of faults [36] were seeded in the application: Logical Faults, Form Faults and Appearance Faults. A logical fault in the program code relates to business logic and control flow e.g., if the

user inputs the same string for the *password* and *confirm password* fields, still the application displays the error message *Password and Confirm Password fields don't match*. Form faults in the program code modifies and displays name-value pairs in forms. Appearance faults controls the way in which a web page is displayed. We seeded faults in the various modified functionalities like *Registration*, *Members*, *MyInfo*, *Login* and *Books* and assumed that the faults behave like real faults.

We used C# to implement our proposed approach. We generated 130 test cases from the UML Activity diagrams for the *Online bookstore* and converted them to C# test scripts readable by the Selenium test tool for automatic test suite execution [92]. The generated test cases were assumed to be non-redundant and were generated according to the functional specifications.

We partitioned the test suite into different test sets and associated these test sets with the different functional modules. Each test set associated with a functional module is composed of test cases related to that specific functionality. The *Online bookstore* application has 14 functional modules (Figure 7.2) and hence the test suite was divided into 14 test sets. Though our framework is general and can be applied when only some of the functional modules are modified, we modified all the 14 functional modules to evaluate the worst-case performance of our parallel prioritisation scheme. Hence according to the discussion in Section 7.2.3, all the nodes had the highest priority. We used three computers for running the tests. The first two computers were allocated five functional modules each and the last was allocated the remaining four modules. The test cases were executed in parallel on their respective machines. We performed the experiments using Selenium Grid [8]. For comparing our results with a random distribution of test cases, we also randomly distributed the test cases among the three machines and executed them in parallel.

We collected the test execution results and used the APFD metric as described in Section 2.7 to determine whether our approach detects

faults earlier and faster compared to the random ordering of the test cases. We applied the APFD metric separately for the test cases run on each of the three participating machines.

Threats to Validity: We used three different machines with different hardware configurations to execute the test cases in parallel. All these machines had different workload conditions when we were running our experiments by publishing the software application on the virtual web server of the school. We noticed that the test execution times differed when we repeated our experiments. We manually seeded the faults in the software application. The faults may not be evenly distributed among the functionalities. Although they are considered to be faults of equal severity, faults with different severity levels may vary the results. The functional test case execution time may differ due to the varying lengths of test cases.

7.5 Results and analysis

Table 7.1: Results from random ordering

%age of test suite run	**Machine 1**	**Machine 2**	**Machine 3**
10%	0	0	21.68
20%	0	0	61.28
30%	0	0	66.85
40%	0	0	66.85
50%	0	32.44	66.85
60%	19.80	47.06	66.85
70%	46.90	47.06	66.85
80%	46.90	47.06	66.85
90%	46.90	47.06	66.85
100%	50.61	47.06	67.18

We compared the test results for the random approach with our suggested approach using the APFD metric. We recall that we distributed the test cases randomly among the three participating machines in the random approach. The first two machines were allocated five modules each and the last machine was allotted the remaining four modules. Table 7.1 shows the results for the random approach.

We have shown the results in 10% increments. We use the APFD metric to explain our results. We note that the random ordering of the test set in the first machine has not detected any faults for the first 50% of the test set execution. The APFD results for 100% test set execution is 50.61. Similarly, the random ordering of the test set in the second machine has not detected any fault for the first 40% of the test set execution. The APFD result for 100% test set execution is 47.06.

Table 7.2: Results from our approach

%age of test suite run	**Machine 1**	**Machine 2**	**Machine 3**
10%	77.64	74.95	97.17
20%	77.64	74.95	97.17
30%	92.35	91.15	97.17
40%	92.35	91.55	97.17
50%	92.35	91.55	97.17
60%	92.35	91.55	97.17
70%	92.35	91.55	97.17
80%	92.35	91.55	97.17
90%	92.35	91.55	97.17
100%	92.35	91.55	97.17

Table 7.2 shows the APFD results using our prioritisation approach. The test sets allocated to the first machine are able to detect all the faults in the first 30% of the test set execution. The APFD result for 100% test set execution using our prioritisation approach is 92.35. The test sets allocated to the second machine are able to detect all the faults

in the first 30% of the test set execution. The APFD result for 100% test set execution in using our prioritisation approach is 91.55. Similarly, the test sets allocated to the third machine are able to detect all the faults in the first 10% of the test set execution. The APFD result for 100% test set execution for this test set is 97.17.

Figure 7.5 shows the execution results of random ordering of all three test sets. Figure 7.5 shows that many faults were detected after executing close to 100% of the test cases.

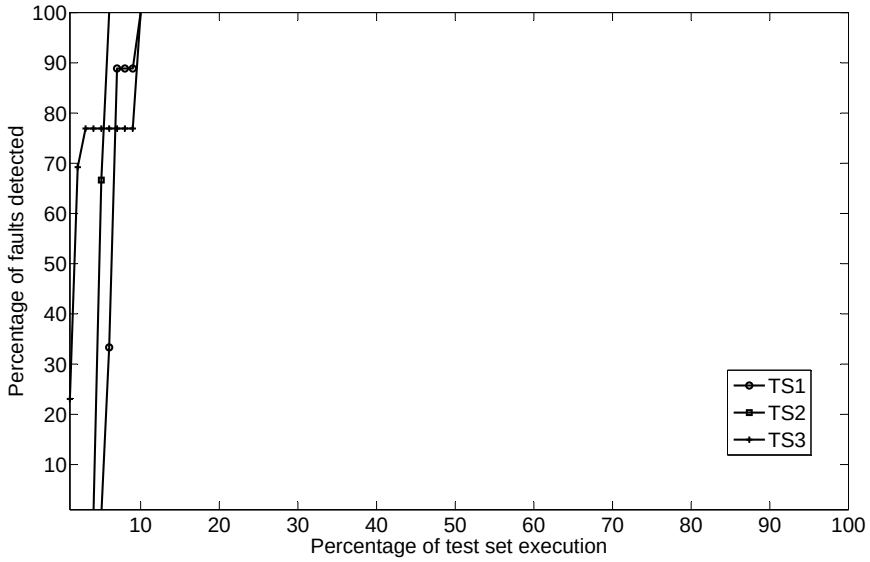

Figure 7.5: Faults detected (Random ordering of test sets)

Figure 7.6 shows the execution results of the test sets that are prioritised using our approach. Figure 7.6 shows that many of the faults were detected in the first 30% of the test set execution.

The execution for the entire test suite on a single machine took approximately 9 hours. After implementing our suggested approach, we were able to execute all the test cases in less than 3 hours and could detect all the faults in the first hour.

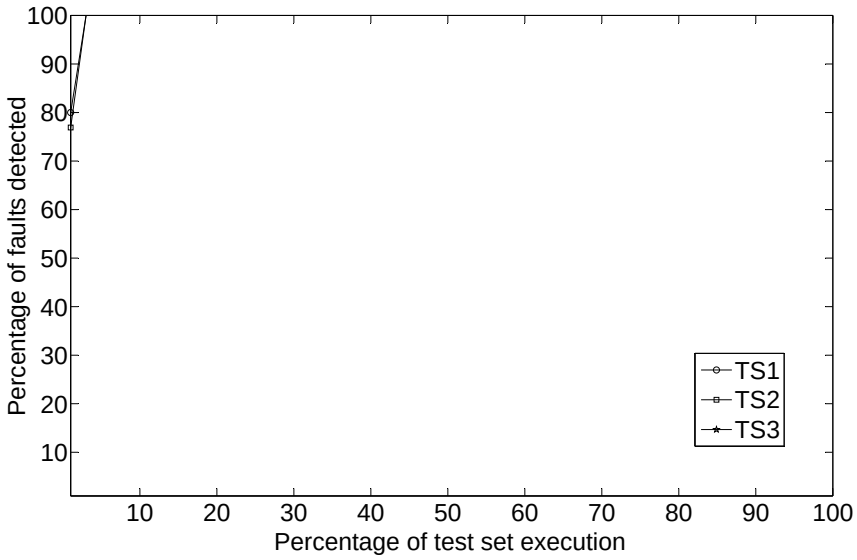

Figure 7.6: Faults detected using prioritisation approach

7.6 Conclusions and future work

We have proposed a novel parallel prioritisation approach for regression testing of complex software applications. We prioritise the test case executions at two levels, by choosing and prioritising the functional modules from the FDG and then ordering the test cases within each test set by using the control flow graphs at the code level. We then distributed the functional modules and their associated test sets among different machines. We measured the performance of our approach using the APFD metric.

We validated the results using various different test combinations and found that our approach is able to detect the faults early and within a small amount of time. In the future, we will validate our results on several other software applications. We will consider real faults with different cost levels. As we have shown, our first 10% execution of test sets detects most of the faults. In the future, we will suggest

a technique that will select the test cases related only to the modified functionalities in software applications and execute only those test cases that will provide maximum fault detection. This may help to reduce the total test execution time even more, as we need to execute only a subset of test cases.

Chapter 8

New metric to detect faults in distributed software development

In the previous chapters, we have suggested techniques to detect faults earlier due to modifications in source code and database of a multi-tier application. We have also suggested a new framework to execute a large number of test cases in parallel on multiple machines.

In this chapter, we suggest a new approach to detect faults in distributed software development environments. In these environments, the software development teams are spread across different locations. The users from different locations raise faults. Due to the rapid development process, it becomes very difficult to test the software application within a specified period of time as this kind of software development requires a tremendous amount of testing. Due to large number of test cases, it becomes very difficult for the software engineers to execute all these test cases and detect issues raised by end-users.

We suggest this approach by dividing the software application into different partitions. We assign weights to the faults according to the severity assigned to the faults. We suggest a new metric that identifies the

severity of a fault and calculates the severity of faults within a partition. We prioritise the test cases according to the rank score obtained from the metric. The new approach helps in detecting faults earlier when the applications are developed and tested in a distributed development environment.

8.1 Our approach

We divide the entire application into different partitions. Each individual partition contains the source code of one web page. The source code includes the design of a web page, database related information and the logic behind that web page. We consider .aspx (extended active server page) file as one partition in a software application. But depending upon the technologies, other methods of dividing the software application could be applied. Our software partition method is a general method that may integrate many of these technologies.

We categorised these faults according to their impacts. We assumed if many source code statements are affected by the fault then that fault has higher severity. The detected faults are distributed among all the software development teams spread across the various locations to resolve these faults. We calculated the fault severity in a partition using our suggested metric and prioritised the test sets and the test cases that belong to a test set using our suggested new prioritisation approach.

We used the *Online bookstore* [1] application to test our approach due to the simplicity of this application. We assume that *Online bookstore* application is based on the distributed software testing and development environment. The other software application that we used is *Moodle* [2]. *Moodle* is an open source community based tool for learning management. *Moodle* is a complex application and has more than 200

[1] available freely at `www.gotocode.com`
[2] available freely at `www.moodle.org`

tables and support many functionalities. This application is developed and tested by many different software engineers.

We generated test cases as described in 2.3. Our approach is independent of test tools. It can be implemented using any of the test case execution tools.

8.1.1 Division of software application into partitions

Modern multi-tiered applications are comprised of different web pages. The test suite is composed of the different test cases of that software application. We suggest our approach by dividing the application into different partitions and each partition corresponds to the group of test cases that are required to test that partition. The test cases that belong to these partitions are further prioritised.

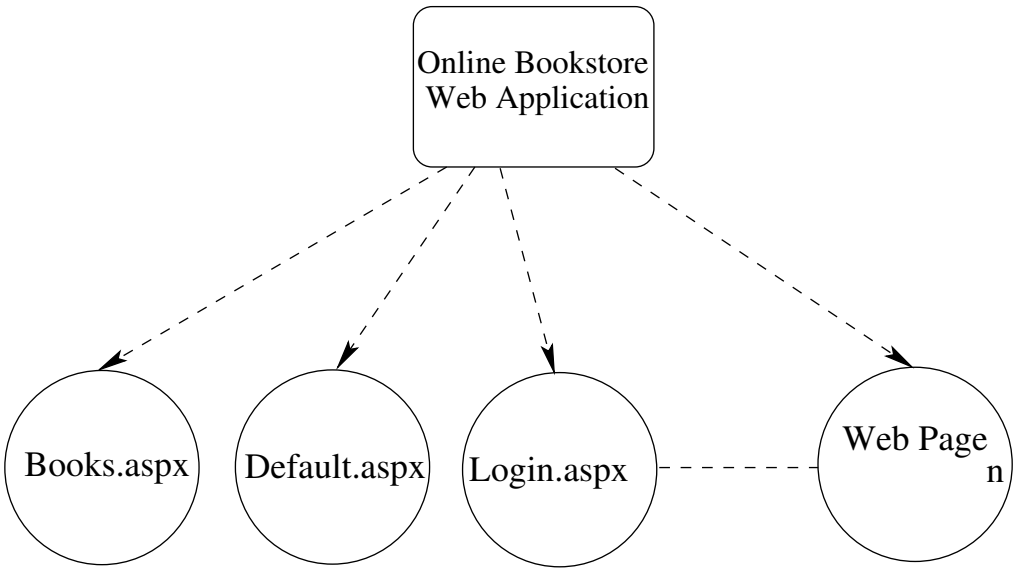

Figure 8.1: Partitioned Software Application (PSA) - *Online bookstore* application

PP stands *for paged partition.* The web page in an application belongs to one partition PP. The partition corresponds to one .aspx file. This

file further contains two different files called .cs (C#) file and designer.cs file. This partition also contains information about the data layer and meta layer related to that particular web page. An example *Partitioned Software Application* (PSA) is shown in Figure 8.1.

8.1.2 Modification of software application

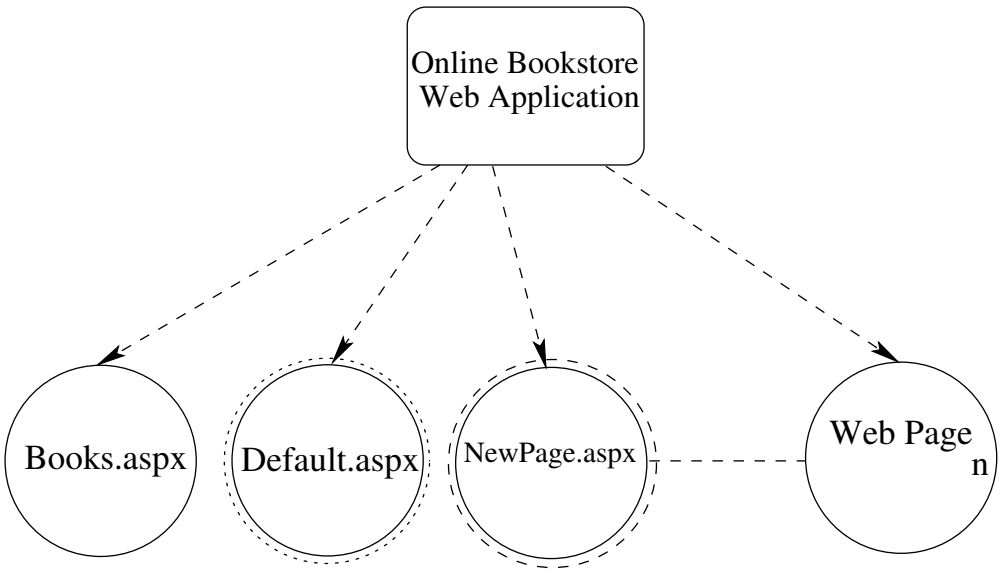

Figure 8.2: Modified Partitioned Software Application (PSA) - *Online bookstore* application

The development of new features and functionalities is an ongoing process in software applications. The requirement of new features from the users or the usage of new technologies results into the modification of software applications. The modifications also relate to the changes in the user interface, database schema or changes in the meta-layer or business layer. These modifications results into the insertion/modification of web pages. These kind of modifications result into new test cases.

Sometimes, while modifying/inserting a new web page, the whole software application becomes non-functional. There are the chances that

already fixed faults appear again or may give rise to new faults. In Figure 8.2, the dashed outline depicts the insertion of a new page and the dotted outline depicts the modification of an existing web page in *Online bookstore application*.

8.2 Faults from the users across different locations

In distributed software applications, the software development and testing teams are deployed at various different locations. The software applications are connected to the internet and are accessed by users from different parts of the world and these users raise different kind of faults. These faults may be related to technical issues, design issues, customised issues or the requirement of new features/enhancements. All of these faults raised by users are stored in a centralised software application called bug tracker. This bug tracker is accessible to all the software development teams that are distributed across different locations. The bug tracker has different privileges assigned to the users and software development teams. The users has limited rights and they can access the bug tracker to raise the issues but the software engineers can access it to resolve those raised issues. Here, we consider an example of *Moodle* where users raise faults using a bug tracker software system [3].

The software development team tracks the issue raised by users and identifies whether it is a fault or enhancement. Sometimes, the end-users raise the same issue many times. Depending upon the raised issue, the software developers act on that particular issue and identifies whether it is a new issue or already raised issue. If the identified issue is a fault, then the software development team assigns the severity to that fault. The severity depends upon many factors, like the impact of fault on other working functionalities, importance of that fault according to the business logic process and the time required to fix that fault. If

[3] `https://tracker.moodle.org/issues`

a fault is blocking all other working functionalities and has a major impact on that application, it is assigned a higher severity and requires immediate attention of software engineers. In our case, we are assigning the severity according to the impact of the fault. If the faults is affecting more lines of source code, it is assigned a higher priority.

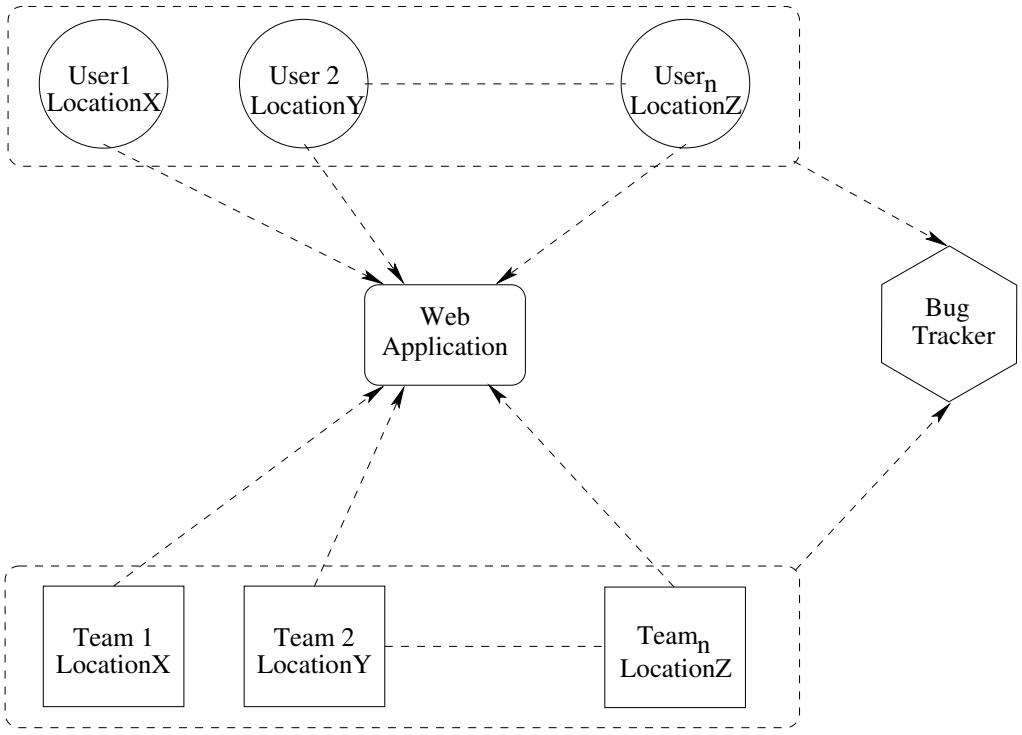

Figure 8.3: Faults raised by different users and software engineering teams across the globe

After identifying the severity of that fault, the fault is assigned to a software engineer that is a part of global software development team. As these software engineers keep on changing from time to time and due to the lack of prior knowledge and understanding about the software architecture process, the software engineers fix these issues. Due to this ignorance of software engineer, it gives rise to a new fault i.e. logical fault. Sometimes the inaccurate implementation of new features or modification of existing features also results into new faults.

Figure 8.3 show the users accessing the software application from various different locations. The teams of software engineers are deployed at different places. Both the users and the software development teams access the same bug tracker software. If the software engineers identified that issue as a part of functional requirements, they give feedback to the user by explaining that functionality otherwise they act on that issue.

8.2.1 Assigning weights to the severity of faults

The severity of a fault indicates the nature of a fault and defines how soon it needs to be fixed [52]. The frequent updates and changes in a software application result into the release of a new version. The users access this new release and report faults. Due to the massive raising of issues in that particular release, it becomes difficult for the software engineers to identify the nature of these faults. Some of these issues require major modifications and makes it difficult for the software engineers to resolve them. Software projects usually have clear guidelines on how to assign a severity to a fault. High severity represents fatal errors and crashes and low severity typically represent cosmetic issues or trivial issues are dependent on the project several intermediate categories as well.

In our approach, we consider blocker faults as the faults that require immediate attention to resolve them as these faults block most of the functionalities in an application. The resolution of these faults result into the resolution of many faults as most of the major or trivial faults are dependent upon the resolution of blocker faults. Critical faults affect some particular components but they don't usually affect other functionalities. Major faults affect some minor portion of an application. Minor faults bothers very few people. Trivial faults are the faults but they does not cause any real problem.

As every new version of an application results into new faults or sometimes the reopening of the existing faults. We collected the faults raised by the users in the previous version of a software application. We identified the severity assigned to that faults through the bug reporting tool or bug tracker. We assign different weights to the faults depending upon the severity of a fault. The weights are ranked as per the criticality of fault. The faults that need urgent attention of the software developer to fix it has been assigned a maximum weight. The faults that are not very urgent to fix as that faults has less impact on the other functionalities. These kind of faults are ranked below than the previous ones and they are assigned less weights. The weights assigned to the different levels of severity are as under:

Table 8.1: Weights to the faults

Fault Type	Assigned Weight
Blocker	5
Critical	4
Major	3
Minor	2
Trivial	1

Table 8.1 shows the weights assigned to the various severity levels. We assigned the weights to the severity according to its nature as *Blocker* is assigned the highest weight of *5* as it blocks most of the functionality and requires urgent attention of the software developers to resolve it. The critical faults are assigned a weight of *4* as they have less impact on the application as compared to blocker faults. The major faults are assigned a weight of *3*. The minor faults are assigned a weight of *2* and trivial faults are assigned *1*. The trivial faults are assigned the least weight as these faults do not cause any real problem.

8.2.2 Metric to calculate the severity of faults

We divided the entire software application into different partitions as in Section 3.1. We assume that if more faults were detected in that

partition in the previous regression testing cycle, then that partition has more chances of faults. Sometimes, the fixing of the faults result into new faults. To investigate the issues raised in a particular partition, we introduced a new metric called Fault Severity Metric *FSM* to calculate the severity of faults in a partition. The metric calculates the total of the severity of all the faults that are related to that partition. The fraction of one partition and the total of the severity of the faults in that particular release in a software application is used to identify the count. The metric is defined as follows:

Fault severity $(PP_i) = (5*Blocker_n) + (4*Critical_n) + (3*Major_n) + (2*Minor_n) + (1*Trivial_n)$

$$P_t = \frac{PP_{\mathrm{i}}}{PP_1 + PP_2 + \cdots + PP_{\mathrm{n}}}$$

In the above metric, PP_i is the calculated severity of faults in a partition. We calculate the fault severity for every partition. P_t is the fraction of one partition and the total of severity of the faults. The values obtained from metric ranges from 0 to 1. P_t with result *1* has more chances of faults and the P_t with result *0* has least chances of faults. We have shown our results obtained from *FSM* in Table 8.2 and Table 8.3.

8.3 Division and execution of test suite into test sets

As shown in Figure 8.1, we divided the entire application into different partitions. We divided the entire test suite into different test sets. Each test set contains test cases related to a partition. The motive of dividing the entire test suite into test sets is to test the source code that has more chances of faults earlier in the test suite execution. The test cases in a test set are further prioritised as per our suggested prioritisation approach *(explained in Section 8.4)*.

We divide the entire test suite execution of a software application into two different test processes. We automatically execute these two different test processes simultaneously on two different machines. The first test process deals with the test cases that are selected and prioritised using our new suggested approach and these test cases are considered as a part of the *active test execution process*. The first test process is considered as a part of the front-end activity in test suite execution. The results from the front-end activity are reported in the active test suite execution report.

The second test process includes the remaining test cases. These remaining test cases are considered as a part of the back-end test suite execution process. The results from the back-end test suite execution process are not reported in the *active test execution report* but these results are stored for future reference for understanding the effect of the overall test suite execution.

8.4 Prioritisation approach

Prioritisation of test cases is to reorder test cases in an order to achieve some specific goal like early detection of faults, better code coverage or reduction of test suite execution time. Although, there are many prioritisation approaches are suggested in the literature to fulfil these goals but we suggest a new approach that is based on our metric and reorder test cases to detect faults early in distributed software applications.

We suggest a new prioritisation approach *fault severity prioritisation fsp* to prioritise the entire test suite of a software application. We prioritise the test sets according to the rank score obtained from our metric. We further reschedule the test cases as per the detection of faults in that particular partition. The test sets are prioritised as under:

1. The test cases that belong to the test set with highest rank score according to the metric are executed first.

2. The remaining test sets are prioritised as per the score obtained from the matric until the least metric score. If two or more partitions have the same metric score then they are assigned the priorities in an arbitrary order.

3. All the remaining test sets are prioritised randomly.

The test cases inside the test set are further prioritised as follows:

1. The test case that detected maximum faults in the previous test execution cycle are executed in priority in a test set.

2. If two or more test cases detected the same number of faults in the previous test execution are prioritised in an arbitrary order.

3. If the test case has not detected any fault in the previous test execution cycle are executed randomly.

We prioritise all the test cases using our new prioritisation approach *fsp*. We reorder the test cases using *fsp* for both of these software applications and compare our results using the prioritisation techniques suggested in existing state of art.

8.5 Experimental evaluation

In this section, we detect faults using various prioritisation strategies and evaluate which prioritisation strategy detects faults early in the test suite execution. We discuss our experimental set up using our suggested approach. To perform the experimental evaluation, we used two different applications, *Online bookstore* and *Moodle*. We have modified some of the functionalities of *Online bookstore* to make them more complex. We used *Moodle 2.4 stable* version for our experiments. We used *Moodle 1.7 stable* version to make use of the modified functionalities between these two different versions. We identified the faults raised in both of these versions.

Online bookstore application is an online shopping portal for buying books. This application uses ASPX for its frontend and MySQL for its backend connectivity. The application allows the users to search for books by different keywords, add to the shopping cart and proceed to orders.

Moodle is an abbreviation for *Modular Object-Oriented Dynamic Learning Environment*. It is a free open-source e-learning software. This application uses PHP and MySQL for this backend connectivity. This application has a huge database comprising more than 200 tables. This application allows the users to submit their assignments and grades and to enrol students in their courses, in addition to many other services associated with a learning management system.

We randomly seeded 10 possible database faults in various functionalities of *Online bookstore*. We seeded 10 possible different faults in the various different functionalities of *Moodle*.

We used C# to implement our proposed approach. We generated 130 functional test cases for the *Online bookstore* and converted them to C# test scripts readable by the Selenium test tool for automatic test suite execution [92]. The generated test cases were assumed to be non-redundant and were generated according to the functional specifications. We generated 260 test cases for *Moodle* using Java. We detect the faults using our suggested prioritisation approach for test cases.

We initially seeded 10 faults. Five different kinds of faults were seeded in the application: Blocker Faults, Critical Faults, Major Faults, Minor Faults and Trivial Faults. We detected all these 10 faults in test suite execution.

Table 8.2 shows the results obtained from metric *FSM* for *Online bookstore*. The partition names are assigned according to the webpages as we have assigned *MEM* to Members webpage. In Table 8.2, the partitions *MEM* and *RT* show the same result multiple times as these faults belong to the same partition.

Table 8.2: Results from FSM metric (Online bookstore)

Faults	**Partition affected**	**Severity of Fault**	**Weight assigned**	**Result from FSM**
F1	SC	Minor	2	0.068
F2	MI	Trivial	1	0.068
F3	MEM	Major	3	0.344
F4	MEM	Critical	4	0.344
F5	CT	Blocker	5	0.172
F6	MEM	Major	3	0.344
F7	REG	Blocker	5	0.172
F8	RT	Minor	2	0.103
F9	RT	Trivial	1	0.103
F10	CAT	Major	3	0.103

Table 8.3 shows the results obtained from metric *FSM* for *Moodle*. In Table 8.3, the partitions *RL* and *PR* show the same result multiple times as the faults belong to the same partition.

We ordered the test sets according to the score obtained from Table 8.2 for *Online bookstore*. The test sets belong to *MEM* are prioritised first and *SC* are prioritised at the last position for *active test execution process*. For *Moodle*, we prioritise the test sets from the score obtained from Table 8.3, the test cases belong to partition *RL* are executed in priority and the test cases that belong to partition *NV* are scheduled at the last position for *active test execution process*.

Evaluation Metrics: We collected the test execution results and used the APFD metric as described in Section 2.7 to evaluate our approach and compare results with the existing state of art. We used the APFD metric to determine whether our approach detects faults earlier and faster as compared to other prioritisation approaches.

Threats to Validity: We manually seeded the faults. The faults severity depends upon the different applications in a real environment.

Table 8.3: Results from FSM metric (Moodle)

Faults	**Partition affected**	**Severity of Fault**	**Weight assigned**	**Result from FSM**
F1	CR	Blocker	5	0.142
F2	BP	Major	3	0.0857
F3	RL	Blocker	5	0.257
F4	PR	Critical	4	0.228
F5	RL	Critical	4	0.257
F6	PR	Critical	4	0.228
F7	CH	Minor	2	0.057
F8	GR	Critical	4	0.114
F9	NV	Trivial	1	0.028
F10	UP	Major	3	0.085

The partitions may vary in size depending upon the size of the webpage. The functional test case execution time may differ due to the varying lengths of test cases.

8.6 Results and analysis

To show the validity of our results, we compared our new prioritisation approaches with the existing prioritisation approaches in the literature. We use prioritisation techniques suggested in [80] [85] to validate our results. We show the results using the techniques: *fsp* and the existing prioritisation approaches: *random*, *stmt-total*. In *random*, we consider the random ordering of test cases [80] [85]. In *stmt-total*, we prioritise on the basis of coverage of statements [80].

We show our results in Table 8.4 and Table 8.5. Table 8.4 shows the test execution results using various prioritisation techniques on *Online bookstore*. Table 8.5 shows the test execution results using various pri-

oritisation techniques on *Moodle*. Table 8.4 and Table 8.5 shows the results in steps of 10% increments. These results were collected using the APFD metric.

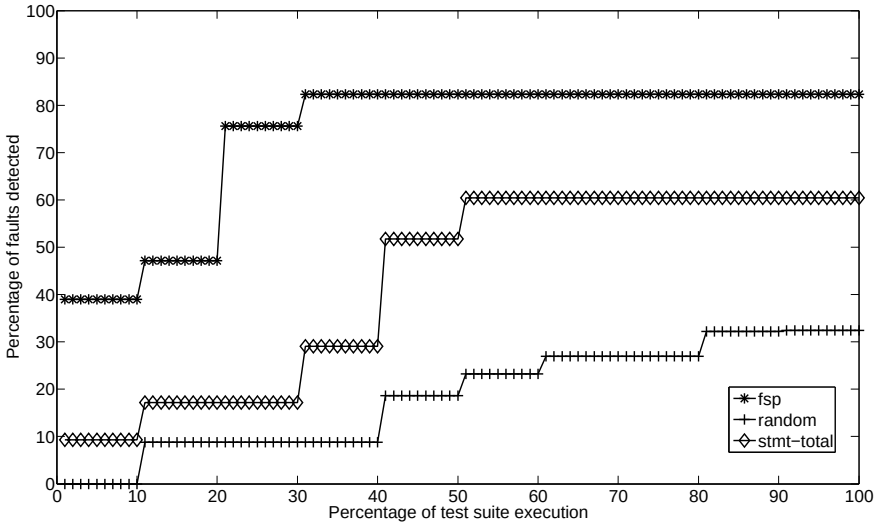

Figure 8.4: Faults detected using various prioritisation techniques *(Online bookstore)*

Figure 8.4 shows the faults detected in *Online bookstore* using three different prioritisation techniques. Our new suggested prioritisation approach *fsp* delivers the highest APFD of 82.29 in 100% test suite execution. More than 30% of the faults were detected in the first 10% test suite execution using *fsp*. The *random* approach attains the lowest APFD of 32.39 among all the three prioritisation approaches. *stmt-total* attains an APFD of 60.46. Surprisingly, *random* was not able to detect any faults in the first 10% of test suite execution.

Figure 8.5 shows the faults detected in *Moodle* using three different prioritisation approaches. Our new prioritisation technique *fsp* has shown the highest APFD of 89.03 for 100% test suite execution. *fsp* detected more than 30% of faults in first 10% of test suite execution and performed better among all these prioritisation approaches. *stmt-total* delivered the APFD of 85.15 in 100% test suite execution. The *random*

Table 8.4: Results from prioritisation techniques (Online bookstore)

%age of test suite run	**fsp**	**random**	**stmt-total**
10%	38.96	0	9.28
20%	47.13	8.80	17.21
30%	75.62	8.80	17.21
40%	82.29	8.80	29.03
50%	82.29	18.64	51.73
60%	82.29	23.24	60.46
70%	82.29	26.97	60.46
80%	82.29	26.97	60.46
90%	82.29	32.21	60.46
100%	82.29	32.39	60.46

Table 8.5: Results from prioritisation techniques (Moodle)

%age of test suite run	**fsp**	**random**	**stmt-total**
10%	38.89	9.33	28.94
20%	73.47	43.87	62.77
30%	89.03	73.56	78.45
40%	89.03	73.56	85.15
50%	89.03	73.56	85.15
60%	89.03	77.57	85.15
70%	89.03	77.57	85.15
80%	89.03	77.57	85.15
90%	89.03	77.57	85.15
100%	89.03	77.57	85.15

attained the APFD of 77.57. Random detected the least number of

faults in first 10% of test suite execution among all these prioritisation approaches.

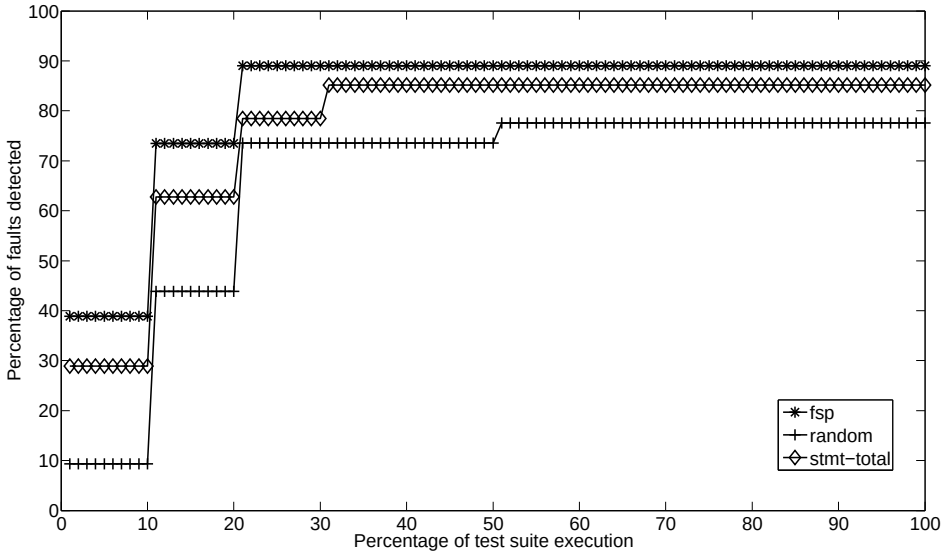

Figure 8.5: Faults detected using various prioritisation techniques *(Moodle)*

8.7 Conclusions and future work

We have proposed a general approach that can be used for regression testing of any application based on the distributed software development and testing. Furthermore, our approach is applicable to all kinds of test cases, not only for those based on the Selenium tool.

We have shown the results of prioritisation technique using two different applications. *Moodle* is a widely used learning management application and *Online bookstore* is a simple and small application and has already been used in the literature. For *Moodle*, we used JUnit test cases with Selenium and for *Online bookstore*, we used NUnit test cases with Selenium. The *fsp* approach has delivered good results for both of these applications and has attained the highest APFD. We conclude that al-

ready resolved faults sometimes result into new faults. The parts of the application that has more blocker faults have more chances of introducing new faults while resolving old faults. We validated our results using various test combinations. In future, we will validate these results on other complex applications. We will consider more complex distributed testing and development scenarios.

Chapter 9

General conclusions and future work

This chapter highlights the problems that we attempted to solve. We also revisit some of the assumptions and limitations of the approaches proposed in this thesis, discussing opportunities for future work.

9.1 Summary of contributions

In this thesis, we proposed and presented new techniques for prioritisation and selection of test cases. We suggested new techniques to detect faults due to modifications in source code and database in multi-tier software applications. The suggested techniques help in detecting faults early. Moreover, the new techniques have shown good results that provide better test coverage than the techniques available in literature. The new techniques reduce test suite execution time. Hence, the techniques help in reducing software testing costs.

The main components and contributions of this thesis are summarised below:

1. Two-level approach to detect faults earlier due to modifications in source code:

   The two-level approach is based on the FDG and CFG graphs. The FDG is a collection of functional modules and is based on the application functional specifications and is independent of the application program code. The CFG is the graph extracted from the source code of the functional module. CFG is dependent on the code of the programming language that is used to develop that module. FDG is the combination of various CFGs and help software engineers to identify the dependency of various source code modules. This further helps in automatic test suite execution of software application written using multiple programming languages. This technique helps in detecting the faults earlier.

2. Selection and prioritisation of test cases due to modifications in source code:

   This approach selects and prioritises the test cases when the mapping between test cases and source code is known or it is possible to obtain the mapping. This new approach is based on matching in bipartite graphs. This approach helps in detecting the modified parts of the source code and selects the subset of test cases that are related to the modified source code. The selected subset of test cases is further prioritised on the basis of the modifications in source code. This approach helps in earlier fault detection when are faults are due to modifications in the source code. This technique helps in reducing test suite execution time and provides better test coverage as compared to the existing techniques.

3. Selection and prioritisation of test cases to detect faults early due to modifications in database:

   This approach automatically selects a subset of test cases from the entire test suite. This approach selects only those test cases that are related to the database of a multi-tier application. This approach helps in determining the modifications in database and

the program source code that is affected by the modifications in database. The selected test cases are further prioritised on the basis of degree of modifications in the database. This technique helps in detecting the faults earlier due to database within a short time.

4. Prioritisation of test cases to detect faults due to modifications in database:

   This approach is useful when the mapping between test cases and source code is unknown or it is not possible to obtain such a mapping. This approach is based on FDG and CFG of a software application. This is based on a relationship between FDG and database schema. This approach extracts database schema of an application and detects the modifications in a database with the help of the extracted database schema. The modifications in the database helps in prioritising the test cases. The test cases related to new/modified tables in a database are executed earlier. This novel approach helps in detecting the faults that are related to modifications in a database earlier.

5. Parallel execution of test cases:

   Due to the increase in size of a software application, it becomes very difficult to execute a large number of test cases on a single machine. This technique automatically partitions the set of test cases and distributes them among different machines. The distributed test cases are executed simultaneously on different machines. Using this approach, a large number of test cases can be executed within a short time. This approach helps in detecting the faults earlier.

6. New metric to prioritise test cases in distributed software development and testing:

   Most modern software applications are developed using distributed software development environments. Due to the massive develop-

ment costs, it becomes very difficult to test the software applications as the users from different locations detect the faults. To overcome this issue, we have suggested a new metric to prioritise the test cases based on the ranking score obtained from that metric. The metric calculates the severity of the faults in a partition and reorders the test cases based on the score obtained from the metric. This approach helps in detecting faults earlier when the applications are developed and tested in distributed software development environments.

9.2 Future work

There are various avenues of future work. There is a need to perform these experiments by setting up more complex applications using real environments to understand the results in a more realistic and better way. The experiments can be performed using real faults instead of seeded faults.

The extraction of FDG (Functional Dependency Graph) can be automated and can be used for partitioning of any software application based on functional specifications. The functional modules in FDG can be further broken down into ICG's and ICG can be used to generate graph of a particular module as it is based on the source code of that particular functional module. FDG is comprised of various ICG's and can be used to test the software applications that are comprised of modules built using different programming languages.

FSM metric can be applied on more complex and large applications that are developed using distributed software development and testing. This helps in the realistic understanding of the results obtained using that metric.

9.3 Concluding remarks

In this thesis, we have demonstrated that our new prioritisation and selection approaches detect faults early. The faults may be related to modifications in source code or database. We have evaluated these techniques using various test combinations and compared our techniques with the existing approaches that are available in literature. Most of these techniques were able to detect faults in less than 30% of test suite execution. Our suggested techniques can be applied to any kind of software applications that are developed using any programming language. Our techniques can be applied for the selection and prioritisation of test cases that are written in any format. The suggested techniques can be fully automated. The techniques help in early detection of faults, reduce test suite execution time and provide better test coverage.

Bibliography

[1] B. B. Agarwal. *Software engineering and testing, an introduction*, volume 1. Jones and Bartlett Publishers, June 2010.

[2] N. Alshahwan, M. Harman, A. Marchetto, and P. Tonella. Improving web application testing using testability measures. In *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 49 –58, September 2009.

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

[4] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.

[5] Sandler Corey; Glenford J Myers ; Tom Badgett. *The Art of Software Testing*. Hoboken : John Wiley & Sons, Inc., 2004.

[6] Luciano Baresi and Mauro Pezzè. An introduction to software testing. *Electron. Notes Theor. Comput. Sci.*, 148:89–111, February 2006.

[7] A. Beer and R. Ramler. The role of experience in software testing practice. In *Proceedings of the 34th Euromicro Conference*

on Software Engineering and Advanced Applications, SEAA '08, pages 258 –265, September 2008.

[8] A. Bruns, A. Kornstadt, and D. Wichmann. Web application tests with selenium. In *Software, IEEE*, volume 26, pages 88 –91, September-October 2009.

[9] Renee C. Bryce, Sreedevi Sampath, and Atif M. Memon. Developing a single model and test prioritization strategies for event-driven software. In *IEEE Transactions on Software Engineering*, volume 37, pages 48–64, Los Alamitos, CA, USA, January-February 2011. IEEE Computer Society.

[10] R. Carlson, Hyunsook Do, and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 382 –391, September 2011.

[11] E.G. Cartaxo, F.G.O. Neto, and P.D.L. Machado. Test case generation by means of uml sequence diagrams and labeled transition systems. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, ISIC*, pages 1292 –1297, October 2007.

[12] S.S. Chakraborty and V. Shah. Towards an approach and framework for test-execution plan derivation. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 488 –491, November 2011.

[13] S.R. Choudhary, H. Versee, and A. Orso. A cross-browser web application testing tool. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1 –6, September 2010.

[14] S.R. Clark, J. Cobb, G.M. Kapfhammer, J.A. Jones, and M.J. Harrold. Localizing sql faults in database applications. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 213 –222, November 2011.

[15] Kate Connolly. 2010 bug hits millions of germans. http://www.guardian.co.uk/world/2010/jan/06/2010-bug-millions-germans, January 2010.

[16] G.A. Di Lucca, A.R. Fasolino, F. Faralli, and U. De Carlini. Testing web applications. In *Proceedings of the International Conference on Software Maintenance*, pages 310 – 319, 2002.

[17] G.A. Di Lucca, A.R. Fasolino, P. Tramontana, and U. De Carlini. Abstracting business level uml diagrams from web applications. In *Proceedings of the Fifth IEEE International Workshop on Web Site Evolution*, pages 12 – 19, September 2003.

[18] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche. Coverage-based test case prioritisation: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pages 302–311, 2013.

[19] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. In *Empirical Softw. Engg.*, volume 11, pages 33–70, Hingham, MA, USA, March 2006. Kluwer Academic Publishers.

[20] T. Eilam, M.H. Kalantar, A.V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the configuration complexity of distributed applications in internet data centers. *IEEE Communications Magazine*, 44(3):166 – 177, March 2006.

[21] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, pages 49 – 59, May 2003.

[22] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. In *IEEE Transactions on Software Engineering*, volume 28, pages 159 –182, Piscataway, NJ, USA, February 2002. IEEE Press.

[23] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[24] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 102–112, New York, NY, USA, 2000. ACM.

[25] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. In *Software Quality Control*, volume 12, pages 185–210, Hingham, MA, USA, September 2004. Kluwer Academic Publishers.

[26] H. Freeman. Software testing. volume 5, pages 48 – 50, September 2002.

[27] A. Gantait. Test case generation and prioritization from uml models. In *Proceedings of the 2011 Second International Conference on Emerging Applications of Information Technology (EAIT)*, pages 345 –350, February 2011.

[28] D. Garg and A. Datta. Test case prioritization due to database changes in web applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 726 –730, April 2012.

[29] D. Garg and A. Datta. Early detection of faults related to database schematic changes. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*, JAMAICA 2013, pages 1–6, New York, NY, USA, 2013. ACM.

[30] D. Garg and A. Datta. Parallel execution of prioritized test cases for regression testing of web applications. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135*, ACSC '13, pages 61–68, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc.

[31] D. Garg, A. Datta, and T. French. New test case prioritization strategies for regression testing of web applications. *International Journal of System Assurance Engineering and Management*, 3(4):300–309, 2012.

[32] D. Garg, A. Datta, and T. French. A two-level prioritization approach for regression testing of web applications. In *19th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 150–153, 2012.

[33] D. Garg, A. Datta, and T. French. A novel bipartite graph approach for selection and prioritisation of test cases. *ACM SIGSOFT Softw. Eng. Notes*, 38(6):1–6, November 2013.

[34] Alberto Gonzalez-Sanchez, ric Piel, Rui Abreu, Hans-Gerhard Gross, and Arjan J. C. van Gemund. Prioritizing tests for software fault diagnosis. In *Software: Practice and Experience*, pages 42–51. John Wiley & Sons, Ltd., April 2011.

[35] Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[36] Yuepu Guo and Sreedevi Sampath. Web application fault classification - an exploratory study. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 303–305, New York, NY, USA, 2008. ACM.

[37] Florian Haftmann, Donald Kossmann, and Eric Lo. Parallel execution of test runs for database application systems. In *Proceed-*

ings of the 31st international conference on Very large data bases, VLDB '05, pages 589–600. VLDB Endowment, 2005.

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

[39] William G. J. Halfond and Alessandro Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 145–154, New York, NY, USA, 2007. ACM.

[40] Mary Jean Harrold. Testing: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 61–72, New York, NY, USA, 2000. ACM.

[41] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for Java software. *SIGPLAN Not.*, 36:312–326, October 2001.

[42] A. Heinecke, T. Bruckmann, T. Griebe, and V. Gruhn. Generating test plans for acceptance tests from uml activity diagrams. In *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pages 57 –66, March 2010.

[43] Pierre Henry. *The Testing Network: An Integral Approach to Test Activities in Large Software Projects*. Springer, 1 edition, October 2008.

[44] John E. Hopcroft and Richard M. Karp. A n5/2 algorithm for maximum matchings in bipartite. In *12th Annual Symposium on Switching and Automata Theory*, pages 122–125, 1971.

[45] Randal Jackson. Software bug mixes patient health data. http://computerworld.co.nz/news.nsf/news/software-bug-mixes-patient-health-data, April 2010.

[46] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *Proceedings of the 30th Annual International Computer Software and Applications Conference, COMPSAC '06*, volume 1, pages 411 –420, September 2006.

[47] D. Jeffrey and Neelam Gupta. Test suite reduction with selective redundancy. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM'05*, pages 549 – 558, September 2005.

[48] M.I. Kayes. Test case prioritization for regression testing based on fault dependency. In *3rd International Conference on Electronics Computer Technology (ICECT)*, volume 5, pages 48–52, 2011.

[49] Alireza Khalilian, Mohammad Abdollahi Azgomi, and Yalda Fazlalizadeh. An improved method for test case prioritization by incorporating historical test case data. *Science of Computer Programming*, 78(1):93 – 116, 2012.

[50] E. Kirda, M. Jazayeri, C. Kerer, and M. Schranz. Experiences in engineering flexible web services. *Multimedia, IEEE*, 8(1):58 –65, January-March 2001.

[51] B. Korel, L.H. Tahat, and M. Harman. Test prioritization using system models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.*, pages 559 – 568, September 2005.

[52] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 1 –10, May 2010.

[53] Alexey Lastovetsky. Parallel testing of distributed software. *Inf. Softw. Technol.*, 47:657–662, July 2005.

[54] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 417–420, New York, NY, USA, 2007. ACM.

[55] H.K.N. Leung and L. White. Insights into regression testing [software testing]. In *Proceedings of the Conference on Software Maintenance*, pages 60 –69, October 1989.

[56] H.K.N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance*, pages 290 –301, November 1990.

[57] William E. Lewis. *Software Testing and Continuous Quality Improvement*. Taylor & Francis, 2008.

[58] Gen Li, Kai Lu, Ying Zhang, Xicheng Lu, and Wei Zhang. Decoupling binary-level dynamic test generation from specific architecture details. In *Proceedings of the Fourth International Conference on Computer Sciences and Convergence Information Technology, ICCIT '09*, pages 1041 –1046, November 2009.

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

[60] Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. In *Quality Assurance and Testing of Web-Based Applications*, volume 48, pages 1172 – 1186, December 2006.

[61] Y.K. Malaiya, Naixin Li, J. Bieman, R. Karcich, and B. Skibbe. The relationship between test coverage and reliability. In *Pro-*

ceedings of 5th International Symposium on Software Reliability Engineering, pages 186 –195, November 1994.

[62] C. Malz, N. Jazdi, and P. Gohner. Prioritization of test cases using software agents and fuzzy logic. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 483 –486, April 2012.

[63] Nashat Mansour and Manal Houri. Testing web applications. In *Information and Software Technology*, volume 48, pages 31 – 42, 2006.

[64] Paul K. ; Walters Gene F. McCall, Jim A. ; Richards. Factors in software quality. Technical Report ADA049014, General Electric Co Sunnyvale CA, November 1977. Final technical rept. Aug 1976-Jul 1977.

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

[66] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[67] Siavash Mirarab and Ladan Tahvildari. A prioritization approach for software testcases based on bayesian networks. In MatthewB. Dwyer and Antnia Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, pages 276–290. Springer Berlin Heidelberg, 2007.

[68] A. Nanda, S. Mani, S. Sinha, M.J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, pages 21 –30, March 2011.

[69] A.J. Offutt. Software testing: from theory to practice. In *Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS '97 'Are We Making Progress Towards Computer Assurance?'*, pages 48 –51, June 1997.

[70] Chhabi Rani Panigrahi and Rajib Mall. Model-based regression test case prioritization. *SIGSOFT Softw. Eng. Notes*, 35(6):1–7, November 2010.

[71] ChhabiRani Panigrahi and Rajib Mall. An approach to prioritize the regression test cases of object-oriented programs. *CSI Transactions on ICT*, pages 1–15, 2013.

[72] Prem Parashar, Arvind Kalia, and Rajesh Bhatia. Pair-wise time-aware test case prioritization for regression testing. In Sumeet Dua, Aryya Gangopadhyay, Parimala Thulasiraman, Umberto Straccia, Michael Shepherd, and Benno Stein, editors, *Information Systems, Technology and Management*, volume 285 of *Communications in Computer and Information Science*, pages 176–186. Springer Berlin Heidelberg, 2012.

[73] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Proceedings of the 15th International Conference on Software Engineering*, pages 287 –301, May 1993.

[74] Jan Ploski, Matthias Rohr, Peter Schwenkenberg, and Wilhelm Hasselbring. Research issues in software fault categorization. In *ACM SIGSOFT Software Engineering Notes*, volume 32, New York, NY, USA, November 2007. ACM.

[75] U. Praphamontripong and J. Offutt. Applying mutation testing to web applications. In *Proceedings of the Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 132 –141, April 2010.

[76] Bo Qu, Changhai Nie, and Baowen Xu. Test case prioritization for multiple processing queues. In *Proceedings of the International*

Symposium on Information Science and Engineering, ISISE '08, volume 2, pages 646 –649, December 2008.

[77] F. Ricca and P. Tonella. Web testing: a roadmap for the empirical research. In *Proceedings of the Seventh IEEE International Symposium on Web Site Evolution, (WSE 2005)*, pages 63 – 70, September 2005.

[78] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[79] E. Rogstad, L. Briand, R. Dalberg, M. Rynning, and E. Arisholm. Industrial experiences with automated regression testing of a legacy database application. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 362 –371, September 2011.

[80] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. In *IEEE Transactions on Software Engineering*, volume 27 of *10*, pages 929 –948, October 2001.

[81] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. In *IEEE Transactions on Software Engineering*, volume 22 of *8*, pages 529–551, Piscataway, NJ, USA, August 1996. IEEE Press.

[82] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[83] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, 2000.

[84] Mazeiar Salehie, Sen Li, Ladan Tahvildari, Rozita Dara, Shimin Li, and Mark Moore. Prioritizing requirements-based regression test cases: A goal-driven practice. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329 –332, March 2011.

[85] S. Sampath, R.C. Bryce, G. Viswanath, V. Kandimalla, and A.G. Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, pages 141 –150, April 2008.

[86] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A.S. Greenwald. Applying concept analysis to user-session-based testing of web applications. In *IEEE Transactions on Software Engineering*, volume 33 of *10*, pages 643 –658, October 2007.

[87] P. Samuel, R. Mall, and S. Sahoo. Uml sequence diagram based testing using slicing. In *2005 Annual IEEE, INDICON*, pages 176 – 178, December 2005.

[88] Hema Srikanth, Laurie Williams, and Jason Osborne. System test case prioritization of new and regression test cases. In *ISESE*, pages 64–73, 2005.

[89] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 97–106, New York, NY, USA, 2002. ACM.

[90] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *22nd IEEE International Conference on Software Maintenance, 2006. ICSM '06*, pages 123–133, 2006.

[91] Paolo Tonella and Filippo Ricca. Statistical testing of web applications. In *Journal of Software Maintenance and Evolution:*

Research and Practice - Special issue: Web site evolution, volume 16 of *1-2*, pages 103–127. John Wiley & Sons, Ltd., 2004.

[92] A.-M. Torsel. Automated test case generation for web applications from a domain specific model. In *Proceedings of the IEEE 35th Annual Conference on Computer Software and Applications Conference Workshops (COMPSACW)*, pages 137 –142, July 2011.

[93] Sagar Naik; Piyu Tripathy. *Software Testing and Quality Assurance*. John Wiley & Sons, Inc., 2008.

[94] Yuan-Hsin Tung, Shian-Shyong Tseng, Tsung-Ju Lee, and Jui-Feng Weng. A novel approach to automatic test case generation for web applications. In *Proceedings of the 10th International Conference on Quality Software (QSIC)*, pages 399 –404, July 2010.

[95] Arie van Deursen and Ali Mesbah. Research issues in the automated testing of ajax applications. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorn, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *Lecture Notes in Computer Science*, pages 16–28. Springer Berlin / Heidelberg, 2010.

[96] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 1–12, New York, NY, USA, 2006. ACM.

[97] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of The Eighth International Symposium On Software Reliability Engineering*, pages 264 –274, 2-5 1997.

[98] Ye Wu and Jeff Offutt. Modeling and testing web-based applications. Technical report, George Mason University, 2002.

[99] Dianxiang Xu. A tool for automated test code generation from high-level petri nets. In *Proceedings of the 32nd international conference on Applications and theory of Petri Nets*, PETRI NETS'11, pages 308–317, Berlin, Heidelberg, 2011. Springer-Verlag.

[100] Mu Xuequan. Google says software bug causing email service failure. http://news.xinhuanet.com/, March 2011.

[101] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201, Piscataway, NJ, USA, 2013. IEEE Press.

[102] Xiaoyu Zheng and Mei-Hwa Chen. Maintaining multi-tier web applications. In *IEEE International Conference on Software Maintenance*, pages 355 –364, October 2007.

[103] T. Zimmermann and N. Nagappan. Predicting subsystem failures using dependency graph complexities. In *Proceedings of the 18th IEEE International Symposium on Software Reliability, ISSRE '07*, pages 227 –236, November 2007.