A METHOD FOR SELECTING REGRESSION TEST CASES BASED ON SOFTWARE CHANGES AND SOFTWARE FAULTS

REGRESYON TESTLERİNİN SEÇİMİ İÇİN YAZILIM DEĞİŞİKLİKLERİNE VE YAZILIM HATALARINA DAYALI BİR YÖNTEM

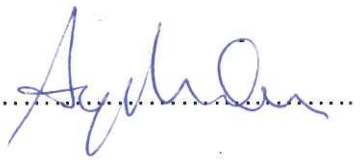
UĞUR YILMAZ

Assist. Prof. Dr. AYÇA TARHAN
Supervisor

Submitted to
Graduate School of Science and Engineering of Hacettepe University
as a Partial Fulfillment to the Requirements
for the Award of the Degree of Master of Science
in Computer Engineering

2019

This work titled "**A Method For Selecting Regression Test Cases Based On Software Changes And Software Faults**" by **UĞUR YILMAZ** has been approved as a thesis for the Degree of **Master of Science** in **Computer Engineering** by the Examining Committee Members mentioned below.

Assoc. Prof. Dr. Aysu BETİN CAN
Head ..... 

Assist. Prof. Dr. Ayça TARHAN
Supervisor ..... 

Assoc. Prof. Dr. Lale ÖZKAHYA
Member ..... 

Assoc. Prof. Dr. Oumout CHOUSEINOGLOU
Member ..... 

Assist. Prof. Dr. Adnan ÖZSOY
Member ..... 

This thesis has been approved as a thesis for the Degree of **Master of Science** in **Computer Engineering** by Board of Directors of the Institute of Graduate School of Science and Engineering on ..... / ..... / .....

Prof. Dr. Menemşe GÜMÜŞDERELİOĞLU
Director of the Institute of
Graduate School of Science and Engineering

ETHICS

In this thesis study, prepared in accordance with the spelling rules of Institute of Graduate School of Science and Engineering of Hacettepe University,

I declare that

- all the information on documents have been obtained in the base of the academic rules
- all audio-visual and written information and results have been presented according to the rules of scientific ethics
- in case of using others works, related studies have been cited in accordance with the scientific standards
- all cited studies have been fully referenced
- I did not do any distortion in the data set
- and any part of this thesis has not been presented as another thesis study at this or any other university.

18/02/2019

UĞUR YILMAZ

YAYIMLAMA VE FİKRİ MÜLKİYET HAKLARI BEYANI

Enstitü tarafından onaylanan lisansüstü tezimin/raporumun tamamını veya herhangi bir kısmını, basılı (kağıt) ve elektronik formatta arşivleme ve aşağıda verilen koşullarla kullanıma açma iznini Hacettepe üniversitesine verdiğimi bildiririm. Bu izinle Üniversiteye verilen kullanım hakları dışındaki tüm fikri mülkiyet haklarım bende kalacak, tezimin tamamının ya da bir bölümünün gelecekteki çalışmalarda (makale, kitap, lisans ve patent vb.) kullanım hakları bana ait olacaktır.

Tezin kendi orijinal çalışmam olduğunu, başkalarının haklarını ihlal etmediğimi ve tezimin tek yetkili sahibi olduğumu beyan ve taahhüt ederim. Tezimde yer alan telif hakkı bulunan ve sahiplerinden yazılı izin alınarak kullanması zorunlu metinlerin yazılı izin alarak kullandığımı ve istenildiğinde suretlerini Üniversiteye teslim etmeyi taahhüt ederim.

Yükseköğretim Kurulu tarafından yayınlanan "***Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge***" kapsamında tezim aşağıda belirtilen koşullar haricince YÖK Ulusal Tez Merkezi / H. Ü. Kütüphaneleri Açık Erişim Sisteminde erişime açılır.

- ☐ Enstitü / Fakülte yönetim kurulu kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren 2 yıl ertelenmiştir.
- ☐ Enstitü / Fakülte yönetim kurulu gerekçeli kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren .... ay ertelenmiştir.
- ☐ Tezim ile ilgili gizlilik kararı verilmiştir.

18 / 02 / 2019

UĞUR YILMAZ

ABSTRACT

A METHOD FOR SELECTING REGRESSION TEST CASES BASED ON SOFTWARE CHANGES AND SOFTWARE FAULTS

UĞUR YILMAZ

Master of Science, Computer Engineering Department
Supervisor: Assist. Prof. Dr. Ayça TARHAN

January 2019, 79 pages

Regression testing is the type of testing in which a modified software is validated to ensure its functionality is not broken. With the increase of modern, agile and large size software systems, regression test selection needs to be efficient, effective and practical to coexist within the software development cycle. To this need, a modern hybrid technique for regression test selection is proposed in this thesis. A detailed literature analysis and a conceptual model are presented in order to better visualize and identify the target concepts of the field. We introduce a technique operating on different granularity levels using difference based techniques of files for both class files and third-party text files. Our technique uses lexical comparison methods for readable files and checksum comparison for any binary files with file or method level granularity. A tunable similarity threshold is offered to users to be used in fulfilling different performance needs. Any available test or fault history data is also used to increase the effectiveness of the proposed technique. We provide an extensive evaluation study in the form of embedded, multiple case study of the proposed technique with other state-of-the-art techniques with respect to performance and cost-efficiency using different open source projects. The results showed that the proposed approach is effective as

other state-of-the-art techniques and selects fewer tests while keeping the fault detection rate at a high level.

Keywords: regression test selection, regression testing, dynamic analysis, text difference based regression testing, conceptual model

ÖZET

REGRESYON TESTLERİNİN SEÇİMİ İÇİN YAZILIM DEĞİŞİKLİKLERİNE VE YAZILIM HATALARINA DAYALI BİR YÖNTEM

UĞUR YILMAZ

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Danışmanı: Dr. Öğr. Üyesi Ayça TARHAN
Ocak 2019, 79 sayfa

Regresyon testi, değiştirilmiş bir yazılımda tüm parçaların işlevlerinin doğru çalıştığını güvence etmek için gerçekleştirilen bir test türüdür. Artan modern, çevik ve büyük kapsamlı yazılım sistemleri ile birlikte yazılım geliştirme döngüsünün bir parçası olabilmek için regresyon test seçiminin de etkili, hızlı, verimli ve pratik olması gerekmektedir. Bu amaç doğrultusunda bu tez kapsamında modern hibrit bir regresyon test seçim yöntemi önerilmiştir. Regresyon test alanının etkin kavramlarını tanımlamak ve görsel olarak daha iyi anlamak için ayrıntılı bir literatür taraması ile birlikte bir konsept modeli sunulmuştur. Hem sınıf hem de üçüncü-parti metin dosyaları için fark tabanlı teknikler kullanarak farklı detay katmanlarında çalışabilen bir teknik anlatılmıştır. Önerilen teknik okunabilir dosyalar için sözcük tabanlı karşılaştırma metotları kullanırken, herhangi bir ikili dosyalar için de sağlama toplamı yöntemlerini dosya veya metot detay seviyelerinde kullanmaktadır. Kullanıcıların farklı performans isterlerini karşılamak amacıyla ayarlanabilir bir benzerlik eşiği sunulmuştur. Ayrıca, önerilen tekniğin verimini artırmak için varsa test ve hata verileri de kullanılmıştır. Önerilen teknik ve diğer modern, gelişmiş teknikler, açık kaynak

kodlu projeler kullanılarak gömülü, çoklu bir durum çalışması şeklinde geniş çaplı bir değerlendirmeye tabii tutulmuştur. Sonuçlar önerilen tekniğin, diğer teknikler kadar etkili olduğunu ve hata tespit oranını yüksek seviyede tutarken daha az test seçtiğini göstermiştir.

Anahtar Kelimeler: regresyon test seçimi, regresyon testi, dinamik analiz, metin farkı tabanlı regresyon testi, konsept model

ACKNOWLEDGEMENT

I would first like to express my sincere gratitude to my advisor, Assist. Prof. Dr. Ayça TARHAN, for her continuous support throughout my study and for giving me the honor of working with her. The door to her office was always open whenever I ran into a trouble spot or had a question. She consistently steered me in the right direction with her wisdom and experience. Without her valuable insights, it was impossible to perform this work.

In addition, I would like to thank my cats, Şalgam and Ecevit, for being there with me on countless sleepless nights and their never-ending spiritual support.

Last but not least, I must express my very profound gratitude to my patient wife, Emine ÇALIŞKAN YILMAZ, for providing me with unfailing support and continuous encouragement throughout my study and always being there for me in my time of need even though it meant sacrificing her own priorities.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Modern software projects are developed in a fast pace to ensure the demands of the customers are met [1]. To illustrate developers working at Google make 16,000 changes on average daily and automated systems make another 24,000 changes to the different software systems [2]. Developers need to perform the given tasks in a timely concise manner and they need to do so in such a way that software quality should not be compromised. To keep up with the rapid development cycle without breaking any existing functionality and ensuring newly added functions operate as expected, software projects are tested rigorously [3]. The testing process takes as much as two-thirds of the overall software development life cycle [4]. One type of the tests carried out is regression testing which is performed to give confidence that no defects are present in modified software and no existing functionality is affected by the changes [5]. With many changes introduced to the system, to be able to perform regression testing in a short time is an essential need for many projects. It is estimated that 80% of software testing activities consist of regression testing [6]. In order to manage the size, cost, coverage, and fault rate of the regression test cases, one of the three following method is selected for regression testing: (1) Test Suite Minimization (Reduction), (2) Regression Test Selection, and (3) Test Case Prioritization. Regression test selection techniques select a subset of tests that ensures the affected parts from changes are working as expected. Using only a part of the all tests can reduce different costs of the testing while giving confidence [3]. However, many regression test selection techniques are tailored for only specific circumstances, making them unusable in a generic manner [6], [7]. In addition, many techniques are only applicable and tested in academic area making them difficult to use in industry and open-source projects where time and/or workload is scarce [6]. This calls for cost and resource aware, easy to implement/maintain solution that incorporates an efficient and effective technique. In this thesis, we introduce a novel and cost-efficient regression testing technique which satisfy several industry and open-source projects' needs. We achieve this by thoroughly analyzing state-of-the-art regression testing techniques, identifying their application areas and their drawbacks and finally constructing a technique to

include a variety of parameters to provide an effective way of regression testing technique.

Although many Regression Test Selection (RTS) techniques are presented in the literature, the adoption rate in the industry and open-source community remains low [3], [6], [8]–[11]. In order to overcome this problem, we suggest a cost-efficient, fast and easily maintainable RTS technique based on work carried out in several other prominent techniques [3], [9], [12]–[18]. In this thesis, we aim to better understand the difficulties in RTS space, classify the techniques used in the field to propose a technique to overcome the obstacles put forward. We suggest a RTS technique based on code changes, test history and external dependencies which we derived from a conceptual framework developed using grounded theory methodology [19], [20]. Furthermore, the experimental results of the proposed technique with other prominent techniques which approach the problem from different viewpoints in the field are presented as a case study so that a base for evaluation of our proposal could be enabled.

More specifically, the contributions of the thesis can be summarized as follows: (i) We propose an RTS technique based on code changes, test history and external dependencies with a effective and efficient approach in terms of time and fault detection. (ii) A conceptual framework for RTS techniques is presented to better illustrate the state of the RTS space. (iii) An extensive evaluation of the proposed technique with other state-of-the-art RTS techniques are given as a case study and the results are discussed throughout the thesis.

The thesis is organized as follows: In Chapter 2, we provide the background information for terminology of regression testing. The basic concepts of regression testing strategies and different classifications of the RTS techniques are also discussed in this chapter. In Chapter 3, we introduce the conceptual model and discuss the various aspects of the model. In Chapter 4, the proposed technique is presented. The motivation for the thesis is also given in this chapter. In Chapter 5, a case study is presented to evaluate the effectiveness of the proposed technique with state-of-the-art RTS techniques. The result of the study

is discussed with the advantages and disadvantages of the proposed technique. Threats to validity of the thesis and concluding remarks are made in Chapter 6.

2. BACKGROUND

2.1. Basic Concepts

Regression testing is defined by IEEE [21]:

“Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or components still complies with its specified requirement.”

Regression testing covers a variety of changes not limited to code changes such as requirement changes, configuration changes and database/third party changes [22], [23].

From the viewpoint of what changed, regression testing can be classified as *(i) progressive regression testing,* in which modified changes results in creation of new test cases, such as requirement change, code structural changes, adding new features, behavioral changes etc., and *(ii) corrective regression testing,* in which modified changes such as bug fixing, refactoring, etc. does not affect previous test cases so that they can be selected and reused for the modified program [6]. The differences of corrective and progressive regression testing are summarized in the Table 2.1.

Table 2.1. Corrective Regression vs Progressive Regression Testing

Corrective Regression Testing	**Progressive Regression Testing**
Software requirement is not changed	Software requirement is changed
Minor modifications (Statement level)	Major modifications (Module level)
During Development	After Release, Between Releases
Many tests are reusable	Many tests are obsolete
Executed irregularly	Executed regularly

In regression testing, test cases can be classified as reusable, retestable, and obsolete [24]. Reusable test cases inspect the unchanged parts of the program. They may not be used in the current regression testing but they can be used in later revisions of the program. Retestable test cases test changed parts of the

program. These tests are to be selected when testing the modified version of the program. Obsolete test cases are redundant test cases to the modified version of the program. These test should not be selected for regression testing [25]. The summarized information about test case types is given Table 2.2 [24].

Table 2.2. A summary of Test Case Types [24]

Test Case	**Specification**	**Target Construct**	**Test Type**
Reusable	unchanged	unchanged	Structural, specification
Retestable	unchanged	changed	Structural, specification
Obsolete	unchanged	changed	Structural
	changed	Unchanged/changed	Specification

For evaluation purposes of regression testing, Rothermel and Harrold [5] developed a framework in which many fundamental definitions used by other studies is structured [1], [3], [6], [8]–[11], [25]–[28]. There are four main properties to consider when evaluating a regression testing technique [5]:

Inclusiveness. Inclusiveness measures the capability of selecting modification-revealing tests. Defining n as total number of modification revealing tests, and if m tests are selected, then inclusiveness is calculated as ((m/n)*100).

Safety. Safety of a technique is the capability of the technique's selection of all modification related tests. A safe technique guarantees all of the faults due to modifications will be found because the modification-revealing tests are proven to be a superset of fault-revealing tests [5], [29]. Therefore, if inclusiveness is %100, then the technique is called safe.

Precision. Precision defines the ratio of selected modification revealing tests to total modification revealing tests. Higher precision results in less time since only the modification revealing tests are selected [5]. It is defined as:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Efficiency. Efficiency is considered in terms of time and space requirements of the technique. If a technique requires less time or reveals more faults in the same time than other techniques, the technique is accepted as more efficient [5].

Generality. Generality defines applicability of the technique to a different selection of programs [5].

Based on these four properties several other metrics are also proposed and used widely. Mostly used metrics are as follows:

Recall. Recall is the percentage of the selected failed tests from all failed tests [6]. It is defined as:

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

F-Measure. F-Measure is a widely used metric in statistics. It combines precision and recall and measures the fault detection capability and cost reduction [6]. It is defined as:

$$F - Measure = \frac{2 * Recall * Precision}{Recall + Precision}$$

FDR. Fault Detection Rate measures the ratio of detected faults [30]. In order to calculate this, all tests need to be run and if the technique is safe this becomes %100.

APFD. Average Percentage of Faults Detected is a value between 0-100 and calculates average percentage of faults which only occurred a later version of the project [6], [30]. There are various versions of APFD, such as APFDc which is cost-cognizant, and NAPFD, which normalizes the APFD between versions [3], [6], [30].

End-to-end Test Time: Mostly used in comparison studies, this metric measures the total time including any preparation and execution of test case [14], [31], [32].

Total Selected Test Percentage: This metric measures the selected number of tests from all test cases. It does not include modification-awareness or fail-status of the test case and therefore easy to calculate and widely used in comparison studies [12], [33], [34].

Regression testing techniques are classified as *fine* and *coarse* granularity at which level selected technique operates. The fine level granularity is generally more precise but takes more time due to more extensive analysis of the program. In this thesis, we use fine level granularity to refer anything below class level analysis like code block, method, statement, line level inspection of the modified program. While coarse level granularity is generally less precise but have smaller overhead and hence results in less end-to-end test time. In this thesis, we use coarse level granularity to refer anything above class level analysis like class, file, and configuration files level inspection of the modified program. There is a quid pro quo involving accuracy and time to consider when choosing the granularity level of the regression testing strategy [35].

2.2. Regression Testing Strategies

The simplest way of regression testing is, *retest all* approach which runs all of the test cases. However, as software size increases this approach becomes expensive in terms of time and workload [25]. For example, Google's test automation system runs 150 million tests daily and the result of tests are delivered to developers in 45 minutes' averages. Furthermore, the system observed delays up to 9 hours which could result in catastrophic failures in the shipped products [36].

A better approach to regression testing is the *random testing*. Test cases are selected at random and executed. This approach is suited if any other technique is not applicable and the time is running short since with random testing many important bugs could make their way to production.

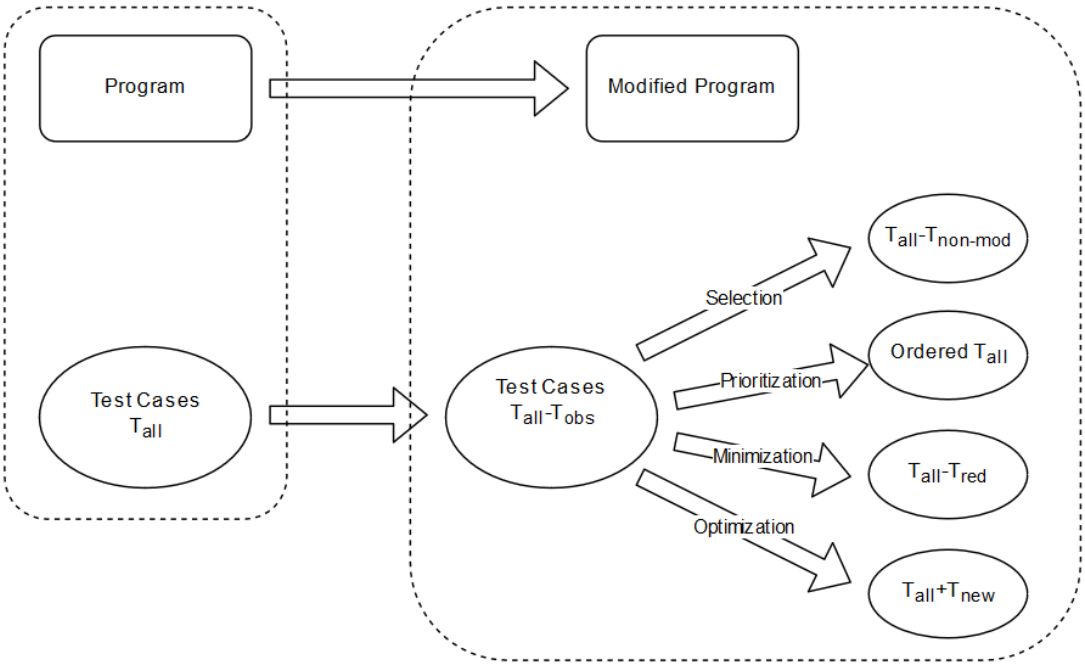


Figure 2.1. Regression Testing Overview

In order to increase the effectiveness of regression testing, four main problems are presented as summarized in Figure 2.1:

Test Case Minimization. Given a test suite, T, a set of test requirements $\{r_1, r_2, \ldots, r_n\}$, that must be satisfied and subsets of $T, T_1, T_2, \ldots, T_n$, one of each is linked with requirements such that any of the test cases t_j which is a part of T_i can be used to achieve requirement r_i. Then the problem becomes finding a set, T', of test cases from T that satisfies all r_is [7], [25]. Test Case Minimization (TCM) aims to decrease the site of regression test suite by eliminating redundant test cases while achieving maximum requirement coverage with minimum set [3]. It is also known as Test Case Reduction (TCR), Regression Test Reduction (RTR) and Regression Test Minimization (RTM) [37].

Regression Test Selection. Given the program, P, the modified version of the program, P', and a test suite, T. Then the problem becomes finding a subset of T, T', which tests modified program P' [7]. Regression Test Selection (RTS) focuses on the changed parts of the program and tries to find the optimal

combination of test cases to ensure modified parts and affected parts are functioning as expected. It is also known as Test Case Selection (TCS).

Test Case Prioritization. Given a test suite, T, the set of ordered tests, PT, and a function from PT to real numbers, $f: PT \rightarrow R$. Then the problem becomes finding $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' != T')[f(T') \geq f(T'')]$ [7]. Test Case Prioritization (TCP) aims to order the test cases with respect to a fitness function so that the ordered test cases yield the best results according the predefined criteria such as time, cost, fault detection rate. By this way, the important bugs could be discovered early for lengthy projects [3]. It is also known as Regression Test Prioritization (RTP).

There are other approaches that combine one or more techniques presented above. For example, Gupta [38] uses RTS and TCP to test highest priority requirements.

Test Case Optimization. Test Case Optimization combines one or more techniques above with each other or with a different testing field. For example, a hybrid approach is proposed by first applying RTS to the all test cases and applying TCP to selected test cases in order to maximize the APFD, hence, reducing the time to gather important defect information [39]. Another example would be combining test case selection techniques to determine the change impacts and then using test case generation for producing test cases [40].

Since TCM is not modification-aware and TCP is inherently fails to find all the bugs, both The TSM and TCP are out of context of this thesis.

2.3. Regression Test Selection

Regression test selection is NP-hard since it can be reduced to well-known Set Covering Problem [41]. Following the Rothermel's formula of the regression testing [5], there are many different solutions proposed in the literature [7]. In this section, we will explain basic approaches used in various RTS techniques so as to build a better foundation of understanding for the rest of the thesis. Note that

there are many approaches not mentioned in here that are chosen to specific programs like database regression testing, business specific programs, and etc. The aim of this section is to lay the groundwork for the readers to better understand and comprehend the rest of the thesis so problem-specific approaches are omitted. A summary is given in the Table 2.3 with key points, main advantages, disadvantages and safety for each approach.

Table 2.3. A summary of Regression Test Selection Approaches

Name	Key Points	Advantages	Disadvantages	Safety
Slicing	Based on dependency graphs and slicing criteria	Different range of analysis. i.e. intramodule or intermodule level modifications	Imprecise, computationally expensive	Unsafe
Data-Flow	Based on data coverage criteria	Different range of analysis. i.e. intramodule or intermodule level modifications	Lack of analysis of non-data-flow changes	Unsafe
Firewall	Based on interactions/dependencies of modules	Different range of analysis depending on firewall definition	Lack of analysis where changes propagate from outside of firewall, imprecise	Unsafe
Difference	Based on code/text differences	Easy to implement and fast depending on program size	Imprecise, computationally expensive depending on program size	Safe
Cluster	Based on grouping similar test case or code modifications	Variety of grouping methods, fast	Imprecise	Safe
Model	Based on a representation of program. i.e. UML, BPEL, class diagram, use cases, etc.	No need for source code, integration testing. Used as a part of another framework, faster than code level approaches	Imprecise, need consistent model updates/generation	Unsafe
Graph	Based on flow/control/dependency graphs of program	Precise, adaptive granularity level, fast for procedural and small programs	Computationally expensive, memory inefficient for large programs	Safe

Learning	Based in machine learning algorithms such as genetic algorithms	Multi objective	Imprecise, slow depending on approach	Safe/unsafe depending on approach
Fault	Based on fault data, test case history, logs	Combination fault data with other approaches	Imprecise depending of lack of fault data, time consuming depending of the granularity	Safe
Hybrid	Based on combination of different approaches	Combination of strengths of different approaches, Different range of analysis	time-consuming than other methods	Safe/unsafe depending on approach

2.3.1. Slicing Approach

Program Slicing is an analysis technique mainly used in debugging. First slicing criteria is defined as a pair <p, V> where p is the program and V is the program variables. Then for every possible input the program is analyzed and the statements affected with respect to the slicing criteria is collected, thus enabling the test cases that produces different outputs for modified program [3]. There are many levels of slicing, ranging from object slicing in object oriented programs to statement level slicing in procedural languages [42]. Slicing is also used to derive flow/dependency graphs when the source code is not available [43], [44].

Slicing techniques are precise as they only select test cases with different outputs but since they omit the changes caused by statement deletions, they are unsafe [3], [7], [43].

2.3.2. Data-Flow Approach

Data-flow analysis uses definition-use pairs that are either modified, added, or deleted in changed program and selects test cases accordingly [3], [7]. The processing of the whole program is costly and doing this process over and over again introduces extra overhead. To overcome this problem, incremental data-flow analysis is used in the literature [43]. In incremental analysis, after a change is processed, appropriate test case is selected and it is updated in the dataflow information. Then, the steps are repeated for all changes [3].This approach is

also combined with slicing techniques and other regression testing techniques to further decrease the cost of analysis [43], [45].

Major downside of the data-flow techniques is the lack of analysis about non-data-flow changes. If the program is changed in such a way that no variable (definition-use pair) is used, calling an argumentless method or changing configuration, the approach fails to detect the related test cases. Consequently, these approaches are unsafe.

However, it is reported that data-flow based techniques suits spreadsheet based programs particularly well since there is no change occurring without dataflow information [7].

2.3.3. Firewall Approach

In firewall approach, the aim is to draw a firewall around the units of the system as to determine which parts of the program are affected and need retesting. The modules interacting in the firewall are selected in regression testing. Leung and White [27] proposed mainly three categories for the firewall approach: (i) No Change: module has not been modified, $NoCh()$. (ii). Only Code Change: module has the same specifications but code has been modified, $CodeCh()$. (iii) Spec Change: module has changed specifications, $SpecCh()$. Using these definitions, a basic firewall can be described as follows: let A and B be the two modules of a software that have interaction with each other. There are nine possible configurations between A and B. If none of the modules are modified, $NoCh(A) \cap NoCh(B)$, then no tests are selected. If both A and B is modified either by code or by specification change, $SpecCh(A) \cup SpecCh(B)$, $CodeCh(A) \cup CodeCh(B)$, $CodeCh(A) \cup SpecCh(B)$, $SpecCh(A) \cup SpecCh(B)$, then each modules' tests and any test which tests the interaction between A and B is selected. Then, only four configurations remain in which a changed module calls the unchanged module and the so called firewall line is drawn at this boundary [7]. The granularity of the technique differs as where the firewall is drawn. Figure 2.2 shows another example of firewall approach. In the figure, solid lines show related classes and boxes show test cases related to the classes. The firewall calculated for class D

is shown by dashed line. Only TC1 and TC2 are affected by changes made to the D class since they are related to the classes within the firewall [46].

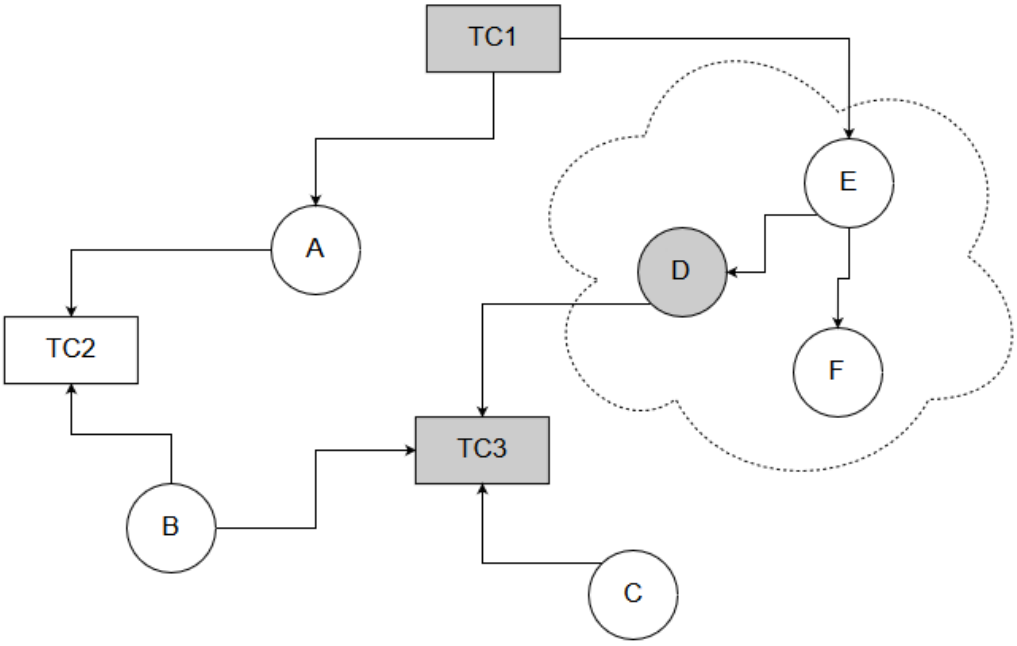


Figure 2.2. An Example of a Firewall Approach

The firewall approach does not select test cases where changes are propagated from outside of firewall. Since any non-modification-traversing tests could be omitted, the firewall approach is not safe.

2.3.4. Difference Based Approaches

2.3.4.1. Code Based Modification Approaches

Modification based techniques rely on analyzing source code to a selected granularity to identify the modified code entity. After selecting the granularity level, a test coverage matrix is constructed by monitoring the execution of test cases. After determining the modified code entity, associated test cases are selected using the test coverage matrix [3], [7]. Many different approaches are proposed using modification analysis directly or indirectly since the approach is safe [8], [11], [35]. Because it selects all the test cases relating to any modified code entity [3], [7].

2.3.4.2. Text Based Modification Approaches

Text based approaches are regarding the source code as plain text and comparing the original and modified program code after some preprocessing such as deleting comments, whitespaces etc. to reduce the risk of false positives [47]. These approaches are also safe like code based modification approaches since they operate similarly.

Although these approaches are fast in terms of time but they are imprecise since differentiation only base upon basic syntax and disregards any language/program specific structures [3].

2.3.5. Cluster Based Approaches

Cluster based approaches tries to group either similar program modifications or similar test cases. After initial analysis of the program clusters are defined and compared to the modified program's clusters. Then, changed clusters are selected for testing [3], [7]. Determining what entities will be in clusters defines the structure of the method, clusters can be code blocks[48], test cases [49][50], and etc. Cluster based approaches uses traditional methods like control flow graphs [3] or more recent methods such as k-means machine learning algorithm [50].

Cluster based approaches are deemed safe since they select all modified code blocks/test cases [7]. However, based on the size of the clusters they can also be imprecise and can select redundant test cases.

2.3.6. Model Based Approaches

Model abstraction of a system is a widely used technique in the industry and highly used in software development [11], [51]. Model based approaches relies on program representation in a standard notation like UML diagrams and BPEL models [37], [31]. They use class-sequence diagrams, use cases, user stories, state machine models, flowcharts to analyze and determine which test cases to select [3], [6]–[8], [35], [37], [51]. Traditionally they use models of the original

program and the modified program and a mapping of test cases to models to identify modified parts and select appropriate test cases, and in recent years several genetic algorithms are also used in conjunction with model based approaches [6], [48]. An example use case diagram is given in the Figure 2.3 [52].

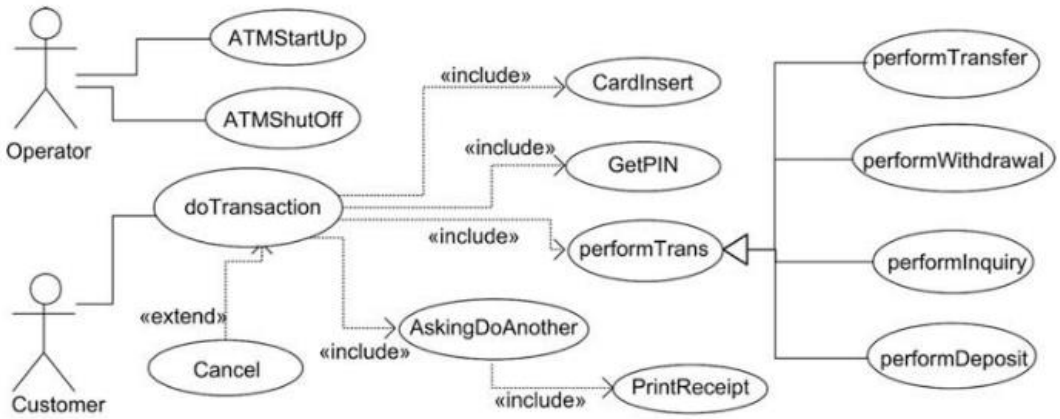


Figure 2.3. An Example of Use Case Diagram [51]

Model based approaches are mainly preferred when source code is not accessible or regression testing is done at the integration test level [53]. They are also used as inputs of the proposed regression test selection technique to generalize to more applicable areas [52], [54]. For example, Kandil et al. uses user stories for their clustering based approaches as shown in the Figure 2.4 [55]. They are also used as a part of a bigger framework including regression test selection. This approach is highly selected within a bigger framework to support regression test selection. An example is shown in the Figure 2.5 representing how SeTGaM [54] tool uses two models to generate tests and classify regression test cases with respect to modified version using impact analysis.

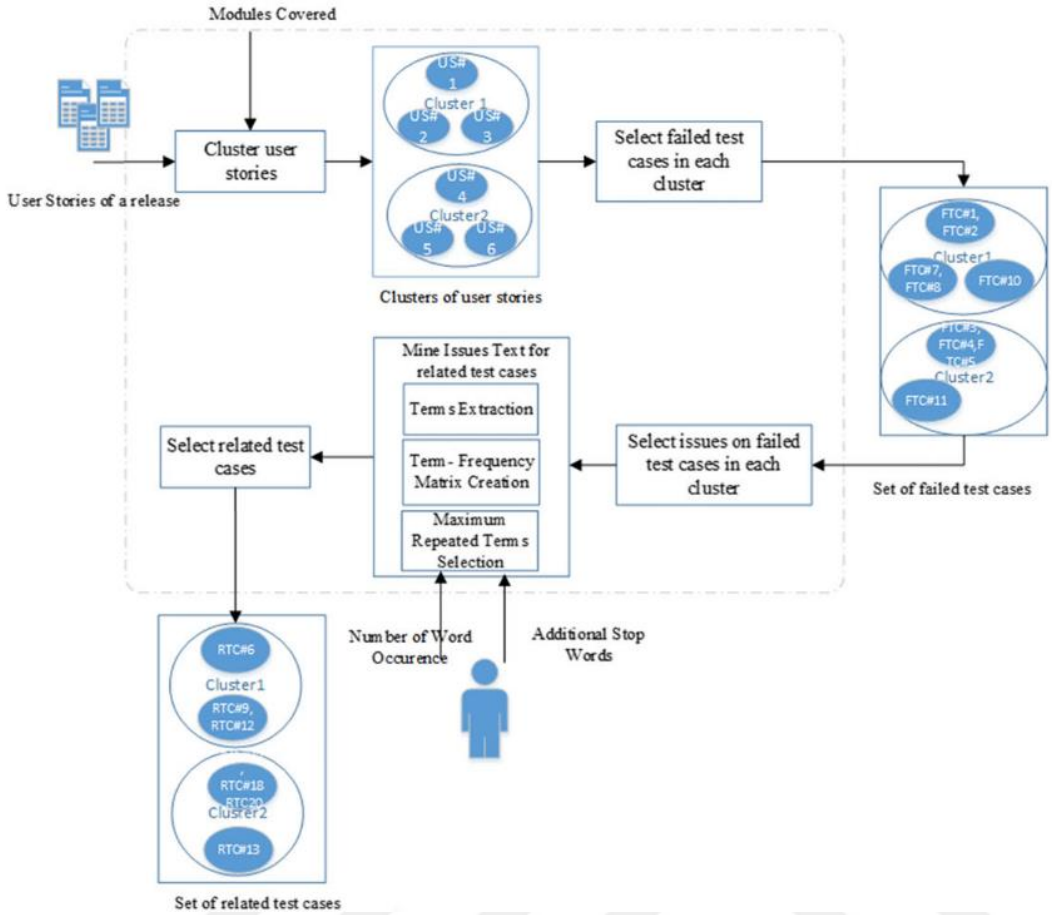


Figure 2.4. Clustering Approach Overview used in the Kandil et al. [54] study

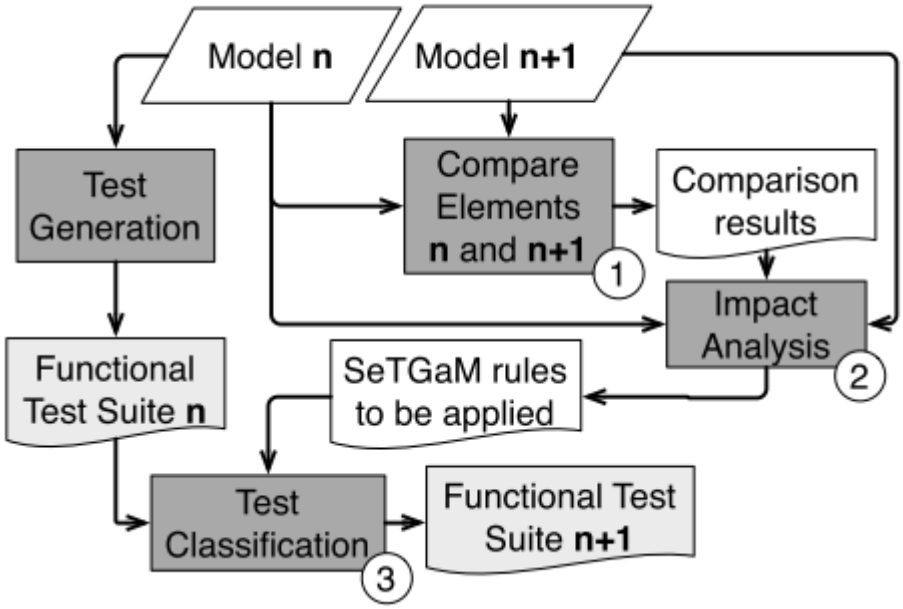


Figure 2.5. Overview of Model Based Approach used in the SeTGAM tool [53]

Model based approaches are preferable since they operate at a higher abstraction level and generally easy to generalize and faster than code level approaches [3]. They also include non-code artifacts like configurations, traceability information in the model and are able to select test cases affecting non code modules of the system [6]. However, as a drawback of operating at a higher level, model based approaches are also imprecise and can be safe or unsafe depending on the technique details. Moreover, the models are always needed to be up to date for technique to effectively function and when model to be used is generated from the code, it can introduce overhead processing to the overall time [3], [6], [11].

2.3.7. Graph Walk Approaches

Graph walk approaches are the first widely adopted approaches used in the literature [3], [29]. Graph walk approaches are also used with other approaches such as slicing, model based and data flow approaches [3], [56], [57]. This approach is also adopted to various languages such as C++, Java and AspectJ [58], [59], [60]. There are several aspects of the software like dependency relation, data flow, control flow, module dependency etc. could be represented. As a result, there are multiple types of graphs generated with this approach [6].
Control flow graph (CFG) is a directed graph where granularity level corresponds to nodes and the relationship between entities is represented as edges. The graph is generated as the program executes normally. While test cases are executing, nodes are also associated with the test cases. Then when the modified program is analyzed, changes in the nodes are detected and corresponding test cases are executed [29]. An example of a CFG is shown in the Figure 2.6.

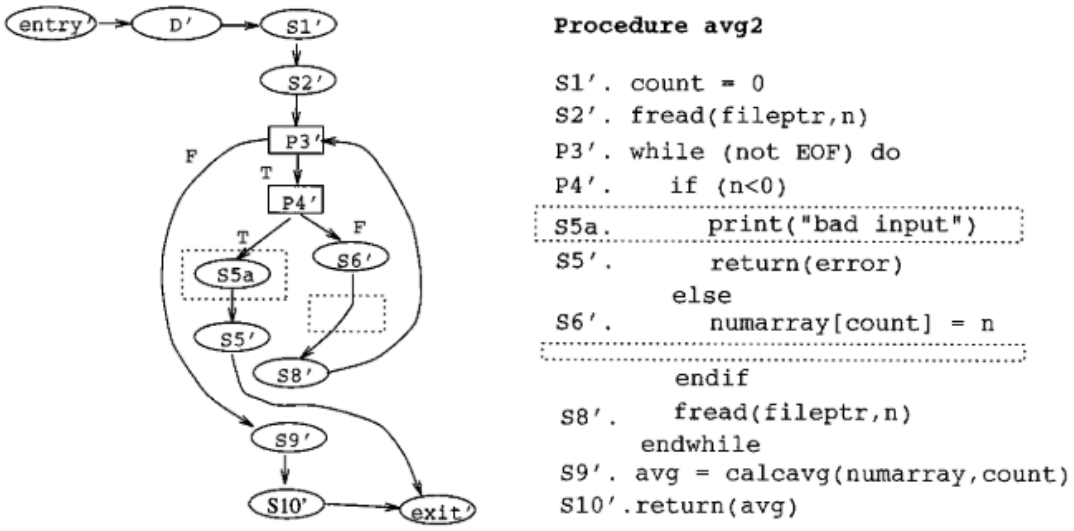


Figure 2.6. Control flow graph example [55]

Control dependency graphs (CDG) and data dependency graphs (DDG) are constructed the same way as above differing only what to trace when identifying nodes and edges. Higher level graphs are also proposed such as program dependency graphs, system dependence graphs, file dependency graphs each only differing what to analyze. There are also user story graphs generated approaches using user interaction graphs, i.e. tracing the user action on a website [61].

In different languages, different iterations of graphs are computed and used accordingly. In Java language, Java Interclass Graph (JIG) is highly used to include java properties to traditional CFGs [58]. An example JIG operation on method calls is shown in the Figure 2.7 [3].

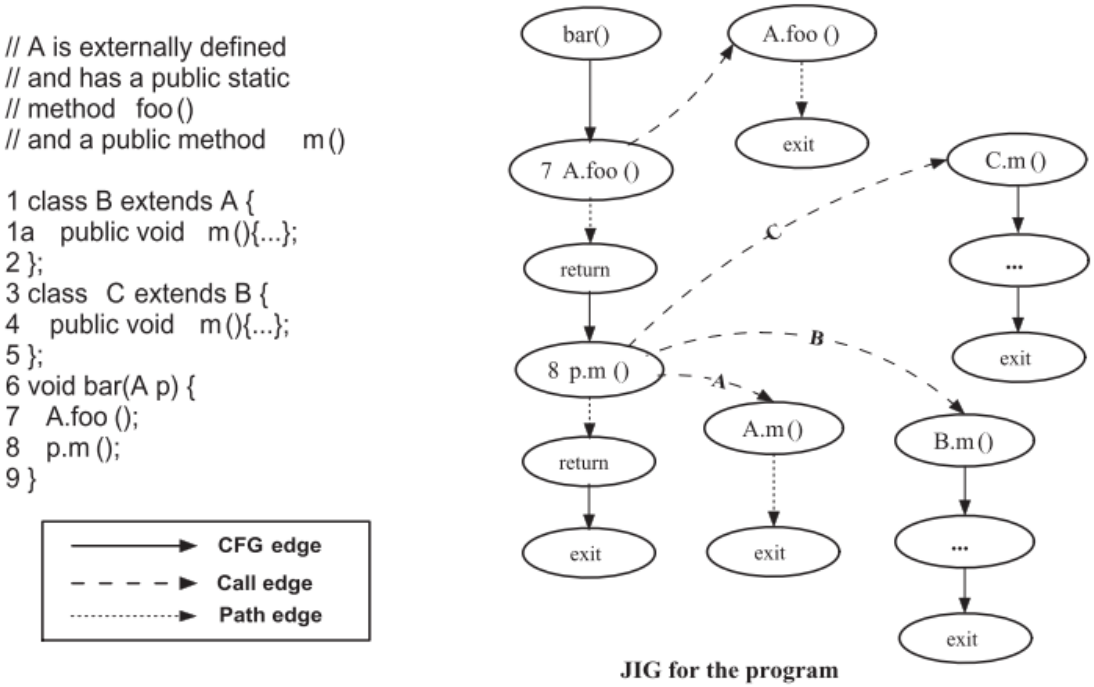


Figure 2.7. Java Interclass Graph Example [3]

Graph based approaches are safe since they guarantee to include modification revealing test cases. Depending on the granularity of the approach, they can be more precise than other approaches. However, since the comparison of the whole graphs of two versions of the program is computed it can be a costly operation with increasing size of the program [62].

2.3.8. Learning Based Approaches

With the advancements made in the recent years in the fields of machine learning and artificial intelligence, there are some approaches applying methodologies from respected fields [6]. Genetic algorithms and fuzzy logic based approaches are successfully applied as to select test cases [63], [64].

Genetic algorithms operate following the nature's example of evolving. It consists of mainly five steps: (i) initial population, (ii) fitness function, (iii) selection, (iv) crossover, and (v) mutation. In initial population, pseudo chromosomes are computed using test cases in a stream representation. Then, a fitness function is defined to evolve the process to the desired goal. An example fitness function

would be similarity function of test cases and modified code blocks [37]. In selection phase, using a fitness function highest valued (fittest) chromosomes are selected for next generation. In next phase, pairs of chromosomes are crossed with a crossover point and a probability. Resultants are then mutated with a mutation probability to diversify the population. In RTS, a chromosome is combinations of test cases, representing a candidate set as explained in the Figure 2.8.

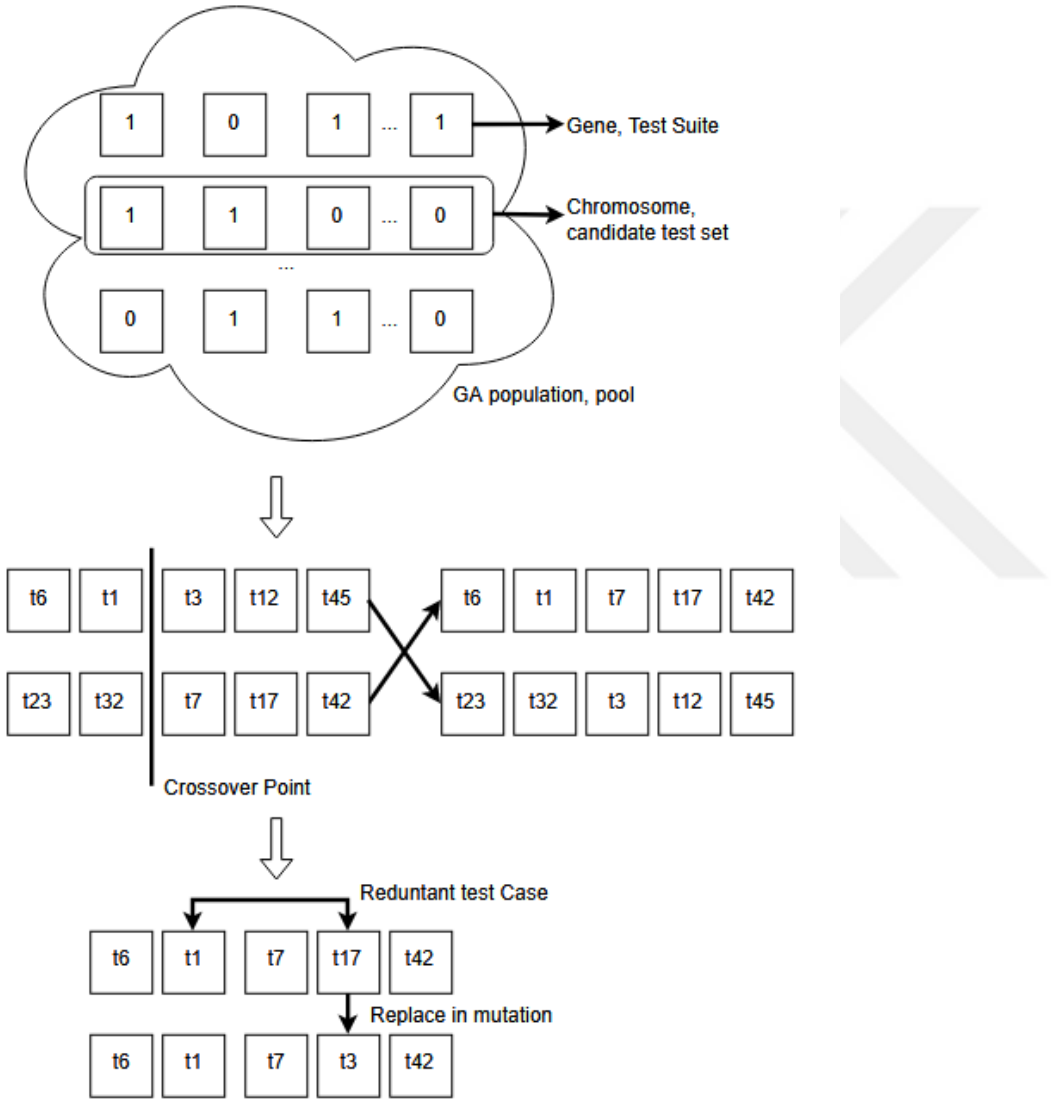


Figure 2.8. Genetic Algorithm based Regression Testing Example

Learning based approaches are also used to develop multi objective regression test selection techniques [48], [65].

2.3.9. Fault Based Approaches

Another approach is using fault data available in the software history to assist in regression test selection process. Build metadata, test case history, modification logs, bug reports are used with other regression test selection approaches in order to improve fault detection capabilities of the proposed technique. They are used as an input to graph walking, difference computing, learning and hybrid approaches and sometimes used for fault prevention mechanisms to determine the possibly faulty locations and generate test cases for future software [66], [11].

2.3.10. Hybrid Approaches

Hybrid approaches combine two or more RTS approaches and selects the appropriate approach after an initial analysis of the modified program [11], [17].

3. CONCEPTUAL MODEL

In order to better visualize and explore the studied areas in regression testing space and to identify the target concepts to include in our proposal, a conceptual model is constructed.

3.1. Research Method

The motivation for creating a conceptual model is to create visual representation of the current state of the regression test selection field. There are several secondary studies such as SLRs, Surveys, etc. carried out in recent years about RTS techniques [1], [3], [6], [8], [9], [11], [25], [28]. Unfortunately, these studies focus on papers and their individual contributions to the field, thus failing to provide an overall picture of the techniques, approaches used in the field in spite of classifying and emphasizing on current trends and future directions. To this end, the grounded theory research method is selected because it is suitable for creating a general understanding of a field [20]. Grounded theory is described as "a general methodology of analysis linked with data collection that uses a systematically applied set of methods to generate an inductive theory about a substantive area" [67]. Grounded theory consists of three main stages: (i) coding, (ii) memoing and theorizing, and (iii) integrating, refining and writing up the theory [67]. In grounded theory according to Glaser's method [68] which we have selected to follow, it is advised to start the investigation without defining a research problem. The aim of developing a conceptual model is detecting patterns and categories to emerge in a flexible way which grounded theory is suitable for [20].

In the first stage of the grounded theory method, coding, first an open coding is performed to identify core category and concepts for the research to focus on and secondly a selective coding is performed to refine the core category. Then in memoing stage, identified concepts and notes for each concept are compared with each other to shape the core category. At the last stage, all of the findings are integrated and linked together with a comparative method and a theory is formed [67]. There are three terms used in grounded theory: code, concept and

category [68]. Code is the fundamental observation taken from the statements, concept is the group of codes and category is the group of concepts [20]. An example of the three level of data abstraction is shown in the Figure 3.1.

Although grounded theory is mostly used with data in the form of questionnaire, interview or observations and codes are deduced from line by line statements. We have used written articles as our data and we have assigned codes to the regression testing approaches, inputs, outputs and artifacts. Then we determined the concepts from codes as a higher grouping element and finally we decided on the categories. Grounded theory dictates that after the coding stage, a core category must be selected and the research should continue accordingly [68]. However, since we are only interested in the core categories emerging from the process for our conceptual model, we did not proceed with the rest of grounded theory methodology. Our process is illustrated in the Figure 3.1. After finalizing all of the categories, related categories are mapped according to emerging patterns in the conceptual model.

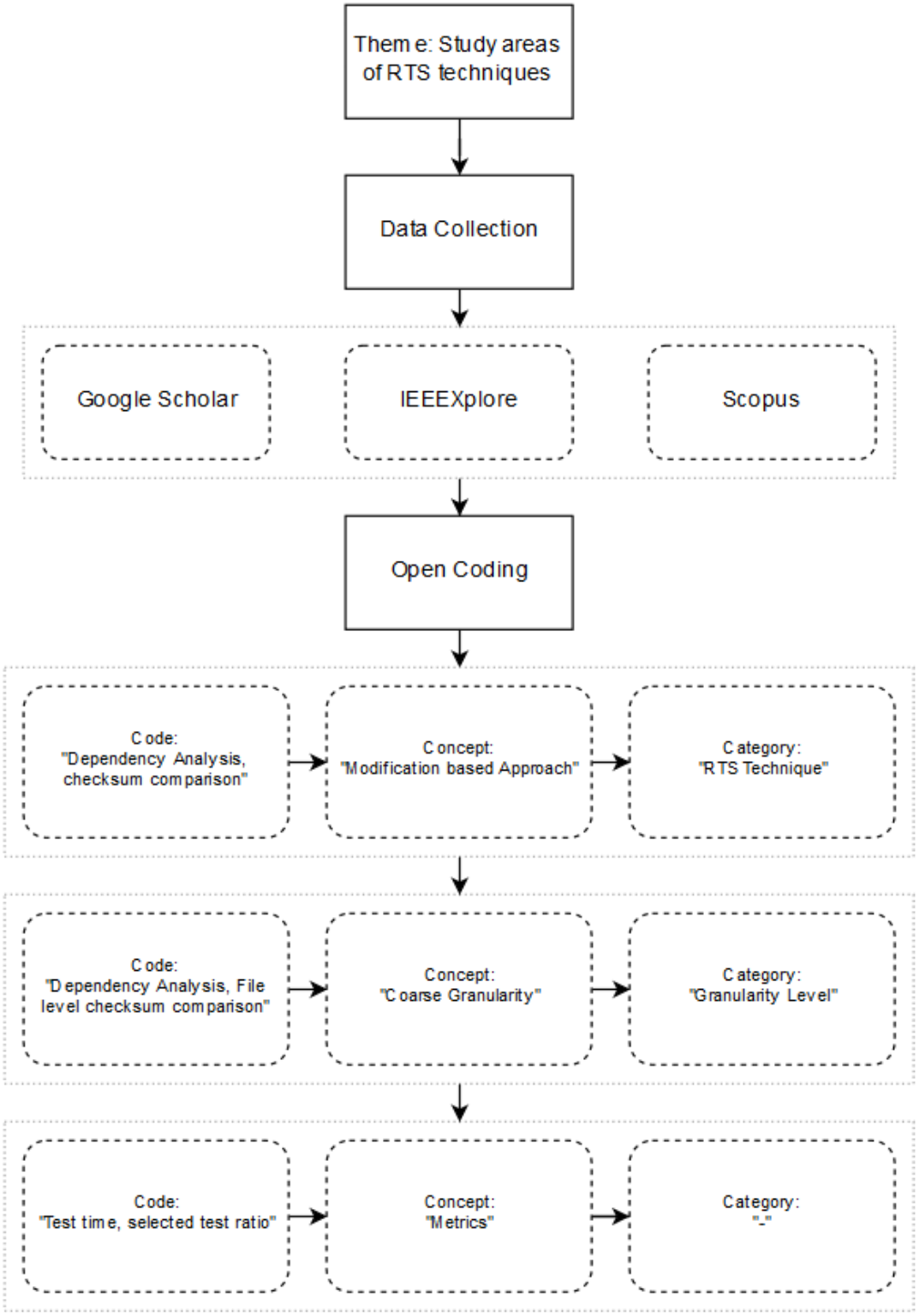


Figure 3.1. Grounded theory research process and three level of data abstraction.

3.2. Conceptual Model

First, data is collected from Google Scholar, IEEEXplore and Scopus databases with the following search string:

"regression" AND ("test" OR "testing") AND ("technique" OR "techniques" OR "method" OR "validation" OR "incremental" OR "select" OR "selection" OR "selective" OR "software")

Second, articles returned from search string are analyzed and any article (i) not related to regression test selection field, or (ii) not contributing a method/ framework/technique or an empirical evaluation/comparison study is excluded. Then articles mentioned in second degree studies such as SLRs, surveys, SMs are extracted via snowballing technique and processed by the exclusion criteria defined above.

Third, coding process is performed following the grounded theory principles. Each article is evaluated at least once since we had no preconceived codes at the start and could not risk missing any relevant data point. Coding is performed following comparative methods to eliminate any terminology differences between articles, i.e. relevant papers are grouped together and coded again to identify and extract exact terminologies which might have been used differently by authors [20]. Resulting conceptual model and intermediate models are given in the Appendix section. Furthermore, an initial abstract version of the conceptual model to visualize the relationships between different blocks is presented Figure 3.2.

RTS Techniques mainly differ depending on whether the technique is code based or not. Further classification of non-code techniques are also available such as requirement based, data based, bug based in the literature but the techniques used in sub-classifications does not differ from each other so we concluded with three main classifications [37], [51]. Furthermore, dynamic and static approaches could be differentiated but we decided not to do so because the underlying approaches are the same and only difference would be the performance, especially regarding time and cost.

The main approaches used in RTS techniques are explained in section 2.3 of the thesis. To select an RTS technique, the application domain, software artifact and granularity level need to be considered. For example, if the software artifact size is small such as web or GUI applications, fine granularity level approaches like graph walk, firewall or data-flow could be selected for an efficient technique. On the other hand, if the technique needs to be adopted to a wide range of systems, technique having high generality, then choosing fine granularity level approaches could result in spending more time on analysis of the test suite than actually executing the test cases. In such case, the technique is highly likely to be dropped from usage.

Most studies uses evaluation metrics that are suited for their respected techniques [6], [7]. It is observed that many safe RTS techniques do not compute metrics such as precision, recall, or inclusiveness [8], [25], [37]. Fault related techniques -whether it uses fault data, the technique is part of a fault detection technique, or technique is a hybrid approach designed to increase early detection of faults- focuses on fault related metrics such as APFD, FDR [1], [6], [11], [26]. Furthermore, if the software artifact has a large size then time related metrics are preferred since time is an important aspect in large software systems [69].

The relationship between granularity level and software artifact can be thought of as having an inverse relation. If the software size gets bigger, time and cost of regression testing become more important so that the technique may sacrifice granularity level to operate faster. If the software size is smaller, the technique may perform deep analysis to refine selected test cases. Likewise, if the language of the software is procedural then the analysis of the source code requires less computing time, thus more precise approaches can be selected [14], [18], [22], [37], [70].

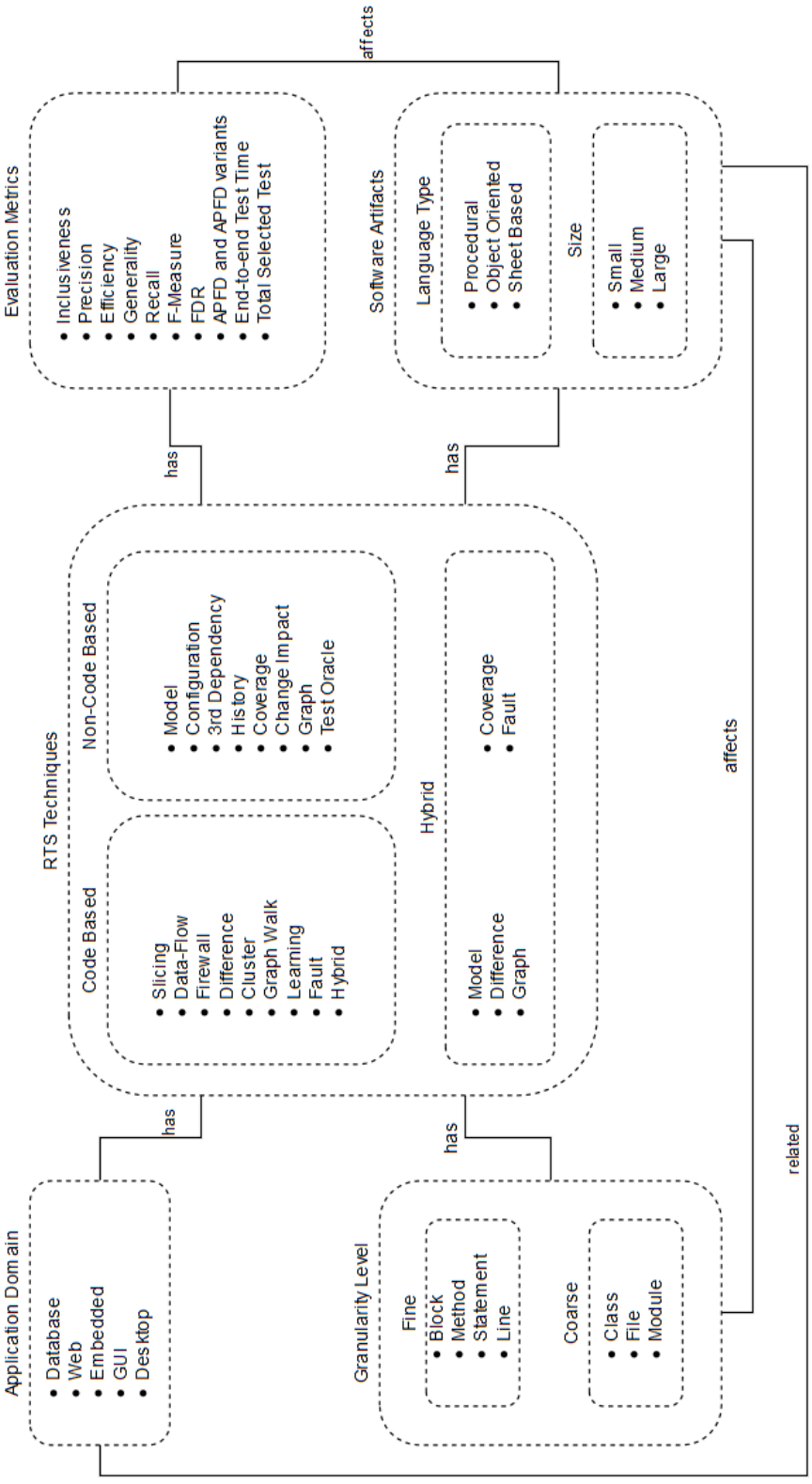


Figure 3.2. The Conceptual Model

Graph based approaches are the earliest techniques presented in the regression test selection field. Rothermel et al. [29] first presented graph walk algorithms

using dependency graphs for procedural languages. Then the approach is extended to use control flow graphs rather than dependency graphs to perform more efficiently [59]. Graph walked approaches are also extended to include object oriented languages [58], [59]. Orso [71] developed DejaVoo tool using safe graph based comparison approach. Then, some model based techniques are also represented as graphs to include other parameters such as traceability information [72]. While El-Hamid [73] used method level granularity to construct CFGs, Willmor [22] used database queries and Huang et. al. [74] used config files as nodes in CFG. CFGs are adopted to include various artifacts as input to better select test cases such as false test cases [75], third party libraries [76], hash code representations [61]. This article supports that graph based approaches are used widely for various projects in literature as also shown in conceptual model. Graph based approach is the only approach observed to have adopted to all granularity levels for a wide range of software projects.

Slicing approach is mostly used in combination with graph based approaches [42]–[44]. Slicing is mostly used with procedural languages since there is no object-oriented structure so it is more efficient to analyze the software. Firewall approach is also another approach that is adopted to different languages and different application domains [1], [27], [46]. It is observed that both slicing and firewall approach are used successfully with class or method level granularities with different software languages and applications.

With the increase of distributed systems and data focused applications, the regression test selection techniques have also shown a change in direction considering used approaches and inputs/outputs of the techniques [9], [11]. Cluster based approaches in cooperation with learning based algorithms such as genetic algorithms [77]–[79] and artificial intelligence approaches [15], [18], [50], [55], [80], [65] shown effective performances with different levels of granularities for different sized projects. But they are mostly used in desktop, GUI and embedded applications. In database applications, model based and data flow based approaches are used extensively with a similarity function or coverage criteria to determine the test cases [22], [81]–[83].

As software complexity is increasing, the analysis of the source code and RTS techniques becomes a time consuming task without decreasing in importance to the quality of the system [84], [36]. Since any fault that made its way to a production environment could result in devastating consequences, many RTS techniques incorporating other sources than code have been developed. Ruth [61] proposed a privacy aware, CFG based web framework for RTS. Change impact analysis and fault detection based RTS techniques with the inclusion of historical data are proposed in recent years to eliminate the risk of high weighted faults not being detected while increasing the APFD score of the systems [85]–[92], [93]. These systems are mostly used with static methods on fine granularity level for modern applications with different sizes. Another solution to this problem is model based regression test selection techniques. Model based regression test selection techniques are used extensively where source code is not available or the analysis of the source code is not feasible [6]. They are used with dependency graphs, configuration differences and requirement analysis in a variety of application domains with a fine granularity [40], [52], [64], [80], [94]–[101], [23].

The availability of the source code or the changes of non-code parts of the program also impacts the selected technique since the approach used would be different. For example, if fault history and traceability of the software is available, a learning based approach could be applied with high fault detection rate. However, if a model based approach is selected with no model available then computing the model from source code would introduce unwanted overhead to the end-to-end test time while resulting in lower precision. To overcome such problems, hybrid approaches are presented in recent years. Hybrid approaches tries to achieve a balance between granularity level and performance of the technique [17], [38], [39], [45], [74].

Granularity level measures how precise the selected technique will be and if the granularity level is finer, then more overhead analysis would be computed resulting in more time spent on overall [24]. But if the software size is small, then the overhead time could be neglected, and hence both precise and efficient technique could be selected. So it is an important aspect to identify the software artifact for which the technique will be adapted [6], [26], [102], [103].

Application domain and software artifact language type greatly reduce the available RTS approaches. For example, many embedded applications are developed with procedural languages, in which object-oriented approaches like model based or object dependency graph based are not available, and need to be tested rigorously if they are safety-critical systems. There are only a few approaches available for embedded applications like slicing or firewall to increase the precision while keeping the performance at acceptable levels. Another example would be many database applications are programmed by sheet based languages which greatly reduces the available approaches.

Evaluation metrics for the approaches depends on the inner characteristics of the RTS technique. To illustrate, if the technique is safe then there is no need to calculate precision and inclusiveness. Type of the software is another important factor in deciding the evaluation metrics to be computed. For example, an object-oriented web application working with a high number of users could prioritize f-measure and weighted fault detection rate since any high weighted fault impact could result in revenue loss of the system.

3.3. State-of-the-Art Summary

The conceptual model shows that traditional approaches are dominant in the RTS field and widely adapted to different problem spaces. On the other hand, recent studies show that new techniques adopt more coarse granularity level dependency [69], difference [15], coverage [104], fault [85] and learning based approaches [13], [93]. It is deduced the reason for this shift is pointed to agile development and comparison studies of traditional approaches with emerging techniques. Agile development has been widely accepted and become the number one development practice across software practitioners [105]. In agile development, software is shipped frequently and most of the build and test process of software is automated and controlled by a CI/CD platform [105]. To be able to keep up with the fast pace of the software life cycle, any RTS technique also should be fast enough. This is only achieved by coarse granularity level approaches such as difference based and learning based approaches. Moreover, the studies carried out in recent years showed that coarse granularity level (class

level) approaches are outperforming fine granularity level (method level) approaches in real life projects [62], [1]. In this section, techniques used for comparison with other techniques, techniques that have high citation impacts, techniques developed in recent years and used in real life projects are discussed as state-of-the-art techniques. A summary of the techniques with key implementation points, advantages and disadvantages of each technique is given in the Table 3.1

Table 3.1. A summary of the state-of-the-art techniques

Name	Key Points	Advantages	Disadvantages	Safety
Ekstazi	Dynamic, file level checksum based dependency analysis	Fast, light-weight	imprecise w.r.t. other state-of-the-art techniques	Safe
SPIRITuS	Dynamic, methold level difference based analysis	Adjustable, fast, language-independent	computationally expensive w.r.t. other state-of-the-art techniques	Unsafe
HyRTS	Dynamic, hybrid level checksum based dependency analysis	Fast, light-weight	computationally expensive w.r.t. other state-of-the-art techniques, highly language dependent	Safe
REKS and RIT	Dynamic, refactor aware graph based data flow analysis	Fast, light-weight, efficient	computationally expensive w.r.t. other state-of-the-art techniques, highly language dependent, refactor	Safe
STARTS	Static, file level checksum based dependency analysis	Fast, light-weight	Imprecise, high test selection ratio w.r.t. other state-of-the-art techniques	Unsafe

Ekstazi [14] is a dynamic file level dependency analysis approach that gained popularity in recent years because it is light-weight and efficient. Ekstazi calculates checksums for each file depending on a test case and stores them for further use. It does not differentiate between class files or configuration files. On later versions of the software, if the checksum of a file is not changed then matched test case is not selected for regression testing. It is easy to include in the build process since it supports tools like Maven, junit out of the box and starts to work automatically without human intervention.

SPIRITuS [15], SimPle Information Retrieval regression Test Selection technique, is difference and learning based approach with coverage information using lexical modifications to detect changed methods in order to select test cases. It is easy to adopt to other programming languages, tunable and fast. After constructing a test case coverage matrix on method level, it uses vector space model by textually analyzing the methods. Then it uses the coverage matrix and vector space model to perform a method similarity computation and selects test cases corresponding to changed methods. It is a fine granularity level, safe approach.

HyRTS [17], is a hybrid approach which operates at multiple granularities to better select test cases. HyRTS performs file level dependency analysis like Ekstazi and an additional method level analysis. Then it overlaps the results of two process to determine which test cases to execute. It is easy to include in projects since it supports build/test tools like Maven, junit. Because it includes method level analysis, it outperforms Ekstazi in precision but test time depends on the software tested. The authors also propose variants changing granularity level for performance gain.

REKS [13] and RIT [91] are both recently published techniques that are refactor aware techniques. They try to skip test cases that are selected due to behavior-preserving changes made to the code. RIT uses abstract syntax trees to identify the refactoring changes and then uses data flow and change impact analysis approaches to select test cases. RIT computes refactorings by its own approach whereas REKS relies refactoring engines of IDEs to inform itself about changes and updates the dependencies to skip the related test. RIT focuses on identifying refactoring based faults by informing the developer about the changes. On the other hand, REKS is built on top of Ekstazi and modifies Ekstazi dependency files to skip the tests.

STARTS [12], is static RTS technique that uses type dependence graphs to calculate checksums of the files each test case depends on like Ekstazi. Since this technique is statically applied, it is more imprecise and unsafe compared to

dynamic techniques. STARTS perform similar to Ekstazi and easily pluggable to build environment.

Hafez [16] proposed caching potential fault revealing files and using this information when dynamically selecting test cases. Oliveria et. al. [90] uses machine learning algorithms to predict whether a commit affects the performance regression tests. Kandil et. al. [55] proposes cluster based approach for regression test selection after test case prioritization phase. Aman et. al. [93] uses natural language processing for topic extraction from test cases and then recommends an order to regression test selection process to increase performance.

4. PROPOSED TECHNIQUE

4.1. The Goal of the Thesis

The traditional models are used extensively in early years of RTS field and the techniques put forth by Rothermel [29], Leung [27] and Orso [71] are adopted to different problem spaces with high precision since the approaches are safe [7], [17], [22], [29], [31], [42], [43], [57]–[60], [73]–[76], [87], [96], [106]–[111]. But the overhead computing time of these techniques, emerging new software engineering practices like machine learning and artificial intelligence approaches and data oriented software caused a shift in the approaches used in regression test selection process [6], [9], [11], [37].

During the grounded theory process, it is observed that most of the techniques presented by the academia is not available to download [102], [112]. Because of this, there are only a few of easily maintainable/user friendly RTS techniques used widely in open-source communities, such as Ekstazi [14], HyRTS [17] and Starts [12]. Although these techniques are efficient, have low precision but high end-to-end test time since they use coarse granularity level approaches and not safe, they are accepted in large open source projects since they are easily to download/use and have little overhead in preparation. To improve the adoption rate of RTS techniques on open-source and private projects without sacrificing precision, a fine level granularity, fast, efficient with lower overhead analysis technique is needed.

The conceptual model and state of the art techniques revealed that although there are many traditional and effective techniques such as based on graph walk, slicing and model approaches, these techniques fail to work adequately with modern large, multi-language and multi-domain software systems [6], [11], [37]. To address the issue, more modern approaches are proposed in recent years like clustering, machine learning and hybrid techniques. In addition, while the inclusiveness and precision of a technique is important for modern systems cost metrics such as time, fault detection and computation power are more preferable from an industry point of view [113]. Furthermore, we observed even though,

learning and hybrid based techniques are used in recent years, the area of fault related approaches using state of the art different based approaches is not explored fully [1]. As to comply with the standards of the modern software, a technique which uses all of the information (test history, fault logs, configuration files) is needed.

To fulfill the goals explained above, a regression test selection technique using a hybrid approach using code modifications with test/fault history and external dependency analysis is proposed in this thesis. We chose to develop a hybrid approach with approaches already addressing different problems encountered in the field as a result of conceptual model because we aim use the advantages of each technique in their proven area and with the combined strength of the hybrid approach, it is also possible to carry out a comprehensive case study comparing state-of-the-art techniques.

4.2. Proposed Technique

The proposed approach is inspired from the techniques mentioned in Ekstazi [14], SPIRITuS [15] and HyRTS [17]. The process of the proposed technique is given in the Figure 4.1. The technique is a hybrid approach which uses text similarity of files for any third-party text files, checksum comparison for any third-party binary files and file/method level checksum with lexical comparison for source files. It also identifies any non-deterministic and high priority test cases using test/fault history. Although there are earlier studies using text based difference approaches, there is no file level text similarity approach presented [47]. The SPIRITuS uses text similarity based on method level analysis [15].

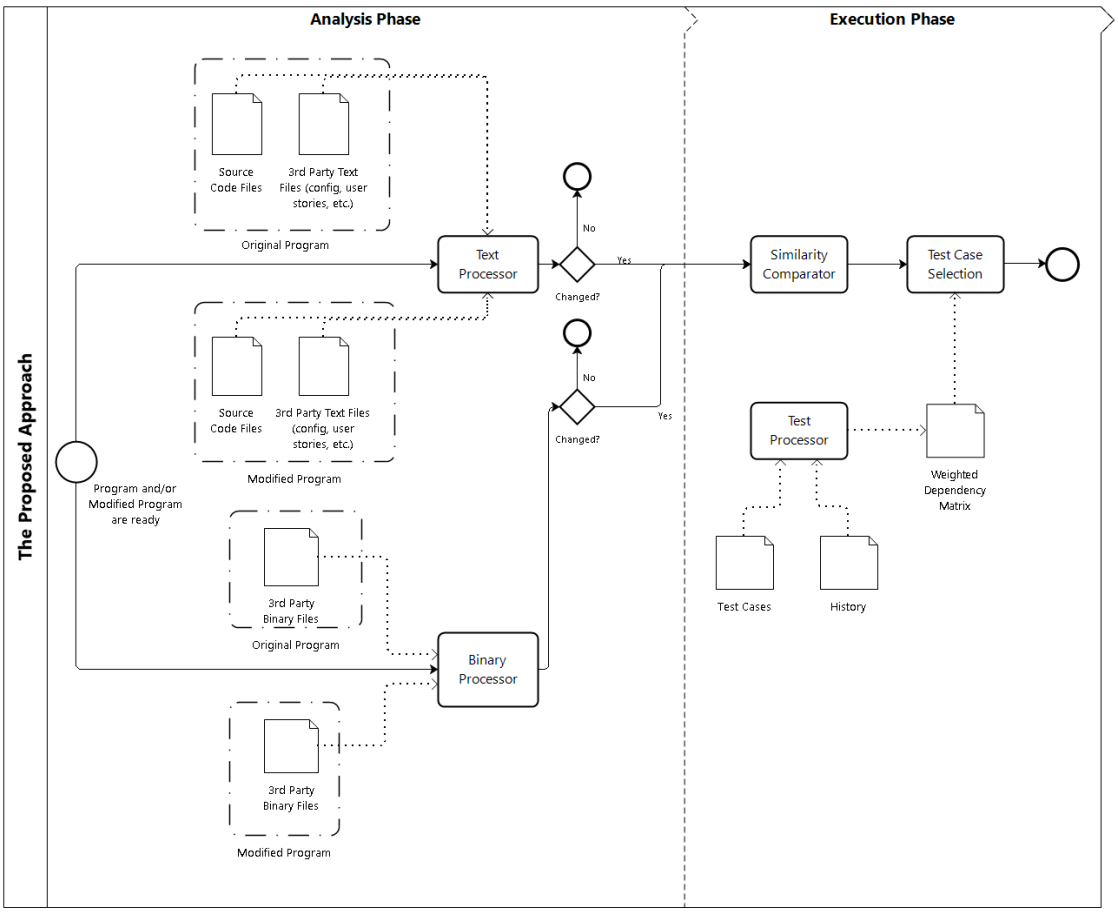


Figure 4.1. Flow model of the proposed technique

Analysis phase could either be online or offline since it does not affect test case selection. In analysis phase, preliminary analysis of the source code and 3rd party files are processed and indexed to be used in execution phase. In execution phase, weighted dependency/coverage matrix is calculated, changed parts of the program are detected via similarity comparator and appropriate test cases are selected.

Text Processor. Text processor uses natural language processing methods based on vector space model (VSM).

Firstly, each class file, third-party file is extracted and converted to documents. In the JVM languages used in this study, this is done by observing the classpath and **.class* files. Third-party files are left untouched since they are regarded not as source codes and treated accordingly.

Secondly, data preprocessing and tokenization is applied to each document such as cleaning (deleting whitespace, redundant characters). For data preprocessing, all the code comments, whitespaces, redundant characters are removed. Different rulesets are applied to source code fields to gain better performance. However, only basic cleaning could be applied in favor of being language neutral. For example, *map.get(key);* is tokenized into: {“map”, ”.”, ”get”, “(“, “key”, “)”, “;”} for java language. Any other language specific tokenization could be performed for better results in different languages. Since tokenization module could be replaced with other methods, any third-party library could also be used to automate the process and reducing the development time.

Finally, using vector space model (VSM) approach the document is indexed via Apache Lucene[1]. In VSM, the documents are represented as vectors.

$$d_j = (w_{1,j}, w_{2,j}, \dots, w_{i,j})$$

Each dimension consists of a term which can be words, keywords, or phases. Several approaches are proposed how to calculate term weights. We are using tf-idf weights which uses following formula. Note that that in Apache Lucene the calculations are automated.

$$w_{t,d} = tf_{t,d} * log \frac{|D|}{|\{d' \in D \mid t \in d'\}|}$$

Where tf is the term frequency if term t in document d as in the following formula:

$$TF(t,d) = \frac{t}{d}$$

The logarithmic expression is the inverse document frequency in which D represents total number of documents and dividend part represents the number of documents containing term t [114] as presented in the below formula:

$$IDF(t, D) = log \frac{|D|}{|d \in D : t \in D|}$$

[1] https://lucene.apache.org

We have decided to use VSM for text processing since it is a proven, effective method used in regression testing [15].

Binary Processor. Any file that is not text based is fed to binary processor. Binary processor calculates fast checksums of each file and stores the result in a file. Since binary files could be large files and we treat any file outside of the project files such as third-party libraries, it is efficient to only use checksums. Moreover, if any binary file is changed/updated it could easily be identified with checksum values.

Test Processor. Test cases and test/fault history is used to derive a weighted dependency matrix. In this thesis we define test case in a test class level since many test methods could depend on other tests in the same class or they could be parameterized with more recent test frameworks for ease of use [14], [1], [17]. The weights are calculated as optional since the history may not be always available. The tool aims to use its execution results to incrementally build history. The test cases are first weighted by the user with the flakiness and fault tolerance of the tests in range of 0-1. Then file dependencies are extracted by executing test cases and recording the used files in specific test cases and associated weights. If no test/fault history is available then all the weights are set to 1. Then coverage matrix is obtained with test cases vs files and weight values. In test selection, if two test cases are affected by the same file, the more weighted test case is selected to ensure cost-efficiency.

If there are flaky tests [104] discovered in the test history, relevant test cases are given lower weights since they are not to be trusted. A flaky test is a test that can fail or pass given same configurations. Flaky tests could occur from unknown data dependencies, concurrencies, non-deterministic behaviors etc. Similarly, if a test is shown as a fault producing test then the weight of the test is increased since it should be prioritized. The weight association is used when there are multiple test cases fulfilling the selection criteria to select the more promising test cases. In order to ensure an objective weight is assigned to the specific test case, we have developed and weight assessment matrix as shown in Figure 4.2.

WEIGHT RATING KEY	LOW (0.33) UNACCEPTABLE This should not be selected	MEDIUM (0.66) REASONABLE This should be selected if no HIGH priority test case does not exist.	HIGH (1) ACCEPTABLE This should always be selected first

	Faultiness		
	Negative The test case is not faulty	Neutral The test case is observed to be faulty in the past but not recently or vice versa	Positive The test case is always faulty
Flakiness			
Negative Test case is not flaky	LOW	MEDIUM	MEDIUM
Neutral Test case is observed to be flaky in the past but not in recently or vice versa	MEDIUM	MEDIUM	HIGH
Positive Test case is always flaky	MEDIUM	HIGH	HIGH

Figure 4.2. Weighted matrix key, flakiness vs faultiness

Similarity Comparison. If binary files are to be used, then the checksum of the original and modified program is compared and if there are any changes then changed file and thus related test cases are selected. On the other hand, similarity between two text based files, documents, is calculated using Jaccard

or Cosine similarity index. The Jaccard similarity is defined as number of shared terms over the number of all unique terms in both documents [115].

$$J_w(d_1, d_2) = \frac{\sum_i min(d_1, d_2)}{\sum_i max(d_1, d_2)}$$

The cosine similarity is defined as cosine of the degree of two vectors [115]:

$$cos(d_1, d_2) = \frac{d_1 \cdot d_2}{|d_1| * |d_2|} = \frac{\sum_i (d_1 * d_2)}{\sqrt{\sum_i d_1^2} * \sqrt{\sum_i d_2^2}}$$

First the original and modified program is processed according to the text processor and vector form of each file is obtained. Then similarity comparison is performed according the above formulas and similarity index is calculated. After calculating the similarity, it is compared to a threshold in test case selection.

Test Case Selection. After setting a similarity threshold between 0 and 1, any test case whose similarity value is smaller than the threshold is selected for execution according to the formula where tc is test case:

$$TCS(tc) = \begin{cases} Don't\ pick, & simScore > threshold \\ Pick, & simScore \leq threshold \end{cases}$$

If the threshold is 1, then the approach becomes a safe RTS technique since it will automatically select all the test cases. The threshold could be well-tuned for different projects to obtain better performance.

In this thesis, we set the threshold to 0.975 to gain comparable performance with other state-of-the-art techniques on average [15]. We have selected the threshold value after experimenting with different threshold values ranging between 0.95-1.00 for each case study program and calculating the best threshold value which has highest fault detection rate and lowest test selection ratio as illustrated in the Figure 4.3.

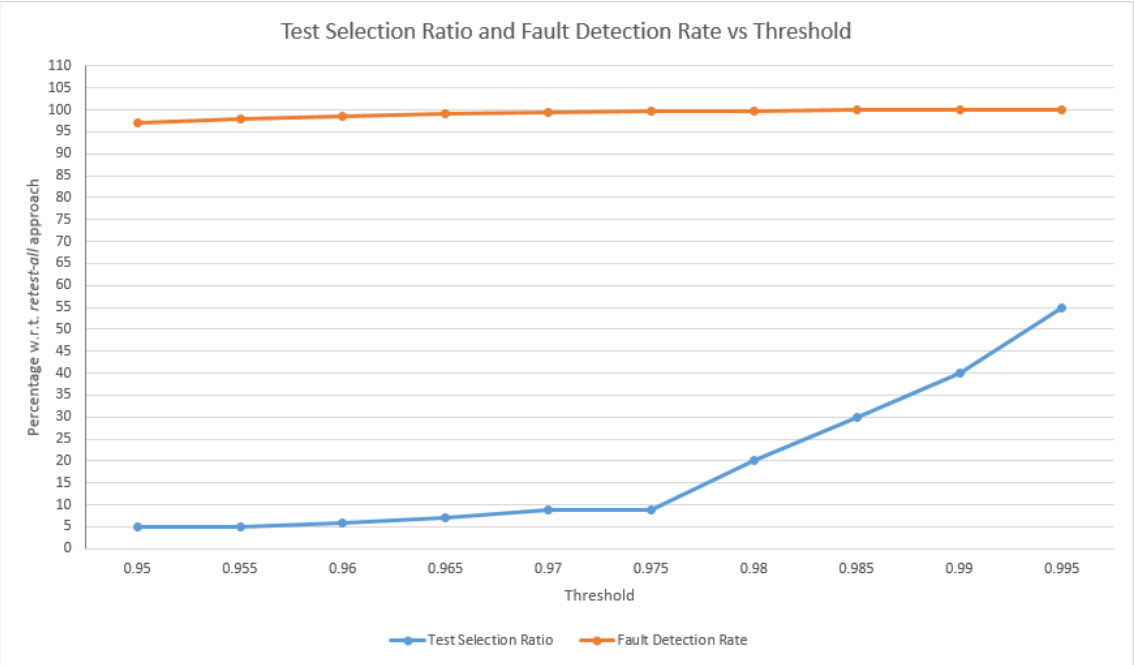


Figure 4.3. Average test selection and fault detection rate calculated for different threshold values computed for each program.

The analysis phase is the most time-consuming of the approach but since the analysis is required for the previous versions of the software, it can be computed as offline.

The technique is developed as a junit extension for the Java language so that it is easy to adopt to existing projects. It can also detect projects controlled in version control, namely git, and process previous versions for building historical data.

5. CASE STUDY

The proposed technique is evaluated with the other state-of-the-art approaches detailed in section 3.3. Case study is selected as an evaluation method. Case studies are performed to research a contemporary phenomenon [116]. We followed the Yin's [117] case study definitions with the help of Runeson's [118] guidelines. Since multiple units are analyzed in our evaluation, embedded case study is carried out. Since we will focus on different cases, the case study is selected as embedded, multiple case study. Case study design approaches are summarized in the Figure 5.1.

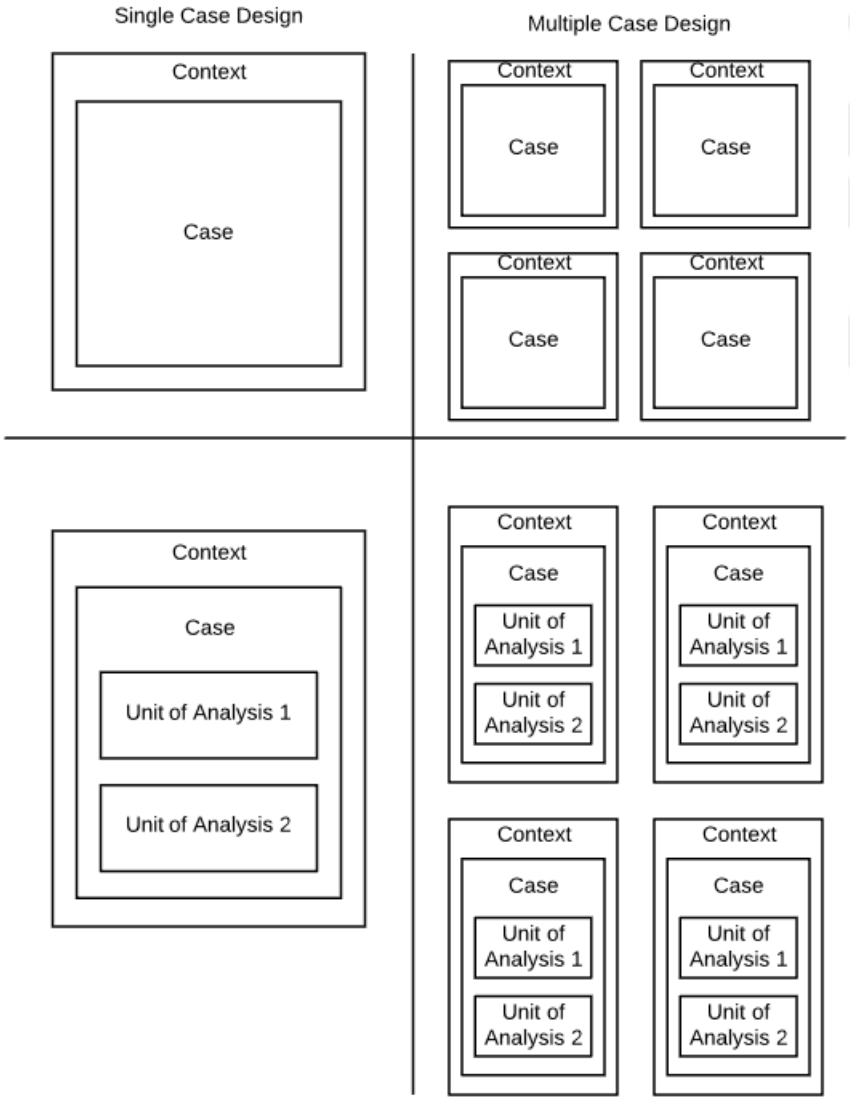


Figure 5.1. Overview of case study design approaches

5.1. Objective

The purpose of the case study is to compare performance, usability and effectiveness of the techniques. To this end, following research questions (RQ) are formed:

RQ1: *How does the proposed approach compare with state-of-the-art approaches in terms of performance with respect to time, selected tests and fault detection capabilities?* This question aims to detail a comparison of the performances between different techniques in terms of total test time, selected test ratio and fault detection rate.

RQ2: *How does the proposed approach compare with state-of-the-art approaches in terms of cost-efficiency?* This question aims to study the proposed approach's efficiency whether it is beneficial to use or not depending on different cost criteria.

5.2. Design

The evaluation study was designed as an embedded, multiple case study. The techniques are applied to large open source projects shown in the Table 5.1.

Many traditional techniques use programs provided by Software Infrastructure Repository (SIR) programs [8], [9]. Although SIR programs are stored with real fault information, many of the programs are only representative of small samples and therefore do not reflect real life world cases [37]. Instead, the real-life projects from GitHub repository is used in this thesis. The projects are taken from other studies of state-of-the-art techniques in literature in order to verify the replication [12], [15], [1]. We have chosen 21 Maven projects since most of the state-of-the-art techniques have support for Maven projects. We have chosen projects with long running tests since they are more likely to benefit from RTS. We have only included revisions in which the programs are compiled and all of tests are run successfully (i.e. mvn test command is executed without failure). Finally, any revision in which one of the techniques failed to operate are also excluded. This resulted in: on average 32.3 kLOCs, 44.8 revisions with 74,790 tests, and 175.8 seconds; in total 677.4 kLOCs with 1,570,600 tests which run in 3691 seconds.

Table 5.1. Statistics of open source projects used in the study

#	Program	# of Version	kLOC	# of Tests (k)	Test Time (s)
1	headius/invokebinder	66	2.0	2.2	3.91
2	google/compile-testing	30	3.0	7.6	8.51
3	apache/commons-cli	50	5.9	23.0	8.74
4	logstash/logstash-logback-encoder	43	3.2	18.7	7.82
5	apache/commons-dbutils	33	5.4	23.2	9.43
6	apache/commons-validator	19	11.9	61.0	11.04
7	apache/commons-fileupload	54	4.3	12.0	11.04
8	apache/commons-codec	63	17.0	47.5	14.95
9	srt/asterisk-java	59	34.5	38.1	14.03
10	apache/commons-compress	12	32.5	89.4	21.62
11	apache/commons-email	23	6.5	17.0	28.29
12	apache/commons-lang	61	69.0	133.8	49.68
13	apache/commons-collections	66	54.3	149.6	45.77
14	apache/commons-imaging	87	37.1	58.9	66.47
15	apache/commons-math	57	185.4	450.2	251.39
16	addthis/stream-lib	5	8.3	24.0	241.04
17	apache/commons-io	49	27.7	93.9	210.22
18	brettwooldridge/hikaricp	49	9.4	21.0	184.46
19	opentripplanner/opentripplanner	20	79.3	135.8	639.17
20	apache/commons-pool	51	12.8	19.5	677.58
21	jankotek/mapdb	57	67.9	144.2	1186.57

After the selection of the programs and revisions, we applied each RTS technique to each revision of programs and gathered related metrics. The overall process of evaluation steps is shown in Figure 5.2. The process is automated and the results are generated for each technique with detailed metric results. The flaky tests and fault history are manually extracted since the project repositories did

not have the information. Mutation testing is used to inject faults to the versions so as to gather fault related metrics [119].

We have used state-of-the-art techniques summarized in the Table 5.2 for our evaluation purposes. Retest-all is selected as a base version to clearly observe the RTS gains. We used HyRTS base variant and we used the same similarity threshold for both SPIRITuS and our approach not to compromise the replicability of the study with SPIRITuS study results [15].

Table 5.2. Techniques used in the case study

Technique	**Description**
Retest-all	Base
Ekstazi	-
STARTS	-
HyRTS	Base Variant
SPIRITuS	Threshold is set to 0.975
Our approach	Threshold is set to 0.975

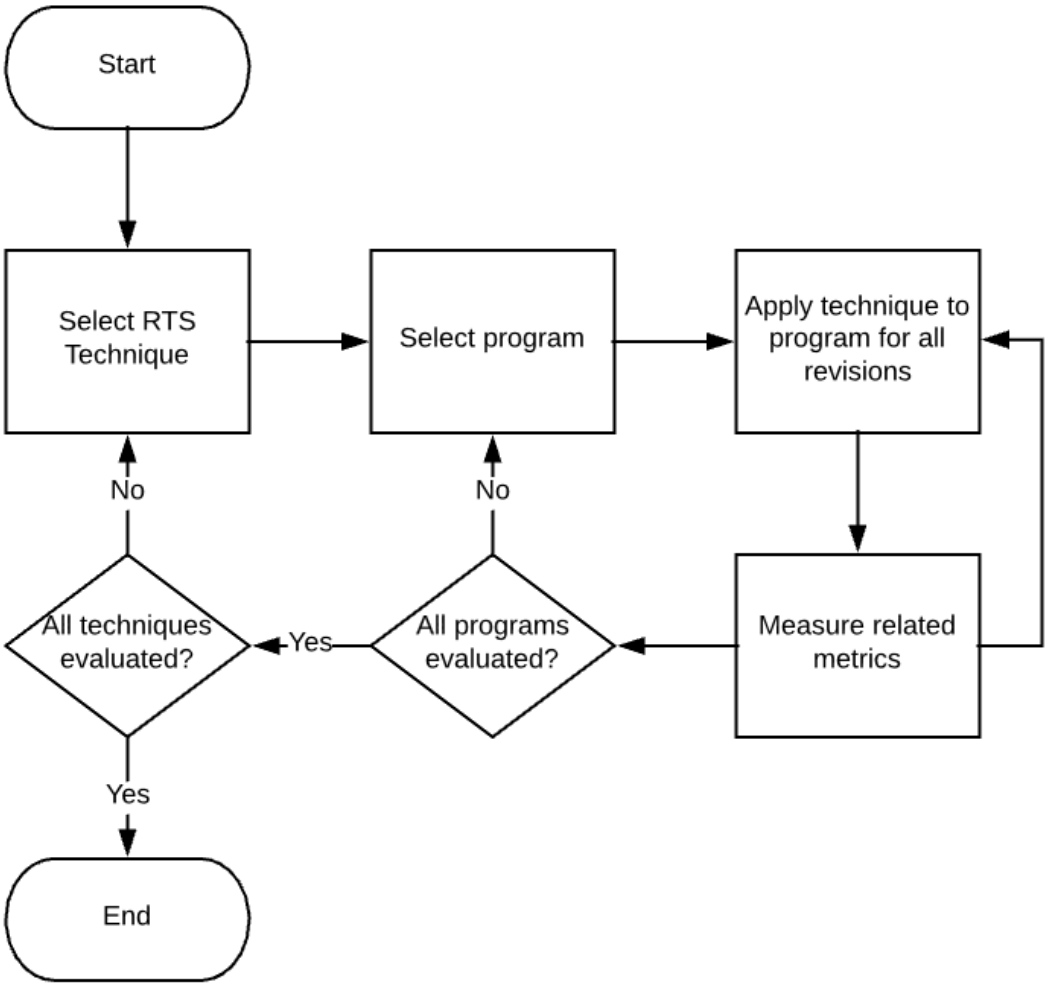


Figure 5.2. Overview of automated evaluation steps of subjects and techniques

The experiments are run on a computer with 4 GHz Intel i5-4670K with 16 GB of RAM, running Windows 10 with Java 8 64 bit JDK.

The following metrics are computed during experiments:

- End-to-end runtime of tests, both offline and online parts of the techniques
- Percentage of tests selected
- FDR, Fault Detection Rate

5.3. Results

5.3.1. RQ1: How does the proposed approach compare with state-of-the-art approaches in terms of performance with respect to time, selected tests and fault detection capabilities?

5.3.1.1. Time

The Table 5.3 summarizes the end-to-end runtime of tests results. The programs are listed in the ascending order of the test execution time regarding *retest*-all approach. Note that, *online* measurement of Ekstazi, STARTS and HyRTS represents the end-to-end time calculated with analysis, execution and collection phases of the techniques. Whereas *offline* measurement of the respected techniques is calculated with analysis and execution, leaving out the collection phase. Analysis phase conforms to selecting test cases, execution phase represents the execution of the running of the selected tests and collection phase includes building or updating of the coverage/dependency matrix while executing test cases in the current version. On the other hand, SPIRITuS and our approach works differently. In the *offline* phase, text parsing/data preparation processing are done. In the *online* phase, in addition to offline phase, selecting test cases and building or updating coverage matrix steps are carried out.

By looking at the end-to-end time in *offline* mode, the overall fastest technique is HyRTS which is computing checksums of differences based multiple granularity level approach. Other approaches using checksum are also faster compared to our approach. Our approach is on average 23% slower than checksum based approaches as expected since there is more computational work done in our approach. When compared to SPIRITuS, it is 11% faster as it calculates changes in file level, which means there is less overhead.

By looking at the end-to-end time in *online* mode, the overall fastest technique is Ekstazi which is computing checksums and building coverage matrix on class level. It is faster than HyRTS since HyRTS needs to analyze methods for building dependency matrix. Our approach is on average 27% slower than checksum based approaches as expected since the overhead calculations from *offline* mode is also reflected in this mode. When compared to SPIRITuS, it is 7% faster

as it calculates changes in file level but it also uses extra parameters when calculating coverage matrix such as test history.

The slowness of the proposed approach is explained by the offline phase being responsible for almost %99 of the total time. The analysis phase could be improved via testing other NLP methods instead of VSM - more basic methods could be fast and effective at the same time - since the analysis phase is only needed if a new modified version is available. Analysis part could be integrated to version control systems to automate the process and reduce the perceived time. On the contrast, this approach is not applicable to other state-of-the-art techniques since there is only small overhead of online processing. This phenomenon is better shown in the Table 5.3. The programs are listed in ascending order with respect to the *retest-all* time. The gray areas on the table represents the time that took longer than *retest-all* approach. To illustrate, all of the techniques took longer times for Apache Commons CLI project than retest-all approach.

Table 5.3. Experimental results of the case study

			Ekstazi		STARTS		HyRTS		SPIRITuS		Our Approach	
#	Program	*Retest-all*	*Offline*	*Online*	*Offline*	*Online*	*Offline*	*Online*	*Offline*	*Online*	*Offline*	*Online*
1	headius/invokebinder	3.9100	4.4965	4.9266	4.2228	5.0830	6.4906	6.7643	10.5540	11.7537	10.09828	11.2314
4	logstash/logtash-logback-encoder	7.8200	8.2110	9.3840	8.7584	10.1660	6.8816	6.9598	10.8720	11.5689	10.37241	11.0255
2	google/compile-testing	8.5100	8.7653	9.6163	9.7865	11.4885	8.4249	8.7653	9.3120	10.6800	9.168421	10.2953
3	apache/commons-cli	8.7400	9.2644	9.9636	9.5266	10.9250	8.8274	9.7888	12.8520	14.2632	12.07931	13.3488
5	apache/commons-dbutils	9.4300	9.4300	9.9958	10.2787	11.3160	9.0528	9.1471	10.3200	11.0613	10.21429	10.8685
6	apache/commons-validator	11.0400	10.2672	10.8192	10.2672	11.8128	8.3904	8.7216	11.5260	12.5765	11.47818	12.3255
7	apache/commons-fileupload	11.0400	9.7152	10.4880	10.8192	11.2608	10.7088	12.6960	12.0900	13.3079	11.51304	12.4329
9	srt/asterisk-java	14.0300	6.1732	6.8747	9.9613	10.9434	6.5941	7.0150	7.5120	7.8355	7.707143	8.0715
8	apache/commons-codec	14.9500	8.3720	8.6710	10.1660	10.7640	6.4285	6.7275	15.0780	15.3548	14.4625	14.9050
10	apache/commons-compress	21.6200	11.6748	12.5396	17.0798	17.7284	12.9720	14.2692	17.0460	18.3040	15.95263	17.2324
11	apache/commons-email	28.2900	14.1450	14.1450	19.5201	19.8030	22.6320	24.0465	24.9120	26.1736	23.64727	24.6177
13	apache/commons-collections	45.7700	20.1388	21.0542	26.5466	26.5466	18.7657	19.2234	24.0120	25.6111	22.43929	24.0485
12	apache/commons-lang	49.6800	21.8592	23.8464	36.2664	38.2536	17.3880	17.8848	36.1260	38.0208	32.68947	34.1841
14	apache/commons-imaging	66.4700	30.5762	32.5703	43.2055	43.8702	33.2350	37.8879	42.5820	48.0923	37.70862	41.4398
18	brettwooldridge/hikaricp	184.4600	175.2370	178.9262	177.0816	175.2370	101.4530	101.4530	120.3120	186.5852	109.3892	156.6984
17	apache/commons-io	210.2200	42.0440	44.1462	90.3946	90.3946	46.2484	48.3506	181.2900	193.4948	161.4336	177.3956
16	addthis/stream-lib	241.0400	84.3640	86.7744	113.2888	115.6992	62.6704	72.3120	120.2460	134.4432	105.5617	118.1772
15	apache/commons-math	251.3900	45.2502	50.2780	72.9031	72.9031	67.8753	140.7784	145.4940	156.4714	127.5165	137.0139
19	opentripplanner/opentripplanner	639.1700	428.2439	453.8107	530.5111	536.9028	325.9767	524.1194	410.9880	447.8617	364.7062	393.7629
20	apache/commons-pool	677.5800	352.3416	359.1174	386.2206	392.9964	352.3416	345.5658	447.4560	529.3566	396.9788	483.2285
21	jankotek/mapdb	1186.5700	486.4937	498.3594	806.8676	795.0019	450.8966	628.8821	506.2560	648.8005	461.2327	648.9515
	Average	**NA**	**85.0982**	**88.3956**	**114.4606**	**115.1951**	**75.4407**	**97.6837**	**103.6589**	**121.9818**	**93.1595**	**112.4407**

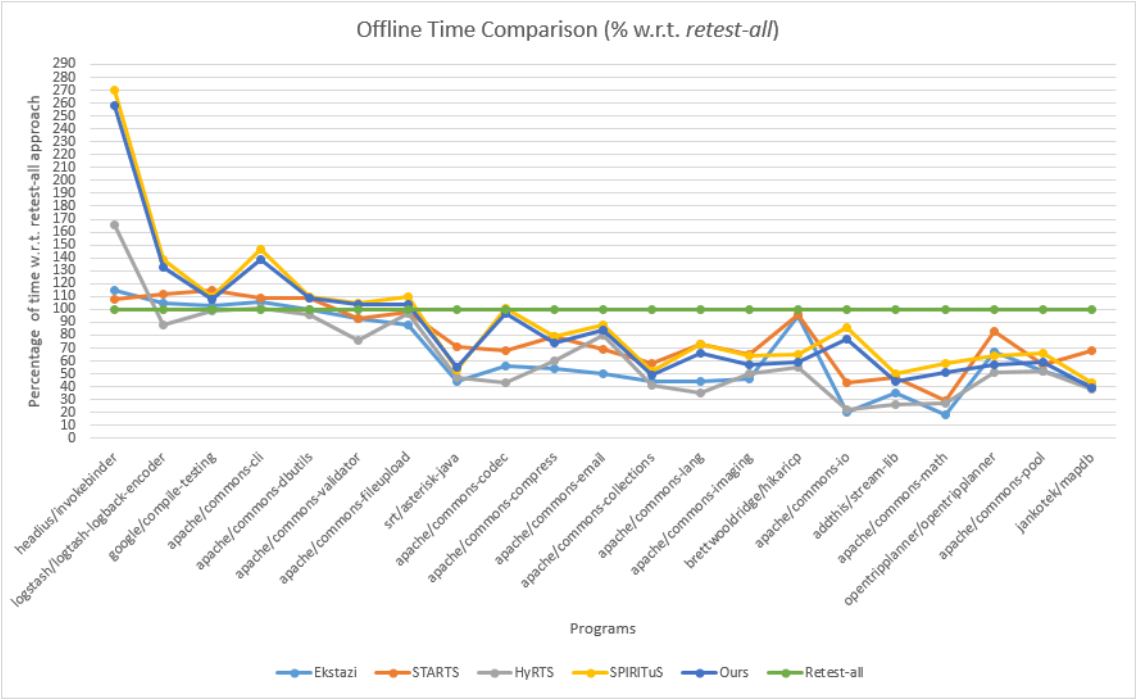


Figure 5.3. Offline time comparison of case study with respect to *retest-all*

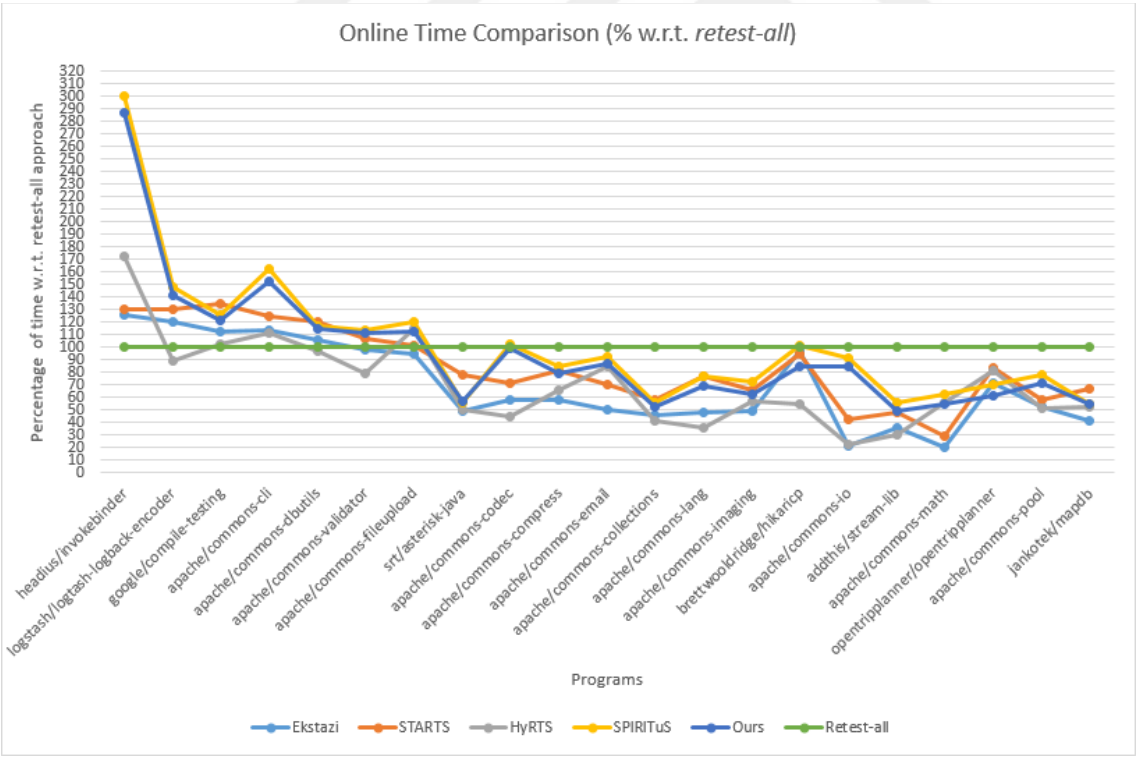


Figure 5.4. Online time comparison of case study with respect to retest-all

When compared with respect to the *retest-all* approach it is observed that some of the techniques takes longer times than testing all of the test cases. Further

investigation reveals that when the test execution times are low it is not feasible to use any RTS technique since it hurts the performance. It is advised that when the running times of tests are relatively low, 10-20 sec, instead of using an RTS technique, all test cases should run. Although in general it can be said that when the software size is relatively low it is preferable not to use any RTS technique, we could not detect a pattern with size of the programs in lines of code so that after certain threshold it is beneficial to perform regression testing. In our experiment we observed that while some programs such as #11, #16, #18 is fairly low on size compared to other subjects have reduced time results using RTS techniques, the programs such as #4 did not benefit from using RTS techniques at all. Similar conclusions are also deducted from online time analysis. However, note that in online analysis our technique also includes test execution times like other techniques and still manages to perform reasonably well in spite of having a much larger offline overhead performance as shown in Figure 5.3 and Figure 5.4.

Table 5.4. Analysis, Selection and Total time results of proposed approach

		Our Approach		
#	Program	*Analysis*	*Selection*	*Total*
1	headius/invokebinder	10.09828	0.0021	**10.1004**
4	logstash/logtash-logback-encoder	10.37241	0.0024	**10.3749**
2	google/compile-testing	9.168421	0.0027	**9.1712**
3	apache/commons-cli	12.07931	0.0046	**12.0839**
5	apache/commons-dbutils	10.21429	0.0015	**10.2158**
6	apache/commons-validator	11.47818	0.0041	**11.4823**
7	apache/commons-fileupload	11.51304	0.0032	**11.5162**
9	srt/asterisk-java	7.707143	0.0027	**7.7098**
8	apache/commons-codec	14.4625	0.0052	**14.4677**
10	apache/commons-compress	15.95263	0.0038	**15.9564**
11	apache/commons-email	23.64727	0.0075	**23.6548**
13	apache/commons-collections	22.43929	0.0135	**22.4528**
12	apache/commons-lang	32.68947	0.0144	**32.7039**
14	apache/commons-imaging	37.70862	0.0181	**37.7267**
18	brettwooldridge/hikaricp	109.3892	0.0857	**109.4749**
17	apache/commons-io	161.4336	0.0003	**161.4340**
16	addthis/stream-lib	105.5617	0.0423	**105.6041**
15	apache/commons-math	127.5165	0.0061	**127.5227**
19	opentripplanner/opentripplanner	364.7062	0.0210	**364.7272**
20	apache/commons-pool	396.9788	0.0165	**396.9952**
21	jankotek/mapdb	461.2327	0.2047	**461.4374**
	Average	**93.1595**	**0.0220**	93.1815

Table 5.4 shows the analysis and selection phase time results of our approach. In analysis phase, the operations described in section 4.2 analysis phase and initial computation of coverage matrix (first coverage matrix building) are carried out and time measurements are taken. In selection phase, all the operations in section 4.2 execution phase except computation of coverage matrix is timed. This is to show that if the analysis phase and computations of coverage matrix could be performed in the background, the test selection is fairly fast compared to other techniques. %99 of the cost of our technique comes from the analysis phase.

5.3.1.2. Selected Test Ratio

The percentage of the selected tests are shown in the Table 5.5. The figure is given as tests selected with respect to retest-all approach for ease of understanding. STARTS, static approach, selects most test cases on average,

while HyRTS outperforms coarse granularity approaches like Ekstazi. SPIRITuS selects almost twice less tests then HyRTS since it follows a more granular approach. Our approach performs closely to SPIRITuS but selects 2% less tests than SPIRITuS compared to retest all approach. This is expected since our approach is finer granularity approach than other techniques.

Table 5.5. Selected test ratio with percentages with respect to *retest-all*

#	Program	*Ekstazi (%)*	*STARTS (%)*	*HyRTS (%)*	*SPIRITuS (%)*	*Ours(%)*
1	headius/invokebinder	78.83	76.19	63.99	29.33	28.15
4	logstash/logtash-logback-encoder	22.29	23.31	12.35	9.41	8.88
2	google/compile-testing	40.30	44.26	35.69	15.35	13.55
3	apache/commons-cli	37.44	44.07	27.48	16.65	14.61
5	apache/commons-dbutils	20.59	33.72	14.26	8.74	6.46
6	apache/commons-validator	18.83	22.10	4.51	9.01	7.83
7	apache/commons-fileupload	21.76	31.88	25.38	10.55	8.16
9	srt/asterisk-java	4.72	6.41	2.67	2.25	2.81
8	apache/commons-codec	3.86	4.46	2.64	1.76	2.98
10	apache/commons-compress	12.01	26.83	8.98	5.72	5.97
11	apache/commons-email	10.62	30.23	22.10	5.23	3.85
13	apache/commons-collections	8.42	4.30	1.65	3.29	3.31
12	apache/commons-lang	9.27	32.36	4.80	4.18	3.11
14	apache/commons-imaging	17.04	37.26	13.89	8.65	5.05
18	brettwooldridge/hikaricp	77.95	84.23	48.21	35.44	25.26
17	apache/commons-io	13.20	23.13	8.33	6.38	7.38
16	addthis/stream-lib	13.47	22.92	11.43	4.88	5.29
15	apache/commons-math	8.76	9.45	3.16	4.20	3.39
19	opentripplanner/opentripplanner	11.58	56.75	18.54	5.10	4.79
20	apache/commons-pool	28.81	33.86	29.26	11.69	12.20
21	jankotek/mapdb	32.59	47.70	28.68	12.03	15.54
	Average	23.44	33.12	18.48	9.99	8.98

5.3.1.3. Fault Detection Rate

Another set of experiments are performed on the mutated versions of the programs. Mutations are performed with PIT tool [119] and the coverage of the mutations are given in the Table 5.6. PIT tool is used with default mutation operators listed as;

- **Conditionals Boundary:** Relational operators are replaced with each other such as <, <=, >, >=.
- **Increments:** Any increment is replaced with decrements and vice versa such as ++, --.
- **Invert Negatives:** Integer and floating point numbers are inverted such as variable with (-)variable.

- **Math:** Binary arithmetic operations are replaced with another operation such as +, -, *, /, %, &, |, ^, <<, >>, >>>.
- **Negate Conditionals:** All conditionals are replaced with other conditionals such as ==, !=, <=, >=, <, >.
- **Return Values:** The return types of the functions are replaced with its counterparts such as boolean true with false.
- **Void Method Call:** void return typed function calls are removed.

To conduct a more controlled comparison of the techniques, fault mutation is selected instead of using actual faults reported in the studies. It is shown that mutation faults could be used to assess software systems instead if real faults [120]. We have chosen a random number of mutations between 0-20 to be able to reflect the real world situations [30], [121]. We generated 20 faulty versions of each program and if the program has less than 20 versions we used all versions to generate its faulty counterpart. Before applying each technique, it is checked and guaranteed that all the mutations could be killed with the test cases, meaning no mutation that could escape the retest-all approach is selected. Note that, any safe RTS technique will have %100 FDR since it is supposed to capture all the modification revealing test cases by definition.

The safety of an RTS technique is measured by its inclusiveness, meaning if the technique is able to detect all modification revealing test cases, then it is deemed as a safe RTS technique. Most of the traditional techniques are safe since they are focused on modifications on the program. However, any exclusion of third-party changes could result in an unsafe technique as observed in some of the method level static analysis approaches [12]. When the techniques used in the case study, following results are acquired:

Ekstazi and HyRTS are safe since both of the techniques evaluate dynamic dependencies at different granularity levels, thus ensuring any modification will be detected. Nevertheless, the safe techniques are also evaluated in this section.

STARTS is an unsafe technique since it analyses the program statically, which can neglect any runtime modifications made to the program.

SPIRITuS and our approach follow a flexible approach in order to be cost-efficient in terms of time and fault detection capabilities. By increasing the similarity threshold, both of the techniques could be made safer. If the threshold is equal to 1, then all of the modifications would be captured by definition and both of the techniques can be considered safe. If the threshold is lowered then there is a risk some fault revealing test cases could be neglected, thus making the techniques unsafe. By analyzing FDR, we aim to show that inherently unsafe techniques could be used for cost-efficient reasons.

Table 5.6. Fault detection rate results

#	Program	*Ekstazi (%)*	*STARTS (%)*	*HyRTS (%)*	*SPIRITuS (%)*	*Ours(%)*
1	headius/invokebinder	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
4	logstash/logtash-logback-encoder	**100.00**	**94.70**	**100.00**	**95.30**	**99.00**
2	google/compile-testing	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
3	apache/commons-cli	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
5	apache/commons-dbutils	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
6	apache/commons-validator	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
7	apache/commons-fileupload	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
9	srt/asterisk-java	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
8	apache/commons-codec	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
10	apache/commons-compress	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
11	apache/commons-email	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
13	apache/commons-collections	**100.00**	**99.00**	**100.00**	**100.00**	**97.00**
12	apache/commons-lang	**100.00**	**99.50**	**100.00**	**93.00**	**100.00**
14	apache/commons-imaging	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
18	brettwooldridge/hikaricp	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
17	apache/commons-io	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
16	addthis/stream-lib	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
15	apache/commons-math	**100.00**	**97.30**	**100.00**	**98.30**	**100.00**
19	opentripplanner/opentripplanner	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
20	apache/commons-pool	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
21	jankotek/mapdb	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
	Average	**100.00**	**99.55**	**100.00**	**99.36**	**99.81**

We prove that using our approach is highly safe only failing in two of the programs. We have improved the SPIRITuS safeness because our approach uses class/file level analysis instead of method level analysis and therefore is more robust. Method level granularity analysis could be affected from exclusion of the third-party libraries, whereas class level granularity approaches are immune proven in Legunsen's study [1].

5.3.1.4. Detailed Time Analysis

We observed that our technique performs poorly on timewise whereas it has a lower selected test ratio and maintains a comparative fault detection rate. Since our technique is also finding the same faults with other techniques on such a small number of tests, we decided to further investigate the selected tests for different approaches. To this end, we analyzed some of the programs in detail, namely: #15 (apache/commons-math) and #11 (apache/commons-email). We compared our technique with the HyRTS technique for ten consecutive revisions (excluding initial run since all test are executed on initial run) of the programs. Program #15 has the highest number of tests and smallest test selection ratio difference whereas program #11 has highest test selection ratio difference. Note that, FDR is 100% for both programs for all revisions with each technique.

First, we have look at whether the selected test cases with our approach is a subset of the selected test cases by other approaches. If this were to be the case, it could be deduced that although the analysis part takes longer, the test execution part is always takes smaller time in our technique compared to others. We have observed this to be the case for most of the revisions but we have also encountered 3% difference between selected test cases on average as shown in Table 5.7. By only our approach represents the tests selected by our approach and not selected by HyRTS. By our approach and HyRTS represents tests that are common in both techniques. HyRTS represents all tests selected by HyRTS.

Table 5.7 Number of selected test results

	Programs/Revisions	1	2	3	4	5	6	7	8	9	10
Email	By only our approach	2	0	1	0	2	0	1	0	0	1
	By our approach and HyRTS	37	35	28	23	30	27	29	33	31	20
	HyRTS	178	175	163	155	165	164	168	171	160	140
Math	By only our approach	11	11	10	12	9	10	11	13	13	12
	By our approach and HyRTS	265	275	268	263	240	255	260	280	274	278
	HyRTS	255	260	262	249	224	247	241	264	247	254

Tests selected by our technique are not a subset of tests selected by other techniques. However, providing the test execution time comparison of each technique we observed that test execution times are in parallel with selected test

ratio as shown in Figure 5.5. We have compared 10 consecutive revisions of each program with each technique and analyzed only the test execution times with respect to *retest-all* approach.

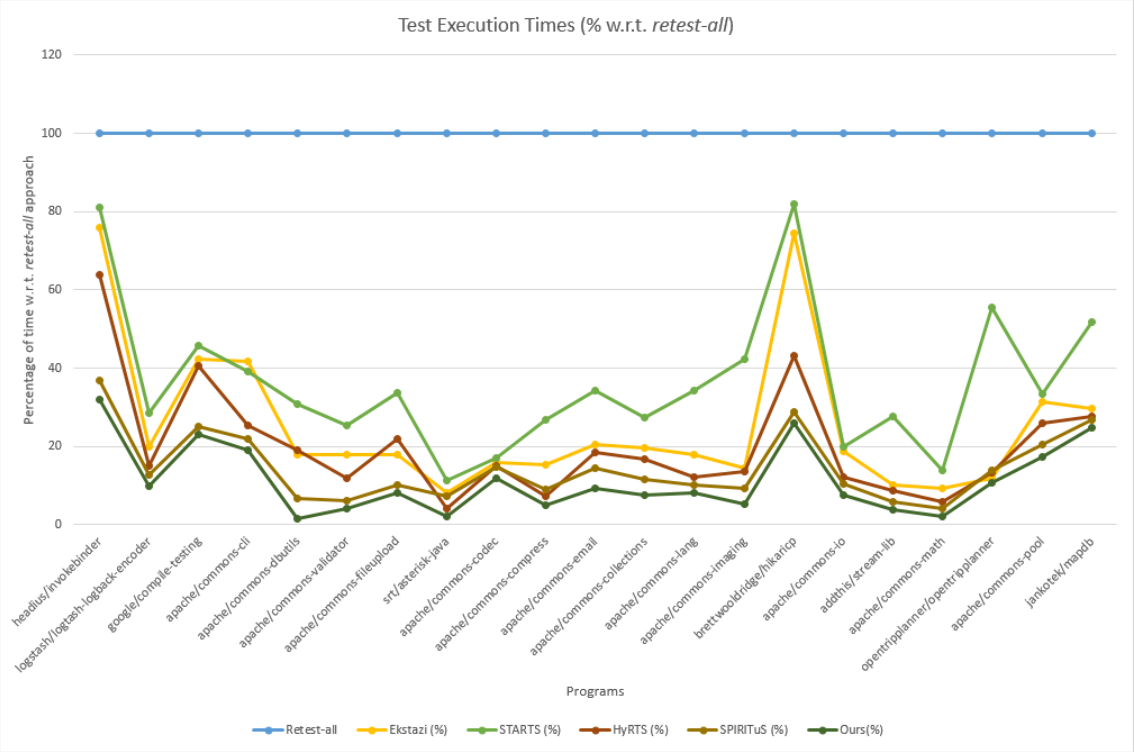


Figure 5.5 Test execution time results with respect to the retest-all approach

Compared to the selected test ratio, the test execution time is observed to be following in a parallel line closely for each technique. Test execution time and selected test ratio is given in Figure 5.6. By the observation of the results, it is safe to assume that selected test ratio and test execution times are closely related; ergo, it can be inferred that our technique is effective in test execution time since it has the lowest selected test ratio. Furthermore, considering the analysis phase could be computed offline, the test execution time results could be deciding factor when choosing an RTS technique if the tests are executed on a shared or resource limited machine.

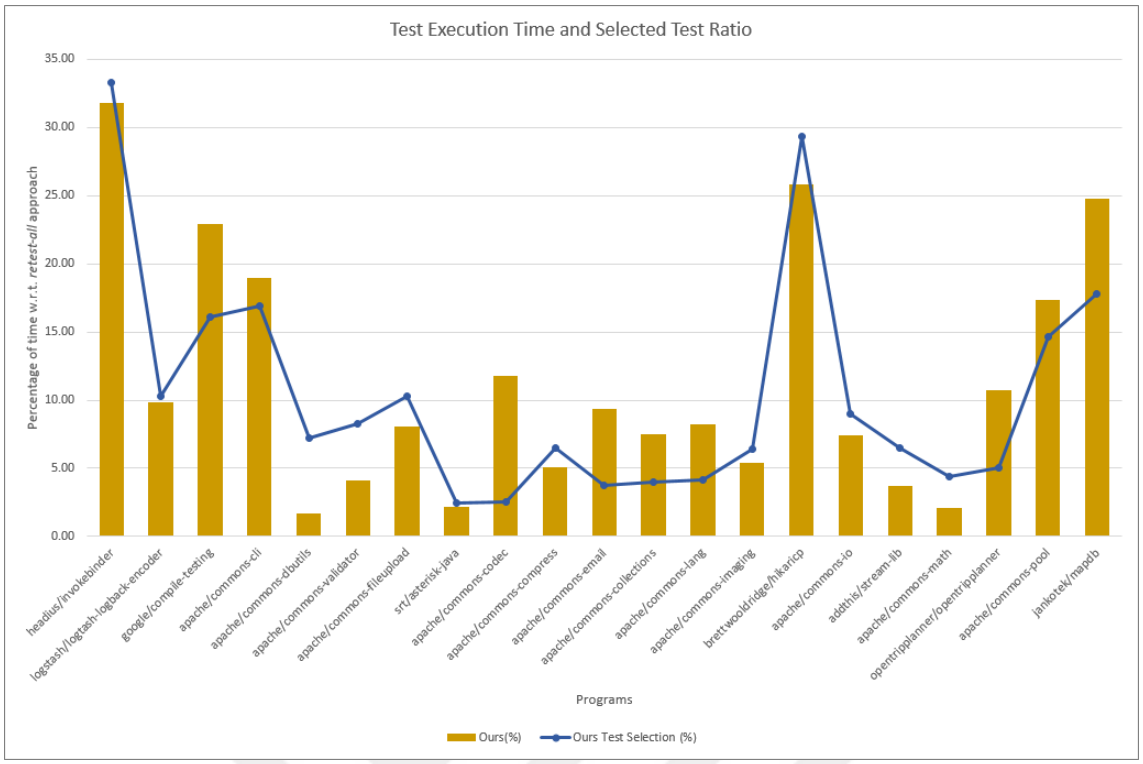


Figure 5.6 Test execution time and test selection ratio for our approach

5.3.2. RQ2: How does the proposed approach compare with state-of-the-art approaches in terms of cost-efficiency?

By combining the results we acquired in the RQ1, we can deduce the following:

- Timewise our approach has a much higher overhead than state-of-the-art approaches. Our technique performs 23% poorer than its counterparts. This makes our approach not feasible if the time is of the essence by focusing purely on time analysis.
- Our approach selects less tests than other state of the art techniques. Our technique performs almost three times better than Ekstazi and almost two times better than HyRTS approach in terms of selected test ratio. Moreover, by tuning the similarity threshold the ratio of test selection can be adjusted. For example, if the computing power is an important aspect of regression testing, user could sacrifice fault detection capabilities in favor of selecting smaller number of test cases. We observed that the threshold of 0.975 is adequate for most cases based on the threshold analysis [15].

- We have observed that by choosing much less tests the same fault detection rate of safe methods could be achieved. This is achieved since our approach is finer granularity level than other approaches.

The cost of the regression test selection could be defined as multifold as studies with multiple objectives show [48], [65], [79], [108], [122]–[125]. Time, selected test number, computation power are the main costs considered in most studies but the cost could be different from project to project even in the same company. Therefore, we proposed a tunable regression test selection that answers to the needs of the user. If time is determining factor, the approach could be adopted to perform faster by changing similarity approaches. Or if the execution of tests is computationally expensive, fewer tests could be selected at the expense of fault detection. By this reasoning, we consider the proposed technique as cost-efficient once the required adjustments are made.

5.4. Threats to Validity

In this section, the validity of the thesis is discussed in terms of construct, internal and external validities and reliability [117].

5.4.1. Construct Validity

The construct validity is related to measurement instruments not being capturing the correct concepts [117]. In our thesis, the measurements of the metrics in evaluation can be considered as a threat to the construct validity. To alleviate this thread, the measurements of the related metrics are calculated with respect to the retest-all approach so as to define a common ground for comparison. Furthermore, the used metrics that are widely accepted in literature and adopted to several studies as discussed in section 2.1.

5.4.2. Internal Validity

The internal validity is related to inner instruments used in study that can affect author's judgement [117]. This can apply to the thesis in the following ways:

- Case Study: The approaches we have used in the case study are well documented and partially available online. If the technique/tool is

available, we have used the technique and the tools as instructed by the authors of respective studies. However, it is still possible these tools contain bugs.

- Proposed Approach: Our own developed technique is another threat to the internal validity of the study. This study is part of a thesis and the technique is developed and implemented by the sole author. It is possible for the approach to contain bugs that affect performance of the technique. Moreover, although state-of-the-art and widely accepted frameworks/tools are used in implementation of the technique, it is possible writer's knowledge of the software ecosystem and personal tendencies could introduce threats to the approach.
- Conceptual Model: The grounded theory is used to alleviate any threats manifesting in the construction of the conceptual model. Additionally, the proposed technique is only shaped and begin implementation phase after the conceptual model is analyzed to disregard any misdirection that could occur.

5.4.3. External Validity

The external validity is related to conditions that limit the researchers' generalization capabilities [117]. To eliminate any external threats, the units of analysis used in the case study is strictly limited to other projects used in the RTS field and publicly available. The revisions of the projects are carefully selected following the processes of other techniques closely [15], [69], [102], [1]. In order to eliminate data bias that could occur, we used different versions of the programs from other studies. However, the units of analysis are fairly large and modern projects. As a result, the threat of generalizing the proposed approach to smaller projects persists in the study.

5.4.4. Reliability

Reliability requires that the study can be repeated with the same steps by other researchers. To counteract this, all of the steps of the case study is properly defined with assisting figures in section 5.2. In addition, there should be more

empirical studies conducted in various projects such as having size ranging from small to large and in different programming languages.

6. CONCLUSION

Regression testing is an essential part of the modern software cycle. In order to keep up with the fast pace of development without breaking any changes made to the software, regression testing must be performed in a timely and cost-efficient manner. In this thesis, to study the area of regression test selection, we first constructed a conceptual model, bridging the gap between traditional and modern approaches and to visually understand and identify the parts of the field that could be improved.

The detailed analysis of the conceptual model revealed that there are many scenario specific solutions to the regression test selection but the techniques are not adequate for large, complex and multi-lingual software projects. Thus, a fast, efficient, language-independent, easy to use and manageable technique is needed for the high adoption rate in the industry. To this end, an adjustable, hybrid regression test selection technique is presented for teams working with large, complex software systems that are under continuous integration and deployment cycles. The proposed technique aims to improve different goals in different times such as computation power, time and fault detection rate and also working with various different software modules written in different languages.

The proposed technique uses vector space models to lexically compare different class and configuration-like files at a coarse granularity while using binary checksum comparison for any third-party dependencies. This approach envisions the needs of a modern software project which is version controlled, uses different programming languages and developed in an agile framework. The proposed approach answers these demands by being language independent, easy-to-use and able to work in the background without interfering with the development process.

In order to compare the proposed technique with state-of-the-art techniques, an embedded, multiple case study is performed. The techniques are evaluated in terms of performance with respect to time, selected tests and fault detection capabilities and in terms of cost-efficiency. Time results showed that although by

end-to-end time our technique performs poorer than some of the other techniques, the test execution is faster in our approach. Hence, the slowness of our approach is caused by the analysis phase which can be fastened with background analysis, different processing techniques and adjusting to threshold for various time needs. The proposed technique is superior in terms of selected test ratio, meaning it selects much fewer tests compared to other state-of-the-art techniques while maintaining a relatively safe fault detection rate. Moreover, the fault detection rate is highly safe, 99.75%, and it can be higher or lower by adjusting the threshold value. The proposed approach is suited to comply with different cost definitions, such as time, computation power, selected test number via a tunable threshold parameter. This results in a cost-efficient approach. After a comprehensive multiple case study, the results showed that the proposed approach is effective as other state-of-the-art techniques and selects fewer tests while keeping the fault detection rate at a high level.

As future work, the proposed technique is expected to be integrated with highly used different tools such as integrated development environments (IDEs), junit test framework, dependency management systems and continuous integration servers in favor of the increase of usability and maintainability. Moreover, text processing part could be tested with emerging natural language processing techniques so the time performance of the analysis part could be improved.

REFERENCES

[1] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, “An extensive study of static regression test selection in modern software evolution,” *Proc. 2016 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. - FSE 2016*, pp. 583–594, 2016.

[2] R. Potvin and J. Levenberg, “Why Google stores billions of lines of code in a single repository,” *Commun. ACM*, vol. 59, no. 7, pp. 78–87, Jun. 2016.

[3] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, “Regression test selection techniques: A survey,” *Inform.*, vol. 35, no. 3, pp. 289–321, 2011.

[4] R. S. Pressman, *Software Engineering A Practitioner’s Approach*. New York: McGraw Hill, 2002.

[5] G. Rothermel and M. J. Harrold, “A framework for evaluating regression test selection techniques,” *Proc. 16th Int. Conf. Softw. Eng.*, pp. 201–210, 1994.

[6] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, “Effective Regression Test Case Selection: A Systematic Literature Review,” *ACM Comput. Surv.*, vol. 50, no. 2, pp. 1–32, 2017.

[7] S. Yoo and M. Harman, “Regression Testing Minimisation, Selection and Prioritisation : A Survey,” *Test. Verif. Reliab*, vol. 00, pp. 1–7, 2007.

[8] E. Engström, P. Runeson, and M. Skoglund, “A systematic review on regression test selection techniques,” *Inf. Softw. Technol.*, vol. 52, no. 1, pp. 14–30, Jan. 2010.

[9] A. Orso and G. Rothermel, “Software testing: a research travelogue (2000–2014),” in *Proceedings of the on Future of Software Engineering - FOSE 2014*, 2014, pp. 117–132.

[10] A. Shi *et al.*, “Comparing and combining test-suite reduction and regression test selection,” *Proc. 2015 10th Jt. Meet. Found. Softw. Eng.*, pp. 237–247, 2015.

[11] R. H. Rosero, O. S. Gómez, and G. Rodríguez, “15 Years of Software Regression Testing Techniques — A Survey,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 26, no. 05, pp. 675–689, 2016.

[12] O. Legunsen, A. Shi, and D. Marinov, “STARTS: STAtic regression test selection,” *ASE 2017 - Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, no. iv, pp. 949–954, 2017.

[13] K. Wang *et al.*, “Towards Refactoring-Aware Regression Test Selection,” p. 12, 2018.

[14] M. Gligoric, L. Eloussi, and D. Marinov, “Ekstazi: Lightweight Test Selection,” *Proc. - Int. Conf. Softw. Eng.*, vol. 2, pp. 713–716, 2015.

[15] S. Romano, G. Scanniello, G. Antoniol, and A. Marchetto, “SPIRITuS: a SimPle Information Retrieval regressIon Test Selection approach,” *Inf. Softw. Technol.*, vol. 99, pp. 62–80, Jul. 2018.

[16] S. Hafez, M. Elnainay, M. Abougabal, and S. Elshehaby, “Potential-fault cache-based regression test selection,” *Proc. IEEE/ACS Int. Conf. Comput. Syst. Appl. AICCSA*, 2017.

[17] L. Zhang, “Hybrid Regression Test Selection,” pp. 199–209, 2018.

[18] Y. Pang, X. Xue, and A. S. Namin, “Identifying effective test cases through K-means clustering for enhancing regression testing,” *Proc. - 2013 12th Int. Conf. Mach. Learn. Appl. ICMLA 2013*, vol. 2, pp. 78–83, 2013.

[19] D. Walker and F. Myrick, “Grounded theory: An exploration of process and procedure,” *Qual. Health Res.*, 2006.

[20] V. Stray, D. I. K. Sjøberg, and T. Dybå, “The daily stand-up meeting: A grounded theory study,” *J. Syst. Softw.*, vol. 114, pp. 101–124, 2016.

[21] IEEE, “IEEE Standard Glossary of Software Terminology.” p. 84, 1990.

[22] D. Willmor, S. M. Embury, S. E. -, 2005. ICSM’05. Proceedings of The, and U. 2005, “A safe regression test selection technique for database-driven applications,” in *ieeexplore.ieee.org*, 2005, vol. 2005, pp. 421–432.

[23] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso, “Regression testing in the presence of non-code changes,” *Proc. - 4th IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2011*, pp. 21–30, 2011.

[24] H. K. N. Leung, “Insights into Regression Testing,” *Proc. Int. Conf. Softw. Maint.*, pp. 60–69, 1989.

[25] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, “An empirical study of regression test selection techniques,” *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 184–208, 2001.

[26] L. Chen and L. Zhang, “Speeding up Mutation Testing via Regression Test

Selection : An Extensive Study," *Proc. - 2018 IEEE 11th Int. Conf. Softw. Testing, Verif. Validation, ICST 2018*, pp. 58–69, 2018.

[27] H. K. N. Leung and L. White, "A study of integration testing and software regression at the integration level," *Proc. Conf. Softw. Maint. 1990*, pp. 290–301, 1990.

[28] H. Do and G. Rothermel, "An Empirical Study of Regression Testing Techniques Incorporating Context and Lifetime Factors and Improved Cost-Benefit Models," *Sigsoft'06/Fse-*, pp. 141–151, 2006.

[29] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, 1997.

[30] S. Mirarab, S. Akhlaghi, and L. Tahvildari, "Size-constrained regression test case selection using multicriteria optimization," *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 936–956, 2012.

[31] R. C. Ruth M., "A privacy-aware, end-to-end, CFG-based regression test selection framework for web services using only local information," *4th Int. Conf. Appl. Digit. Inf. Web Technol. ICADIWT 2011*, pp. 13–18, 2011.

[32] S. Huang, J. Zhu, and Y. Ni, "ORTS: a tool for optimized regression testing selection," *Proc. 24th ACM SIGPLAN Conf. companion Object oriented Program. Syst. Lang. Appl.*, pp. 803–804, 2009.

[33] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," *ICST 2010 - 3rd Int. Conf. Softw. Testing, Verif. Valid.*, pp. 137–146, 2010.

[34] W. Fu, H. Yu, G. Fan, X. Ji, and X. Pei, "A Regression Test Case Prioritization Algorithm Based on Program Changes and Method Invocation Relationship," *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, vol. 2017–Decem, pp. 169–178, 2018.

[35] P. Dhareula and A. Ganpati, "Prevalent Criteria's in Regression Test Case Selection Techniques: An Exploratory Study," 2015.

[36] A. Memon and Z. Gao, "Taming Google-Scale Continuous Testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, 2017.

[37] H. Do, "Recent Advances in Regression Testing Techniques," *Adv. Comput.*, vol. 103, pp. 53–77, Jan. 2016.

[38] V. Gupta and D. S. Chauhan, "Hybrid regression testing technique: A multi

layered approach," *Proc. - 2011 Annu. IEEE India Conf. Eng. Sustain. Solut. INDICON-2011*, 2011.

[39] A. S. A. Ansari, K. K. Devadkar, and P. Gharpure, "Optimization of test suite-test case in regression test," *2013 IEEE Int. Conf. Comput. Intell. Comput. Res. IEEE ICCIC 2013*, pp. 3–6, 2013.

[40] N. Ye, X. Chen, P. Jiang, W. Ding, and X. Li, "Automatic regression test selection based on activity diagrams," *2011 5th Int. Conf. Secur. Softw. Integr. Reliab. Improv. - Companion, SSIRI-C 2011*, pp. 166–171, 2011.

[41] H. Ural and H. Yenigün, "Regression test suite selection using dependence analysis," *J. Softw. Evol. Process*, vol. 25, no. 12, pp. 689–709, 2013.

[42] C. Tao, B. Li, X. Sun, and C. Zhang, "An Approach to Regression Test Selection Based on Hierarchical Slicing Technique," in *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, 2010, pp. 347–352.

[43] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," *Softw. Maintenance, 1992. Proceerdings., Conf.*, no. November, pp. 299–308, 1992.

[44] Vedpal and N. Chauhan, "Regression test selection for object oriented systems using OPDG and slicing technique," *2015 2nd Int. Conf. Comput. Sustain. Glob. Dev.*, pp. 1372–1378, 2015.

[45] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," *Proc. Eighth Int. Symp. Softw. Reliab. Eng.*, pp. 264–274, 1997.

[46] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of COMPONENT IDENTIFICATION METHODS APPLYING METHOD CAL....," no. October, 1995.

[47] F. I. Vokolos and P. G. Frankl, "Pythia: A regression test selection tool based on textual differencing," *Reliab. Qual. Saf. Software-Intensive Syst.*, pp. 3–21, 1997.

[48] V. Garousi, R. Özkan, and A. Betin-Can, "Multi-objective regression test selection in practice: An empirical study in the defense software industry," *Inf. Softw. Technol.*, no. June, pp. 0–1, 2018.

[49] B. Guo, M. Subramaniam, and H. F. Guo, "An approach to regression test selection of adaptive EFSM tests," *Proc. - 5th Int. Conf. Theor. Asp. Softw.*

Eng. TASE 2011, pp. 217–220, 2011.

[50] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, “Using semi-supervised clustering to improve regression test selection techniques,” *2011 Fourth IEEE Int. Conf. Softw. Testing, Verif. Valid.*, pp. 1–10, 2011.

[51] L. Yu, C. Liu, and Y. Zhang, “A multidimensional classification of safe regression test selection techniques,” *2012 Int. Conf. Syst. Informatics, ICSAI 2012*, no. Icsai, pp. 2516–2520, 2012.

[52] L. C. Briand, Y. Labiche, and S. He, “Automating regression test selection based on UML designs,” *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 16–30, 2009.

[53] A. Andrews, S. Elakeili, and A. Alhaddad, “Selective Regression Testing of Safety-Critical Systems: A Black Box Approach,” *Proc. - 2015 IEEE Int. Conf. Softw. Qual. Reliab. Secur. QRS-C 2015*, pp. 22–31, 2015.

[54] E. Fourneret, J. Cantenot, F. Bouquet, B. Legeard, and J. Botella, “SeTGaM: Generalized technique for regression testing based on UML/OCL models,” *Proc. - 8th Int. Conf. Softw. Secur. Reliab. SERE 2014*, pp. 147–156, 2014.

[55] P. Kandil, S. Moussa, and N. Badr, “Cluster-based test cases prioritization and selection technique for agile regression testing,” *J. Softw. Evol. Process*, vol. 29, no. 6, pp. 1–19, 2017.

[56] M. J. Harrold, A. Orso, and M. Lou Soffa, “Using Component Metadata to Support the Regression Testing of Component-Based Software Component Metadata for Regression Test,” *Proc. IEEE Int. Conf. Softw. Maint. (ICSM 2001); Florence; Italy; 7 Novemb. 2001 through 9 Novemb. 2001; Code 60554*, 2001.

[57] M. Skoglund and P. Runeson, “A case study of the class firewall regression test selection technique on a large scale distributed software system,” *IEEE/ACM Int. Symp. Empir. Softw. Eng. Meas. - ESEM*, vol. 00, no. c, pp. 72–81, 2005.

[58] M. J. Harrold *et al.*, “Regression test selection for Java software,” *ACM SIGPLAN Not.*, vol. 36, pp. 312–326, 2001.

[59] G. Rothermel, M. J. Harrold, and J. Dedhia, “Regression test selection for C++ software,” *Softw. Test. Verif. Reliab.*, vol. 10, no. 2, pp. 77–109, 2000.

[60] G. Xu and A. Rountev, “Regression test selection for AspectJ software,”

Proc. - Int. Conf. Softw. Eng., pp. 65–74, 2007.

[61] M. Ruth *et al.*, “A Safe Regression Test Selection Technique for Web Services,” in *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, 2007, pp. 0–5.

[62] M. Gligoric, L. Eloussi, and D. Marinov, “Practical regression test selection with dynamic file dependencies,” *Proc. 2015 Int. Symp. Softw. Test. Anal. - ISSTA 2015*, vol. 520, pp. 211–222, 2015.

[63] C. Sharma, S. Sabharwal, and R. Sibal, “A Survey on Software Testing Techniques using Genetic Algorithm,” *Int. J. Comput. Sci. Issues*, vol. 10, no. 1, pp. 381–393, 2013.

[64] M. Al-Refai, “Improving model-based regression test selection,” *CEUR Workshop Proc.*, vol. 2019, pp. 507–510, 2017.

[65] L. S. De Souza and R. B. C. Prudˆ, “Multi-Objective Test Case Selection : A study of the influence of the Catfish effect on PSO based strategies,” *An. do XV Work. Testes e Tolerância a Falhas -WTF 2014*, pp. 3–58, 2014.

[66] D. Rai and K. Tyagi, “Estimating the Regression Test Case Selection Probability using Fuzzy Rules,” pp. 603–611, 2013.

[67] B. Glaser, “Basics of grounded theory analysis: Emergence vs forcing,” 1992.

[68] B. Glaser, “Getting out of the data: Grounded theory conceptualization,” 2011.

[69] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, “Regression Test Selection Across JVM Boundaries,” *Proc. 2017 11th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2017*, pp. 809–820, 2017.

[70] Sujata and G. N. Purohit, “A schema support for selection of test case prioritization techniques,” *Int. Conf. Adv. Comput. Commun. Technol. ACCT*, vol. 2015–April, pp. 547–551, 2015.

[71] A. Orso, N. Shi, and M. J. Harrold, “Scaling regression testing to large software systems,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, p. 241, 2004.

[72] M. Al-Refai, “MaRTS : A Model-Based Regression Test Selection Approach,” 2017.

[73] W. El-hamid and S. El-etriby, “Regression test selection technique for multi-programming language,” *INFOS2010 2010 7th Int. Conf. Informatics Syst.*,

2010.

[74] S. Huang, Z. J. Li, J. Zhu, Y. Xiao, and W. Wang, “A novel approach to regression test selection for J2EE applications,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, no. 61003001, pp. 13–22, 2011.

[75] B. Srisura and A. Lawanna, “False test case selection: Improvement of regression testing approach,” *2016 13th Int. Conf. Electr. Eng. Comput. Telecommun. Inf. Technol.*, pp. 1–6, 2016.

[76] M. E. Ruth *et al.*, “Towards Automatic Regression Test Selection for Web Services,” *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf. - Vol. 02*, vol. 2, p. 1265, 2007.

[77] M. Huang, S. Technology, S. Guo, X. Liang, S. Technology, and X. Jiao, “Research on Regression Test Case Selection Based on Improved Genetic Algorithm,” pp. 256–259, 2013.

[78] T. Apiwattanapong, A. Orso, and M. J. Harrold, “A differencing algorithm for object-oriented programs,” *Proc. - 19th Int. Conf. Autom. Softw. Eng. ASE 2004*, pp. 2–13, 2004.

[79] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia, “Improving multi-objective test case selection by injecting diversity in genetic algorithms,” *IEEE Trans. Softw. Eng.*, vol. 41, no. 4, pp. 358–383, 2015.

[80] M. Al-refai, W. Cazzola, and S. Ghosh, “A Fuzzy Logic Based Approach for Model-based Regression Test Selection,” 2017.

[81] M. Kim *et al.*, “Efficient regression testing of ontology-driven systems,” *Proc. 2012 Int. Symp. Softw. Test. Anal. - ISSTA 2012*, p. 320, 2012.

[82] R. H. Rosero, O. S. Gomez, and G. Rodriguez, “An Approach for Regression Testing of Database Applications in Incremental Development Settings,” in *2017 6th International Conference on Software Process Improvement (CIMPS)*, 2017.

[83] E. Rogstad, L. Briand, and R. Torkar, “Test case selection for black-box regression testing of database applications,” *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1781–1795, 2013.

[84] M. Khattar, Y. Lamba, and A. Sureka, “SARATHI : Characterization Study on Regression Bugs and Identification of Regression Bug Inducing Changes : A Case-Study on Google Chromium Project Categories and Subject Descriptors,” *Proc. 8th India Softw. Eng. Conf. XXX - ISEC ’15*, pp.

50–59, 2015.

[85] L. Zhang, M. Kim, and S. Khurshid, “FaultTracer: A Change Impact and Regression Fault Analysis Tool for Evolving Java Programs,” *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, p. 40:1-40:4, 2012.

[86] B. G. Ryder and F. Tip, “Change Impact Analysis for Object-Oriented Programs,” *ACM SIGPLAN-SIGSOFT Work. Progr. Anal. Softw. tools Eng.*, pp. 46–53, 2001.

[87] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, “Chianti: A Tool for Change Impact Analysis of Java Programs,” *Proceeding OOPSLA ’04 Proc. 19th Annu. ACM SIGPLAN Conf. Object-oriented Program. Syst. Lang. Appl.*, vol. 39, no. 10, pp. 432–448, 2004.

[88] S. Ni and Y. Liu, “A progressive refinement approach for JAVA regression test selection,” *2009 WRI World Congr. Softw. Eng. WCSE 2009*, vol. 4, no. 60773105, pp. 170–174, 2009.

[89] A. Larprattanakul and T. Suwannasart, “An Approach for Regression Test Case Selection Using Object Dependency Graph,” *2013 5th Int. Conf. Intell. Netw. Collab. Syst.*, pp. 617–621, 2013.

[90] A. B. De Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, “Perphecy: Performance Regression Test Selection Made Simple but Effective,” *Proc. - 10th IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2017*, pp. 103–113, 2017.

[91] Z. Chen, H. F. Guo, and M. Song, “Improving regression test efficiency with an awareness of refactoring changes,” *Inf. Softw. Technol.*, vol. 103, no. July, pp. 174–187, 2018.

[92] Q. D. Soetens, S. Demeyer, and A. Zaidman, “Change-based test selection in the presence of developer tests,” *Proc. Eur. Conf. Softw. Maint. Reengineering, CSMR*, pp. 101–110, 2013.

[93] H. Aman, T. Nakano, H. Ogasawara, and M. Kawahara, “A Topic Model and Test History-Based Test Case Recommendation Method for Regression Testing,” *2018 IEEE Int. Conf. Softw. Testing, Verif. Valid. Work.*, pp. 392–397, 2018.

[94] M. Al-Refai, S. Ghosh, and W. Cazzola, “Model-Based Regression Test Selection for Validating Runtime Adaptation of Software Systems,” *Proc. - 2016 IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2016*, no. 3,

288–298, 2016.

[95] Y. Chen, R. R. L. R. R. L. Probert, and D. P. D. Sims, “Specification-based regression test selection with risk analysis,” *Proc. 2002 Conf. …*, p. 1, 2002.

[96] Q. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and A. Nadeem, “An approach for selective state machine based regression testing,” *Proc. 3rd Int. Work. Adv. Model. Test. - A-MOST ’07*, no. July, pp. 44–52, 2007.

[97] F. Ahmad and Z. H. Qaisar, “Scenario based functional regression testing using Petri net models,” *2013 12th Int. Conf. Mach. Learn. Appl. ICMLA 2013*, vol. 2, pp. 572–577, 2013.

[98] Q.-U.-A. Farooq, S. Lehnert, and M. Riebisch, “Analyzing model dependencies for rule-based regression test selection,” *Model. 2014*, vol. P225, pp. 305–320, 2014.

[99] R. K. Bhatia, S. Dahiya, and D. Rattan, “Regression test selection using class, sequence and activity diagrams,” *IET Softw.*, vol. 10, no. 3, pp. 72–80, 2016.

[100] L. Naslavsky, H. Ziv, and D. J. Richardson, “MbSRT2: Model-based selective regression testing with traceability,” *ICST 2010 - 3rd Int. Conf. Softw. Testing, Verif. Valid.*, pp. 89–98, 2010.

[101] Janhavi and A. Singh, “Efficient regression test selection and recommendation approach for component based software,” *Proc. 2014 Int. Conf. Adv. Comput. Commun. Informatics, ICACCI 2014*, pp. 1547–1553, 2014.

[102] Q. Luo, K. Moran, L. Zhang, and D. Poshyvanyk, “How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects,” *IEEE Trans. Softw. Eng.*, vol. X, no. X, pp. 559–570, 2018.

[103] M. Khatibsyarbini, M. A. Isa, D. N. A. Jawawi, and R. Tumeng, “Test case prioritization approaches in regression testing: A systematic literature review,” *Inf. Softw. Technol.*, vol. 93, pp. 74–93, 2018.

[104] A. Gambi, J. Bell, and A. Zeller, “Practical Test Dependency Detection,” *Proc. - 2018 IEEE 11th Int. Conf. Softw. Testing, Verif. Validation, ICST 2018*, pp. 1–11, 2018.

[105] “Stack Overflow Developer Survey 2018.” [Online]. Available: https://insights.stackoverflow.com/survey/2018/. [Accessed: 02-Jan-2019].

[106] B. Guo, M. Subramaniam, and P. Chundi, “Analysis of test clusters for regression testing,” *Proc. - IEEE 5th Int. Conf. Softw. Testing, Verif. Validation, ICST 2012*, p. 736, 2012.

[107] C. Science and S. Marcos, “Redroid : A Regression Test Selection Approach for Android Applications,” 2016.

[108] V. Channakeshava, S. Lakshmanan, A. Panigrahi, and V. Shanbhag, “ChiARTS - Safe subset-regression test selection for C#,” *4th IASTED Int. Conf. Adv. Comput. Sci. Technol. ACST 2008*, no. December, pp. 37–42, 2008.

[109] N. Chouhan, M. Dutta, and M. Singh, “A Code Analysis Base Regression Test Selection Technique for D Programming Language,” *2014 Int. Conf. Comput. Intell. Commun. Networks*, pp. 1106–1112, 2014.

[110] P. R. Srivastava, K. A. V, S. R. V, and R. G, “Regression Testing Techniques for Agent Oriented Software,” *2008 Int. Conf. Inf. Technol.*, pp. 221–225, 2008.

[111] S. Akimoto, R. Yaegashi, and T. Takagi, “Test Case Selection Technique for Regression Testing Using Differential Control Flow Graphs,” pp. 1–3, 2015.

[112] Z. Tóth, P. Gyimesi, and R. Ferenc, “A public bug database of GitHub projects and its application in bug prediction,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9789, pp. 625–638, 2016.

[113] S. Elbaum, G. Rothermel, and J. Penix, “Techniques for improving regression testing in continuous integration development environments,” *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. - FSE 2014*, pp. 235–245, 2014.

[114] G. Salton, A. Wong, C. Y.-C. of the ACM, and undefined 1975, “A vector space model for automatic indexing,” *dl.acm.org*.

[115] Wael H. Gomaa and Aly A. Fahmy, “A Survey of Text Similarity Approaches,” *Int. J. Comput. Appl.*, vol. 68, no. 13, pp. 13–18, 2013.

[116] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering*. 2012.

[117] R. K. Yin, *Case Study Research. Design and Methods.*, vol. 5, no. 5. 2009.

[118] P. Runeson and M. Höst, “Guidelines for conducting and reporting case

study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2009.

[119] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for Java (demo)," *Proc. 25th Int. Symp. Softw. Test. Anal. - ISSTA 2016*, pp. 449–452, 2016.

[120] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," *Proc. 27th Int. Conf. Softw. Eng. - ICSE '05*, p. 402, 2005.

[121] J. Weston, S. Chopra, and A. Bordes, "Memory Networks," pp. 1–15, 2014.

[122] M. Harman, "Making the case for MORTO: Multi objective regression test optimization," *Proc. - 4th IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2011*, no. March 2011, pp. 111–114, 2011.

[123] A. Choudhary, A. P. Agrawal, and A. Kaur, "An Effective Approach for Regression Test Case Selection using Pareto based Multi-Objective Harmony Search," in *1th International Workshop on Search-Based Software Testing*, 2018, pp. 1–8.

[124] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "CBGA-ES: A Cluster-Based Genetic Algorithm with Elitist Selection for Supporting Multi-Objective Test Optimization," *Proc. - 10th IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2017*, pp. 367–378, 2017.

[125] Q. Gu, B. Tang, and D. Chen, "Optimal regression testing based on selective coverage of test requirements," *Proc. - Int. Symp. Parallel Distrib. Process. with Appl. ISPA 2010*, pp. 419–426, 2010.

APPENDICES

Appendix 1 - Intermediate Conceptual Model Figures

In Figure A.6.1, the full conceptual model is presented. In order to make conceptual model easier to read, color codes and line styles are used extensively. Hybrid approaches are linked with green lines between approaches to identify which approaches are used. However, the figure can be difficult to read. To alleviate any misconceptions about the model, two simplified versions are also presented with only code based approaches, and with only non-code based approaches in Figure A.6.2 and Figure A.6.3 respectively. The links of evaluation metrics are not drawn to reduce the complexity of the conceptual model.

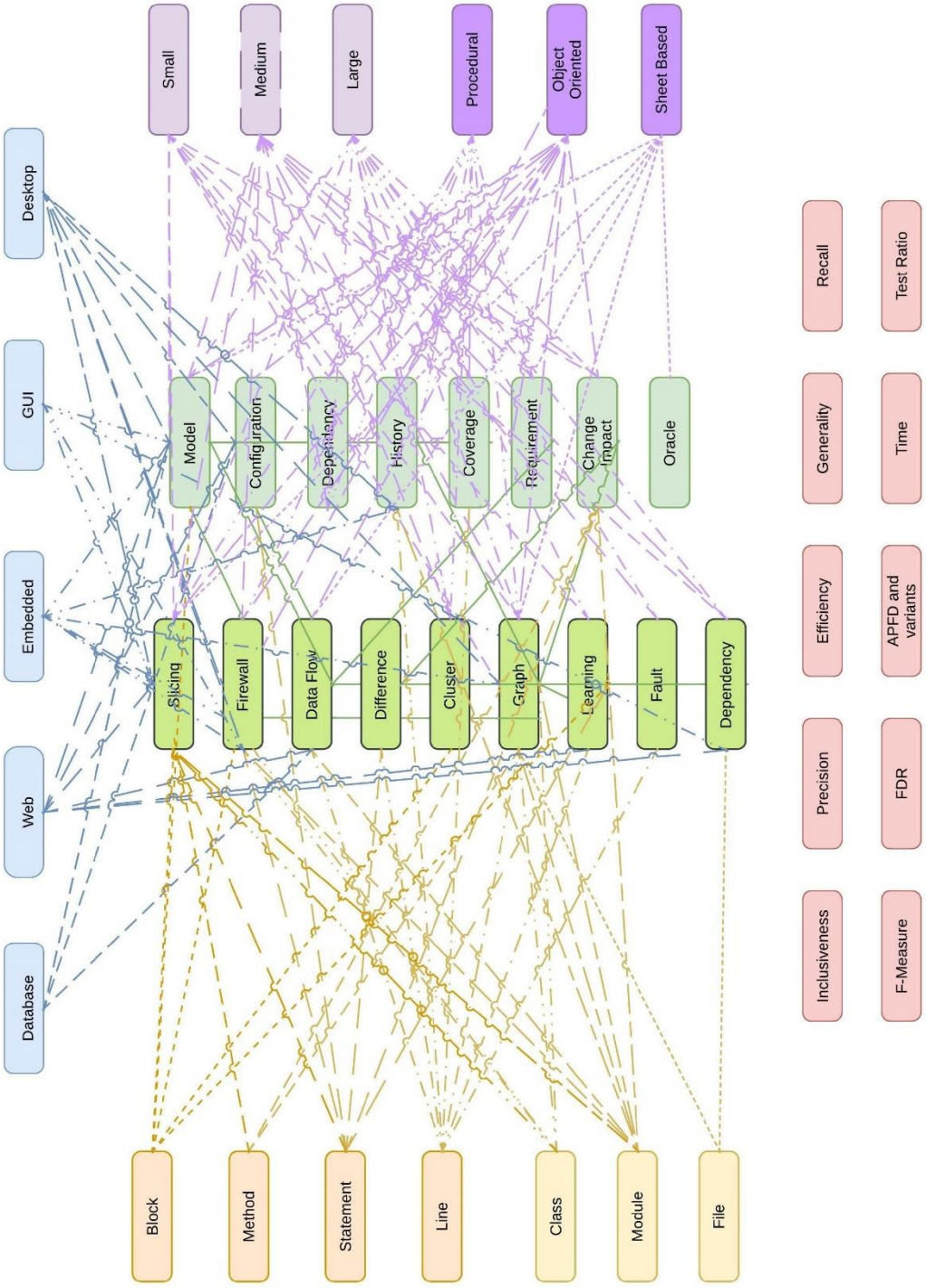


Figure A.6.1. The Conceptual Model

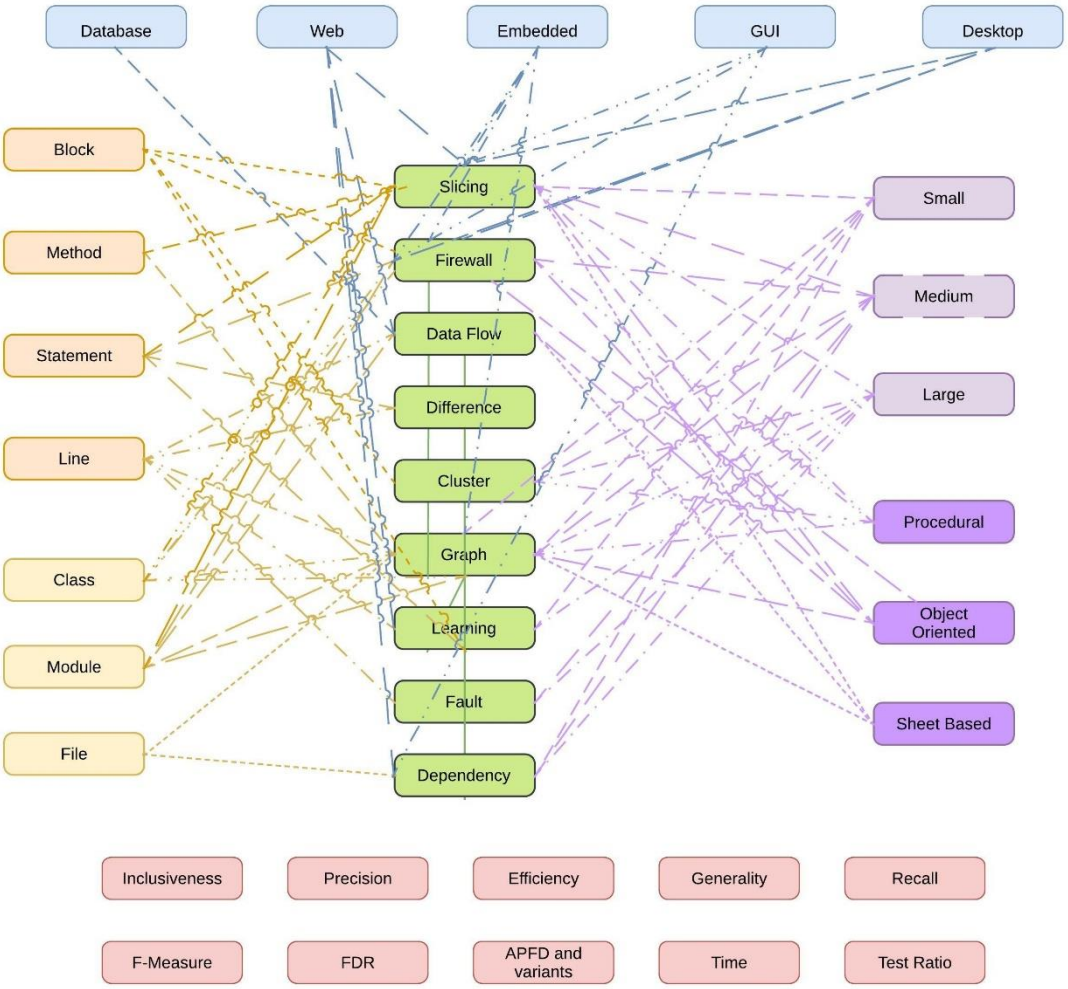


Figure A.6.2. Conceptual model representing only code-based approaches

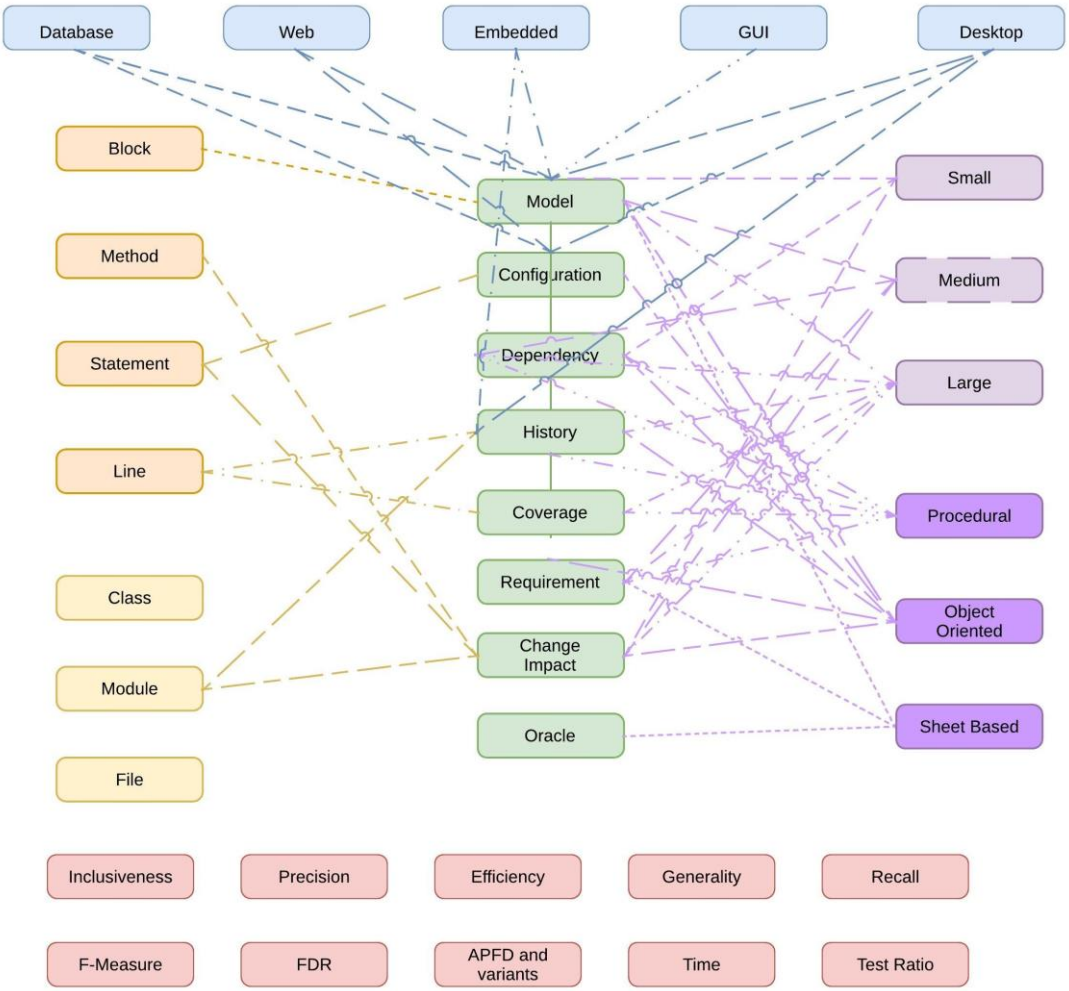


Figure A.6.3. Conceptual model representing only non-code based approaches

Appendix 2 - Papers driven from thesis

U. Yilmaz and A. Tarhan, “A Case Study to Compare Regression Test Selection Techniques on Open-Source Software Projects,” in *Proceedings of the 12th Turkish National Software Engineering Symposium*, 2018.

HACETTEPE UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING
THESIS/DISSERTATION ORIGINALITY REPORT

HACETTEPE UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING
TO THE DEPARTMENT OF COMPUTER ENGINEERING

Date: 18/02/2019

Thesis Title / Topic: A METHOD FOR SELECTING REGRESSION TEST CASES BASED ON SOFTWARE CHANGES AND SOFTWARE FAULTS

According to the originality report obtained by my thesis advisor by using the *Turnitin* plagiarism detection software and by applying the filtering options stated below on 18/02/2019 for the total of 79 pages including the a) Title Page, b) Introduction, c) Main Chapters, d) Conclusion sections of my thesis entitled as above, the similarity index of my thesis is 6 %.

Filtering options applied:

1. Bibliography/Works Cited excluded
2. Quotes excluded / ~~included~~
3. Match size up to 5 words excluded

I declare that I have carefully read Hacettepe University Graduate School of Science and Engineering Guidelines for Obtaining and Using Thesis Originality Reports; that according to the maximum similarity index values specified in the Guidelines, my thesis does not include any form of plagiarism; that in any future detection of possible infringement of the regulations I accept all legal responsibility; and that all the information I have provided is correct to the best of my knowledge.

I respectfully submit this for approval.

Date and Signature
18/02/2019

Name Surname: Uğur YILMAZ
Student No: N14327492
Department: Computer Engineering
Program: Computer Engineering MSc. with Thesis
Status: ☒ Masters ☐ Ph.D. ☐ Integrated Ph.D.

ADVISOR APPROVAL

APPROVED.

Assist. Prof. Dr. Ayça TARHAN
(Title, Name Surname, Signature)

CURRICULUM VITAE

Name Surname : Uğur YILMAZ

Place of Birth : Ankara

Date of Birth : 28/04/1991

Marital Status : Married

Correspondence Address : Çiğdem Mah. 1570. Cd. 8B/42 Çankaya/Ankara

Phone : 0505 112 5794

E-mail Address : uguryilmaz@aselsan.com.tr

EDUCATIONAL BACKGROUND

Bachelor's Degree :

Middle East Technical University, Ankara, Turkey 2009 – 2014

B. Sc. in Electrical and Electronics Engineering CGPA: 3.25 / 4.00

Master's Degree :

Hacettepe University, Ankara, Turkey, 2015 – 2019

M. Sc. in Computer Engineering CGPA: 3.58 / 4.00

WORK EXPERIENCE

Aselsan Inc., Ankara, Turkey 2014 – Present

Software Engineer in Test

Aselsan Inc., Ankara, Turkey 2013 – 2013

Intern

Karel Electronics Inc., Ankara, Turkey 2012 – 2012

Intern

PUBLICATIONS

V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz, "What we know about testing embedded software", IEEE Software, 2018.

U. Yilmaz and A. Tarhan, "A Case Study to Compare Regression Test Selection Techniques on Open-Source Software Projects," in *Proceedings of the 12th Turkish National Software Engineering Symposium*, 2018.