Estimating and Improving the Performance of Prediction Models for Regression Test Selection

by

Weining Liu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

© Weining Liu 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Researchers have proposed models to predict the percentage of the selected regression test cases when a Regression Test Selection (RTS) technique is used. One of the most successful and best performing RTS predictors is the Rosenblum and Weyuker (RW) coverage-based prediction model. However, previous evaluation results on the RW predictor show that although it performs well on some subject programs, it deviates from actual percentage significantly on others. To understand the factors impacting he RW predictor's performance, this thesis presents a set of experiments on four factors that can potentially impact the RTS prediction performance. We setup two different set of experiments on several Java open-source test subjects and three RTS techniques. Our study on the effect of each factor on the RW performance reveals that large amount of code changes and significant code coverage overlaps between test cases are the two factors contributing to the RW predictor's prediction error. Based on the experimental results and through regression analysis of the impacting factors, we propose a RW error estimator that can help testers and developers to gain a better understanding of RW predictor's confidence level and get insight into the applicability of the RW predictor to different organizations products and processes.

To further improve the RW predictor's performance, we propose an improved RW prediction model utilizing the error estimator to compensate the prediction error. We also present a RTS technique-specific improvement to the RW predictor and evaluate Harrold *et al.*'s weighted predictor which incorporates change history. Our experiments on these improved RW predictors demonstrate that they can reduce the RW prediction error and improve performance.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Professor Ladan Tahvildari for her guidance and support over this long journey. Her advice and encouragement have made this work possible.

Special thanks to Professor Lin Tan and Professor Reid Holmes for their valuable time reading this thesis and providing feedback.

I acknowledge my gratitude to all the researchers I have collaborated with, especially Dr. Mehdi Amoui.

Dedication

This is dedicated to my family, for their support and lots of sacrifices.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today, customers expect a reliable and up to date service anywhere, anytime. To meet customers expectations, software organizations strive to update existing services or delivery new features promptly and continuously. One of the new approaches in the industry to deliver high-quality, low-cost software in a continuous pipeline is Continuous Delivery (CD) [18]. The basic idea behind CD is to develop a software product through many small cycles and the product must be in a high quality state to be released to customers at the end of each cycle. CD is extended from Continuous Integration with rigorous testing on each build. For organizations adopted CD practices, regression testing plays a critical role in the software development process.

Regression testing is the process of running existing test suite to validate that the behaviours of software are not adversely impacted by software changes. Regression testing is a frequently executed activity and can account for significant portion of software development costs. In CD, regression testing is becoming more frequent and costly than ever. Figure 1.1 depicts the regression testing cycle in a CD environment. Typically in CD, a new build is created whenever there is a commit to the code repository and regression testing is performed against the new build to confirm the new build has met the quality bar for release. As depicted in Figure 1.1, there are test and development cycles run in parallel. In the development cycle a new build is created as soon as code is committed, while in the test cycle each new build is regression tested. The duration between two consecutive commits is the maximum time allocated for regression testing.

The classic approach of regression testing is to execute entire regression test suite(retest-all). However, CD practices pose new challenges to the retest-all approach. First, there are typically many commits which produce many builds to be regression tested. As a result,

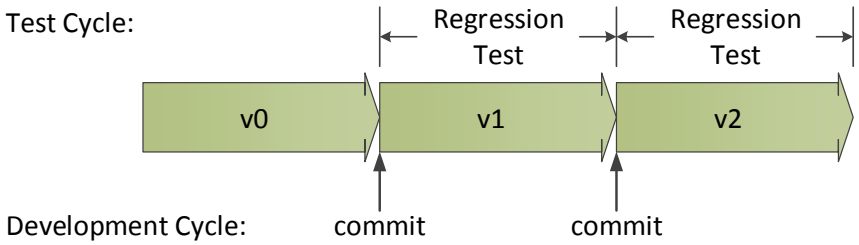

Figure 1.1: The Regression Testing Cycle in Continuous Delivery Environment

the overall regression effort increases dramatically as the total number of builds increases. Second, as the commit becomes more frequent, the allocated regression testing time gets shorter. It could reach to a point that there isn't enough time to execute entire regression test suite between two commits. For example, Google has 6000 engineers work on the same code base and there are over 20 commits per minute [22]. The allocated time for each build is only 3 seconds which is challenging for completing any average size test suite.

One strategy to deal with these challenges is to choose an appropriate subset of test cases from existing test suite. This strategy is called Regression Test Selection (RTS). Most RTS techniques are modification-aware, which means test cases are selected based on changes between program revisions. Empirical evaluation results of RTS techniques indicate that these techniques can be very effective in reducing the size of the test suite while maintaining the effectiveness of the original test suite. Thus, it is beneficial for the organizations to adopt RTS techniques in their testing practices.

However, organizations adopted RTS techniques are facing a new challenge of estimating the regression testing effort. When retest-all approach is used, test teams can provide reasonable estimate based on previous execution effort of the entire test suite. However, with RTS techniques in place, the number of tests selected for regression is unknown until the technique is applied to the test suite. Since most RTS techniques require code change information, the number of tests selected for regression is unknown until the code change is committed. Consequently, prior to commit, test managers would have to estimate the regression testing effort without knowing how many tests need to be executed in the regression testing cycle. The accuracy of the estimation can greatly impact software project's success. Underestimating regression testing effort may force organizations to delay a committed release date; Overestimating regression testing effort may cause organizations to miss invaluable market opportunities.

To help with the challenge of estimating the regression test effort, Rosenblum and Weyuker proposed a coverage based prediction model to predict the percentage of test

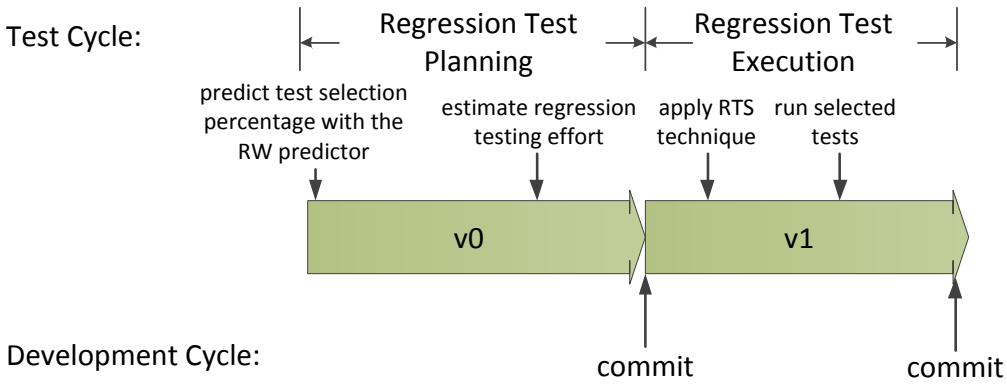

Figure 1.2: An Improved Regression Testing Cycle Utilizing the RW Predictor

cases would be selected by a RTS technique[28]. Their case study demonstrated that the predicted average selection percentage by the Rosenblum and Weyuker (RW) predictor is very close to the actual average selection percentage. If the RW predictor consistently produces accurate predictions, it can be used to predict percentage of tests selected for regression by a RTS technique. Figure 1.2 shows an improved regression testing process utilizing the RW predictor. In regression test planning phase when code change has not yet been committed, test team could use the RW predictor to predict percentage of test cases will be selected by a RTS technique. Based on the predicted selection percentage and the total execution efforts of entire test suite in previous release, test team could come up with a regression effort estimation. Once code change is committed, test team moves into regression test execution phase to first apply RTS technique to the test suite and then run selected test cases.

1.1 The Problem

We have shown in Figure 1.2 that if the RW predictor consistently performs well on all subject programs and all RTS techniques, it can be used to predict the test selection percentage at the beginning of regression test planning phase. Test organizations can benefit from the RW predictor to produce more accurate regression test effort estimations. Unfortunately, empirical studies of the RW predictor shows inconsistent prediction performance.

Rosenblum and Weyuker evaluated the RW prediction model and the evaluation result is shown in Figure 1.3. The study was performed on 31 versions of KronShell88 program

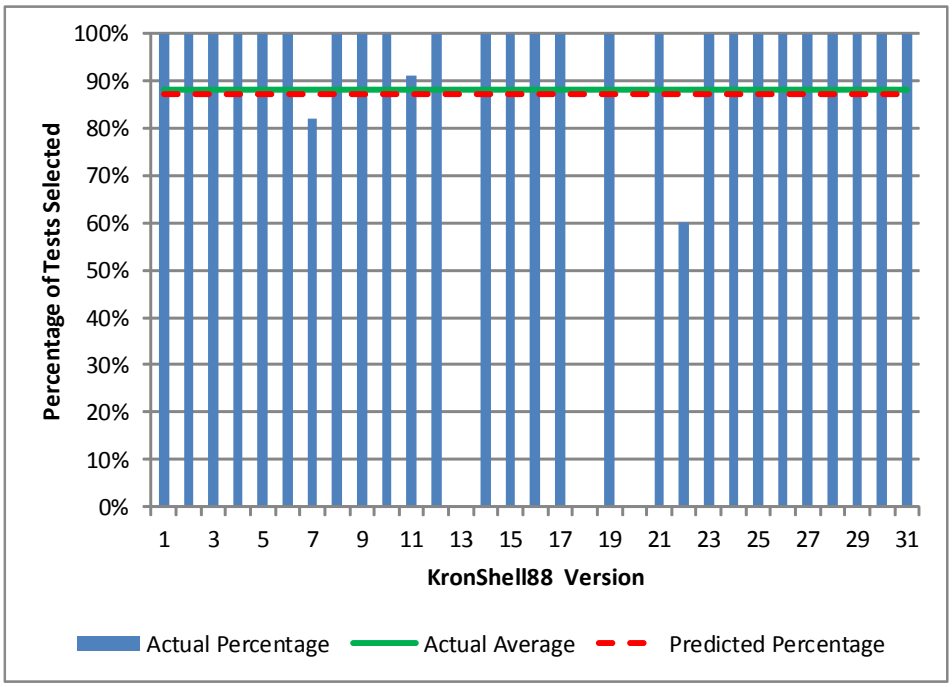

Figure 1.3: RW Predictor Performance in the KornShell88 Study

with *TestTube* [10] as the RTS technique. Each bar in Figure 1.3 represents the actual percentage of tests selected by *TestTube* on each version. A bar of 100% indicates all test cases were selected for the version, while a bar of 0% (no bar) indicates no test case was selected for the version. In average, *TestTube* selected 88.1% of the tests, which is represented by the green solid line in Figure 1.3. The RW prediction model computed a value of 87.3%, which is represented by the red dash line. Since the difference between blue solid line (actual) and red dash line (predicted) is very small, clearly, the RW predictor was very accurate in this study.

Later empirical studies conducted by Harrold *et al.* [15] evaluated the RW predictor on one additional RTS technique and seven C programs. The study results suggested that the RW predictor can yield acceptable accuracy for some subject programs but can deviate significantly for others. The results of their study showed that deviation between the RW predicted test selection percentage and actual percentage could be up to 40% for some versions.

The inconsistent performance of the RW predictor makes it unreliable to be used as a generic solution for test effort estimation and hinders its industry adoption. The goal of this thesis is to first identify the factors contributing to the RW prediction error and estimate the error, then come up with models to improve the RW prediction performance.

1.2 Thesis Contributions

The focus of this thesis is to address the issue of RW predictor's inconsistent performance. We perform regression analysis to understand the contributing factors to the RW prediction errors, and then try to come up with models to estimate the errors. We then propose improved prediction models and compare the performance of improved models from the original RW predictor. The contributions of this thesis are summarized as follows:

- Identify two major contributing factors to the RW prediction errors are the amount of code changes between revisions and code coverage overlaps between test cases.
- Propose two linear models to estimate the RW prediction errors.
- Propose two improved RW prediction models and demonstrate their effectiveness through case studies.
- Propose an extensible RTS Prediction Models Performance Evaluation Framework (PRIME), which is used to evaluate the performance of prediction models.
- Realize three RTS techniques and four RTS prediction models, which are added into the PRIME framework as plug-ins.

1.3 Thesis Organization

This thesis is organized as follows.

- Chapter 2 overviews related works on regression testing and regression test selection. It also covers related works on prediction models with focus on regression test selection prediction models.
- Chapter 3 describes the Prediction Models Performance Evaluation Framework which we implemented to evaluate the performance of prediction models.

- Chapter 4 lists the research questions and covers design of our experiments including the procedures for our studies and the factors have potential impact to the RW predictor's performance.
- Chapter 5 presents the results of the two studies and the analysis of the results.
- Chapter 6 presents three improved prediction models and the performance comparison of the improved predictors from the original RW predictor.
- Chapter 7 concludes the thesis and highlights future works.

Chapter 2

Related Works

This chapter provides background knowledge and related works of two major research domains closely related to this thesis work. The chapter starts with an overview of regression testing and regression test selection. Then, it is followed by prediction models with focus on Regression Test Selection prediction models.

2.1 Regression Test Selection

Regression testing is the process of running existing test suite to validate that the behaviours of software are not adversely impacted by software changes. Regression testing is a costly activity due to two factors: i) regression test suite sizes tend to grow as software becomes more complex. ii) regression testing is a recurring activity. The total regression testing cost increases as the number of repetition increases.

There are three related research domains aiming at reducing regression testing cost: test suite minimization, test case selection, and test case prioritization. Test suite minimization seeks to eliminate redundant test cases in order to reduce the size of regression test suite. Test case selection seeks to select an effective set of test case to cover changes in the current revision. Test case prioritization seeks to order test cases in such a way that some testing goals, such as fault detection or coverage of important features, are reached sooner. This thesis work is closely related to Regression Test Selection which we discuss in more details in this section.

Regression Test Selection (RTS) is a research area aimed at selecting a subset of test cases from existing test suite for regression testing. In this section, we will start with some

background knowledge of notations and concepts then discuss RTS techniques.

2.1.1 Definitions

Following notation are used in the discussion of Regression Test Selection problem: Let P be the current version of the program under test , and P' be the next version of P. Let S be the current set of specification of P and S' be the set of specifications for P'. T is the set of existing test suite and t is the individual test case in T. $P(t)$ represents execution result of P with test case t. We define an execution trace $ET(P(t))$ for t on P to consist of the sequence of code entities in P that are executed when P is executed with t.

Formally, Regression Test Selection problem is defined as follows -

Given the program, P, the modified version of P, P' and a test suite, T. Find a subset of T, T', with which to test P'.

Rothermel and Harrold defined RTS as a problem of choosing all *modification-revealing* [29] test cases from T and gave the following definition of modification-revealing: A test case t is modification-revealing for P and P' if and only if $P(t) \neq P'(t)$. Rothermel and Harrold pointed out that unfortunately we cannot in general find an algorithm that will identify the set of modification-revealing tests in T as it is an undecidable problem. Instead, authors suggested to use a weaker criterion to select all *modification-traversing* test cases and gave the following definition of modification-traversing:

A test $t_i \in T$ is modification-traversing if and only if:

1. it executes a new or modified code entity in P', or
2. it exectues a code entity in P that had been deleted in P'

RTS techniques focusing on selecting all modification-traversing tests are called *safe* techniques [33]. A safe RTS technique must not exclude any modification-traversing test cases. Note that although modification-revealing tests are necessarily modification-traversing, not all modification-traversing tests are modification-revealing. Therefore safe techniques may not be precise, *i.e.* not all selected modification-traversing test case will reveal fault.

2.1.2 RTS Techniques

There has been significant amount of research on Regression Test Selection techniques in the recent years and many techniques have been proposed. Engström *et al.* [13] systematically reviewed 2923 papers published before 2008 and identified 32 different RTS techniques. From a high level view, most RTS techniques consist of similar processes as depicted in Figure 2.1 (adapted from [23]). A RTS technique typically has one process to model and identify changes between two program versions, and another process to create traceability linkage between test case and the program. Then test cases cover changed part of the program are selected for regression testing.

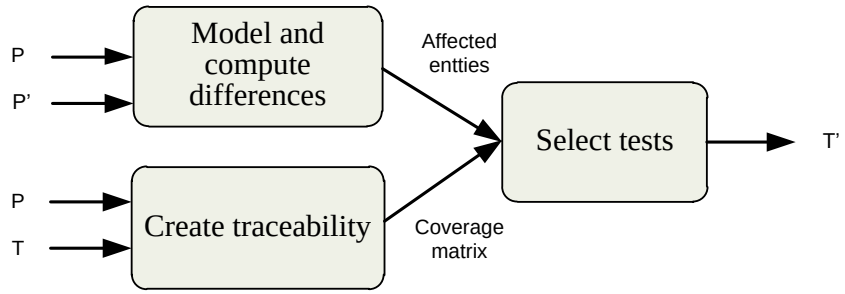

Figure 2.1: A General Regression Test Selection Technique [23]

RTS techniques however vary on how they model the program and identify changes. Examples include test case selection based on code changes [5, 16, 32], based on specification changes [6, 9, 25], and based on test case similarities [8, 17]. As there is a large set of RTS techniques available in the literature, we will not be able to discuss them all in this section. Our focus here is to present some RTS techniques closely related to the RTS techniques we implemented in our evaluation framework. A more comprehensive survey of RTS techniques can be found in Engström *et al.*'s systematic review [13], or Yoo and Harmna's paper [43].

Textual Differencing Technique One of the RTS techniques used in our experiments is the texture differencing technique proposed by Volkolos and Fankl [37, 38]. The technique utilizes UNIX diff tool to compare source code of P and P' to identify modified program statements. The source code is pre-processed into canonical forms to remove the impact of cosmetic differences. Authors implemented the technique into a tool called Pythia. Pythia is a safe RTS tool and capable of analyzing large software systems written in C.

The empirical study results provided evidence that textual differencing technique is very fast and can achieve substantial reductions in the size of the regression test suite. The fact that Pythia is an integration of standard, well known UNIX programs makes it very attractive for industry adoption. Although authors conducted analytical studies to show that texture differencing technique is as precise as techniques using graph representations, there was no empirical evaluation to support the claim.

Firewall Techniques Leung and White proposed the firewall concept in a serial of papers [21, 20, 39, 40]. The firewall concept is basically that given a modified module, draw a virtual firewall around the modules need to be retested so that defect should be kept away from spreading into more modules if possible. Authors proposed approach to identify the firewall based on call graph. Skoglund and Runeson implemented the class firewall technique which adapted the firewall concept to the object-oriented paradigm considering a Java class as a module [36]. By creating a class dependency diagram, any classes dependent or transitively dependent on the modified class are considered inside the firewall. Authors implemented the technique by integrating DependencyFinder[1] tool which operates on compiled Java byte codes. Their evaluation results shown that the class firewall technique can reduce the regression suite and scale to large software systems. Since the class firewall technique operates on binary code, it can be used even when source code is not available, e.g., Commercial Off The Shelf(COTS) components. In our study, we implemented the class firewall techniques based on Skoglund and Runeson's paper.

Graph-walk Techniques Rothermel and Harrold *et al.* presented several closely related RTS techniques based on different graph representation of the program [16, 30, 34, 32, 35]. These techniques are classified as graph-walk as their algorithms walk through a sequence of vertices and edges of the graph in order to identify changes between program versions. Similar to class firewall and textual differencing techniques, these techniques identify tests that execute new or modified code. Different from class firewall and textual differencing, these techniques use an intermediate program representation to identify modification in the program. The algorithms presented in these techniques construct graph representation G for P and G' for its modified version P'. The algorithms then synchronously traverse the graphs that begin with the entry node of G and G', looking for pairs of nodes N and N' whose labels are not lexicographically equivalent. This mismatch node pair represents the modification point of the program. When such a pair (N,N') is found, algorithms select all tests in T that have reached N based on execution trace $ET(P(t))$.

[1] http://depfind.sourceforge.net/

In [30], Control Dependence Graph (CDG) is constructed and in [34] Program Dependence Graph (PDG) is used. CDG is very similar to PDG but only captures control dependency of the program. CDG algorithm does not address interprocedural programs. PDG represents both control dependency and data dependency in a single graph. PDG algorithm can be applied to single procedure and interprocedural level. Rothermle and Harrold later extended the graph traversing algorithm based on Control Flow Graph (CFGs) program representation[32]. CFG is a simpler representation of the program. Therefore, the algorithm is more efficient. Similar to CDG, CFG does not capture data dependency information. If a variable definition is modified but never used, the modification cannot contribute to any changes in the output. However, CFG/CDG based test selection techniques may select test cases that have reached the variable definition modification, and these test cases may not reveal fault.

Rothermal *et al.* extended the CFG based technique for C++ software using Interprocedural Control Flow Graph(ICFG) and Class Control Flow Graph (CCFG) [35]. Beydeda and Gruhn integrated black-box data flow information to the CCFG to test object orientated software [5]. Harrold and Jones *et al.* [16] presented a safe technique for Java that efficiently handles the features of Java language. Its an extension of [32] using Java Interclass Graph (JIG) as program representation. JIG is suitable to efficiently handle Java features such as exception-handling constructs, external libraries and polymorphism. Orso *et al.* proposed two-phase approach in graph construction and traversing [24]. The first phase performs initial analysis based on high-level Interclass Relation Graph (IRG) to identify parts of the system to be further analyzed. The second phase performs in-depth analysis based on JIG and select test cases based on modification. Under certain assumptions, this technique is still safe and scalable for large software systems.

Specification-based Techniques Several researchers proposed RTS techniques based on specification change. Briand *et al.* presented a black box design level test selection technique based on UML [6]. Authors proposed approaches to identify changes between two versions of class diagram, sequence diagrams and use case diagrams. Then based on the traceability information between the UML design and test cases, regression test cases can be classified into three categories: reusable, retestable and obsolete. Chen *et al.* presented a RTS technique based on changes of activity diagram [9]. An activity diagram is a UML model to describe requirements on customer features or behaviours. Since the activity diagram is very similar to CFG, authors apply the CFG-base algorithm to identify changes in activity diagram between two revisions. Then regression test cases are selected based on traceability information.

Comparing to RTS techniques based on source code or an intermediate representation

of the source code, these specification based techniques are efficient, scalable and can be applied to Commercial Off-The-Shelf (COTS) software when source code is not available. These techniques could potentially be applied to the test suite even before source code is implemented. The disadvantage of specification change based techniques is that they may not be as precise as code modification base techniques and these techniques require accurate and up-to-date UML design models of every version of the program.

Defect Correction-based Techniques Wikstand *et al.* [41] proposed a test selection technique that uses a cache to monitor fault-prone files and recommends test cases to rerun to cover updated files. This technique requires source code of P and P, test suite T and linkage information between defects, test cases found the defects and source files updated to fix the defects. Empirical study on a large system with five digit number of source files has shown effective selection of test cases covering fault-prone files.

Comparing to other code modification based techniques; this does not require execution trace information. This technique can be used in practice when code coverage information for each test case is not readily available, such as in real-time and embedded systems, or any environment collecting coverage information is too expensive. Test case code coverage information is instead computed from what files were updated to fix a fault found by the test case. Its reasonable to believe this techniques fault detection effectiveness is lower than graph-walk techniques as this technique is not safe. In practice, this may be used as a test case recommendation system compliment to other fault detection approaches. Authors have not completed evaluating the quality of the test cases recommend by this technique, therefore the precision of the selection is unknown.

Techniques for Non-code Changes Nanda *et al.* [23] proposed a technique to address the issue that most existing RTS techniques focus on changes made to code components and completely ignore non-code elements. Non-code elements , such as configurations file and databases, can also change and affect the system behaviour. Authors proposed a technique to first build abstract models of configuration files and databases and perform differencing on these models to identify the modifications made to these external entities. Second, the technique creates traceability links from test cases to external entities by tracing code read from or write to external entities. Finally, test cases linked to modified external entities are selected for regression. Empirical evaluation results indicate that this technique can select modification-traversing tests that are missed by code-centric RTS techniques. Haraty *et al.* [14] proposed a RTS technique for selecting test cases in the presence of database changes. Their technique is applicable to stored procedure only. Willmor and Embury [42]

present a safe RTS technique for database-drive applications. Their technique considers the scenario where code change happens on a database-interacting statement, which in turn affect the behaviour of other statements that read the modified persistent state.

These techniques are nice complements to the existing code-centric RTS techniques, as modern software systems often contains many complex database logic and configuration files.

Techniques for Component-based Software Some researchers proposed techniques to address challenges in RTS of component-based software. Component-based software development often integrates black-box components developed by a third party. Application developers may not have the access to the internal information of the components, such source code and specifications, due to intellectual property issues. Orso *et al.* [25] proposed code based and specification based techniques utilizing component meta data to support regression test selection of component-based software. The code-based technique assumes each software component was capable of providing structural coverage information and change information as metadata. The specification-based technique assumes the component specification is represented by UML state-chart diagrams, which were used by the graph-walk techniques. The empirical results of the technique show that component metadata can feasibly be used to produce savings in retesting cost. This technique built upon existing graph-walk techniques and specification based techniques to solve specific issues in component-based software.

Techniques for Different Testing Processes There are many researches focus on applicability of RTS techniques in industry environment, particularly applicability of RTS techniques on different testing processes. These works generally use code modification based techniques and are mainly concern with real-world challenges. Buchgeher *et al.* presented experience of applying RTS technique in an industry manual testing process [7]. Authors implemented a tool to identify changed source code file based on information from version control systems then select test cases based on test case to source file coverage information. The results showed that selecting test cases based on code coverage information often leads to a large set of test cases being selected. Some preliminary study suggested that performing RTS on method level may improve the precision. However, the version control tools used in the study(Subversion or the Microsoft Team Foundation Server) only provide information on modified lines of code, but not which methods these lines belong to. As the result, they were not able to implement the RTS technique at method level. Authors also discussed many challenges in their work which are relevant for implementing

RTS techniques in practice.

In addition to the papers present the original RTS techniques, there are significant research efforts in conducting empirical evaluation of these RTS techniques on different programming languages and applications. These research can be valuable resources to understand the effectiveness of RTS techniques and challenges in evaluating these techniques.

2.2 Prediction Models

Software engineering researcher have long been proposing prediction models to help with defect prediction, software development efforts prediction and software performance prediction etc. In this thesis work, our focus is on prediction models to predict the percentage of test cases will be selected by a RTS technique. We discuss two closely related RTS prediction models in this section.

2.2.1 Rosenblum and Weyuker Predictor

Rosenblum and Weyuker [28] presented a prediction model to support the determination of the cost-effectiveness of RTS techniques. Using coverage information, the Rosenblum and Weyuker(RW) model computes the percentage of test cases will be selected by a RTS technique. Based on some simplified assumptions, the predicted percentage can then be used to determine whether it is worthwhile to apply the RTS technique.

As in Rosenblum and Weyuker's model, given a program P and its regression test suite T, let M be the RTS technique used to choose a subset of T_M from T. The RW predictor predicts the value of $|T_M|/|T|$, which is the percentage of test cases selected by technique M.

Let E be the set of code entities of P that are considered by technique M for coverage analysis. The type of code entity could be statement, method, class or basic block etc. It is assumed that T and E are nonempty and that every syntactic element of P belongs to at least one entity in E.

Let E^C denote the set of covered entities, *i.e.* code entities exercised by at least one test case during test execution. Formally, E^C is defined as follows:

$$E^C = \{e \in E | \exists t \in T(covers_M(t, e))\} \tag{2.1}$$

RW model defines $covers_M(t, e)$ as the coverage relation induced by technique M for P and defined over $T \times E$. $covers_M(t, e)$ is true if and only if the execution of test case t on P causes entity e to be exercised at least once. This relation is represented by a 0-1 *coverage matrix* C, in which the rows represent test cases in T and the columns represent entities in E. Then, element $C_{i,j}$ of C is defined as:

$$C_{i,j} = \begin{cases} 1 & \text{if } covers_M(i, j) \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

and *cumulative coverage* CC is the sum of all ones in the coverage matrix C

$$CC = \sum_{i=1}^{|T|} \sum_{j=1}^{|E|} C_{i,j} \tag{2.3}$$

For a single entity change in P, assuming all entities are equally likely to be changed, *i.e.* a uniform distribution, the expected number of test cases would have to be rerun can be estimated simply as the average number of test cases that exercise an arbitrary entity. RW model defines average number of test cases covers a single entity as N_M:

$$N_M = \frac{CC}{|E|} \tag{2.4}$$

N_M in fact is just the average number of ones of each column in the coverage matrix C.

A slightly refined variant of N_M considers $|E^C|$ rather than $|E|$ as the universe of entities. RW model defines average number of test cases covers a single *covered* entity as N_M^C:

$$N_M^C = \frac{CC}{|E^C|} \tag{2.5}$$

N_M^C is the predicted number of test cases will be selected by technique M for a single entity change in P.

Finally, the predicted percentage of test cases will be selected by technique M for a single entity change in P is denoted π_M, the predictor for $|T_M|/|T|$:

$$\pi_M = \frac{N_M^C}{|T|} = \frac{CC}{|E^C||T|} \tag{2.6}$$

Rosenblum and Weyuker made two important assumptions in their model. One is the RTS technique M is safe, the other is there is only single entity changed from program P to P'. Hereinafter, RW predictor is referring to the safe strategies single entity changed model in equation 2.6.

Rosenblum and Weyuker conducted study of the RW model on 31 versions of KronShell88 program with *TestTube* as the RTS technique. The predictor reported an average of 87.3% of the test cases would be selected while 88.1% was actually selected. Clearly, the RW predictor was very accurate in this case.

2.2.2 Predictor Incorporating Change Frequency

Harrold and Rothermel joined by the two original authors of the RW predictor, Rosenblum and Weyuker, conducted additional evaluation on the RW prediction model [15]. They evaluated the RW predictor with seven C programs which had been used previously by researchers at Siemens Corporate Research. The test cases of these programs are white-box test cases created to cover every executable statement, edge and definition-use pair. In addition to the *TestTube* RTS technique used in the original study, Harrold *et al.* used one additional RTS technique *DejaVu* [31]. Their results shown that the RW prediction is accurate for some test subjects, but deviates from actual selection percentages significantly for others. Authors pointed out that RW predictor only accounts for test coverage but does not account for the locations of the modifications. Therefore, authors proposed an improvement to the original RW predictor by incorporating relative frequency of changes to the covered entities. However, there were no experiment on the improved predictor due to lack of change history information for their test subjects. We discuss Harrold *et al.*'s prediction model below and present our evaluation results of the predictor in Section 6.3.

The improved predictor extends the RW predictor by adding weights that represent the relative change frequency of the covered entities. For each element $e_j \in E^C$, w_j is the relative frequency with which e_j is modified, and it is defined such that $\sum_{j=1}^{|E^C|} w_j = 1$. The original unweighted RW predictor, discussed in Section 2.2.1, compute the predicted number of test cases will be selected by technique M, N_M^C, as

$$N_M^C = \frac{CC}{|E^C|} \tag{2.7}$$

where CC is simply the number of ones in the Coverage Matrix and $|E^C|$ is the number of covered entities.

The weighted analogue of N_M^C is a weighted average, WN_M^C, which authors define it as follows:

$$WN_M^C = \sum_{j=1}^{|E^C|} w_j \sum_{i=1}^{|T|} C_{i,j} \tag{2.8}$$

where

$$C_{i,j} = \begin{cases} 1 & \text{if } covers_M(i,j) \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

For this weighted predictor, the predicted percentage of test cases will be selected by technique M for a single entity change, denoted by Π_M^w, is given as follows:

$$\Pi_M^w = \frac{WN_M^C}{|T|} \tag{2.10}$$

2.3 Summary

In this section, we presented necessary background knowledge and works from two research domains closely related to this thesis. One is regression test selection with focus on regression test selection techniques. The other is prediction model with focus on RTS prediction models. In the next section, we discuss the framework we implemented to evaluate RTS prediction performance.

Chapter 3

RTS Prediction Models Performance Evaluation (PRIME) Framework

To evaluate the performance of the RW prediction model, we developed a RTS PRedIction Models performance Evaluation framework (PRIME). The PRIME framework compares the predicted test selection percentage with the actual test selection percentage in order to evaluate the accuracy of prediction models. The PRIME framework, as it stands now, contains implementation of three RTS techniques, four prediction models and factory classes to create artifacts from Software-artifact Infrastructure Repository (SIR) [11], GitHub, and Apache SVN. As depict in Figure 3.1, the PRIME framework consists of four major components. *Artifact Extractor* component connects to test subject repositories and extracts software artifacts; *Test Selection Prediction* component predicts test selection percentage using prediction models; *Regression Test Selection* component applies RTS techniques to test subjects to obtain the actual test selection percentage; *Predictors Evaluation* component evaluates prediction models based on some performance metrics. We discuss each component in more details in the following subsections.

3.1 PRIME Framework Component: Artifact Extractor

The purpose of the artifact extractor component is to act as a proxy to connect to various repositories and deal with repository specific tasks. Artifact extractor component consists

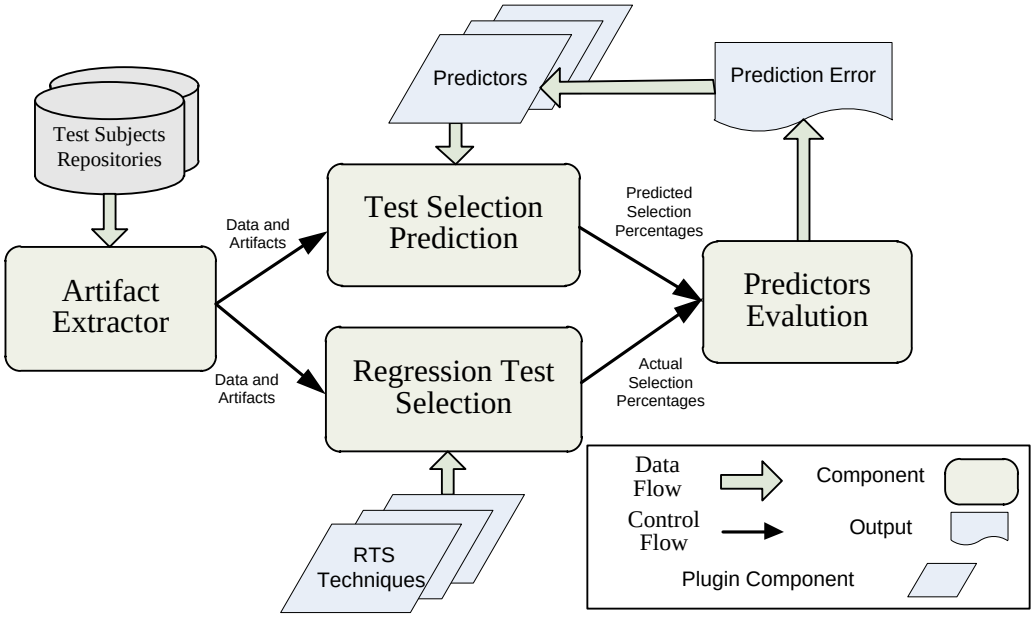

Figure 3.1: RTS PRIME Framework

of factory classes to extract artifacts and store them in a consistent format for other components to use. In this section, we discuss what software artifacts are, how we extract them from repositories, and how we model them once they are extracted into the framework.

3.1.1 Software Artifacts

Software artifacts are one of the many by-products built during the development of software. Some artifacts such as source code, test cases, fault, and UML diagrams help describe or develop the function of the software. Other artifacts are concerned with the process of development itself - such as project plan, business case, and risk assessment [4]. The composition of all software artifacts not only represent the software system itself, but also the development and evolution of the software system. Typically, software artifacts have multiple versions and have one or many traceability links to or from other artifacts. For example, most development organizations use version control systems to automatically create a new version of a source file whenever there is a code check-in. A new source file version may link to a new requirement specification version if the source file change is due to a change in the requirement; A new source file version may link to a defect if the source

file change was to fix the defect. As software evolves, there is a co-evolution of all of its artifacts and the traceability links between them. In the context of this thesis, we consider traceability links as an artifact.

Software artifacts such as source code, test suite, fault and execution trace are critical input to RTS techniques and RTS prediction models. Some metadata in the software development, such as time stamp of a commit, author of a commit, may also be of interest to some RTS techniques or RTS predictors. However, there are many challenges exist with artifacts extraction. First, the availability of various artifacts varies by software development process. For example, requirement specification is a common artifact in waterfall model, but it is generally not available in Agile development process. Second, even when an artifact is available, it may be stored in an array of different tools, therefore its format varies. In some organizations, artifacts are managed in Application Lifecycle Management(ALM) system. In other organizations, artifacts are managed in open source tools or just stored as files on disk.

The difference in artifacts availability and format impose challenges to organizations whom wish to adopt RTS techniques. For an organization wishes to implement a RTS technique that requires code coverage, the first step would be collecting test case to code traceability. If the traceability is not available in the current development process, it would require significant process change and high upfront investment, which could become major road block in adopting RTS techniques.

The PRIME framework tackles these challenges in two ways:

• It filters RTS techniques based on the availability of artifacts. This is discussed in more details in Section 3.2.2

• It models artifacts so that all artifacts have consistent format regardless the tools and processes organizations used. The framework provides an abstraction layer of all software artifacts that are potentially used by RTS techniques or RTS predictors. In the next subsection, we discuss this in more details.

3.1.2 Extracting Artifacts from Repository

Artifacts are generally stored in one or many repositories during software development. A repository can be commercial ALM tools (e.g., HP Quality Center), open source tools (e.g., Apache Subversion) or simply directories on disk with predefined structure (e.g., Software-artifact Infrastructure Repository [11]). The creation of artifacts objects is highly dependent to the repository used. Any changes to the repository, e.g., API changes, tool

version changes, directory structure changes, would impact extraction process. In the PRIME framework, we use factory method pattern to provide an abstraction layer. There is a factory class for each repository to handle specific details of that repository. Various factory classes produce consistent artifact objects. For example, if test cases, requirements and defects are stored in HP Quality Center(QC), then a QCFactory can be implemented to extract all artifacts stored in QC. QCFactory understands all internal details for QC, how each type of artifacts are stored and how the traceability links between artifacts are managed. This factory class is the only place needs to be changes if there is a change in HP QC.

3.1.3 Artifact Modelling

The goal of artifact modelling is to create a model that captures all artifacts available for RTS techniques or RTS predictors to use. The artifact model in the PRIME framework is independent of development process, program language, tools and techniques. In Figure 3.2, we depict the artifact model in a UML diagram. An application in the PRIME framework contains the following artifacts:

• **Specification** is an artifact contains requirement specification, design specification and architecture specification.

• **Fault** can be fault naturally occurred in the original program or fault seeded intentionally.

• **Program** is an artifact represents all artifacts related to program code, including Source File Entity, Class Entity, Method Entity and Statement Entity, binary, database schema and configurations. A program could have several variants. A program variant is an alternative copy of the original program. Seeded is a variant contains faults intentionally added into the program. Alternative is a variant semantically equivalent to original program.

• **Trace** represents traceability links between artifacts. Trace contains code coverage matrix, specification matrix and fault matrix. Code coverage matrix maintains all links between each test case and its covered code entities when executed. Specification matrix maintains all links between each specification and test cases cover that specification. Fault matrix maintains links between each fault and test cases discover that fault.

• **Test suite** is an artifact contains test cases. A test case contains input, output, test run and execution script. A test run contains a test result indicates pass or failure of the test run.

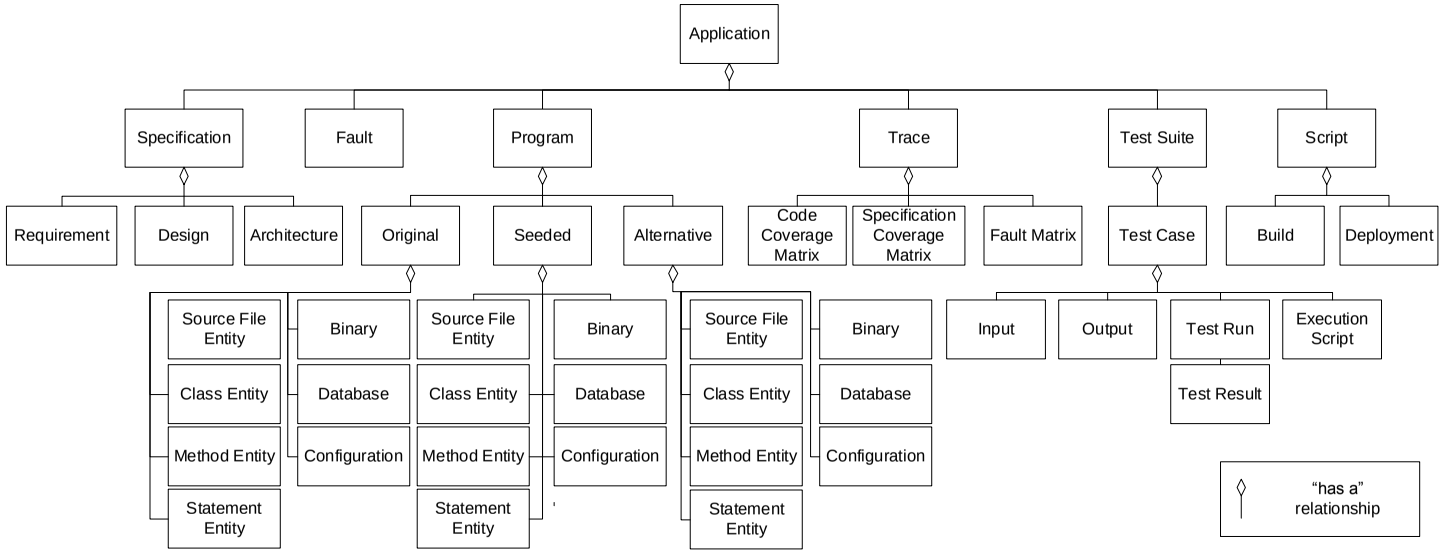

Figure 3.2: Artifact Model

• **Script** is an artifact that contains build script or deployment script.

Each box in Figure 3.2 represents zero to n versions of the artifact. Note that artifacts may not evolve at the same time. For example, program version m may be covered by test case version m' as code version increment may not trigger test case version increment. This could happen when program is refactored to provide same functionality which does not require any test case change to verify the program.

Once artifacts are extracted into the framework and modelled, the PRIME framework performs the following traceability verification step to ensure the integrity of the data.

3.1.4 Traceability Verification

Maintaining software artifacts, especially maintaining traceability links among software artifacts is a tedious and time consuming task. In industry where software development is largely deadline driven, companies mostly focus on delivering the product and left software artifacts poorly maintained [27]. The issue of poor traceability could due to a variety of reasons such as human factor, use of heterogeneous tools or lack of visible benefit [3].

Unfortunately, RTS techniques and RTS predictors are highly dependent on the accuracy of traceability links. Poor traceability would result in poor test selection. Therefore, in the PRIME framework, following traceability verifications are performed to assess and improve the accuracy of traceability links:

- Verify that the total number of test cases in the test suite equals the total number of test cases in the test execution script. This is to ensure every test is executed.
- Verify that the total number of test cases in the test suite equals the total number of test cases in code coverage matrix. This is to ensure every test at least cover some code entities.
- Verify that the total number of faults equals the total number of faults in fault matrix. This is to ensure every fault has at least one test case associated with it.
- Verify that the total number of specifications equals the total number of specifications in specification coverage matrix. This is to ensure every specification is covered by at least one test case.

These verifications are basic sanity checks to ensure the integrity of input data for RTS techniques or RTS predictors. More in-depth verifications can be defined and plugged into

the framework as needed. For organizations using ALM tools to manage all artifacts in one integrated environment, these verifications may not as critical. However, many companies are still managing their software artifacts in a variety of tools. Artifact Verification is a must in order to maintain the overall integrity of the data.

In addition to traceability verification, we could borrow techniques from traceability recovery domain to reconstruct missing traceability links [2, 19]. The PRIME framework does not currently have any traceability recovery capability but it is one of our future works to recover missing traceability links.

In general, Artifact Extractor component in our framework is to provide consistent context for all artifacts. It facilitates rich and accurate input data for RTS techniques or RTS predictors. It also provides processes and models to integrate new applications or new artifacts into the framework. Great care has been taken in this component to reduce test subject setup cost and to improve input data quality.

3.2 PRIME Framework Component: Regression Test Selection

Regression Test Selection is the component applying all RTS techniques on every test subjects and on every versions. All implemented RTS techniques have one common interface which can be used to select tests from regression test suite. Based on the actual tests selected by RTS techniques, the actual selection percentage can be computed and used as an input in Predictors Evaluation component. Other common shared tasks among RTS techniques, such as analyzing code coverage, parsing source code or analyzing code changes are also implemented in this component.

Regression Test Selection component consists of three subcomponents: technique modelling, technique filtering and test case selection. We discuss each subcomponent in details in the following sections.

3.2.1 Technique Meta-modelling

Regression test selection is a very active research area. Over the years, many RTS techniques have been proposed. However, there is no clear definition of what constitutes a *technique* and thus there is a problem in determining which regression test selection techniques exist [13]. Some techniques proposed were considered novel and unique at the time

of first presentation, but over time, many variants also proposed to adapt to different programming languages, different program representations, or different analysis scope. This lack of distinction of each technique is a challenge for test engineers to adopt RTS in practice and for researchers to evaluate RTS techniques consistently.

Facing this challenge, we propose a technique meta-model to provide a consist representation of various techniques exist in the literate. Our technique model in Figure 3.3 uniquely identifies a technique in the PRIME framework. The key point in our technique model is the separation of techniques and their implementations. RTS technique is a concept and sometimes described in a general manner. However, for a test engineer to apply the technique to the application under test, it must be in a concrete form. Most importantly, the performance of a RTS technique may be very sensitive to changes in the implementation of the technique. Technique implementation on a different programming language, on a different analysis scope or using different utilities would have different precision and cost. This sensitivity determined that in the PRIME framework we have to separate a technique from its implementations. A RTS technique in the PRIME framework is referring to a specific RTS technique *implementation* which can be applied to a test suite directly.

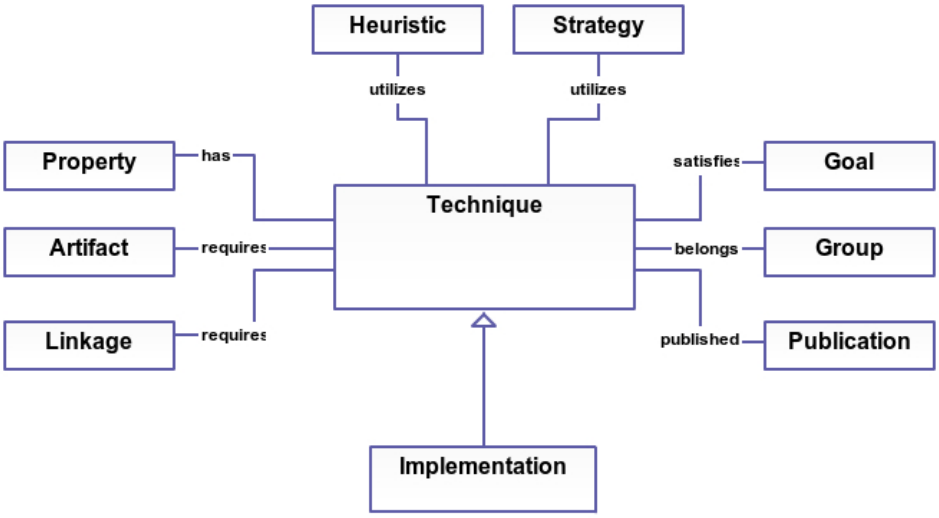

Figure 3.3: RTS Technique Meta-model

3.2.2 Techniques Filtering

In our experiment, we always select all techniques implemented in the framework and apply them to all subject programs. As the number of implemented RTS techniques grow in the framework, it may make sense for a test organization in the industry to select one or a small subset of RTS techniques in the PRIME framework. In that case, techniques can be filtered by the availability of required artifacts. For example, a UML design based RTS technique requires UML diagram of each version of the program. If the subject program does not contain UML diagrams, then the RTS technique should be filtered out. It is worth noting that the technique filtering process is application specific, *i.e.* the availability of artifacts of each application could be different. Therefore this process has to be repeated for each application.

3.2.3 Test Case Selection

Test Case Selection is a subcomponent to apply the test selection algorithm of each RTS technique on subject programs and versions. In the PRIME framework, all techniques implement a *Technique* interface which contains a *selectTests* method. The *selectTests* method takes two versions of the programs p and p' and returns a list of test cases from p that selected for regression test in p'. We have implemented *selectTests* for all three techniques in the PRIME framework. A brief description of the test selection procedures of each RTS techniques can be found in section 4.5.

3.3 PRIME Framework Component: Test Selection Prediction

Test Selection Prediction component runs prediction models and predicts the percentage of test cases will be selected if a RTS technique is applied to a test subject. In the PRIME framework, we have implemented the RW predictor and three improved RW predictors. The RW predictor only requires the Coverage Matrix and regression test suite as input. The improved predictors require some additional data which are fetched from the the Artifact Extractor component. Test Selection Prediction component runs all predictors on all techniques and test subjects combination. The result is a set of predicted selection percentages which is fed into the Predictors Evaluation component. Some performance metrics, such as memory or time consumptions, are also measured in this component.

Each predictor plug-in in the PRIME framework provides a *predictSelectionPercentage* API. RTS techniques utilize the predictors by calling the API and supplying the code coverage matrix and the entire regression test suite. The implementation details of each predictor plug-in varies. We have discussed details of the RW predictor and an improved RW predictor in Section 2.2. The details of the other two improved predictors will be discussed in Chapter 6.

3.4 PRIME Framework Component: Predictors Evaluation

Predictors Evaluation component takes predicted selection percentages from predictors and compare them with actual selection percentages from RTS techniques. It then uses some metrics to evaluate the predictors performance. In our experiment, we use deviation to evaluate the RW predictor and the improved predictors. The result of the evaluation is formatted and exported to a csv file which can be analyzed offline.

3.5 Summary

In this chapter, we described four major components in the RTS Prediction Models Performance Evaluation(PRIME) Framework. The framework is designed to be reusable and extensible that new test subject repositories, RTS techniques, predictors and evaluation metrics can be plugged into the frameworks with minimum efforts. This framework provides generic functions and major control flows which enables us and other researchers to evaluate new predictors in future research.

It's worth noting that our framework is focusing on evaluating the performance of RTS prediction models, not RTS techniques. As a future work, we could expand it to also evaluate RTS techniques based on some characteristics.

In the next chapter, we present the research questions and describe the design of experiments. Utilizing the PRIME framework discussed in this chapter, we conduct the experiments on three Java subjects, the results of the experiments are then discussed in Chapter 5.

Chapter 4

Design of Experiments

In the previous chapters, we pointed out the motivation of this thesis is the inconsistent performance of the RW predictor. We discussed background knowledge and related works, then presented the PRIME framework which is used to evaluate the RW predictors performance. In this chapter, we report on the design of experiments performed to evaluate the RW predictor's performance and understand its prediction errors. The following research questions are pursued through these experiments:

RQ1 : What are the factors impacting the RW predictor's accuracy?

RQ2 : Does a model exist to estimate the RW predictor's prediction error?

RQ3 : Can the RW predictor's performance be improved by incorporating error estimation model?

This chapter describes the design of two studies. In the first study, the RW predictor is used as a general predictor; In the second study, the RW predictor is used as version-specific predictor. The rest of the chapter describes the dependent variable and independent variables used in the regression analysis. Finally, the test subjects and RTS techniques used in the studies are presented.

4.1 Study 1

In this study, we use the RW predictor as a *general* predictor, *i.e.* we only predict the test selection percentage once based on the Coverage Matrix of version 0 of the program.

In Rosenblum and Weyuker's study [28], authors found that the Coverage Matrix was extraordinarily stable over the 31 versions of KornShell program they used in the study. Since the RW predictor is computed solely based on the Coverage Matrix, there is no reason to predict again if there is little change to the Coverage Matrix. For this reason, authors argue that to save analysis cost, prediction value from the first version of the program is good enough for all subsequent versions, *i.e.* use the RW predictor as a *general* predictor.

Following Harrold *et al.*'s paper [15], we use the following notations in both studies:

π_{M_i} **:**
The predicted percentage of test cases selected by RTS technique M from T_{i-1} when an arbitrary change is made to create P_i

S_{M_i} **:**
The percentage of test cases actually selected by RTS technique M from T_{i-1} for the change made to create P_i

Here, M is one of the three RTS techniques in our experiment. We will discuss these techniques in details in Section 4.5.

The deviations, D_{M_i}, of the actual percentage of test cases selected by technique M from the predicted test selection percentage is computed by:

$$D_{M_i} = S_{M_i} - \pi_{M_1} \tag{4.1}$$

In Study 1, D_{M_i} represents the RW predictor's accuracy and it is the dependent variable. To answer our research questions, we first use the procedure in Algorithm 1 to compute D_{M_i} and all factors listed in Table 4.1, then we perform regression analysis to explore the relationship between the deviation (dependent variable) and the potential factors (independent variables).

4.2 Study 2

In Study 1, we treat the RW predictor as a *general* predictor. The RW predictor is used only once at the beginning of the development cycle to predict the test selection percentage of all future versions. In Study 2, we use the RW predictor as a *version-specific* predictor. At the beginning of regression cycle of every version, we use the RW predictor to predict the test selection percentage of the upcoming version. Obviously, this would

Algorithm 1 Study 1 Procedure

```
1: for all subject programs do
2:   for all release/snapshot build types do
3:     for all test class/test method test suite types do
4:       for all RTS techniques do
5:         predict selection percentage π_{M_1}
6:         for all versions do
7:           compute actual selection percentage S_{M_i}
8:           compute deviation D_{M_i}
9:           compute values of all factors for version i
10:        end for
11:      end for
12:    end for
13:  end for
14: end for
```

Algorithm 2 Study 2 Procedure

```
1: for all subject programs do
2:   for all release/snapshot build types do
3:     for all test class/test method test suite types do
4:       for all RTS techniques do
5:         for all versions do
6:           predict selection percentage π_{M_i}
7:           compute actual selection percentage S_{M_i}
8:           compute deviation D_{M_i}
9:           compute values of all factors for version i
10:        end for
11:      end for
12:    end for
13:  end for
14: end for
```

require more accumulated efforts as prediction is computed on every version. In situations where the Coverage Matrix is not stable, this maybe necessary. Organizations may also use the RW predictor as a *version-specific* predictor when recomputing the RW predictor for each version would not cause significant overhead.

Similar to Study 1, we compute the deviation of the actual percentage of test cases selected by technique M for each version P_i from the percentage of test cases predicted by the RW predictor.

$$D_{M_i} = \pi_{M_i} - S_{M_i} \tag{4.2}$$

In study 2, D_{M_i} represents the RW predictor's accuracy and it is the dependent variable. To answer our research questions, we first use the procedure in Algorithm 2 to compute D_{M_i} and all factors listed in Table 4.1, then we perform regression analysis on the results.

4.3 Impacting Factors (Independent Variables)

In order to answer our research questions, we first define a list of factors may impact the RW predictor's accuracy based on the nature of the predictor and our survey of existing literature. These factors are summarized in Table 4.1. We describe details of each factor and how it is computed in our experiment below.

Table 4.1: Summary of Potential Factors Impacting the RW Predictor's Accuracy

Factor Name	**Description**
f1_entity_change_percentage	Percentage of covered entity changed
f2_element_change_percentage	Percentage of elements changed in the Coverage Matrix
f3_num_tests_per_entity	Average number of tests covers each covered entity
f4_RTS_technique	RTS technique

- ***f1_entity_change_percentage*** is the percentage of covered code entity changed between two consecutive versions of the program. It represents the amount of code changes. Since changes on code entities not covered by any test cases would not impact regression test selection, we measure code changes of covered entities only. This factor is computed using number of covered code entity changed divided by total number of covered code entities. Since each RTS technique may operate on different code entity levels (statement, source, class or basic block etc.), the code entity level used in this factor matches the code entity level of the RTS technique in factor f4_RTS_technique.

• ***f2_element_change_percentage*** is the percentage of elements changed in the Coverage Matrix between two consecutive versions of the program. It represents the stability of coverage relations between test case and its covered entities. Recall in Section 2.2.1, we discussed Coverage Matrix $C_{i,j}$ is a 0-1 matrix whose rows represent test cases of T whose columns represent entities of program P. A change from a 0 to a 1 or from a 1 to a 0 between $C_{i,j}$ of a pair of consecutive versions represents one change in the coverage relation. The total number of changes in the coverage relation over total number of elements is the percentage of elements changed. To simplify the analysis, the total number of elements is the size of Cartesian product between the union of the sets of covered entities and the union of the sets of test cases.

• ***f3_num_tests_per_entity*** is the average number of test cases covers each covered entity. It represents the degree of code coverage overlaps between test cases. If each entity is covered by one and only one unique test case, then there is minimum coverage overlap. One entity change would result in a selection of exactly one test case. On the other side of the spectrum, if each entity is covered by all test cases in the test suite, then the coverage overlap is at its maximum. One entity change would result in a selection of all test cases. This factor is computed from the Coverage Matrix of each version and it is basically the average number of ones in each column.

• ***f4_RTS_technique*** is one of the RTS techniques used in the study. We discuss more details of RTS techniques in section 4.5 and a list of implemented RTS techniques used in our work can be found in Table 4.3.

The factors described above are the independent variables in the regression analysis. Our dependent variable is the deviation between test case selection percentage predicted by RW predictor and the actual percentage of test cases selected by the RTS technique. The values of dependent variable is calculated differently for study 1 and study 2 based on Algorithm 1 and 2 while the values of independent variables are the same for the two studies.

4.4 Subject Programs

For our experiments, we have selected three Java subject programs that were developed in recent years. Special interests have been given to subject programs developed using some flavour of Continuous Delivery methodologies. Table 4.2 presents information about our test subjects. For each subject, we list its name, a brief description, the number of lines of code, the number of classes, which repository the subject program was extracted from, the number of versions, and the number of test classes and test methods.

Table 4.2: Summary of Test Subject Programs

Subject Name	**Description**	**Lines of Code**	**Num. of Classes**	**Repository**	**Num. of Versions**	**Num. of Test Classes (Methods)**
apache-xml-security	XML security library	16,800	143	SIR	4 releases	13(84)
jacoco-core	Java code coverage tool	5,067	78	Github	14 releases 9 snapshots	61(736) 62(746)
apache-solr-core	search platform	80,039	568	Apache SVN	7 releases 17 snapshots	166(667) 169(691)

We have selected a well-known Java test subject from Software-artifact Infrastructure Repository (SIR) [11]. The subject program has been used in many regression testing empirical studies such as [26, 12]. We have directly used the source code and developer supplied JUnit test cases without any modifications.

• **apache-xml-security** is a component library implementing XML signature and encryption standards, supplied by the XML sub-project of the open source Apache project. Versions used in this study map to XML-security original version v1.0.0 to v1.0.8 developed between year 2000 and 2001.

In addition, we have selected two open source Java test subjects developed in recent years using some flavour of Continuous Delivery methodology. Since we have access to the version control system used in the development, in addition to major release builds, we have also extract snapshot builds from the version control system. Snapshot builds are created either on a regular interval (e.g., nightly) or created after each commit. For the purpose of this study, we removed snapshot builds that do not have any code changes (e.g. documentation changes or build configuration changes). For each build, we extracted source code, build files and unit tests.

• **JaCoCo** is a free open source code coverage library for Java. It employs test driven development approach and every commit will trigger a build and every build is considered fully functional. As a code coverage library, its code coverage is very high. Most of the packages in JaCoCo has achieved over 85% of statement coverage. Source code of JaCoCo is maintained in a Git repository at GitHub. JaCoCo is under active development and the release versions we extracted for our study were developed between October 2010 and February 2013. The snapshot builds were taken from the development of version v0.6.2 and v0.6.3 between January 8, 2013 and March 22, 2013.

• **Apache Solr** is a open source search platform from the Apache Lucene project.

Apache Solr uses Jenkins Continuous Integration system to produce nightly snapshot build. Apache Solr project wiki page suggests contributors to run all unit tests prior to commit. Its test suite contains functional and some performance tests. The core package we used in our study has achieved 61% of statement coverage. Apache Solr is under active development and the release versions we extracted for our study were developed between January 2012 and April 2013. The snapshot builds were taken from the development of version 4.2.0 between June 13, 2013 and May 29, 2013.

For each test subject, we first created a regression test suite including all JUnit test classes supplied by original developers. Then, we created a second regression test suite by adding every test methods in JUnit test classes. So in the first regression suite, each test case is a JUnit test class, while in the second regression suite, each test case is a JUnit test method. Since there is no standard of what is considered as a test case, we made sure that our subject programs contain both test class test suite and test method test suite. Since there are two regression suites for each test subjects, we list the number of test classes and the number of test methods separately in Table 4.2.

For jacoco-core and apache-solr-core, we provide separate count for the number of release versions and the number of snapshot versions. For apache-xml-security, SIR repository only provides release builds. Based on the time frame it was developed, we assume there were no snapshot builds available for apache-xml-security.

4.5 Regression Test Selection Techniques

Table 4.3 lists three RTS techniques we have implemented based on two research papers [37, 36].

Table 4.3: RTS Techniques

Technique	Description
td_src	Textual Difference on source file level
td_stm	Textual Difference on statement level
cf_ext	Extended Class Firewall on class level

As we presented in Section 2.1.2, there is a large set of RTS techniques proposed in the literature. Initially, we hoped to find RTS techniques implemented into tools that can be readily integrated into the PRIME framework. However, we did not find any RTS technique implementation capable of analyzing Java program. So we decided to implement

several RTS techniques by ourselves. In the process of determining which RTS techniques to implement in the PRIME framework, we have considered following factors:

• **Data Availability:** Different RTS techniques require different set of input data. For example, specification change based techniques like [6] and [9] require UML designs which are generally not available in open source projects. Therefore, we most likely have to select among code modification based techniques. The three techniques we chose to implement only need source code, binary code and test cases, which are available in every open source projects.

• **Empirical Study Results:** We'd like to implement well known RTS techniques backed up by empirical studies. Vokolos and Frankl evaluated Textual Differencing technique with a test subject of 11,000 LOC. Skoglund and Runeson evaluated the Class Firewall technique on an industry software with about 50,000 classes. Both evaluation results shown the techniques are fast and effective.

• **Implementation Effort:** Textual Differencing technique utilizes UNIX diff tool to perform the change analysis; Class Firewall technique utilizes DepedencyFinder tool to perform dependency analysis. By integrating standard, well known programs, we reduce the implementation effort and improve the overall quality of the framework.

We provide a description of the RTS techniques we implemented below with focus on changes we made from the original paper. More information can be found from the corresponding reference papers of the techniques.

4.5.1 Textual Differencing Technique

Textual Differencing [37] is a safe and scalable technique by comparing source files of two versions of the program. Vokolos and Frankly implemented the technique in a tool named Pythia, which is capable of analyzing C programs. We have implemented the technique following the procedures described in the original paper but made several modifications in order to analyze Java programs. Both td_src and td_stm RTS technqiues consist of following high level steps:

• **Beautify Source Files:** Source files are first beautified to ensure syntax and style consistency between versions. This would minimize irrelevant differences when comparing source files of two versions. For our study, we use Jindent[1] to beautify Java source files.

• **Change Analysis:** Similar to Pythia, we used Unix *diff* utility to compare source files of program P and P' to identify program entities that have been modified. Modified

[1] http://jindent.com/

entities include entities added, changed, or deleted in the new version of the program. For td_src, the results of the change analysis is a list of modified source files in P. For td_stm, the results of the change analysis is a list of modified statements in P.

• **Coverage Analysis:** We use Java code coverage tool JaCoCo[2] to instrument Java classes and collect execution trace $ET(P(t))$ of each individual test case on all versions of program P. Coverage reports from JaCoCo are in HTML and XML format which contains statements, methods and classes exercised by executing each test case. As in equation 2.9, a Coverage Matrix is generated for each version of the test subject.

• **Test Selection:** Based on the output from previous steps, each modified entity (*i.e.* source file for td_src, statement for td_stm) in P is used to lookup the Coverage Matrix of P and retrieve all test cases that cover the modified entity. The union of all test cases that cover the modified entities is selected for regression.

4.5.2 Class Firewall Technique

The Class Firewall concept assumes that only changed classes and its dependencies need to be re-tested. Skoglund and Runeson [36] implemented a Class Firewall technique using Java byte code in the analysis. Our cf_ext technique follows the same procedure as Skoglund and Runeson with the exception that in the coverage analysis step, instead of using EMMA, we use JaCoCo for performance reasons. Details of EMMA performance issue are discussed in Section 7.2.3. RTS technique cf_ext consists of following high level steps:

• **Extracting Dependency Information:** We use DependencyFinder[3] tool to analyze class dependency information. Command line interface of DependencyFinder is used to extract every class's inbound dependencies including transitively dependent classes. This tool operates on compiled class files and extracted dependency information is reported in XML format.

• **Change Analysis:** Changes are identified by comparing MD5 signature of class files between two versions of the program. If MD5 hash value has changed between versions, the class file must be changed. We use Google HashFunction in Google Guava library to compute MD5 hash of class files.[4]

• **Coverage Analysis:** This step is very similar to the coverage analysis step in textual difference technique. We use JaCoCo to instrument Java classes and collect execution trace

[2]http://www.eclemma.org/jacoco/index.html
[3]http://depfind.sourceforge.net/
[4]https://code.google.com/p/guava-libraries/

$ET(P(t))$ of each individual test case on all versions of program P. A coverage matrix between test cases and classes is generated for each version of the test subject.

• **Test Selection:** Based on the output from previous steps, each modified class in P is used to query in the Coverage Matrix of P to retrieve all test cases that covered the modified class. The union of all test cases that have exercised modified classes is selected for regression.

4.6 Summary

In this chapter, we presented the design of experiments we followed in the two studies. We introduced three research questions and the procedures to explore the answers of those questions. We then discussed the dependent variable and the independent variables in the experiments, following by three open source Java subject programs we have selected. Finally, we presented the three RTS techniques implemented for the experiments.

Based on the experiments setup defined in this chapter, we discuss the results of investigating RQ1 and RQ2 in Chapter 5 and RQ3 in Chapter 6.

Chapter 5

Results and Analysis

As the first step of our experiments, we focus on understanding the prediction error of the original RW model. In this chapter, we present and discuss the results of pursuing the first two research questions we presented in Chapter 4.

RQ1 : What are the factors impacting the RW predictor's accuracy?

RQ2 : Does a model exist to estimate the RW predictor's prediction error?

We start the chapter by discussing experiments results of Study 1 and Study 2, then perform a causal relation analysis to gain further understanding of the results. At the end, we discuss lessons learned from the two studies and threats to validity.

5.1 Study 1 - General Prediction

5.1.1 Obtained Results on Impacting Factors

Based on the procedures and equations outlined in Section 4.1, we computed the deviation, D_{M_i}, of the actual percentage of test cases selected by technique M from the predicted test selection percentage. For the purpose of visually compare the prediction performance between different test subjects, we calculated average absolute deviation $\overline{D_M}$ as follows:

$$\overline{D_M} = \frac{\sum_{i=1}^{n} |D_{M_i}|}{n} \tag{5.1}$$

where i is the version number, n is the version number of the last version.

It's noteworthy that the version number starts from 0 and all test cases in version 0 are executed for the first time. Regression test selection only starts from version 1 therefore in our equation above, i starts from 1. Also it's worth mentioning that our dependent variable is *deviation*, which has positive and negative values. The average absolute deviation is only calculated for the purpose of presenting the results on a chart.

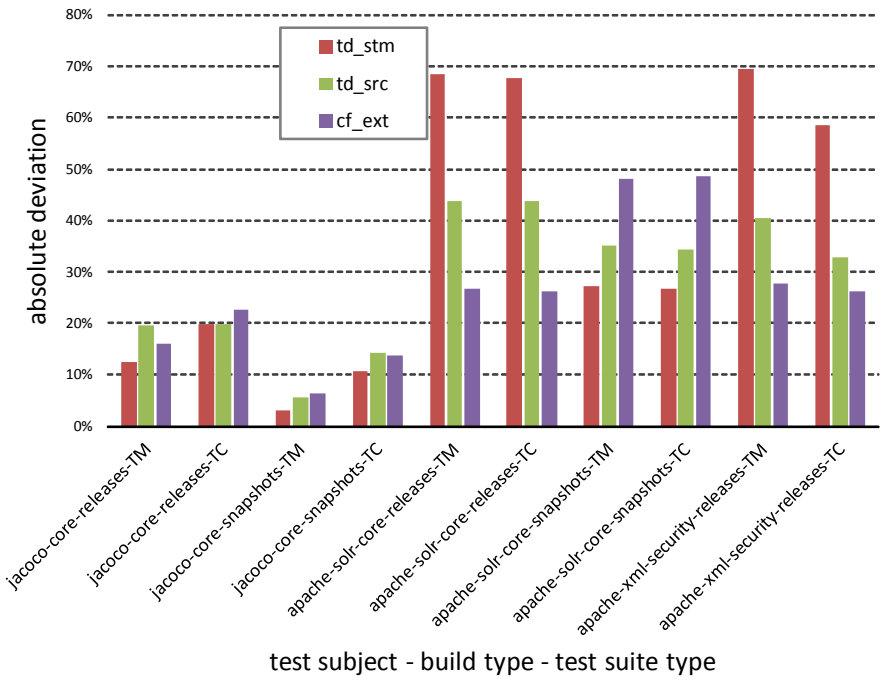

Figure 5.1: Average Absolute Deviation - General Prediction

Figure 5.1 shows the results of average absolute deviation $\overline{D_M}$ when the RW predictor is used as a general predictor. The x-axis label of Figure 5.1 indicates the test subject, build type, and test suite type combination. Test subject is one of the subject names listed in Table 4.2; build type is either releases or snapshots build; test suite type is either TC for test class or TM for test method. The y-axis is the average absolute deviation of each test subject, build type and test suite type combination. For each combination, there are three bars in the graph represent the average absolute deviation $\overline{D_M}$ of the three RTS techniques.

The prediction deviation in Figure 5.1 shows that, for some test subjects and techniques, the RW predictor was quite successful. For example, for jacoco-core-snapshots-TM, the average absolute deviations of all three RTS techniques are less than 10%. For jacoco-core-snapshots-TC, the average absolute deviations of all three RTS techniques were are less than 15%. Clearly for these test subjects, the RW predictor produced good enough prediction on test selection percentages. However, Figure 5.1 also shows that absolute deviations are high for some test subjects. The RW predictor was not successful with apache-solr-core-releases on td_stm technique and apache-xml-security-releases-TM on td_stm technique as the deviations are close to 70%. For these test subjects, the RW predictor's prediction values deviate from the actual values significantly.

To understand this performance difference of the RW predictor on different test subjects and RTS techniques, we applied the procedure in Algorithm 1 to compute the values of all dependent variable (*i.e.* deviation) and independent variables (*i.e.* impacting factors). The result of Algorithm 1 is composed into a table to perform regression analysis in MATLAB. The table consists of a unique row identifier as the first column, 4 columns for the factors we defined in Table 4.1, then follow MATLAB's convention, the dependent variable deviation as the last column. We imported the table, which has total of 276 rows and 6 columns, into MATLAB as a dataset array and then applied *fitlm* function to fit a linear regression model to the data. We used robust fit method, which is little affected by outliers.

The result of the fitting model from MATLAB regression analysis is shown in Figure 5.2. More details to help with interpreting the regression analysis result are as follows:

```
Linear regression model (robust fit):
    Deviation ~ [Linear formula with 5 terms in 4 predictors]

Estimated Coefficients:
                                Estimate      SE         tStat      pValue
    (Intercept)                   0.1128    0.0463     2.4378     0.0154
    f1_entity_change_percentage   0.0052    0.0009     5.7019     3.1022e-08
    f2_element_change_percentage -0.0293    0.0173    -1.6948     0.0913
    f3_Num_tests_per_entity       0.0011    0.0002     6.6585     1.5415e-10
    f4_RTS_technique_td_src      -0.0948    0.0529    -1.7922     0.0742
    f4_RTS_technique_cf_ext      -0.0630    0.0543    -1.1590     0.2475

Number of observations: 276, Error degrees of freedom: 270
Root Mean Squared Error: 0.28
R-squared: 0.214, Adjusted R-Squared 0.2
```

Figure 5.2: Fitting Model - General Prediction

• **Estimate** Coefficient estimates of each corresponding term in the model. For example, the estimate for the constant term (intercept) is 0.1128. It is an estimate because what we really interested in is the coefficient values of the population. However, we don't have access to the entire population. In this case, population is all test subjects that the RW predictor could be applied to, which is infinite. We must estimate the coefficient values based on a sample, which is the three test subjects we selected in the experiment. The size of the coefficient for each independent variable gives indication of the size of the effect that each variable is having on the dependent variable. And the sign of the coefficient(positive or negative) gives the direction of the effect. In Figure 5.2, factor *f1_entity_change_percentage* has higher coefficient estimate than factor *f3_Num_tests_per_entity* indicating f1 has higher impact to the deviation than f3. Also factor f1 and f3 both have positive effects on the independent variable, *i.e.* while other independent variables are hold constant, the higher the *entity_change_percentage* that higher the deviation; the higher the *Num_tests_per_entity* the higher the deviation.

• **SE** Standard error of the coefficients. Because of sample variability, the coefficients estimates may be too high or too low from the corresponding population parameters. The standard error gives indication of the degree of variation. Larger SEs mean lower confidence in coefficient estimate. In Figure 5.2, SE values for factor 1 and factor 3 are much less than other factors which indicates good confidence in estimated coefficient values for f1 and f3.

• **tStat** t-statistic for each coefficient to test the null hypothesis that the corresponding coefficient is 0 against the alternative which is different from 0. The null (default) hypothesis is that each independent variable is having absolutely no effect, *i.e.* real coefficient is 0 and the coefficient estimates appear to be non-zero due to random chance. Note that t-statistic is the coefficient divided by its standard error. *i.e.*
tStat = Estimate/SE. If the coefficient is large compared to its standard error, then it is probably different from 0, *i.e.* we can reject the null hypothesis. In Figure 5.2, the t-statistic for f1 and f3 are 5.7019 and 6.6585 which indicates that these two factors do have impact on the deviation, and the non-zero coefficient estimates are not due to a random chance.

• **pValue** p-value for the F-statistic of the hypotheses test that the corresponding coefficient is equal to 0 or not. Small p-value indicates strong evidence against the null hypothesis (*i.e.* the independent variable has absolutely no effect), so you reject the null hypothesis. The result shows that pValues for factor *f1_entity_change_percentage* and *f3_Num_tests_per_entity* are rather small, much smaller than 0.01. In fact, they are a few orders of magnitude less than 0.01. This indicates that these two factors

are likely the impacting factors of the observed deviation. On the other hand, factor *f2_element_change_percentage* and *f4_RTS_technique* have pretty high pValues indicating that these two are not the main factors impacting deviation and may not be necessary in the regression model.

• **Number of observations** Number of rows without any NaN (*i.e.* Not a Number) values. The total number of combinations between test subject, build type, test suite type and RTS technique is 276, therefore we have 276 observations.

• **Error degrees of freedom** $n - p$, where n is the number of observations, and p is the number of coefficients in the model, including the intercept. In the result, the model has six coefficients, so the Error degrees of freedom is 276 - 6 = 270.

• **Root mean squared error** Square root of the mean squared error, which estimates the standard deviation of the error distribution.

• **R-squared and Adjusted R-squared** R^2 is a statistic that gives some information about the goodness of fit of a model. In regression, the R^2 coefficient of determination is a statistical measure of how well the regression line approximates the real data points. An R^2 of 1 indicates that the regression line perfectly fits the data. In the result, the R-squared value suggests that the model explains approximately 21.4% of the variability in the dependent variable.

To visually understand the effects of factor f1 and f3, we plotted a colour coded scatter plot in Figure 5.3. The x-axis of the graph is the percentage of entity changed (factor f1), and y-axis of the graph is the average number of tests per entity (factor f3). The absolute deviation is presented by colour, where red is high deviation, blue is low deviation. Figure 5.3 contains entire 276 data points including data from all test subjects, versions and techniques. Figure 5.3 shows that most of the blue markers are clustered at the left bottom corner where both factors have small values. This suggests that the RW predictor predicts closer to the actual value when the the percentage of entity changed is small and the average number of tests per entity is also small. As either x or y increases, there are more green and red markers which suggests increased deviation.

We have also plotted a scatter chart for Jacoco-core test subjects separately to visually display an ideal case. Figure 5.4 shows the distribution for Jacoco test subjects only. It contains data points from Jacoco-core releases and snapshots build for all RTS techniques. We can see from Figure 5.4 that most of the markers are blue or blueish, indicating low deviation values. In addition, we can also see clearly that as either x or y increases in the plot, data point colours are getting closer to red, which strongly suggests the impact of f1 and f3 to the deviation. The pattern in Figure 5.4 is not surprising as we known Jacoco employed test driven development methodology and its code coverage is very high.

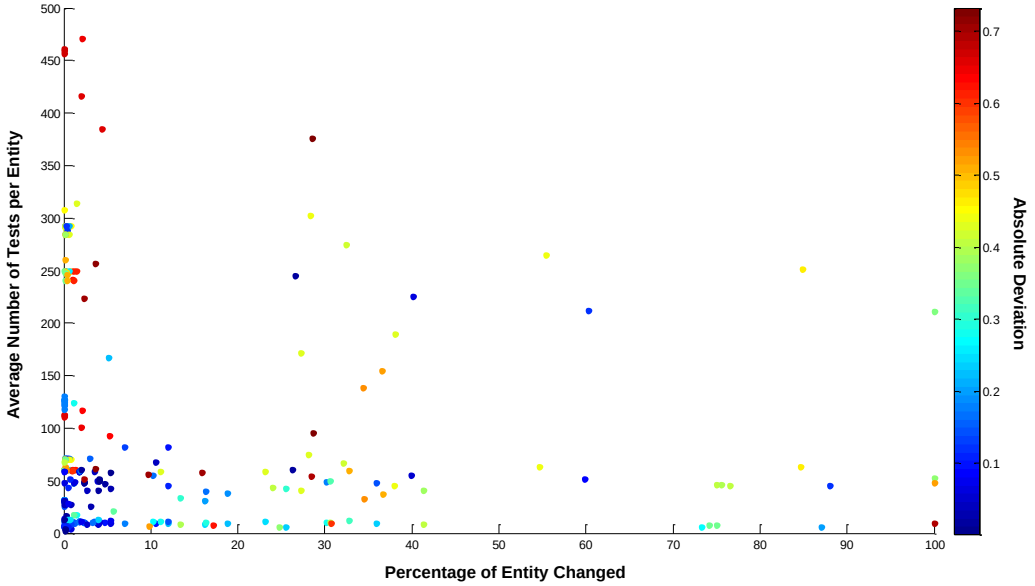

Figure 5.3: Scatter Plot - All

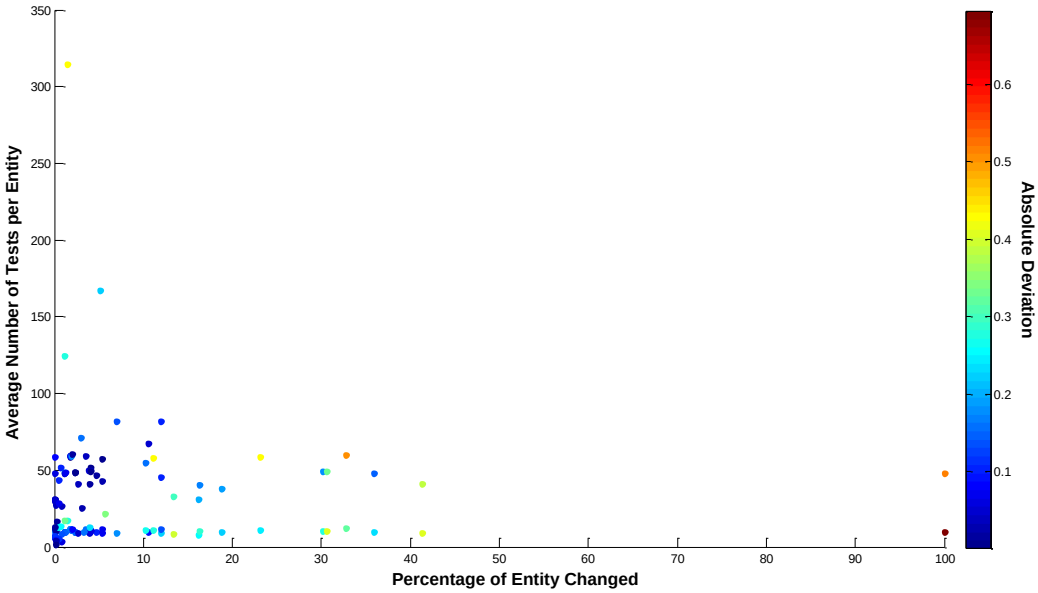

Figure 5.4: Scatter Plot - Jacoco-core

Jacoco builds on every commit which may help to keep the percentage of entity changes low. One hypothesis is that their development process may have encouraged developers to write independent and well-decomposed unit tests which would keep the number of test per entity low.

5.1.2 Obtained Results on Regression Functions

To obtain a simplified regression function, we excluded factor f2 and f4 from the input table, and then created a linear regression model using stepwise regression with *stepwiselm* function in MATLAB. The result of the analysis is a new fitting model. The new model and its coefficients are presented in Figure 5.5.

```
Linear regression model:
    y ~ 1 + x1 + x2 + x1^2

Estimated Coefficients:
                    Estimate      SE            tStat      pValue
    (Intercept)       0.029922      0.024837     1.2048    0.22934
    x1                0.011923     0.0022025     5.4136    1.3572e-07
    x2              0.00086727    0.00015579     5.5669    6.2151e-08
    x1^2            -9.887e-05    2.7214e-05    -3.6331    0.00033468

Number of observations: 276, Error degrees of freedom: 272
Root Mean Squared Error: 0.267
R-squared: 0.207, Adjusted R-Squared 0.198
```

Figure 5.5: Regression Function - General Prediction

The coefficient for x_1^2 is very small so we removed it from the regression function. We also round all coefficients to four decimal places. The simplified regression function is

$$\hat{y} = 0.0299 + 0.0119x_1 + 0.0009x_2 \tag{5.2}$$

where $\hat{y}$ is the estimated prediction error, x_1 is the entity change percentage, x_2 is the average number of tests per entity.

5.2 Study 2 - Version-specific Prediction

5.2.1 Obtained Results on Impacting Factors

Figure 5.6 shows the results of average absolute deviation $\overline{D_M}$ based on procedures and equations outlined in section 4.2 when the RW predictor is used as a version-specific predictor. Similar to the results of Study 1, the graph shows that for some test subjects and techniques, e.g., jacoco-core-snapshots-TM and jacoco-core-snapshots-TC, the RW predictor was quite successful. However, for some other test subjects and techniques, e.g., apache-solr-core-release-TM on td_stm technique, the RW predictor was not successful.

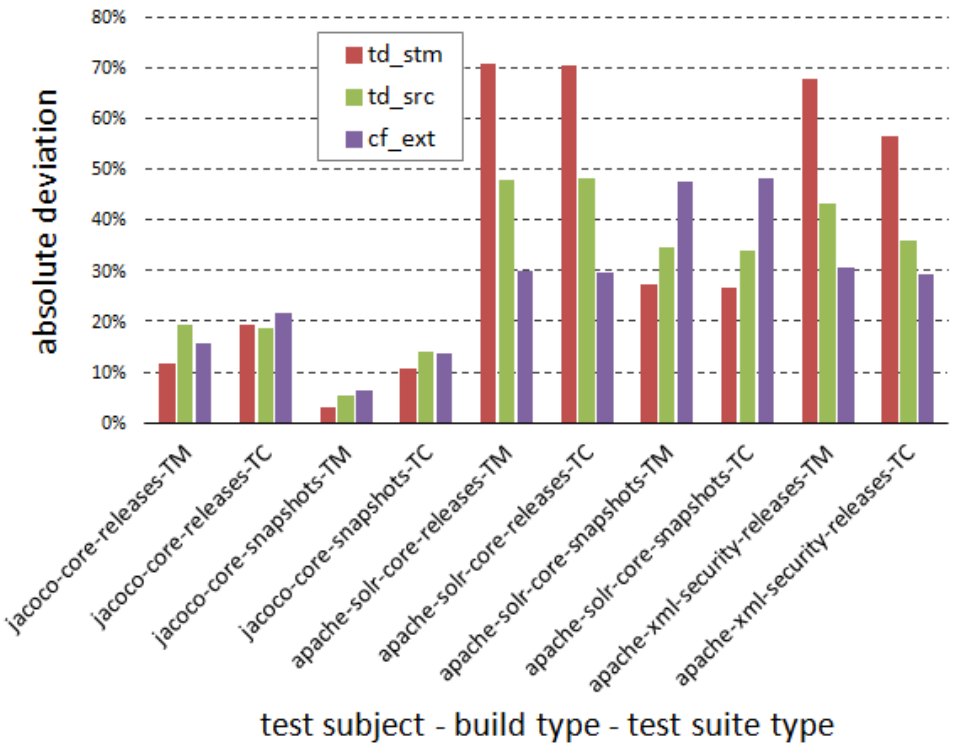

Figure 5.6: Average Absolute Deviation - Version-specific Prediction

Following the procedure in Algorithm 2, we computed the values of all factors and created a table for further analysis in MATLAB. The table consists of a row identifier column, 4 columns for the factors we defined in table 4.1 and the dependent variable deviation as the last column. The results from MATLAB *fitlm* function is shown in Figure 5.7.

```
Linear regression model (robust fit):
    Deviation ~ [Linear formula with 5 terms in 4 predictors]

Estimated Coefficients:
                                  Estimate     SE        tStat      pValue
    (Intercept)                     0.1034    0.0460     2.2467    0.025467
    f1_entity_change_percentage     0.0057    0.0009     6.2040    2.0598e-09
    f2_element_change_percentage   -0.0296    0.0172    -1.7256    0.085571
    f3_Num_tests_per_entity         0.0012    0.0002     7.1692    7.2314e-12
    f4_RTS_technique_td_src        -0.0913    0.0526    -1.7352    0.083853
    f4_RTS_technique_cf_ext        -0.0644    0.0540    -1.1923    0.23419

Number of observations: 276, Error degrees of freedom: 270
Root Mean Squared Error: 0.278
R-squared: 0.242, Adjusted R-Squared 0.228
```

Figure 5.7: Fitting Model - Version-specific Prediction

Figure 5.7 shows the pValues for *f1_entity_change_percentage* and *f3_Num_tests_per_entity* are quite small, much smaller than 0.01. This indicates that f1 and f3 are likely the impacting factors to the deviation. Similarly, factor *f4_RTS_technique* and *f2_element_change_percentage* are likely not the factors impacting the deviation as their pValues are pretty high.

5.2.2 Obtained Results on Regression Functions

Similar to Study 1, we used *stepwiselm* function to get a simplified regression function. The result is shown in Figure 5.8.

```
Linear regression model:
    y ~ 1 + x1 + x2 + x1^2

Estimated Coefficients:
                    Estimate       SE            tStat      pValue
    (Intercept)       0.021273      0.024778    0.85855       0.39135
    x1                0.012823     0.0021973      5.836    1.5172e-08
    x2               0.00091193    0.00015542     5.8675    1.2824e-08
    x1^2            -0.00010489    2.7149e-05    -3.8637    0.00013971


Number of observations: 276, Error degrees of freedom: 272
Root Mean Squared Error: 0.267
R-squared: 0.232, Adjusted R-Squared 0.224
```

Figure 5.8: Regression Function - Version-specific Prediction

After rounding the coefficients, the obtained regression function when the RW predictor is used as version specific predictor is :

$$\hat{y} = 0.0213 + 0.0128x_1 + 0.0009x_2 - 0.0001x_1^2 \tag{5.3}$$

where $\hat{y}$ is the estimated prediction error, x_1 is the entity change percentage, x_2 is the average number of tests per entity.

5.3 Causal Relation Analysis

Linear regression analysis of the dependent variable and the independent variables in Study 1 and Study 2 revealed an interesting relationship between the RW predictor's accuracy and its impacting factors. Both studies suggested that *f1_entity_change_percentage* and *f3_Num_tests_per_entity* are impacting factors to the deviation. However, a correlation between two variables does not directly imply cause and effect relationship. We perform informal casual analysis below to reason the relationship we discovered in our studies.

• **f1: percentage of entity changed**. Recall in Section 2.2.1, we discussed that Rosenblum and Weyuker made two important assumptions in the RW prediction model. One of the assumptions is that there is only single entity changed between two versions. It is reasonable to assume that the RW predictor works the best when there is exactly one entity changed and the deviation increases as the percentage of changed entities increases.

• **f2: average number of tests per entity**. Average number of tests per entity represents the degree of code coverage overlaps between test cases. In an ideal code coverage relation, each entity is covered by one and only one unique test case and there is no coverage overlap between test cases. In this situation, the Coverage Matrix can be arranged into a diagonal matrix in which the elements on the main diagonal are all one and elements outside of main diagonal are all zero. Element $C_{i,j}$ of an ideal Coverage Matrix C^I is defined as

$$C^I_{i,j} = \begin{cases} 1 & \text{if i equals j} \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

In an ideal coverage relation, RTS technique will always select one test case if there is only one entity changed, so the actual selection percentage is $1/|T|$. Since there are only ones

on the main diagonal of the Coverage Matrix, cumulative coverage CC, which is the sum of all ones in the coverage matrix, equals number of covered entity $|E^C|$. Given

$$CC = |E^C| \tag{5.5}$$

Predicted percentage can be simplified into

$$\pi_M = \frac{CC}{|E^C||T|} \tag{5.6}$$

$$= \frac{1}{|T|} \tag{5.7}$$

which is exactly the actual selection percentage. So in an ideal code coverage relation, the deviation between the RW predictor predicted percentage and actual percentage is zero. As the coverage overlap between test cases increases, the predicted percentage starts to deviate from actual and deviation increases.

5.4 Lessons Learned

The research results from the two studies suggest the following for the industry practitioners:

• **RQ1 : Impacting Factors.** In both studies, we have shown that the pValues of factor 1 and 3 are very small which indicate that *percentage of entity changed* and *average number of tests per entity* are the factors impacting the RW predictor's accuracy. This suggests that organizations already applied Continuous Delivery practices are likely to find the RW predictor performs well on their projects, as Continuous Delivery practices often result in small amount of code changes between revisions which improves the RW predictor's accuracy. On top of that, organizations have processes in place to promote independent and well-decomposed tests should be encouraged to adopt the RW predictor, as well-decomposed tests often result in low number of test per entity which improves the RW predictor's accuracy.

• **RQ2 : Regression Functions.** We have presented regression function 5.2 and 5.3 which model the RW's prediction error. In practice, these regression functions can help test mangers to gain a better understanding of the confidence level of the prediction. A

low $\hat{y}$ value indicates good confidence in RW's prediction; A high $\hat{y}$ value indicates that the RW predictor may not perform well for the subject program and an alternative method need to be pursued.

5.5 Threats to Validity

The section describes the threats to the validity of our study and the approaches we used to limit their effects.

5.5.1 Construct Validity

The dependent variable deviation is not the only way to measure the difference between predicted value and actual value. Future research should experiment with deviation and mean squared error. Also, the RW prediction error is measured by the difference in actual and predicted percentage which is relative to the size of the test suite. Care has been taken to select experiments with different test suite size. Furthermore, there is no definitive way to determine what is counted as a test case. We have experimented to use each test class as a test case and each test method as a test case. However, random inspection reveals that some unit test methods contain multiple assertions. Potentially each assertion could also be count as a test case which will change the coverage relation and test suite size.

5.5.2 Internal Validity

Linear regression analysis of dependent and independent variables may reveal interesting relationship between some factors and deviation. However, it may not indicate a cause and effect relationship. We have taken the effort in section 5.4 to analyze the reasoning behind the relationship we discovered in our study. Four potential factors are selected as independent variables in our studies. However, the selection of factors is not systematic, nor comprehensive. The factors we selected is biased on what we know about the internal implementation of the RW predictor. Future studies with additional factors may help validate our results.

5.5.3 External Validity

We have selected test subjects from a range of size (5k-80k LOC), applications, development processes, and different repositories. However these subjects do not represent the entire range of size and complexity of real world software applications. Replicating this study on real industry software may reveal different relationships between dependent variable and independent variables.

Moreover, the RW predictor is highly dependent on the Coverage Matrix and the Coverage Matrix is generated by executing test cases on test subjects. Special care must be taken to ensure test cases are not created to generate a specific coverage pattern in the first place. We have taken the effort to understand the test cases creation processes of our test subjects. As far as we know by reading test subjects' wiki pages, release notes and readme files, these test cases are real function test cases developed by the original development teams and they were not generated by any tools or for any special purpose other than functional testing.

We have implemented three RTS techniques in our study and the test selection percentages from these three implemented techniques are considered actual values. However, RTS techniques rely on a range of tools to parse source code, collect code coverage, and analyze change information. Same RTS technique implemented with different tools may produce different "actual" values. To mitigate this risk, we designed the PRIME framework to be extensible so that same RTS techniques implemented with different tools, same techniques implemented by other researchers, or other RTS techniques can all be plugged into the framework to further validate our experiments.

5.6 Summary

In this chapter, we presented and discussed experiment results around the first two research questions raised in Chapter 4. Through the experiments, we identified two impacting factors to the RW predictor's accuracy and we successfully modelled the RW prediction error with a linear function. We also reasoned that the correlation observed between impacting factors and deviation is likely a cause and effect relationship. We then suggested some actions for organizations in practice in order to improve the RW predictor's accuracy. Lastly, we discussed the threats to validity of our studies.

Since we have the functions to estimate the RW prediction error, the natural next step is to use the error functions to estimate the RW prediction errors on other subject

programs. In the next chapter, we present an improved RW predictor utilizing the error estimation functions, alongside with two other improvements to the RW predictor.

Chapter 6

Improved RW Prediction Models

In Chapter 5, we focused on studying the existing RW predictor's performance by understanding its impacting factors and estimating its prediction errors. The purpose of studying the RW predictor's performance is to improve it. In this chapter, we present three improved RTS prediction models extended from the original RW prediction model. The first two improved models are proposed by us while the last model was originally proposed by Harrold *et al.* [15] but it was never evaluated. Since we have implemented the PRIME framework to evaluate RTS predictors, we take the opportunity to evaluate Harrold's improved predictor together with the predictors we proposed. In the following sections, we discuss the improved prediction models then compare their performance with the original RW prediction model.

6.1 Improved Predictor: Utilizing the Error Estimator

In Chapter 5, we presented two linear functions modelling the RW prediction errors in Equation 5.2 and 5.3. These models are the results of regression analysis of the RW prediction errors on test subjects listed in Table 4.2. If these regression functions are good estimation of the RW prediction errors, we can utilize them to estimate the RW prediction errors of other programs. In this section, we present an improved predictor utilizing the error estimators and conduct experiments to evaluate its effectiveness.

6.1.1 The Improved Predictor

By performing regression analysis of the RW prediction errors, we have created the following two error estimators:

• the RW prediction error when it is used as a general predictor can be estimated by:

$$\hat{y} = 0.0299 + 0.0119x_1 + 0.0009x_2 \tag{6.1}$$

• the RW prediction error when it is used as a version-specific predictor can be estimated by:

$$\begin{aligned} \hat{y} = {} & 0.0213 + 0.0128x_1 + 0.0009x_2 \\ & - 0.0001x_1^2 \end{aligned} \tag{6.2}$$

where $\hat{y}$ is the estimated prediction error, x_1 is the entity change percentage, x_2 is the average number of tests per entity.

Then we define the improved RW predictor utilizing the error estimator as:

$$\Pi_M^e = \pi_M + \hat{y} \tag{6.3}$$

where

Π_M^e is the percentage of test cases that the improved predictor predicts will be selected by RTS technique M

π_M is the percentage of test cases that the RW predictor predicts will be selected by RTS technique M

$\hat{y}$ is the estimated RW prediction error

The procedures to compute Π_e are similar to the procedures outlined in Section 4.1 and 4.2 with some minor changes. Rather than computing values of *all* factors, we only compute entity_change_percentage and num_test_per_entity as the error estimators only require these two factors as input. For entity_change_percentage, we simply use the average of entity_change_percentage of all previous versions to predict the value of current version.

For num_test_per_entity, it is calculated using the Coverage Matrix of the previous version. Detail description of these two factors can be found in Section 4.3.

Since the error estimators are created from the evaluation of test subjects listed in Table 4.2, we can not use the same subjects to evaluate this improved predictor again. We extract apache-ant from SIR repository and list it below in Table 6.1.

Apache-ant is a Java based build tool supplied by Apache project. Versions used in this study map to Apache Ant original version 1.2 to 1.6 developed between Oct 2000 and Oct 2003. We use JUnit test cases supplied by original developers without any modification.

Table 6.1: Additional Test Subject Program

Subject Name	**Description**	**Lines of Code**	**Num. of Classes**	**Repository**	**Num. of Versions**	**Num. of Test Class (Method)**
apache-ant	Java build tool	80,500	627	SIR	9	150(878)

In the next section, we present results of our evaluation on the improved predictor using apache-ant as test subject.

6.1.2 Evaluation Results

Figure 6.1 shows results of comparing the average absolute deviation between the RW predictor and the improved predictor when they used in general prediction. The average absolute deviation of the RW predictor shows as purple bars in the chart; The average absolute deviation of the improved predictor shows as blue bars in the chart. Figure 6.1 shows that blue bars are lower than purple bars in four out of six cases which indicates that the improved predictor has less absolute deviation than the original RW predictor. In the other two cases, the improved predictor's performance is still as good as the original RW predictor's performance.

Similarly, Figure 6.2 shows results of comparing the average absolute deviation between the RW Predictor and the improved predictor when they are used in version-specific prediction. The results show that the improved predictor has less deviation than the original RW predictor in five out of six cases. In the other case, the improved predictor's performance is still as good as the original RW predictor's performance.

In general, we observe similar performance improvement of the improved predictor in general prediction and version-specific prediction. The evaluation results demonstrate that

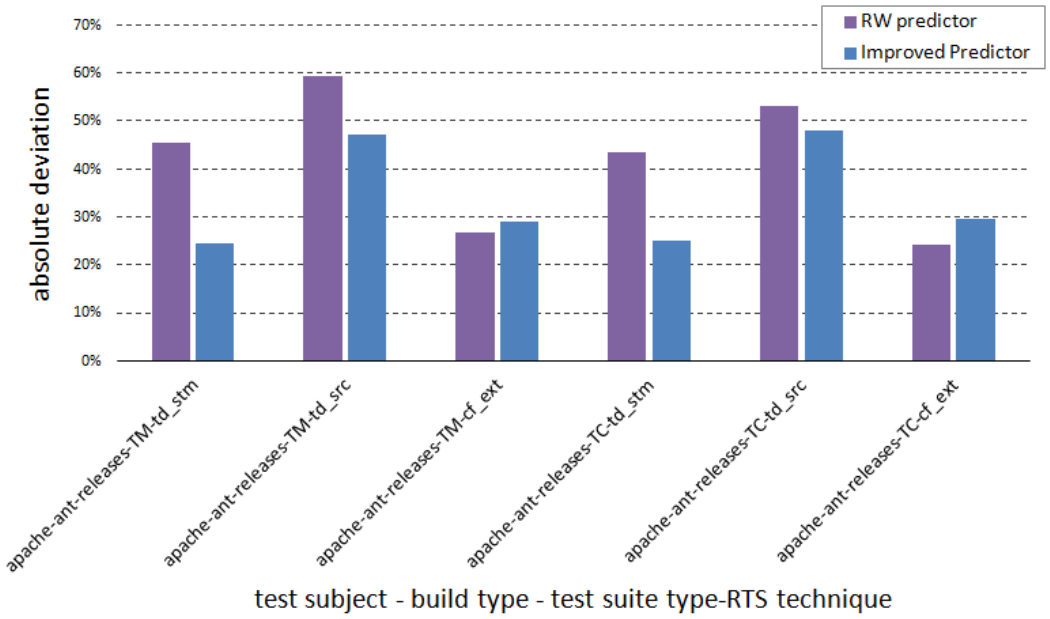

Figure 6.1: Comparison of Average Absolute Deviation - RW Predictor vs. Improved Predictor - General Prediction

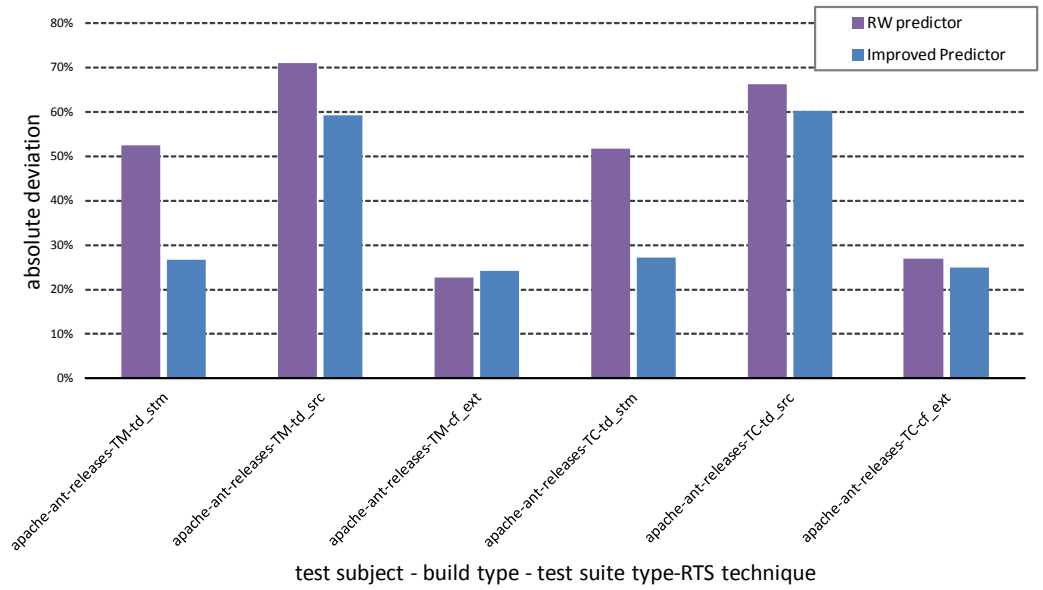

Figure 6.2: Comparison of Average Absolute Deviation - RW Predictor vs. Improved Predictor - Version-specific Prediction

by utilizing the error estimators, the prediction performance can be improved. The three cases where the improved predictor does not perform better than the RW predictor are all with the cf_ext technique. In the next section, we present an improved predictor specific for the cf_ext technique.

6.2 Improved Predictor: Incorporating Class Dependencies

In Section 2.2.1, we have presented the RW predictor equation as follows:

$$\pi_M = \frac{CC}{|E^C||T|} \tag{6.4}$$

where

π_M is the RW predicted percentage of test cases will be selected by technique M

CC is the sum of all ones in the coverage matrix

$|E^C|$ is the number of covered entities

$|T|$ is the size of the test suite.

In this RW predictor equation, both CC and $|E^C|$ are computed from the coverage matrix. Since one test suite covers a fix set of entities, the coverage matrix of a particular code entity level (e.g., statement, basic block, method and class) should remain constant. Therefore, two RTS techniques operate on the same code entity level would have the same predicted test selection percentage based on the RW predictor equation above. However, in reality, different RTS techniques often select different test cases, therefore have different test selection percentages. As we discussed in Section 4.5.2, Class Firewall technique (cf_ext) takes into consideration the dependency information between classes. In addition to select test cases cover modified classes, cf_ext also selects test cases cover classes directly or transitively dependent on the modified classes. In this section, we present an improved RW predictor tailored for cf_ext technique. We then conduct experiments to demonstrate that the improved RTS technique-specific predictor outperforms the original RW predictor on cf_ext technique.

6.2.1 The Improved Predictor

Take a hypothetical Coverage Matrix in Figure 6.3a as an example, the Coverage Matrix represents the coverage relation between program P, which contains 3 classes c_0, c_1, c_2, and test suite T, which contains 4 test cases t_0, t_1, t_2, t_3. Each one in the matrix indicates the entity is covered by the test case of the corresponding column, while each zero indicates the entity is not covered by the test case. Based on Equation 2.6, Cumulative Coverage CC is 6, size of test suite $|T|$ is 4 and the number of covered entity $|E^C|$ is 3. The original RW predictor predicts the percentage of test cases will be selected is $6/(3 * 4)$ which is 50%.

	c_0	c_1	c_2
t_0	1	0	0
t_1	0	1	0
t_2	0	0	1
t_3	1	1	1

(a) Original RW Predictor's Coverage Matrix

	c_0	c_1	c_2
t_0	1	1	0
t_1	0	1	0
t_2	0	0	1
t_3	1	1	1

(b) Coverage Matrix Combining Dependency Information

Figure 6.3: An Example Coverage Matrix

Now assume that based on class dependency analysis, we know c_0 depends on c_1 ($c_0 \rightarrow c_1$). Since c_0 depends on c_1, change in c_1 could potentially impact c_0's functionality. When c_1 is changed, cf_ext technique would also select all test cases cover c_0 for regression test in order to make sure c_0's functionality is not negatively impacted by the change. Therefore, the cf_ext RTS technique would first select t_1 and t_3 to cover c_1 , then select t_0 and t_3 which covers c_0. So the total percentage of tests actually selected by cf_ext is 3/4 which is 75%. In this case, the deviation between actual and RW predicted percentage is 25%.

To improve the RW predictor, we propose an algorithm to merge the class dependency information into the Coverage Matrix. We copy all the ones from column c_0 into column c_1 to create a new Coverage Matrix shows in Figure 6.3b. Cumulative Coverage CC of the new Coverage Matrix is 7 and the predicted percentage by the improved predictor is $7/(3*4)$ which is 58%. The deviation of the improved predictor is 17% which has improved from the original RW predictor's deviation of 25%.

Formally, we define Algorithm 3 to transform a Coverage Matrix to a new Coverage

Matrix that merges all class dependency information. The predicted percentage of test cases will be selected by cf_ext technique is:

$$\Pi_{cf\_ext} = \frac{CC_{cd}}{|E^C||T|} \tag{6.5}$$

where

Π_{cf_ext} is the percentage of test cases that the improved predictor predicts will be selected by RTS technique cf_ext

CC_{cd} is the cumulative coverage of the Coverage Matrix combining dependency information

$|E^C|$ is the total number of covered entities

$|T|$ is the number of tests in the test suite

Algorithm 3 Merge dependency information into coverage matrix

```
1: for all class c in the columns of Coverage Matrix C do
2:    Dependent_c ← find all inbound dependencies of c including transitively dependent
      classes
3:    for all c' ∈ Dependent_c do
4:       for all test case t in the rows of C do
5:          if covers_M(t, c') = 1 then
6:             covers_M(t, c) ← 1
7:          end if
8:       end for
9:    end for
10: end for
```

Once the new Coverage Matrix combining dependency information is created, we follow the same procedures as Study 1 and Study 2 to evaluate this improved predictor. The results of our evaluation are presented in the next section.

6.2.2 Evaluation Results

We have performed evaluation of the improved predictor incorporating class dependency on test subjects listed in Table 4.2. Since the improvement is for Class Firewall technique only, we compare the performance of the improved predictor with the RW predictor on cf_ext technique. Figure 6.4 shows the comparison of average absolute deviation between the RW predictor and the improved predictor when both are used as general predictor. The result shows that the improved predictor outperforms the RW predictor in nine out of ten cases. Notable improvement is observed on apache-xml-security-releases-TC where the deviation has been reduced from 26.3% to 5.7%.

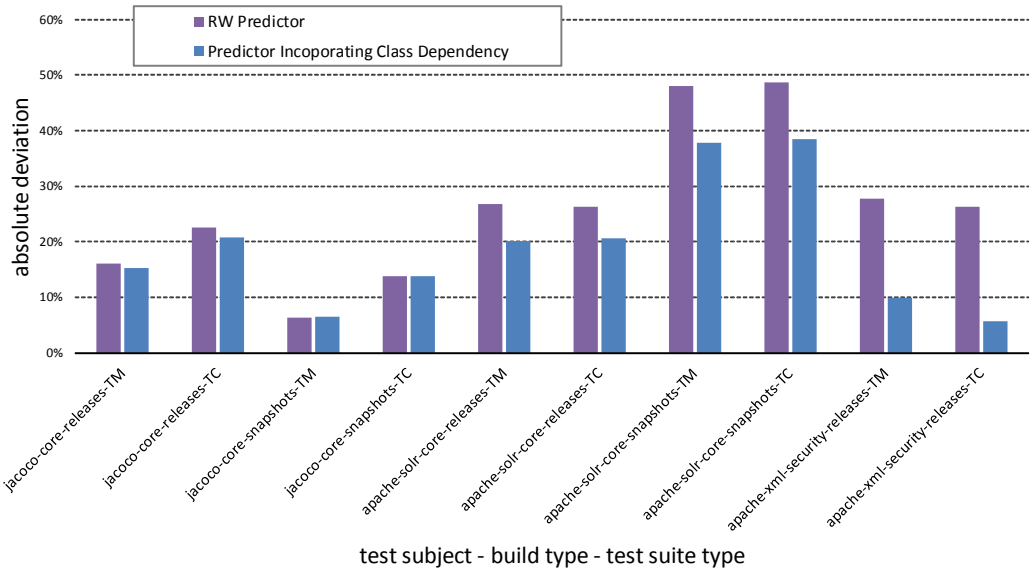

Figure 6.4: Comparison of Average Absolute Deviation - RW Predictor vs. Improved Predictor Incorporating Class Dependency- General Prediction

Similarly, Figure 6.5 shows the the comparison of average absolute deviation between the RW predictor and the improved predictor when both are used as version-specific predictor. The result shows that the improved predictor outperforms the RW predictor in eight out of ten cases.

Note in Algorithm 3 that finding $Dependent_c$ for all classes is computationally expensive. However, this is also the first step in the Class Firewall technique. In practice, class dependencies only needs to be computed once and the results are first used by the improved predictor then used by the Class Firewall technique. So there is no additional

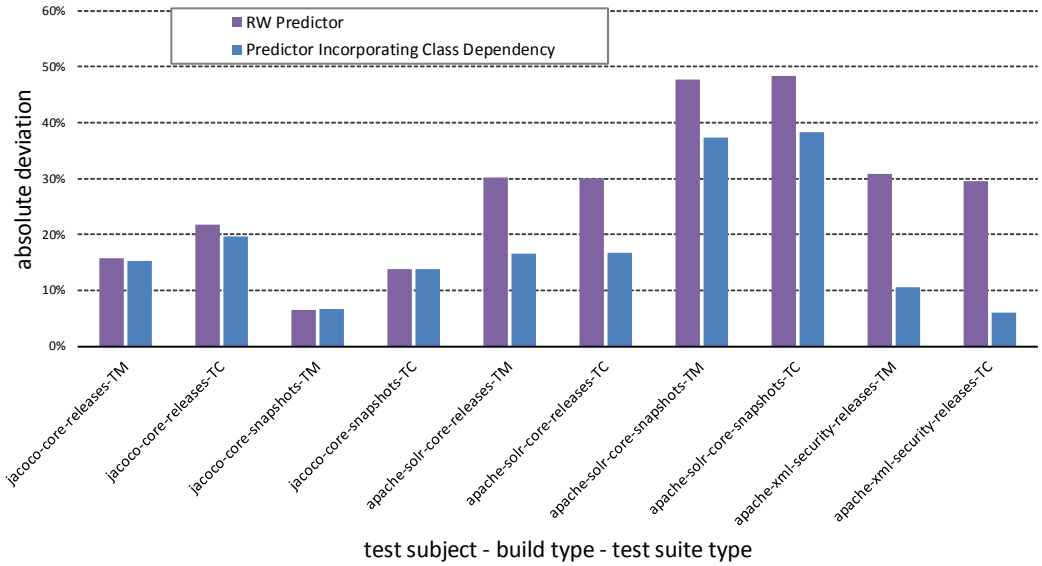

Figure 6.5: Comparison of Average Absolute Deviation - RW Predictor vs. Improved Predictor Incorporating Class Dependency- Version-specific Prediction

overhead to apply this improved predictor. Based on our experimental results, we believe incorporating class dependencies does improve the RW prediction performance. Moreover, this experiment opens up a new direction of creating RTS technique-specific predictors. As the field of Regression Test Selection evolves, new RTS techniques are being proposed and evaluated. We believe there should be a co-evolvement of RTS predictors to incorporate specific characteristic of individual RTS techniques.

6.3 Improved Predictor: Incorporating Change Frequency

The last improvement on the RW predictor was proposed by Harrold *et al.* [15] which we have discussed in Section 2.2.2. Authors proposed an improved prediction model to incorporate relative frequency of changes to the covered entities. Due to the limitation in SIR repository, test subjects in SIR do not have change history information therefore this improved predictor was not evaluated.

Since we extracted apache-solr-core and jacoco-core directly from version control system, we were able to get the change histories of entire development life time of these two

subjects. We use *git log* and *svn log* command to fetch all the commits from the beginning of the projects to the current revision. The output of the commands lists which file is added, modified or deleted on each commit. We parse through the command outputs to increment the change counter for a source file whenever the file is modified or deleted. The change frequency of a source file entity is computed by its number of changes over the total number of changes of all source entities. This guarantees that the sum of change frequency of all source file entities is one.

Once we have the change frequency of each source file entity, similar to previous studies, we conduct two studies to evaluate this improved predictor both as general predictor and version-specific predictor. In the following section, we discuss the evaluation results of this improved predictor.

6.3.1 Evaluation Results

Figure 6.6 shows the comparison of average absolute deviation between the original RW predictor and the improved predictor incorporating change frequency when the predictor is used as general predictor. The results show the improved predictor has less deviation than the orignal RW predictor. The blue bars are lower than the purple bars in all cases. However, the results also show that the improvement is insignificant. The performance difference between the two predictors are less than 1% in all cases.

Similarly, Figure 6.7 shows the comparison of average absolute deviation between the original RW Predictor and the improved predictor incorporating change frequency when the predictor is used as *version-specific* predictor. We can see similar trend as in Figure 6.6 that the improved predictor performs better than the original RW predictor in all cases though the performance improvement is not significant.

6.4 Summary

In this chapter we presented three improvements to the RW predictor and our evaluation shows that all three improved predictors outperformed the original RW predictor in most cases. This is very encouraging as, from research point of view, it opens up many research directions. For example, we can evaluate more subject programs to improve the error estimators; We can research RTS techniques to develop more technique-specific predictors; Or we can take advantage of existing research on change location prediction to generate more accurate weight values. In Section 7.2, we discuss our future works in more details.

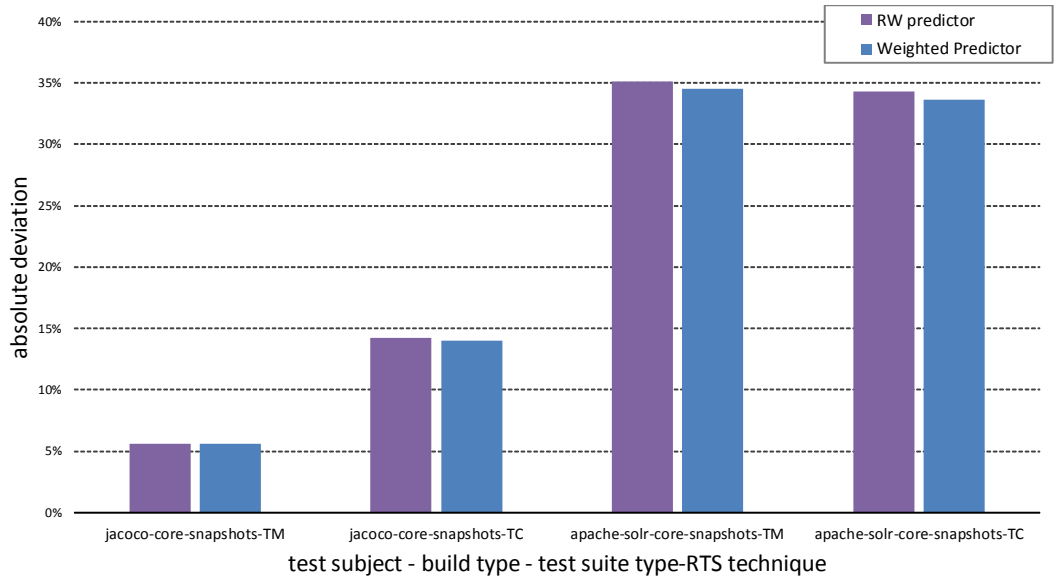

Figure 6.6: Comparison of Average Absolute Deviation - RW Predictor vs. Improved Predictor Incorporating Change Frequency- General Prediction

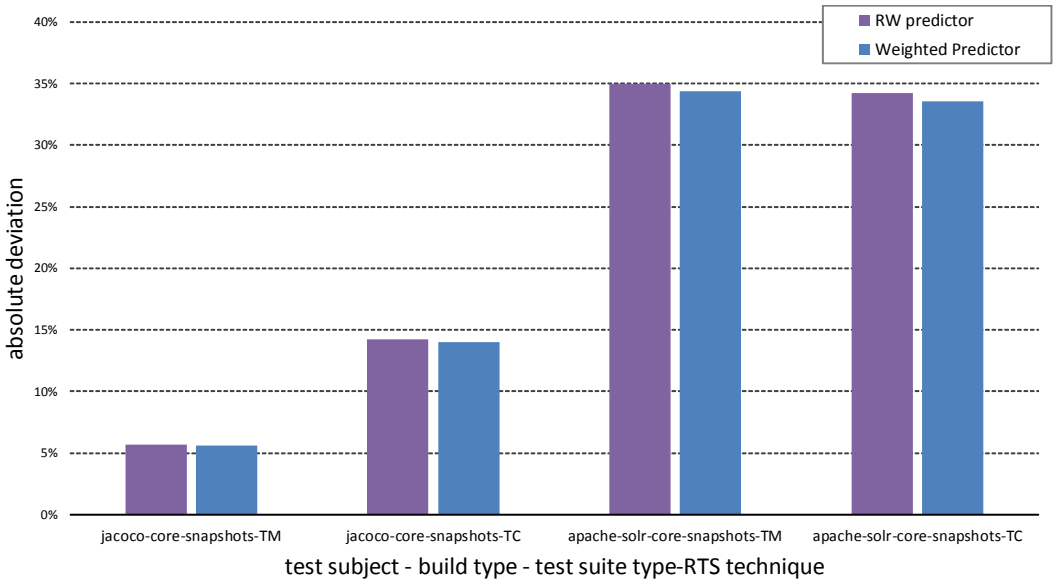

Figure 6.7: Comparison of Average Absolute Deviation - RW Predictor vs. Improved Predictor Incorporating Change Frequency- Version-specific Prediction

Our evaluation results of the improved predictors are also very encouraging to industry practitioners. Depending on organizations development processes, test engineers may adopt one of the improved predictors or a combination of the three. For an organization already has experience with the RW predictor and has been tracking the deviation between actual and predicted percentage over time, it is straightforward to build a regression model to estimate the RW prediction error. Then the improved predictor utilizing the error estimator is probably the best choice. For an organization using the Class Firewall RTS technique, the improved predictor incorporating class dependencies is a good choice as we see visible performance improvement from our evaluation results. For an organization develops a product over a long period of time, the improved predictor incorporating change frequency is also a good pick as change frequency can be extracted from configuration management system fairly easily. Moreover, if an organization has history data with the RW predictor, uses Class Firewall technique and maintains change history data, it can use the combination of all three improved predictors. As one of the proposed future works suggests, we may evaluate the effectiveness of such scenario and provide additional data for industry to adopt these predictors.

Chapter 7

Conclusions and Future Works

In this chapter, we first summarize this thesis and then we discuss potential future directions to extend this work.

7.1 Conclusions

In this thesis, we presented results from two set of experiments that were designed to study the performance of the Rosenblum-Weyuker (RW) prediction model for predicting the percentages of test cases selected by RTS techniques. For this purpose, we implemented three RTS techniques from the literature and evaluated the performance of the RW predictor on a selected set of Java test subjects. In order to study the high deviation in the RW prediction performance, we considered a set of four factors that could potentially impact the prediction performance. Among the studied factors, we concluded that the *amount of code changes* and *code coverage overlaps between test cases* are two factors that actually contribute to the RW performance. Through regression analysis on these factors, we proposed two linear models to estimate the RW prediction errors.

The study of impacting factors and the RW prediction errors opens up opportunities to come up with improved RTS prediction models. We proposed an improved RW predictor utilizing the error estimator and an improved RW predictor incorporating class dependencies. The evaluation results shown that both improved predictors perform better than the original RW predictor. In addition, we evaluated the improved RW predictor incorporating change frequency which was proposed by Harrold *et al.* Our evaluation demonstrated Harrold's improved RW predictor also preform better than the original RW predictor.

7.2 Future Works

"A journey of a thousand miles begins with a single step". In our view, this work is just the first step of a long journey. We looked at the cause of errors of existing prediction models and proposed a few improvements. However, prediction is a hard problem and there are still a lot more we can do to improve the prediction models. In this section, we discuss our thoughts on several future directions.

7.2.1 RTS Predictors

In this work, we have demonstrated the effectiveness of three improved predictors based on the RW predictor. There are continuing research we'd like to work on in the following areas:

• **Improved predictor: utilizing the error estimator** Our error estimators are built from the release and snapshot versions of three test subjects. We'd like to evaluate additional test subjects to further enhance the accuracy of the error estimators. Evaluate test subjects of different code size, test suite size and development process would further improve the error estimators.

• **Improved predictor: incorporating class dependencies** We'd like to develop more technique-specific RTS predictors to incorporate characteristics of individual RTS techniques. As the field of Regression Test Selection evolves, new RTS techniques are being proposed and evaluated. We believe there should be a co-evolvement of RTS predictors.

• **Improved predictor: incorporating change frequency** We have used the average change frequency as weight in this predictor. As a future work, we can take advantage of existing research on change location prediction to generate more accurate weight values. Ideally, the weight factor should reflex the likelihood of a particular entity being changed in the next revision. In addition to the change history, weight factor could also incorporate information such as code complexity, defect patterns and requirement changes.

• **Combination of the three improved predictors** As discussed in Section 6.4, it is possible to combine several predictors. We'd like to evaluate such scenario by combining all improved predictors and evaluate its performance.

The end goal of this research is to come up with more accurate RTS predictors. Rosenblum and Weyuker proposed the RW predictor with a constraint in mind that it should be simple as it does not make economical sense if the cost of prediction is great than the savings (*i.e.* execution cost of test cases not selected for regression). However, knowing is

power. From project planning and management prospective, there is huge benefit of knowing how many test cases would be selected for regression therefore knowing how much effort the regression testing would need. Also as the computing power increases and computing cost decreases, even complex prediction models can be computed in short period of time and low cost. In the future, we'd like to work on new RTS predictors taking advantage of emerging techniques in prediction models domain. We definitely would like to develop prediction models taking into account the amount of code changes and number of test cases per entity, as these are the impacting factors we identified from this work.

Another future research work is to extend the RW prediction models by incorporating techniques from software change prediction domain. Previous work such as one proposed by Amoui *et al.* in [1] could help to predict the number of changes and the locations of changes. Change prediction models would enable us to create new RTS predictors consider multiple entity changes which would make the RTS predictors applicable to a wide range of development processes.

7.2.2 Prediction Models Performance Evaluation Framework

Another line of future work is to utilize the RTS PRIME Framework to conduct more studies. Here are several areas we could explore related to the framework:

• **Test Subjects** It's fairly straightforward to add new test subjects from SIR, GitHub or Apache SVN into the PRIME framework. With reasonable effort, the PRIME framework can also extract test subjects from other repositories, including open source repositories and proprietary repositories. Evaluate prediction models performance with more test subjects could help us to confirm the factors impacting RW prediction accuracy and may also improve our error estimators. Particularly, programs with long development history and industry software applications would be very interesting test subjects.

• **RTS techniques** Regression Test Selection is an active research area. There are lots of RTS techniques exist in the literature and there are also emerging RTS techniques. Our framework contains implementation of three RTS techniques which build the foundation of additional RTS techniques. Though the focus of our framework is to evaluate prediction models performance, with some modification, it can be used to evaluate RTS technique's based on some different performance metrics.

• **Impacting Factors** In our study we considered four factors potentially contributing to the RW prediction error and through regression analysis two factors were identified as impacting factors. There are likely more factors could be considered. In the future, we'd like to create an expanded list of factors and conduct similar studies on these factors.

7.2.3 Framework Performance Improvements

The performance of PRIME framework has room of improvement. For example to evaluate apache-ant-releases-TC, which has 80KLOC and 150 test cases, it takes about 2 hours to complete the full evaluation on an Intel Core i7 machine with 8GB of RAM. This is mostly because in the evaluation run, we apply all prediction models on all versions and on all RTS techniques. With 4 prediction models, 17 version and 3 techniques, it is a combination of 204 runs. If the framework were to use in real industry environment, it only needs to predict once with one prediction model on one RTS technique. For example, in a regression process similar to Figure 1.2, a general prediction model only need to be run once at the beginning of the test cycle and that would dramatically reduce the time required to run the prediction. Nevertheless, for the prediction models to be really useful in an industry environment, the prediction is expected to take only seconds if not milliseconds to complete so that it won't add any delay to the existing processes. Here we outline several areas where performance can be improved.

• **Coverage Analysis** In the PRIME Framework, we have tracked time spent on each sub-process of the prediction. The results show that one of the major bottle necks is the Coverage Analysis. In the Coverage Analysis step, the framework parses the coverage reports generated by code coverage tool and builds the Coverage Matrix between test cases and covered entities. In order to build the Coverage Matrix, we run each test case separately and trigger a coverage dump after each individual test run. As a result, the size of the coverage reports is a function of number of code entities multiplies the number of test cases. In our study, we initially use EMMA code coverage tool[1] to collect code coverage. EMMA reports code coverage in plain text,XML or HTML format. However, EMMA only reports statement coverage in HTML format which takes much more space than the XML format. For apache-ant-releases-TC test subject, EMMA produces 700,000 HTML coverage reports files with total size over 11GB. Not only it consumes lots of disk space but also it takes significant computing time to parse these files in order to create Coverage Matrix. We later mitigate the problem by replacing EMMA with JaCoCo code coverage tool which outputs statement coverage in XML format. The size of the coverage report files has reduced to 1/5th of the size from EMMA and time taken to parse report files has been reduced significant as well. However, ideally we'd like to extend JaCoCo code to generate Coverage Matrix directly in memory which would eliminate the intermediate step of generating and parsing XML files. We expect to change JaCoCo reporting module so that it produces a binary list of zeros (not covered) and ones (covered) for the entire entity set.

[1] http://emma.sourceforge.net/

• **Dependency Analysis** In this thesis, we used Dependency Finder[2] to extract inbound dependent and transitively inbound dependent classes of each covered class. Finding transitive closure is a very expensive operation. In our study, Dependency Analysis takes average of 91 seconds for each version of apache-ant-releases-TC. As a future work, we'd like to explore other tools to reduce the computing time, memory consumption and improve scalability.

Additionally, one area we'd like to explore in the future is to use GPU computing to accelerate Coverage Analysis and Dependency Analysis on a single machine. Both processes seem to consist of many independent processing tasks which are good candidates for parallel programming.

7.2.4 Software Artifact Repository

Preparing test subjects for experiments is not a trivial task. It is very time consuming and any minor mistake would invalidate the entire research work. Do, Elbaum and Rothermel's work on building an infrastructure to support controlled experiment of software testing (SIR) has benefited many researchers over the years [11]. It is a great initiative to support reproducible research. SIR contains many well known test subjects that have been used in many RTS related experiments. Our initial plan was to take all test subjects from SIR but due to some challenges, we eventually only used 2 test subjects from SIR and had to extract other test subjects directly from GitHub and Apache SVN. We discuss our experience with SIR below and outline future work in obtaining test subjects.

• **Folder Structure:** SIR repository is a collection of many test subjects and provides valuable data for research. However, the repository is created and maintained manually. Though artifacts of each test subjects are organized in a predefined folder structure, they are not consistent and accurate enough for program to use. For example, *versions* directory exist in some test subjects but not in others; Compiled class files sometimes are placed under *build/classes* directory, sometimes in *build/ant/classes* directory(various by versions). These variation may be tolerated by a human user, but will cause errors in an automated program.

• **Test Execution Scripts:** For each test subject, SIR contains a test plan file which lists all test cases, and a test execution script(shell script) to run all tests. There are several cases where a test case exists in the test plan file but not in the test execution script, or vice versa. We assume these files are manually generated and there are human errors in creating these files.

[2]http://depfind.sourceforge.net

• **Debug Information:** SIR provides test subjects' source code and compiled versions. However, many subjects were not compiled with full debug data. For code coverage tool to generate statement coverage report, it requires a mapping from byte code to statement line number. And this data is only available if the source were compile with option *debug='true'*. We had to modify build script manually and then recompile every version of the test subjects. Due challenges of dependency management in legacy software, compiling these test subjects is nothing less than a nightmare.

• **Dependency Management:** Many test subjects in SIR are legacy software developed long time ago. There were no dependency management tool like Apache Maven available at the time. Therefore, information such Java version, Ant version and external libraries versions have to be dug out from release notes, readme files etc. When dependency information is not documented, compile becomes a trial and error exercise.

• **Metadata:** SIR basically stores offline copies of subject programs. Though it seems convenient to have a separate repository to keep all experiment subjects, some useful data are lost when programs are downloaded from the original development environment. For example, we compare two versions of the subject program to find out which entity is changed, but we don't know how many times the entity has been changed between the two versions. The change frequency data is lost as soon as test subjects are taken away from their development environment. Similarly, metadata such as who made the check in, when, and commit notes etc are not available in SIR.

We believe the future direction of an infrastructure to support controlled experiment is not to build another perfect repository, rather to build framework to extract programs from their version control systems on the fly. The framework should act as a proxy to various popular version control systems and code repositories. The framework provides one common set of APIs to testing techniques which consume artifacts extracted from test subject programs.

References

[1] M. Amoui, M. Salehie, and L. Tahvildari. Temporal Software Change Prediction Using Neural Networks. *International Journal of Software Engineering and Knowledge Engineering* (2009), pp. 995–1014.

[2] G. Antoniol et al. Recovering Traceability Links Between Code and Documentation. *IEEE Trans. Software Eng.* (2002), pp. 970 –983.

[3] P. Arkley and S. Riddle. Overcoming the Traceability Benefit Problem. In: *Proceedings of the IEEE International Conference on Requirements Engineering (RE)*. 2005, pp. 385–389.

[4] *Artifact (software development)*. 2012. URL: http://en.wikipedia.org/wiki/Artifact_(software_development).

[5] S. Beydeda and V. Gruhn. Integrating White- and Black-Box Techniques for Class-Level Regression Testing. In: *Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC)*. 2001, pp. 357–362.

[6] L. C. Briand, Y. Labiche, and G. Soccar. Automating Impact Analysis and Regression Test Selection Based on UML Designs. In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. 2002, pp. 252–261.

[7] G. Buchgeher et al. Towards Tool-Support for Test Case Selection in Manual Regression Testing. In: *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2013, pp. 74–79.

[8] S. Chen et al. Using Semi-supervised Clustering to Improve Regression Test Selection Techniques. In: *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2011, pp. 1–10.

[9] Y. Chen, R. L. Probert, and D. P. Sims. Specification-based Regression Test Selection with Risk Analysis. In: *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. 2002, pp. 1–14.

[10] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. TestTube: A System for Selective Regression Testing. In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 1994, pp. 211–220.

[11] H. Do, S. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Softw. Engg.* (2005), pp. 405–435.

[12] H. Do et al. The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments. *Software Engineering, IEEE Transactions on* 36.5 (2010), pp. 593–617.

[13] E. Engström, P. Runeson, and M. Skoglund. A Systematic Review on Regression Test Selection Techniques. *Information & Software Technology* (2010), pp. 14–30.

[14] R. A. Haraty, N. Mansour, and B. Daou. Regression Testing of Database Applications. In: *Proceedings of the ACM Symposium on Applied Computing*. 2001, pp. 285–289.

[15] M. J. Harrold et al. Empirical Studies of a Prediction Model for Regression Test Selection. *IEEE Trans. Software Eng.* (2001), pp. 248–263.

[16] M. J. Harrold et al. Regression Test Selection for Java Software. In: *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2001, pp. 312–326.

[17] H. Hemmati, A. Arcuri, and L. C. Briand. Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection. In: *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 2011, pp. 327–336.

[18] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[19] N. Kaushik, L. Tahvildari, and M. Moore. Reconstructing Traceability Between Bugs and Test Cases: An Experimental Study. In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 2011, pp. 411–414.

[20] H. K. N. Leung and L. White. A Study of Integration Testing and Software Regression at the Integration Level. In: *Proceedings of Conference on Software Maintenance (ICSM)*. 1990, pp. 290–301.

[21] H. K. N. Leung and L. White. Insights into Testing and Regression Testing Global Variables. *Journal of Software Maintenance* (Dec. 1990), pp. 209–222.

[22] J. Micco. *Tools for Continuous Integration at Google Scale, Google Tech Talk*. http://youtu.be/KH2_sB1A6lA. 2012.

[23] A. Nanda et al. Regression Testing in the Presence of Non-code Changes. In: *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2011, pp. 21–30.

[24] A. Orso, N. Shi, and M. J. Harrold. Scaling Regression Testing to Large Software Systems. In: *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 2004, pp. 241–251.

[25] A. Orso et al. Using Component Metadata to Regression Test Component-based Software. *Softw. Test., Verif. Reliab.* (2007), pp. 61–94.

[26] N. Rachatasumrit and M. Kim. An Empirical Investigation into the Impact of Refactoring on Regression Testing. In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. 2012, pp. 357–366.

[27] B. Ramesh et al. Implementing Requirements Traceability: a Case Study. In: *Proceedings of the IEEE International Symposium on Requirements Engineering (RE)*. 1995, pp. 89 –95.

[28] D. S. Rosenblum and E. J. Weyuker. Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies. *IEEE Trans. Software Eng.* (1997), pp. 146–156.

[29] G. Rothermel and M. Harrold. A Framework for Evaluating Regression Test Selection Techniques. In: *Proceedings of the International Conference on Software Engineering*. 1994, pp. 201–210.

[30] G. Rothermel and M. Harrold. A Safe, Efficient Algorithm for Regression Test Selection. In: *Proceedings of the Conference on Software Maintenance (ICSM)*. 1993, pp. 358–367.

[31] G. Rothermel and M. Harrold. Empirical Studies of a Safe Regression Test Selection Technique. *IEEE Trans. Software Eng.* (1998), pp. 401–419.

[32] G. Rothermel and M. J. Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Trans. Softw. Eng. Methodol.* (Apr. 1997), pp. 173–210.

[33] G. Rothermel and M. J. Harrold. Analyzing Regression Test Selection Techniques. *IEEE Trans. Software Eng.* (1996), pp. 529–551.

[34] G. Rothermel and M. J. Harrold. Selecting Tests and Identifying Test Coverage Requirements for Modified Software. In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1994, pp. 169–184.

[35] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression Test Selection for C++ Software. *Softw. Test., Verif. Reliab.* (2000), pp. 77–109.

[36] M. Skoglund and P. Runeson. A Case Study of the Class Firewall Regression Test Selection Technique on a Large Scale Distributed Software System. In: *Proceedings of the International Symposium on Empirical Software Engineering*. 2005, pp. 74–83.

[37] F. I. Vokolos and P. G. Frankl. Pythia: A Regression Test Selection Tool Based on Textual Differencing. In: *Proceedings of the Internatinal Conference on on Reliability, Quality and Safety of Software-intensive Systems (ENCRESS)*. 1997, pp. 3–21.

[38] F. I. Vokolos and P. G. Frankl. Empirical Evaluation of the Textual Differencing Regression Testing Technique. In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. 1998, pp. 44–53.

[39] L. White and H. Leung. A Firewall Concept for Both Control-flow and Data-flow in Regression Integration Testing. In: *Proceerdings of the Conference on Software Maintenance (ICSM)*. 1992, pp. 262–271.

[40] L. White et al. Test Manager: A Regression Testing Tool. In: *Proceedings of the Conference on Software Maintenance (ICSM)*. 1993, pp. 338–347.

[41] G. Wikstrand et al. Dynamic Regression Test Selection Based on a File Cache An Industrial Evaluation. In: *Proceedings of the International Conference on Software Testing Verification and Validation (ICST)*. 2009, pp. 299–302.

[42] D. Willmor and S. Embury. A Safe Regression Test Selection Technique for Database-driven Applications. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. 2005, pp. 421–430.

[43] S. Yoo and M. Harman. Regression Testing Minimization, Selection and Prioritization: a Survey. *Softw. Test., Verif. Reliab.* (2012), pp. 67–120.