UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

EDUARDO NORONHA DE ANDRADE FREITAS

SCOUT: A Multi-objective Method to Select Components in Designing Unit Testing

Goiânia
2016

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE TESE EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: SCOUT: A Multi-objective Method to Select Components in Designing Unit Testing

Autor: Eduardo Noronha de Andrade Freitas

Goiânia, 15 de Fevereiro de 2016.

Eduardo Noronha de Andrade Freitas – Autor

Dr. Auri Marcelo Rizzo Vincenzi – Orientador

Dr. Celso Gonçalves Camilo Júnior – Co-Orientador

EDUARDO NORONHA DE ANDRADE FREITAS

SCOUT: A Multi-objective Method to Select Components in Designing Unit Testing

Trabalho apresentado ao Programa de Pós–Graduação em Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação.

Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi

Co-Orientador: Prof. Dr. Celso Gonçalves Camilo Júnior

Goiânia
2016

EDUARDO NORONHA DE ANDRADE FREITAS

SCOUT: A Multi-objective Method to Select Components in Designing Unit Testing

Tese defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Doutor em Ciência da Computação, aprovada em 15 de Fevereiro de 2016, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Auri Marcelo Rizzo Vincenzi
Universidade Federal de Goiás – UFG e
Universidade Federal de São Carlos – UFSCAR
Presidente da Banca Examinadora

Prof. Dr. Celso Gonçalves Camilo Júnior
Universidade Federal de Goiás – UFG

Prof. Dr. Fabiano Cutigi Ferrari
Universidade Federal de São Carlos – UFSCAR

Prof. Dr. Arilo Cláudio Dias Neto
Universidade Federal do Amazonas – UFAM

Prof. Dr. Plínio de Sá Leitão Júnior
Universidade Federal de Goiás – UFG

Prof. Dr. Cássio Leonardo Rodrigues
Universidade Federal de Goiás – UFG

Eduardo Noronha de Andrade Freitas

Eduardo Noronha Andrade Freitas received his degree in Computer Science from the Instituto Unificado de Ensino Superior (IUESO) in 2000; his specialization in Software Quality in 2003, his master's degree in Electrical and Computer Engineering in 2006, and his Ph.D. in Computer Science from the Universidade Federal de Goiás in 2016. From 2013 to 2015, during his Ph.D. studies, he collaborated in the Checkdroid startup (www.checkdroid.com) at the Georgia Institute of Technology in Atlanta, GA. He served as Information Technology Manager at the Secretariat of Public Security of the State of Goiás from 2006 to 2010, participating in the development and implementation of strategic processes. He also developed numerous strategic planning projects and data analysis in the public and private sectors in diverse areas: health, education, security, sports, politics, and religion. Since 2010, he has served as a professor at the Instituto Federal de Goiás (IFG). He has extensive experience in computer science with a focus on computer systems, principally in the following areas: systems development, software engineering with an emphasis on search-based software engineering, Android testing, multiagent systems, strategic management of technology, and computational intelligence. He can be reached at eduardonaf@gmail.com.

To my mother, Gislene, for her noble character, subservience, and indescribable determination.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Prof. Dr. Auri, for his continuous support throughout the course of my thesis, for his patience, humility, motivation, and immense knowledge. His guidance helped me considerably in the research and writing of this thesis.

I would also like to thank my co-advisor, Prof. Dr. Celso, for introducing me to this exciting research topic, for assisting me with timely feedback, practical working structures, and for providing me useful information and encouragement.

In addition, I would like to thank the rest of my thesis committee: Prof. Dr. Arilo Cláudio Dias Neto, Prof. Dr. Fabiano Cutigi Ferrari, Prof. Dr. Plínio de Sá Leitão Júnior, and Prof. Dr. Cássio Leonardo Rodrigues, for their insightful comments, questions, and encouragement.

Thanks, as well, to the professors and staff and my Ph.D. colleagues at the Universidade Federal de Goiás (UFG), my colleagues at the Instituto Federal de Goiás (IFG), and Prof. Dr. Nei Yoshiriro Soma at the Instituto Tecnológico de Aeronáutica (ITA).

I also wish to convey my gratitude to several institutions: OOBJ company and its founder Jonathas Carrijo for their invaluable assistance in sharing data, systems, and workers for the development of experimental studies and Checkdroid for the opportunity to collaborate in a challenging and stimulating research environment. I am indebted to CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) and to FAPEG (Fundação de Amparo à Pesquisa do Estado de Goiás) for their financial support, and to IFG for the paid leave, which enabled me to devote myself entirely to my doctorate.

I would also like to thank my friends who made my thesis possible and an unforgettable experience. First, I thank Kenyo for being the first to encourage me to pursue my Ph.D. and for introducing me to Prof. Auri. I thank Laerte Campos (in memoriam) for his generous, fun-filled, and consistent guidance during countless conversations and project implementations. Thanks also to my beloved friends Edson Ramos and Thiago Campos for sharing their lives with me and to my great friend Jean Chagas for investing in and supporting me spiritually throughout my life. I acknowledge and appreciate the impact of

his life on my journey. I am grateful to the brothers of my last discipleship group, Osmar, Mayko, and Léo, and to Jeuel Alves for sharing thoughts and encouragement.

I would also like to thank Dr. Alessandro Orso for allowing me to participate in his research group (ARKTOS) at the Georgia Institute of Technology and colleagues, in particular, Dr. Shauvik Roy Chouldhary for sharing with me not only his office, but his friendship and extensive knowledge.

Thanks to the dear friends and families who made my time in Atlanta so special: Emerson Patriota, Aster, Tubal, Alan Del Ciel, Dr. Monte Starkes, Bryan Brown, Charles Hooper Jr, Tony Heringer, and to my dear friend Josh in Slidel, LA. Thanks also to the 15 families who showed their love by visiting us in Atlanta. Each one was like a new breath.

Special recognition goes to my family, for their support, patience, and encouragement, during my pursuit of higher levels of education, above all, to my lovely, precious wife, Leticia, who understood her vocation, supporting and encouraging me at all moments with wisdom and caring. It would be much easier to earn a Ph.D. than to find a word that could adequately express my deepest, heartfelt gratitude for her love. To my little boys, Davi and Pedro, who are my greatest friends, and who many times became my extra motivation when the "burden" was heavy, I know you guys will go far! You rock! I would also like to convey my heartfelt dedication to my beloved parents, Eduardo and Gislene, for their lifelong encouragement and support. I know what they faced to make this moment possible, and I will never forget their love. Thanks too to my dear sisters, Kelly, Karlla, and Karen, and brother, Ricardo, for their friendship. I also express my gratitude to my dear parents-in-law, Edilberto and Cida, for their love and support.

First and foremost, I dedicate this work and all the work required to arrive here, in honor of my Lord Jesus Christ, who gave me a new life and called me to follow Him until the last day. He is the Alpha and the Omega, the Beginning and the End, the First and the Last. Thank You, Jesus!

"And I gave my heart to seek and search out by wisdom concerning all things that are done under heaven: this sore travail hath God given to the sons of man to be exercised therewith."

Solomon,
Ecclesiastes 1:13.

Abstract

Freitas, Eduardo Noronha de Andrade. **SCOUT: A Multi-objective Method to Select Components in Designing Unit Testing.**. Goiânia, 2016. 82p. PhD. Thesis
Instituto de Informática, Universidade Federal de Goiás.

The creation of a suite of unit testing is preceded by the selection of which components (code units) should be tested. This selection is a significant challenge, usually made based on the team member's experience or guided by defect prediction or fault localization models. We modeled the selection of components for unit testing with limited resources as a multi-objective problem, addressing two different objectives: maximizing benefits and minimizing cost. To measure the benefit of a component, we made use of important metrics from static analysis (cost of future maintenance), dynamic analysis (risk of fault, and frequency of calls), and business value. We tackled gaps and challenges in the literature to formulate an effective method, the Selector of Software Components for Unit testing (SCOUT). SCOUT was structured in two stages: an automated extraction of all necessary data and a multi-objective optimization process. The Android platform was chosen to perform our experiments, and nine leading open-source applications were used as our subjects. SCOUT was compared with two of the most frequently used strategies in terms of efficacy. We also compared the effectiveness and efficiency of seven algorithms in solving a multi-objective component selection problem: random technique; constructivist heuristic; Gurobi, a commercial tool; genetic algorithm; SPEA_II; NSGA_II; and NSGA_III. The results indicate the benefits of using multi-objective evolutionary approaches such as NSGA_II and demonstrate that SCOUT has a significant potential to reduce market vulnerability. To the best of our knowledge, SCOUT is the first method to assist software testing managers in selecting components at the method level for the development of unit testing in an automated way based on a multi-objective approach, exploring static and dynamic metrics and business value.

Keywords

software testing, unit testing, component selection, Search Based Software Testing (SBST), multiobjective optmization.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

An essential process for software testing is selecting the components to be tested. However, in practice this process has been driven by empiricism on the part of software engineers and by techniques and strategies that were not specifically formulated for this purpose.

This thesis, which lies within the the sub-set of Software-Based Engineering (SBSE) known as Search-Based Software Testing (SBST), proposes an enhanced method to assist professionals in the selection process. To the best of our knowledge, it constitutes original work as no analogous research was found in the literature review.

1.1 Motivation

Among software-engineering activities, verification and validation are the practices most commonly used in software testing and the most expensive, representing more than half the total cost of a project (MYERS, 1979).

Several techniques have been applied to improve software quality. Among them, the most frequently used has been software testing. Despite considerable testing effort delivered through research and tools, automating testing activities remain a major challenge. In automating testing activities, two questions arise: "what to test?" and "how to test?". Much effort has been given by academia and industry to address the first query, but a gap remains in responding to the second, particularly in regards to unit testing.

In his book *Succeeding with Agile* Cohn (2010), Mike Cohn advocates the precedence of unit testing over functional testing in his test automation pyramid, which is divided into three levels according to Figure 1.1.

With many available resources that can be used Fowler (2012), the test pyramid, which depicts test emphasis, proposes focusing on unit as opposed to user interface (UI) testing as unit testing is easier to maintain compared to UI end-to-end testing. According to Fowler, UI tests that run end to end are brittle, expensive to write, and time consuming to run. Accordingly, the pyramid argues that one should do much more automated testing through unit tests than through traditional UI-based testing. UI testing

specifications, which are largely non-formal, may be incomplete or ambiguous as will be the test suite derived from them. UI testing also overlooks important functional properties of the programs that are part of its design or implementation and which are not described in the requirements (HOWDEN, 1980).

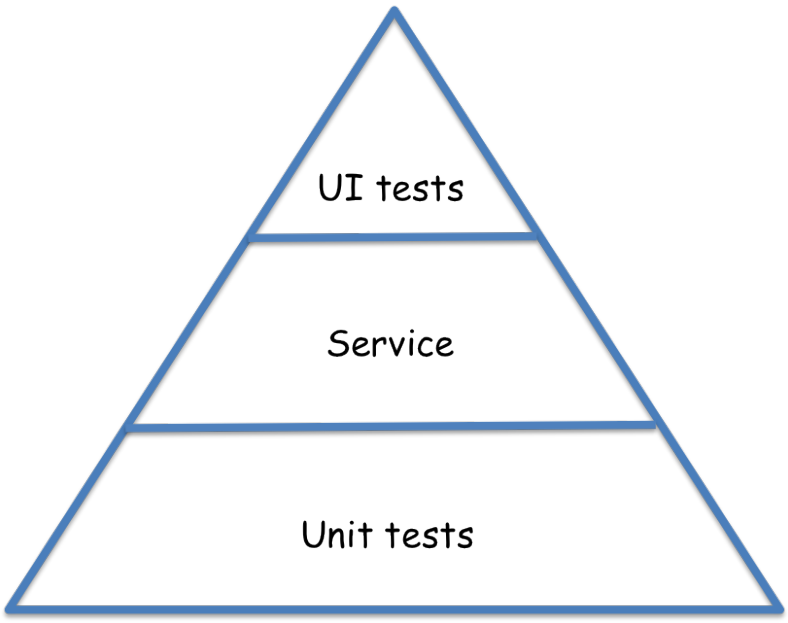

Figure 1.1: *Levels for test automation (COHN, 2010).*

We argue that a bug revealed by a UI test will likely reveal a bug in a unit code or a fault in an intermediate service. By way of example, UI testing, via the Android platform used in our experiments, indicates key drawbacks such as lack of standardization in mobile test infrastructure, scripting languages, and connectivity protocols between mobile test tools and platforms and the lack of a unified test automation infrastructure and solutions that cross platforms and browsers on most mobile devices (GAO et al., 2014). Most tool-testing initiatives using Android involve UI testing. There are a set of frameworks and APIs to assist in the development of UI testing for Android apps, such as UIAutomator API (GOOGLE, 2015), and Espresso API (Espresso, 2015). There are also tools to generate UI testing inputs and to support test case generation, including oracles, as presented in the Chapter 2.

A unit test is simply a method without parameters that performs a sequence of method calls that exercise the code under test and asserts properties of the code's expected behavior (TILLMANN; HALLEUX; XIE, 2010). Ideally the unit testing should be written prior to the code, as done in both methodologies Acceptance Test Driven Development (PUGH, 2010) and in Test Driven Development (TDD) (ASTELS, 2003). In these kind of methodologies, the development is preceded by the creation of unit tests, making the whole system or the most part be covered by unit tests.

Unfortunately, many companies in the software industry own systems devoid of any testing artifacts. On the other hand, the demand for higher quality software has been increasing, indicating the need for increased investment in testing activities in the same proportion. Accordingly, some companies have tried to introduce testing activities incrementally in their processes.

The development of unit testing in this context is a special challenge for professionals charged with the demanding task of deciding which components to select for testing in the limited time available to complete it. In this regard, the development and application of unit testing to the entire system with extensive coverage may be impractical. The identification of components relevant to the system is crucial, especially in legacy systems, large systems, and systems with high maintenance levels.

According to many studies, the incorporation of constraints can change significantly the subset of selected components for unit testing. When these constraints are considered, the problem can be seen as a combinatorial problem. For this case the algorithms used to solve this kind of problem are penalized by high dimensionality. Therefore, the process of selection should consider variables about components, and about the feasibility of the application of tests, and also the existent constraints about the time availability.

In many interviews with practising software testers and developers, we asked which criteria did they use to select components for unit testing. The response concentrates in three group of testers and developers: those who do this selection based on their own experience and technical intuition; those who select the components based on static metrics, such as cyclomatic complexity, lines of code; and those who use a prediction fault model to guide their selection. As an example of the second group, we can mention IBM's Rational Test RealTime software (version 8.0.0). In its online documentation, the selection of components for unit testing is guided by static metrics as follows: "As part of the Component Testing wizard, Test RealTime provides static testability metrics to help you pinpoint the critical components of your application. You can use these static metrics to prioritize your test efforts." (IBM, 2016). In our initial systematic review, we found works related to the criteria used by the third group. For example, we highlight the paper entitled "Using Static Analysis to Determine Where to Focus Dynamic Testing Effort" (WEYUKER; OSTRAND; BELL, 2004), where the authors state the following as their motivation: "Therefore, we want to determine which files in the system are most likely to contain the largest numbers of faults that lead to failures and prioritize our testing effort accordingly." In the systematic review entitled "Reducing test effort: A systematic mapping study on existing approaches," the authors investigate the identification of current approaches able to reduce testing effort. Among them, they confirm the use of predicting defect-prone parts or defective content to focus the testing effort. (Further detail regarding these works and others can be found in Chapter 3

To illustrate the complexity and importance of selecting components for unit testing, consider the Android ecosystem we used to validate the Selector of Software Components for Unit Testing (SCOUT). The worldwide smart-phone market is growing annually, with 341.5 million shipments in the second quarter of 2015, according to data from the International Data Corporation (IDC) (IDC, 2016). Android still dominates the smartphone market with 82.8% as shown in Table 1.1, with a proliferation of brands, generating more than 24,000 different devices, , four generalized screen sizes (small, normal, large, and extra-large), six generalized densities (ldpi, mdpi, hdpi, xhdpi, xxhdpi, and xxxhdpi), presenting a significant challenge for developers and testers: device fragmentation. In addition to the high number of devices, with distinct settings (screen size, memory, functions), the operating system itself is extremely fragmented with more than 20 different APIs at the time this thesis was written.

Table 1.1: *Smartphone OS Market Share.*

Period	Android	iOS	Windows Phone	BlackBerry OS	Others
2015Q2	82.80%	13.90%	2.60%	0.30%	0.40%
2014Q2	84.80%	11.60%	2.50%	0.50%	0.70%
2013Q2	79.80%	12.90%	3.40%	2.80%	1.20%
2012Q2	69.30%	16.60%	3.10%	4.90%	6.10%

This positive moment in the Android market with 1.4 billion users (DMR, 2016) has also leveraged growth in the number of related apps to 1.8 million (STATISTA, 2016) in November 2015, as shown in Figure 1.2.

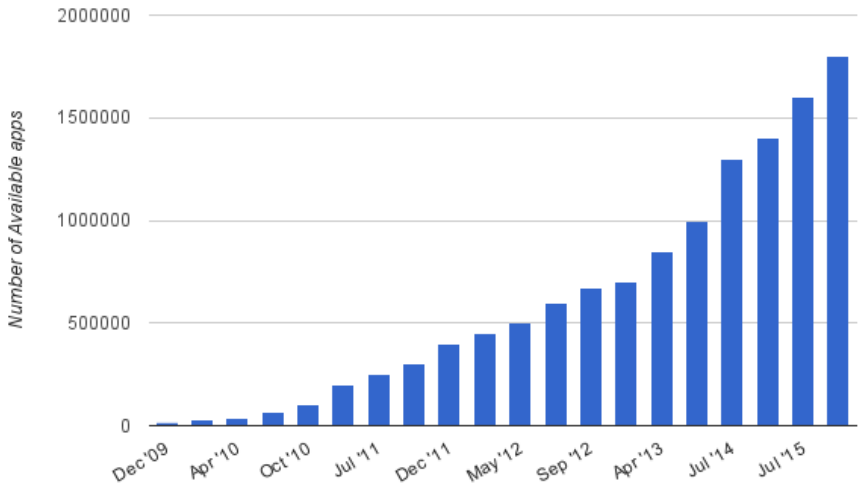

Figure 1.2: *Number of downloaded Android apps.*

Such diversity comes with many challenges for developers and testers alike, particularly in regard to software quality. Delivering a faulty application in this dynamic environment can have a highly negative effect. One way to avoid this is to ensure the

quality of these apps by applying effective software testing techniques, especially as they pertain to the choice of which subset of components to test before the next release.

Much of current software engineering practice and research is done in a value-neutral setting, in which every requirement, use case, object, test case, and defect is equally important (BOEHM, 2006). Sometimes, in Software Testing it is not an exception, and we bring the reflection of: once software testing has as the main goal reveals errors/bugs, do these bugs have the same strategic importance when we think in terms of both technically and the value to the business by which the software was designed?

Motivated to answer this question, we elaborated a multiobjective model (SCOUT) that could incorporate under consideration important variables for the components selection problem, as we detail in the Chapter 4.

As we used the Android platform to validate the method we propose, we realize that generally, in practice many Android developers face some situations as follows:

1. There is a well defined Android market share including more than 24k devices with distinct configurations;
2. There are apps already available at Google Play store;
3. These apps do not have a suite of unit test cases yet;
4. They know they need to increase the software quality minimizing associated risks;
5. Each component has its own strategic importance;
6. The developers and testers understand they should start form Unit Testing;
7. Each component consumes a time to be cover by tests;
8. The available time until the next release for testing activities is less than the sum of the required time to test all components.

Of course that to develop this research we had to provide all data and technologies in order to have the expected results. The possibility to check the impact on real industry environment is considered a plus by researchers at the SBSE area. This is justified because of the wealth of existing details for the possibility of establishing comparative to new research, and the real evaluation of the effectiveness of the research.

We have not identified in the literature any work to assist both developers and testers to select a subset of components for unit testing given a coming deadline, and in a multi-objective approach. Considering tight deadlines, the component selection process can be seen as a optimization problem, suggesting the investigation of Search Based Software Engineering (SBSE) techniques (HARMAN; JONES, 2001) in this context.

Given these findings, and also the challenge of combining static and dynamic metrics, and Android market information to guide this selection, we developed our research.

1.2 Objectives

Based on the motivations presented previously, our main objective in this research is to elaborate a method to select components for Android unit testing. As objectives we have:
1. Model the Component Selection Problem (CSP) for Unit Testing as a multiobjective problem;
2. Investigate the use of both static and dynamic metrics, and also Android market information in a component selection process;
3. Evaluate the performance of a multiobjective model over the methods used in the literature to select components;
4. Investigate the use of Search Based Software Testing (SBST) techniques to solve a CSP;
5. Make a comparison among different solvers in terms of their efficiency and efficacy when applied to solve a CSP;

1.3 Research Methodology

In our research we have adopted the quantitative research method to systematically and empirically investigate the component selection problem for unit testing. Particularly, software testers suffers from insufficient deadlines for the development of unit testing, and by the absence of valuation criteria that allow them differentiate and value the components for the selection process.

The hyphotesis that the selection of components should be multi-objective was tested. Additionally, we developed our research:

- Identifying important objectives for CSP;
- Modeling a multi-objective CSP;
- Identifying and comparing strategies usually used for CSP;
- Identifying and comparing solvers for CSP;
- Carrying out empirical studies on Android platform in order to answer the following research questions:

RQ1 - Which solver is more appropriated to be used in a scenario where benefit and cost have the same strategic importance for the specialist?

RQ2 - What is the impact of using SCOUT in scenarios of different priorities? In contexts:

[**RQ2.1**] - where benefit and cost have the same strategic importance for the specialist.

[**RQ2.2**] - where the specialist prioritizes a high quality of the product instead of a low cost testing strategy.

[**RQ2.3**] - which requires low cost for testing.

RQ3 - What is the efficacy of SCOUT in selecting more important components in terms of their market relevance?

The research questions are answered by the empirical studies based on the quantitative data and analysis of the result.

Despite the main goal of this work is to develop a general method in such way it can be applied in different contexts and platforms, we choose Android platform to validate SCOUT once Android ecosystem own complex and dynamic features as stated in the Section 1.1. The empirical studies on nine different Android apps are conducted on seven solvers.

1.4 Contributions

The results show that SCOUT is an effective method to address the difficulty of selecting components for Android unit testing. In summary, the main contributions of this work are:
(1) A novel multiobjective method that considers important variables for optimizing the selection of components for Android unit testing;
(2) A comparison analysis of both efficacy and efficiency among three strategies and seven solvers to address the problem;
(3) A compiled database containing metrics and algorithms to replicate the experiments done in this research, and also to be a novel benchmark for the problem of components selection for Unit testing;
(4) A strategy for reducing the numbers of devices to test the market vulnerability, based on Orthogonal Array Technique.

In addition, as stated in the recommendation letter Appendix A, as result of this collaboration at Checkdroid/Gerogia Tech we had:
(1) An initial prototype of Capture/Replay tool called Android Mirror Tool (AMT) generating Input Tests written in Espresso API (FREITAS, 2015);
(2) A tool for generating automated UI test cases in Espresso API called Barista (CHOUDHARY, 2015a);

1.5 Publications and Experiences

(1) A paper entitled "A Parallel Genetic Algorithm to Coevolution of the Strategic Evolutionary Parameters" published in the International Conference on Artificial Inteligence (ICAI'13), Las vegas/USA.
(2) A paper entitled "Prioritization of Artifacts for Unit Testing Using Genetic Algorithm Multi-objective Non Pareto" published in the International Conference on Software Engineering Research and Practice (SERP'14), Las vegas/USA.
(3) A paper entitled "Android apps: Reducing Market Vulnerability by Selecting Strategically Units for Testing" submited to IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC/2016), Atlanta/USA.
(4) A paper entitled "Barista: Generation and Execution of Android Tests Made Easy" submited to International Symposium on Software Testing and Analysis (ISSTA/2016), Saarbrücken, Germany.
(5)

During the PhD, I visited the Georgia Tech Institute of Technology (2014 and 2015), working under the supervision of Dr. Alessandro Orso, and also worked at Checkdroid company (CHOUDHARY, 2015b) close to Dr. Shauvik Roy Choudhary who is the Checkdroid founder (Appendix A).

1.6 Thesis Organization

As introduced by this chapter the motivation, objectives, and main contributions of this thesis, the rest of this thesis is organized according to describe in the next paragraphs.

Chapter 2 presents the basic terminology, software testing concepts, a description of Component Selection Problem (CSP), and the field of Search Based Software Testing (SBST).

Chapter 3 summarizes the related works found in the literature, and it presents a discussion of gaps and opportunities for research in this subject field.

In **Chapter 4**, we present in detail the formulation of our method for selecting Android components for unit testing.

In **Chapter 5**, we present the strategy of experimentation to confirm our hypothesis, the baselines, the subjects, and a detailed research questions analysis. Also, we list some threads to validity.

Lastly, in the **Chapter 6** are presented the general conclusions and pointed out some possible future works.

CHAPTER 2

Concepts

This Chapter describes basic concepts to understanding the remaining of this thesis. First, in the Section 2.1 we brief some phases and techniques of software testing, and some challenges to automate them on Android platform. Next, in the Section 2.2 we detail the Component Selection Problem (CSP) and its formulation, and in the Section 2.3 we introduce the Search Based Software Testing (SBST).

2.1 Software Testing

The requirements for higher quality software are increasing in the modern life where systems have given support since basic human routines until complex process. It has motivated the development of software testing activities whose initial idea is probably due to Turing (TURING, 1989) who suggested the use of manually constructed assertions (HARMAN; JIA; ZHANG, 2015). According to (MYERS, 1979) software testing is the process of executing a program with the intent of finding errors. Myers affirm that we should focus on breaking the software instead of confirming that it works. Because testing is a sadistic process of breaking things. It is a destructive process. Moreover, a set of activities such as Verification, Validation and Test (VVT) have been practiced aiming to minimize the incidence of errors and its associated risks (DELAMARO et al., 2007). These activities must be develop throughout the software development process, and in general, they are grouped in different phases or levels of testing as described in the next section.

2.1.1 Levels or Phases of Testing

In the context of procedural software, the software development is done in an incremental way demanding the parallel development of software testing activities to ensure product quality for the user. Thereby, testing activities can be divided into four incremental phases: unit, integration, system, and acceptance testing (PRESSMAN, 2005).

The **Unit Testing** is focused in the smallest piece of code in a system. It searches for finding both logic and implementation errors in each software module, separately, to ensure that their algorithmic aspects are correctly implemented. Due to the presence of dependency among units, in this phase is common the need to develop drivers and stubs. Considering a unit under test as *u*, a stub is a unit that replaces another unit used (called) by *u* during unit testing. Usually, a stub is a unit that simulates the behavior of the used unit with minimum computation effort or data manipulation.

A high overhead to unit testing may be represented by the development of drivers and stubs. There are a large number of "xUnit" frameworks for different programming languages, such as JUnit (JUNIT, 2010). They may provide a test driver for the *u* with the advantage of also providing additional facilities for automating the test execution.

Once the desirable units were separately tested, how can we ensure that they will work adequately together? The target of the **Integration Testing** is to answer this question. A unit may suffer from the adverse influence of another unit. Sub-functions, when combined, may produce unexpected results and global data structures may raise problems.

The **System Testing** is responsible for ensuring that the software and the other elements that are part of the system (hardware and database, for instance) are adequately combined and adequate function and performance are obtained. In **Acceptance testing** is used to check whether the product meets the user's expectations.

All of these previous kind of tests are run during the software development process. However, once new requirements for change come from the users, the required change in the software after its release demands some tests to be rerun to make sure the changes did not introduce any collateral effect in the previous working functionalities. This kind of testing is called **Regression Testing**.

The focus of this thesis is to present a method to assist testers in selecting properly components for unit testing, when there is no enough time to test all of them. We choose to focus on this testing phase due to the reasons presented in our motivation (see Section 1.1).

2.1.2 Testing Techniques

As stated by (MYERS, 1979) one of the most difficult questions to answer when testing a program is determining when to stop, since there is no way of knowing if the error just detected is the last remaining error. In general, it is impractical, often impossible, to find all the errors in a program. Since then, many techniques have been proposed in the literature.

According to (HOWDEN, 1987) testing can be classified in two distinct ways: specification-based testing, and program specification. Based on this, there are three kind of testing techniques: Functional testing, structural testing, and Fault-Based Testing.

Functional or Black-box Testing

Functional or black-box testing is a testing technique based in specification and the goal is to determine whether the requirements (functional or non functional) have been satisfied. It is so named because the software is handled as a box with unknown content, only the external side is visible. A program is considered to be a function and is thought of in terms of input values and corresponding output values. In **Functional Testing** the internal structure of a program is ignored during test data selection. Tests are constructed from the functional properties of the program that are specified in the program's requirements (HOWDEN, 1980). Examples of such criteria are equivalence partition, boundary value, cause-effect graph, and category-partition method (VINCENZI et al., 2010).

Structural Testing

Also known as white box (as opposed to black box) is a testing technique based on program specification. It takes into consideration implementation or structural aspects in order to determine testing requirements. According to (VINCENZI et al., 2010) a common approach to applying structural testing is to abstract the Software Under Test (SUT) using a representation from where required elements are extracted by the testing criteria. For instance, for unit testing, each unit is abstracted as a Control Flow Graph – CFG (also called Program Graph) to represent the SUT. A product P represented by a CFG has a correspondence between the nodes of the graph and blocks of code, and between the edges of the graph and possible control-flow transfers between two blocks of code. It is possible to select elements from the CFG to be exercised during testing, thus characterizing structural testing. For integration testing a different kind of graph is used, and so on. The first structural criteria were based exclusively on control-flow structures. The best known are All-Nodes, All-Edges, and All-Paths (MYERS et al., 2004).

Fault-Based Techniques

Fault-Based techniques use information on the most common mistakes made in the software development process and on the specific types of defects we want to reveal (DEMILLO, 1987). Two criteria that typically concentrate on faults are the error seeding and mutation testing.

Table 2.1: *Number of variables to reveal a fault in the software (WALLACE; KUHN, 2001).*

Variables	Medical Devices	Browser	Server	NASA GSFC	Network Security	TCAS
1	66	29	42	68	20	*
2	97	76	70	93	65	53
3	99	95	89	98	90	74
4	100	97	96	100	98	89
5		99	96		100	100
6		100	100			

Error seeding criteria inserts typical faults into a system, and determines how many of the inserted faults are found. In mutation testing, the criterion uses a set of products that differ slightly from product P under testing, named mutants, in order to evaluate the adequacy of a test suite T . The goal is to find a set of test cases which is able to reveal the differences between P and its mutants, making them behave differently. When a mutant is identified to have a diverse behavior from P it is said to be “dead”, otherwise it is a “live” mutant. A live mutant must be analyzed for one to check whether it is equivalent to P or whether it can be killed by a new test case, thus promoting the improvement of T. (VINCENZI, 2004).

Despite there are many works about different testing techniques and their criteria, only few works propose some strategy to assist the definition of which components will be selected, specially in unit testing level. In this thesis we are proposing a method with the purpose of covering this gap, in order to be used even before the definition of some criteria to define the test cases.

Orthogonal Array Testing (OATS)

Orthogonal Array Testing (OATS) is a special functional testing technique. The resources (time, money) available for the development of testing are often limited. Thus, it is more attractive for developers and testers to identify which areas more fault prone. The work of Wallace e Kuhn (2001) is the first we found in the literature presenting a relationship between the number of variables and system failures. The authors investigated a medical device system, and they concluded that most failures were triggered by the interaction of two variables, and progressively less for 3, 4, or more variables, and that all software failures involve interactions among a small number of variables no more than six. Table 2.1 presents the results of this study.

Based on these evidences, the use of techniques for generating an optimized set of variables instead of using all possible combinations passed to be desired. Among these techniques, we highlight the technique called Pairwise comparison, which is based on the comparison of peers to determine which of them is the most interesting. In one of the pioneering works of pairwise applied in software testing context, Mandl (1985) presents

a technique which attempts to minimize the level of necessary effort to define a set of states to test a compiler.

Also known as OA, OAT, or OATS, Orthogonal Array Testing is a special functional testing technique, designed in a statistical and systematic way. Through the usage of OATS, it is possible to maximize test coverage while minimizing the number of test cases to be considered. For instance, based on the conclusions of (WALLACE; KUHN, 2001) and (KUHN; WALLACE; GALLO, 2004), the number of reduced combinations of User Interface (UI) inputs for black-box testing can be generated with the aid of automated tools with this purpose. The use of this approach allows significant savings of testing costs, increasing the fault detection rates in the system. OATS has been applied in system testing, regression testing, configuration testing, performance testing, and in UI testing. In our method we make use of OATS technique to generate a optimized list of Android devices in order to maximize the market coverage while minimize o number of devices, as presented in the Section 4.3.3.

2.1.3 Automation in Android Testing

There are much benefit when tests are automated. Thus, as argued by Ammann e Offutt (2008) testing should be automated as much as possible, but there are some challenges when it comes to automating the testing process. However, the need for automated testing is still great, since testing plays a big role in software development.

Android User Interface Testing (UI Testing) is a functional testing technique used to identify the presence of faults in a Software Under Test (SUT) by using Graphical user interface (GUI). There are three kind of UI Testing approaches: manual, based on capture-Replay techniques, and model-based testing.

In order to automate Android UI testing some strategies have been implemented embedded in some tools. In addition to manual approach, Capture-Replay is a well known and used approach for recording user interactions into a script that can be later replayed for automatically performing the same interactions on the app. RERAN (GOMEZ et al., 2013) is one of such a tool. It captures low level system events by leveraging the Android GETEVENTS utility and generates a replay script for the same device. RERAN is useful to capture and replay complex multi-touch gestures. However, the generated scripts are not suitable for replay on different devices because they contain screen-coordinate based interactions, which cannot be re-run on a screen with different size. MOSAIC (ZHU; PERI; REDDI, 2015) is another similar tool, which solves this problem by abstracting the low level events into a set of operations on a virtual display. The tool then uses a heuristic to convert these operations into low level events for a device with a different screen size. Both RERAN and MOSAIC do not support adding assertions in their replay

script. Moreover, the replay of captured scripts might not be deterministic. In fact, at replay time the app might not exhibit the same timing characteristics as displayed at capture time.

Related to the tools used to generate input data for UI Testing, according to Choudhary, Gorla e Orso (2015) they can be classified according to their strategy. Basically, four groups of strategies might be found: instrumenting the app/system, triggering system events, black-box testing, and exploration strategy.

The first group is based in the instrumentation strategy. In this strategy the tool has to interact with the app in order to understand the results that come from the interaction. The tool can modify the app by injecting commands, or even modifying Android platform to know what is happening during the app execution.

The second strategy is based in triggering system events. A UI testing generation tool can interact with an app not only through UI components, but through system events. Parts of the apps might be triggered by external notifications, i.e., messages. In order to trigger such functionality, the tools have to trigger system events. Also, even if a tool does not have access to the source code of an app, it can do the testing in black box approach.

In exploration strategy it is a challenge to decide how the tool will explore the states of an app. It can be done in three distinct ways: randomly (Monkey (UI/APPLICATION..., 2015), and Dynodroid (MACHIRY; TAHILIANI; NAIK, 2013)), based in the app model, or in a systematic way. Model-based exploration strategy uses a specific model (e.g., GUI model) of the app to systematically explore finite state machines, where the states are the activities, and the edges are the events representing the transitions among the states. A3E (AZIM; NEAMTIU, 2013), SwiftHand (CHOI; NECULA; SEN, 2013), GUIRipper (AMALFITANO et al., 2012), PUMA (HAO et al., 2014), and Orbit (YANG; PRASAD; XIE, 2013) use this strategy. Despite this strategy reduces the redundancy by not explore the same states more than once, they do not consider events that alter non GUI-state. In the systematic exploration strategy they use sophisticated techniques such as symbolic execution and evolutionary algorithms to cover the states of the application systematically. As example of tools that make use of this strategy we can mention ACTEve (ANAND et al., 2012), and also EvoDroid (MAHMOOD; MIRZAEI; MALEK, 2014) which is based in white box strategy.

In computer programming, an application programming interface (API) is a set of routines, protocols, and tools for building software and applications. There are some APIs in order to assist both Android developers and Android testers in the development of UI testing for Android apps, such as, UIAutomator API (GOOGLE, 2015), Robotium (ZADGAONKAR, 2013), Appium (Sauce Labs, 2015), and the recent API designed by Google called Espresso (Espresso, 2015).

Robotium (ZADGAONKAR, 2013) is an Android test framework that provides a Java API to interact with the UI elements. It is an open source library extending JUnit (JUNIT, 2010) with plenty of useful methods for Android UI testing. Supports native, hybrid and mobile web testing, and it works similar to Selenium, but for Android. Calabash (Calabash, 2015) was designed as cross-platform supporting both Android and native iOS by writing tests either in the Ruby language or in natural language using the Cucumber (Cucumber, 2015) tool and then converted to Robotium at run time. It also includes a command line inspector for finding right UI element names/ids. Appium (Sauce Labs, 2015; SHAH; SHAH; MUCHHALA, 2014) is another cross-platform testing framework, which allows tests to be written in multiple languages. Appium tests run in a distributed fashion on a desktop machine while communicating with an agent on the mobile device. This communication follows the JSON wire protocol standardized by the web testing tool WebDriver, commonly known as Selenium. Selendroid (Selendroid, 2015) is based on Selenium to be able to give full support to both hybrid and native Android applications. It allows tests to be written in Java. UIAutomator is a Google‘s test framework for testing native Android apps across device (GOOGLE, 2015). It works only on Android API level 16 or higher, and it runs JUnit test cases with special privileges. There is no support for web view. Espresso is the latest Android test automation framework from Google. It is a custom Instrumentation Testrunner with special privileges, and it works on API levels 8 or higher on top of Android instrumentation framework. Espresso is becoming a *de-facto* standard in the Android testing world. Espresso synchronizes view operations with the app's main UI thread and with `AsyncTasks` workers, thereby making the replay fast and deterministic.

Also, some tools are available to automate the generation of scripts in some of the APIs listed above. ACRT (LIU et al., 2014) is a research tool that also generates Robotium tests starting from user interactions. ACRT's approach modifies the layout of the Application Under Test (AUT) to intercept user events. The tool also allows for injecting a custom gesture to launch a dialog for capturing assertions for certain UI elements. In practice, injecting such gestures can limit the normal interactions that the tester can have with the AUT. For instance, the default gesture *slide down* can interfere with *scroll* events on an app screen. SPAG (LIN et al., 2014a) is a recent tool that integrates SIKULI (YEH; CHANG; MILLER, 2009) and ANDROID SCREENCAST (ANDROID..., 2015) to develop and run image based tests on a desktop machine connected to a mobile device. SPAG−C (LIN et al., 2014b) is an extension to SPAG that adds visual oracles by automatically capturing reference screen images during test case creation. Such visual techniques are minimally invasive, as they do not modify the app. However, capturing deterministic screenshots is a practical challenge that leads to a high number of false pos-

itives reported by the tools. Moreover, images tend to differ across devices, making such techniques unsuitable for cross-device testing.

In this work we used Barista tool (CHOUDHARY, 2015a) to automate the generation of UI test cases written in Espresso API from user's interactions. Barista allows the user records interactions with an app in a minimally-intrusive way, and easily specifies expected results (assertions) while recording. Barista is able in generating platform-independent test scripts based on the recorded interactions and the specified expected results, and also running the generated test scripts on multiple platforms automatically. With this test cases we could collect some dynamic metrics defined in our model running them cross-device. The automation of our method is describe in more details in the Chapter 4.

2.2 Component Selection Problem (CSP)

The choice of which subset of components[1] are chosen for a next unit testing cycle is always supported by some kind of guidance. This decision is typically made in the planing stage of the process, and its influence can be far reaching.

To the best of our knowledge, the earliest generic formulation for the Component Selection Problem (CSP) in software engineering field was presented in the poster paper (HARMAN et al., 2006), suggesting the usage of automated approaches employing search based software engineering in future works for different instances. Still according to Harman et al. (2006), this problem finds a manager considering several candidate components, and a hard challenge of finding a suitable balance among potentially conflicting objectives. Thus, the component selection solution should assist the manager to decide which set of components will optimize the objectives.

To model a CSP, we define a score for each component, and we combine the cost of testing to a single cost value c_i, and manager desirability and expected revenue to a benefit value b_i, and the value of the item x_i, where i is an index of the components. The objective is to maximize the total score of feasible subsets, trying to figure out a subset that maximizes the total sum of score while minimizing the total cost of the selected components. A subset is feasible if its total cost of unit testing is less or equal to the total available time for unit testing (T). The formulation of a Component Selection Problem (CSP) with n components, and a single objective can be given as follows:

[1]The term components refers to the small piece of code, e.g. a method in object-oriented languages.

$$\max \sum_{i=1}^{n} (b_i - c_i) \cdot x_i \tag{2-1}$$

$$s.t. \sum_{i=1}^{n} c_i \cdot x_i \leq T, \qquad x \in \{0,1\} \tag{2-2}$$

A CSP with a single objective is a Knapsack-type problem, which is known to be NP-hard. However, it can be solved by a pseudo-polynomial algorithm using dynamic programming (PAPADIMITRIOU; STEIGLITZ, 1998). The algorithm runs in $O(n^2t)$ time (where n is the number of components) and therefore depends on the optimum value for t that can be found within T (HARMAN et al., 2006).

In additional to have a single objective, the formulation presented in 2-2 may also be comprised by several objectives that will be optimized simultaneously. In this case the component selection problem can be formulated in the following form:

$$\max \ F(x) \tag{P1}$$

$$\text{subject to } \ g_j(x) \leq r_j, \ j = 1,2,\cdots,m, \tag{P2}$$

where $x = (x_1, x_2, \cdots, x_N)$ with x_i taking value 1 if artifact i is selected and 0 otherwise; $F(x)$ can be a real function defined by any combination of the real functions $f_1, f_2, \cdots, f_n$, or $F(x)$ can be a vector function given by $F(x) = (f_1, f_2, \cdots, f_n)$; inequalities (P2) represent limitations on the availability of resources.

When $F(x)$ is a real function (e.g., $F(x) = f_1(x) + f_2(x) + \cdots + f_n(x)$) the optimization problem (P) might be handled by any standard integer programming solver. However, when $F(x)$ is a vector function we have a many-objective optimization problem (also called multi-objective when n is less or equal to four).

A multiobjective problem may not have a single solution. Indeed, its solution is usually composed by a set of solutions that represents a commitment among the objectives. In component selection optimization context, a solution is a set of code units with different values for each unit f_i.

The precise solution to the Component Selection Problem (CSP) depends on the concept of dominance. Let S denote the set of binary vectors satisfying the constraints (P2). Given x and y in S we say that *x dominates y* if the following conditions hold:
a) $f_i(x)$ is greater than or equal to $f_i(y)$ for all i in $\{1,2,\cdots n\}$;
b) $f_i(x)$ is strictly greater than $f_i(y)$ for at least one i in $\{1,2,\cdots n\}$.
A vector x^* in S is called a dominating solution if it dominates all other solutions. When such a solution exists, it is called a Pareto Optimal. On the other hand, we say that x is not

dominated by y if $f_i(x)$ is strictly greater than $f_i(y)$ for at least one index i. A vector x^* in S is called a non-dominated solution if it is not dominated by any other solution in S.

The set of all non-dominated solutions define the solution of (P) in a N-dimensional solution space. Applying F to each non-dominated solution we obtain a subset in the n-dimensional objective space, which is called Pareto Front.

As an example, consider a problem with only two objectives f_1 and f_2. In Figure 2.1 we have the images under $F(x) = (f_1(x), f_2(x))$ of seven candidate solutions of the problem. In this case, the solution represented by the point B dominates the solutions E, F, and G. However, B does not dominate C, indeed C is non-dominated in the set of solutions plotted in this figure. Likewise, A, B, and D are non-dominated. In particular, if the whole solution set to this problem were composed by those seven points we could conclude that A, B, C, and D formed the Pareto Front of this instance. In our context, each of these points would represent a set of selected components that maximize the objectives f_1 and f_2 simultaneously.

Summarizing, the main goal in CSP is to compute the optimal values of (P) in the case $F(x)$ is a real function, and the Pareto Front in the case $F(x)$ is a vector function.

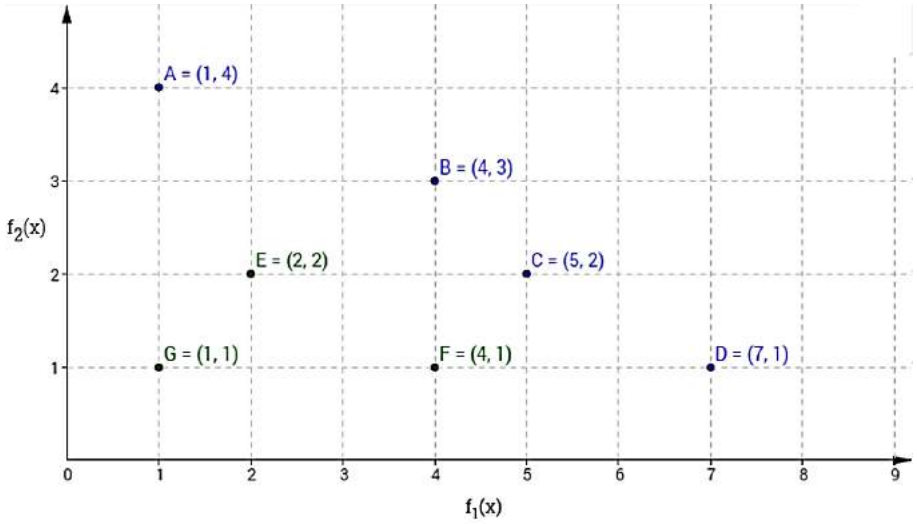

Figure 2.1: *Pareto Front is constituted by the points A, B, C, and D.*

2.3 Search Based Software Testing (SBST)

Search Based Software Engineering (SBSE) is a sub-area of software engineering with origins stretching back to the 1970s but not formally established as a field of study in its own right until 2001, with the publication of the seminal paper in SBSE (HARMAN; JONES, 2001). Search Based Software Engineering (SBSE) seeks to reformulate software engineering problems throughout the Software Engineering life cycle as search-

based optimization problems and applies a variety of Search Based Optimization (SBO) algorithms and meta-heuristics to solve them. The objective is to identify among all possible solutions a set of solutions, which will be sufficiently good according to a set of appropriated metrics. SBSE has been applied in software engineering problems come from requirements and project planning to maintenance and reengineering phases.

A subarea of SBSE is Search Based Software Management (SBSM). Although SBSE was mentioned for the first time by Harman (HARMAN; JONES, 2001), early papers in Search Based in Software Management were done ((CHANG, 1994), (CHANG et al., 1998) , (CHANG et al., 1994), (DOLADO, 2000), and (SHUKLA, 2000)). Recently, Ren J. (REN, 2013) presented a thesis entitled "Search Based Software Project Management" showing how Search Based Software Engineering (SBSE) approach is applied in the field of Software Project Management (SPM).

According to Harman, Mansouri e Zhang (2012) Software Engineering Management is concerned with the management of complex activities being carried out in different stages of the software life cycle, seeking to optimize both the processes of software production as well as the products produced by this process. As detailed in Section 2.2 the component selection for unit testing can be considered a problem studied in this category, once Software Engineering Management has been also used to assist software testing activities.

SBSE has been also applied explicitly to solve problem in Software Testing. By definition, Search Based Software Testing is the use of SBSE search techniques to search large search spaces, guided by a fitness function that captures natural counterparts as test objectives (adapted from (HARMAN; JIA; ZHANG, 2015)). The number of published papers in SBST has increased exponentially, according to presented at the Figure 2.2.

Many meta-heuristic search techniques, such as Genetic Algorithm (GA), Simulated Annealing (SA), and Hill Climbing (HC), SPEA_II, NSGA_II, NSGA_III have become a burgeoning interest to many researchers in recent years. In our work, we evaluated some of this search based techniques applied in order to solve our formulation for CSP as shown in the Section 5.4.

We address our research in a planning phase of Software Testing employing SBSE approach to solve a CSP problem, and although Software Engineering Management in its essence has been used in this context to assist the optimization of software testing activities. Thus, by definition our work is located in SBST field.

In this Chapter we described basic concepts necessary to understand the remaining of this thesis, an a detailed formualation of the Component Selection Problem (CSP). In the next chapter we present the main related works found in the literature, pointing out gaps and opportunities we explore in our thesis.

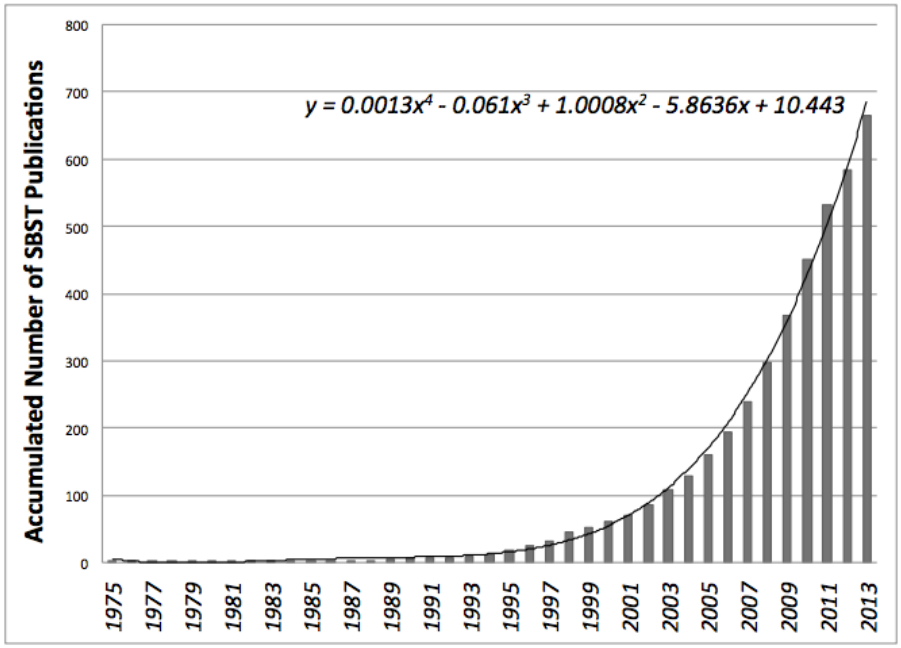

Figure 2.2: *Number of papers in SBST, extracted from (HARMAN; JIA; ZHANG, 2015).*

CHAPTER 3

Related Work

In this Chapter we present a general overview about the state of art of approaches to select components for unit testing. The works we found close to our work can be classified according to their nature of the objectives. Also, there are a few key characteristics such as component level, nature of the problem, number of objectives, algorithms, and focus.

3.1 Nature of the Objectives

Related to the nature of the objectives, some of them have as the main goal reducing the fault proneness using (ELBERZHAGER et al., 2012): (1) Product metrics, e.g., size metrics (e.g., lines of code), complexity metrics (e.g., McCabe complexity), or code structure metrics (e.g., number of if-then-else); (2) Process metrics, e.g., development metrics (e.g., number of code changes), or test metrics (e.g., number of test cases); (3) Object-oriented metrics, e.g., weighted method per class, depth of inheritance; and (4) Defect metrics, e.g., customer defects, or defects from previous releases. Others works seek to increase the software reliability, and others focudes in minimizing the stub creation effort (ASSUNÇÃO et al., 2014).

We found a systematic mapping study presenting different approaches to reduce the test effort (ELBERZHAGER et al., 2012). From this work, we could indentify some gaps and opportunities that reinforced our motivation in proposing a method to solve a multi-objective component selection problem. The authors presented many approaches to predict defect-prone parts of the system. The basic assumption is that if such areas are identified, theoretically testing activities should be focused on those parts to reduce the testing effort. The authors investigated the identification of existing approaches that are able to reduce testing effort, and among them they confirm the use of predict defect-prone parts or defect content to focus the effort testing. They identified five different areas that exploit different ways to reduce testing effort, and among them, approaches that predict defect-prone parts or defect content. According to them predictions can support decisions on how much testing effort is needed or how testing effort should

be distributed. They also presented an overview of the kind of input (i.e., top-level metric) used to perform the predictions, and classified them in four cases that can be distinguished in product metrics, process metrics, development metrics, object-oriented metrics, and defect metrics. Therefore, having the works mentioned in this systematic mapping (ELBERZHAGER et al., 2012), we extended our search seeking to find others works related to our work, as mentioned below.

Confirming the same line of reasoning presented in the systematic mapping, in the paper entitled "Using Static Analysis to Determine Where to Focus Dynamic Testing Effort" (WEYUKER; OSTRAND; BELL, 2004), the authors state the following in their motivation: "Therefore, we want to determine which files in the system are most likely to contain the largest numbers of faults that lead to failures and prioritize our testing effort accordingly.". Exploring historical defect data, they used a static analysis to determine where to focus dynamic testing effort. They developed a negative binomial regression model to predict which files in a large software system are most likely to contain the largest numbers of faults. Shihab et al. (2011) suggest that heuristics based on the statics metrics such as function size, modification frequency and bug fixing frequency should be used to prioritize the unit testing writing on legacy systems. In his another work (SHIHAB et al., 2011) argues even there are a plethora of recent work leverages historical data to help practitioners better prioritize their software quality assurance efforts, the adoption of this in practice remains low. We did not consider neither this strategy nor (SHIHAB et al., 2011) in the comparison in our baselines because differently from our proposal, they work as file level (instead of component level), and they would need the historical of defects per method for all subjects, such information is not available under our subjects. In (HASSAN; HOLT, 2005) the authors present an approach called "The Top Ten List" to assist managers in allocating testing resources by focusing on the subsystems that are likely to have a fault appear in them in the near future. The Top Ten List highlights to managers the ten most susceptible subsystems (directories) to have a fault. Thereby, managers could focus testing resources to the subsystems suggested by the list. The list is updated dynamically as the development of the system progresses. They applied their presented approach to six large open source projects (three operating systems: NetBSD, FreeBSD, OpenBSD; a window manager: KDE; an office productivity suite: KOffice; and a database management system: Postgres). However, they did not defend a especific heuristic as the best, but they just used a few heuristics to validate their proposed Top Ten list approach.

Spectrum-based Fault Localization (SBFL) approaches utilize various program spectra acquired dynamically from software testing, as well as the associated testing result, in terms of failed or passed, and evaluates the risk of containing a fault for each program entity. Among those, we can highlight Tarantula tool (JONES; HARROLD;

STASKO, 2002), a statistics based lightweight fault localization technique using Ochiai coefficient (ABREU et al., 2009); and MZoltar (MACHADO; CAMPOS; ABREU, 2013) is an approach to perform dynamic analyzes in Android apps producing reports to help identifying potential defects quickly.

In (RAY; MOHAPATRA, 2012) the authors propose a testing effort prioritization method to guide tester during software development life cycle. They consider five factors of a component (class) such as influence value (number of components directly or indirectly impacted), average execution time, structural complexity (response for a class - RFC; weighted methods in a class - WMC), severity (severity of damages caused by the failure of the component within a scenario), and business value as inputs and produce the priority value of the component as an output. While they explored operational profile collecting the execution time from test cases execution (average on 100 executions), in our approach we explore frequency of method calls from operational profile through user interactions. Severity and business value need to be collected manually (business value comes from domain analyst), which are expensive, error-prone and very time-consuming. Our method allows to compute severity (cost of future maintenance and the market vulnerability) in an automated way. Another important difference from our work is that they considered class level metrics, while we consider metrics in method level. Lastly, they do not consider the problem as a component selection problem which includes the combinatorial aspect, but as a prioritization problem which gets as a result a ranking of components.

In (LI; BOEHM, 2013) and (LI, 2009) the authors propose a value-based prioritization strategy based on their ratings of business importance, Quality Risk Probability, and Testing Cost. However, these metrics are extremely dependent of the specialist, who manually defines their values and weights, there is no usage of dynamic information, and the result is a ranked list of components (as in (RAY; MOHAPATRA, 2012)) instead of a subset of components.

Elberzhager et al. (ELBERZHAGER et al., 2013) present *In2Test* to integrate inspections with testing, i.e., inspection defect data is explicitly used to predict defect-prone parts in order to focus testing activities on those parts. In addition, they use both code metrics and historical data. However, the inspection process is manual and dependent of certain factors such as inspector experience or process conformance. The are still others papers (ELBERZHAGER; MÜNCH; NHA, 2012), (ELBERZHAGER; MÜNCH; ASSMANN, 2014), (ELBERZHAGER; BAUER, 2012), (ELBERZHAGER et al., 2013), (ELBERZHAGER et al., 2012), (ELBERZHAGER; MÜNCH, 2013), (ELBERZHAGER; ESCHBACH; MÜNCH, 2010), (ELBERZHAGER et al., 2011) from the same group of authors exploring the integration of inspection and testing techniques as a promising research direction for the exploitation of additional synergy effects.

3.2 Others Characteristics

Still, we can also classify the works we found in the literature according to a few characteristics: number of objectives (single-objective or multi-objective), component level (method, class, or file), nature of the problem (selection, prioritization, testing resource allocation), algorithms, and focus.

The most of works were formulated as a single-objective problem, while others as the multi-objective. Even that there are many works in Multi-Objective Search Based Software Testing (MoSBaT) (HARMAN YUE JIA, 2015) presenting strategies for problems concerned with test suite selection and prioritization (ASSUNÇÃO et al., 2014), (BATE; KHAN, 2011), (BRIAND; LABICHE; CHEN, 2013), (MIRARAB; AKHLAGHI; TAHVILDARI, 2012), (SHELBURG; KESSENTINI; TAURITZ, 2013), (SHI et al., 2014), (YOO; HARMAN, 2010), (CZERWONKA et al., 2011) they have different purpose from our work, once we work for selecting components for the development of unit tests, even if there is no test cases written for the system.

The earliest generic formulation for the Component Selection Problem (CSP) in search based software engineering field was presented in the poster paper (HARMAN et al., 2006), suggesting the use of automated approaches employing search based software engineering in future works. After that, many works have been proposed in different fields of the Software Engineering, such as Next Release Problem (NRP) (DURILLO et al., 2011; ZHANG; HARMAN; LIM, 2013; ZHANG, 2010). In these works the NRP is seen as a multi-objective problem, since it minimizes the total cost of including new features into a software package and maximizes the total satisfaction of customers. This poster paper addresses the problem of choosing sets of software components to combine in component–based software engineering. It formulates both ranking and selection problems as feature subset selection problems to which search based software engineering can be applied. They considered the selection and ranking of elements from a set of software components from the component base of a large telecommunications organisation. To the best of our knowledge, there is no instance in the literature working with CSP in Software Testing.

Also, a close research field to our work is Testing Resource Allocation (TRA). Besides allocating resources among components guided by static defect prediction, TRA also has used Software Reliability Growth Models (SRGMs). Some works have been found in this field. In (KAPUR et al., 2009) the authors propose the use of a genetic algorithm in the field of software reliability. They have discussed the optimization problem of allocating testing resources in software having modular structure by minimizing the total software testing cost under the constraints of availability of limited testing resource expenditure and to achieve desired level of reliability for each module. This approach

explores its capability to give optimal results through learning from historical data. In another work, Wang, Tang e Yao (2010) suggest solving Optimal Testing Resource Allocation Problems (OTRAPs) with Multi-Objective Evolutionary Algorithms (MOEAs). They formulated the problem as two types of multi-objective problems. First, they considered the reliability of the system and the testing cost as two objectives. Second, the total testing resource consumed is also taken into account as the third objective.

In (KIPER; FEATHER; RICHARDSON, 2007) the authors applied genetic algorithm and simulated annealing to select optimal subset of Verification and Validation activities in order to reduce risk under budget restrictions, thereby linking the problem domains of testing and management.

Many types of algorithms have been also applied. Among them we can highlight greedy approaches, and evolutionary algorithms such as Genetic Algorithm, NSGA_II, NSGA_III, and SPEA_II. Compared to the levels of the components, we found other works focusing on three different levels: class, and file. No work was found performing the selection on method level. A clear difference between our work and others is regarding to the nature of the problem. While we work as a component selection problem, there are works that the main goals is to prioritize components, i.e., to create a ranked list of components based in their importance. In this case, these works do not take into consideration constraints (the available time for testing activities).

3.3 General Summary

Table 3.1 presents some highlights regarding to the close works found in the literature compared to our work.

Table 3.1: *Close works to CSP.*

Work	Number of Objectives	Nature of Objectives	Algorithms	Component Level	Nature of the Problem	Market Information
(SHIHAB et al., 2011)	Single	Change Metrics (MFM)	Greedy	Method	Prioritization	Not present
(RAY; MOHAPATRA, 2012)	Multi	Influence value; execution time; structural complexity; severity; business value	Greedy	Class	Prioritization	Present (Manually)
(WEYUKER; OSTRAND; BELL, 2004)	Single	Historical Data	Binomial Regression	File	Prioritization	Not present
(ELBERZHAGER et al., 2011)	Single	Inspection and Test Cases	Greedy	Class	Prioritization	Not present
(HASSAN; HOLT, 2005)	Single	Fault Prone	Greedy	Subsystem	Prioritization	Not present
(JHA et al., 2009)	Single	Software reliability	Genetic Algorithm	Module	Resource Allocation	Not present
(LI; BOEHM, 2013)	Single	Business Importance; Quality Risk Probability; Testing Cost.	Greedy	Method	Prioritization	Present (Manually)
(YUAN; XU; WANG, 2014)	Multi	Software reliability; Testing Cost	NSGA_II	Module	Resource Allocation	Not present
(CZERWONKA et al., 2011)	Single	Fault Prone	Greedy	Method	Test Prioritization	Not present

Some works ((JONES; HARROLD; STASKO, 2002), (ABREU et al., 2009), (MACHADO; CAMPOS; ABREU, 2013), (WEYUKER; OSTRAND; BELL, 2004)) are guided only by a single-objective strategy, to define in which components they have to focus their testing effort. Despite few works proposing the usage of multiple

objectives ((LI; BOEHM, 2013), (RAY; MOHAPATRA, 2012)), none of them work at the method level, but in the file or class level, therefore the metrics and the goal are different. Many strategies are dependent of the human intervention to collect the necessary information ((LI; BOEHM, 2013), (RAY; MOHAPATRA, 2012)) not allowing an automated collection. The related works do not see the component selection problem as a combinatorial optimization problem (including tight deadlines), but as a prioritization problem which expects as a result a ranked list of components. None of them works with market vulnerability (especially in the Android ecosystem).

We tackled these gaps and challenges with a Selector of Software Components for Unit testing (SCOUT). The main goal is to optimize two different objectives considering metrics in level of units such as: risk of fault (suspiciousness), frequency of calls (profiling), market vulnerability, cost of future maintenance, and cost of unit testing. We presented our process to automate the use of SCOUT in Android real context. Also, in order to assist the specialist in an automated way, we also investigate some potential solvers for this unit selection problem. Seven algorithms/techniques were analyzed to solve this multiobjective problem: Randomly approach (R), Constructivist Heuristic (CH), Genetic Algorithm (GA), SPEA_II, NSGA_II, NSGA_III, and a heuristic implemented by the Gurobi tool (OPTIMIZATION et al., 2015), as presented in the Chapter 5 on Section 5.4.

In our comparative study, we used Halstead Bugs metric (JHAWK, 2016) as the representative of static metrics, and Tarantula coefficient (JONES; HARROLD; STASKO, 2002) in our baseline as the representative of the SBFL approaches (dynamic techniques). Since fault localization approaches do not handle multiples objectives, we compared the efficacy of SCOUT over these fault localization techniques, as described in the Section 5.6.

To the best of our knowledge, SCOUT is the first method to assist software testing managers to select Android components in method level for unit testing based on many-objective approach exploring both static and dynamic metrics as well as Android market information.

CHAPTER 4

Selector of Software Components for Unit Testing

This chapter presents the Selector of Software Components for Unit Testing, which performs two principal processes: extraction of metrics and multi-objective optimization. The metrics are extracted from Android-user interactions and combined in a unique metrics database, which, according to tester inputs and time constraints, conducts a multi-objective optimization that generates a list of selected components for unit testing that respects the imposed constraints. Figure 4.1 depicts this flow.

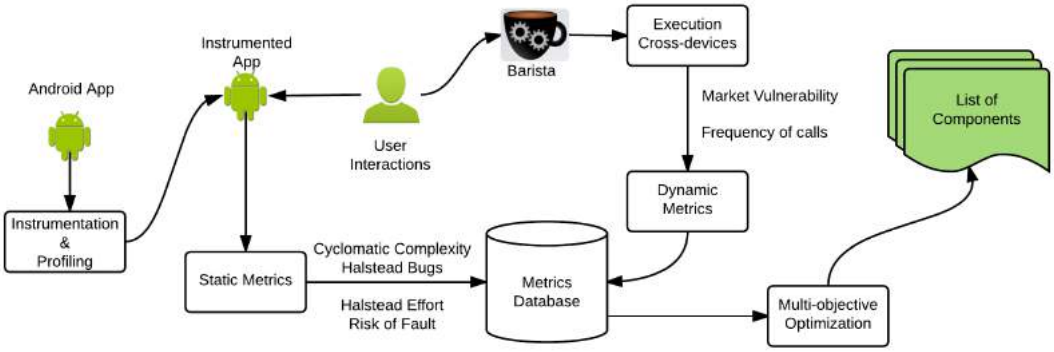

Figure 4.1: *General* SCOUT *flow to select artifacts for unit testing.*

In this chapter, key variables used by SCOUT are discussed; followed by a description of its model formulation and concepts. Aspects to automate this process on Android platforms are provided, followed by the multi-objective optimization phase.

4.1 Metrics Choice

The quality assurance team requires a strategy to guide the selection of components for unit testing. As previously stated, most strategies are based on the experiences of specialists or defect prediction or fault localization models. Three types of approaches are widely used, those based on code metrics, change metrics, and spectrum-based fault

localization. The basic premise is that if critical areas were identified, testing activities could be economized. According to Elberzhager et al. (2012), most previous works use metrics, i.e., statics or dynamics, to predict defect-prone components. Their efficacy is proven through analysis of their proficiency in identifying faults.

No doubt, finding faults is important as it focuses testing on components prone to defects. However, in practice, are these parts equally significant? Even if two components are equally prone to defect, do they have the same strategic importance or are there others factors that should be considered in assessing their relative benefits? If so, what are they?

SCOUT addresses these questions by taking into account metrics that derive from three principal sources: static, dynamic, and market analyses. These sources are used as variables in defining the relative benefit of selected components for unit testing, using the following metrics: cost of future maintenance (static analysis), frequency of calls (dynamic analysis), fault risk (dynamic analysis), market vulnerability (market analysis), and unit testing cost (in terms of time). Each of these metrics is delineated below and Section 4.3 provides an automated process of collecting them.

4.1.1 Unit Testing Cost

The unit testing cost is the variable used to describe the amount of time required to develop the unit testing process for a given component. Inasmuch as testers do not customarily design the software they are testing, they must expend considerable time in learning about it (CHIKOFSKY; CROSS et al., 1990). While rarely calculated as a direct cost, the cost of understanding software is nonetheless tangible. It is manifest in the time required to comprehend it, which includes time lost to misunderstanding. Measuring and estimating the time required to develop unit testing depends on the kind of testing criteria chosen.

4.1.2 Cost of Future Maintenance

The ANSI definition of software maintenance is the modification of a software product after delivery to correct faults, improve performance or other attributes, or adapt the product to a modified environment (COMMITTEE et al., 1998).

Each component has an associated defect proneness, and in case of a failure, those responsible for maintaining the software spend time to understand the system and make appropriate changes. We call the product of the defect proneness of a component and the time required to understand and fix it the cost of future maintenance (*cfm*). The equation below presents its computation.

$$cfm_i = t_i \cdot bugs_i \tag{4-1}$$

where:
t: amount of work in seconds to understand and recode the component (i);
bugs: estimated number of bugs in the component (i).

We choose cfm as an important variable in SCOUT to take into account fault prediction models based on static analysis and to measure the cost impact of this type of fault in case should it occur.

4.1.3 Frequency of Calls

Profiling a software can leverage the analysis of runtime information. The frequency of calls represents the number of times a component is invoked during an execution. In practice, SCOUT computes the frequency of calls inasmuch as it indicates how the software is demanded internally at the method level and which components are more frequently exercised. Despite a component having a high degree of cyclomatic complexity or even a high rate of defect proneness, the impact of these static metrics must be associated in some way with a metric that reflects the level of requisition of a component under execution, i.e., frequency of method calls.

4.1.4 Fault Risk

Fault risk can be computed based on spectrum-based fault localization techniques whose objective is to identify the components responsible for observed software failures. In essence, the coefficient ranks the component in terms of suspiciousness with a risk of fault in the range [0,1] wherein 0 means the lowest and 1 the highest risk based on execution of a test suite.

The data needed to compute this metric comes from the record of the execution of a component in both successful and failed test cases. Methods have been developed to automate this assessment. For example, one can highlight coefficients such as that used by the Tarantula tool (JONES; HARROLD; STASKO, 2002), the Jaccard coefficient used by the Pinpoint tool (CHEN et al., 2002), and the Ochiai coefficient used by the MZoltar tool (MACHADO; CAMPOS; ABREU, 2013).

In this study, the coefficient used by Tarantula (JONES; HARROLD; STASKO, 2002) tool is used. Its metric can be computed by:

$$rf_i = \frac{p_i}{p_i + f_i} \tag{4-2}$$

where:

p_i is a function that returns, as a percentage, the ratio of the number of passed test cases that executed the component to the total number of passed test cases in the test suite; and f_i is a function that returns, as a percentage, the ratio of the number of failed test cases that executed s to the total number of failed test cases in the test suite;

To illustrate fault risk, consider the results from the execution of a test suite with nine test cases as depicted in Table 4.1.

Nine test cases were executed as shown on the table's right. The set of test case executions is indicated by column heads. Component coverage is shown by an "x" in the appropriate column, and the pass (P)/fail (F) result of each test execution is indicated at the bottom of its respective column.

Thus, the second component was invoked by two failed test cases (2 and 3), and the first set of test-case execution (4, 5, and 6), which passes, involves components 2, 3, and 6. Based on the results of the test case executions, the risk of fault for the component 2 is 0.60, once $p_i = 0.33$ and $f_i = 0.22$.

Table 4.1: *Faulty components (left); test cases, component coverage, and test results (right). Adapted from (JONES; HARROLD; STASKO, 2002).*

	Test Cases					
Components	4,5,6	2,3,4	2,3	6,4,5	5,7,8	7,8,9
1		x		x		x
2	x		x			
3	x				x	
4		x	x	x	x	x
5				x	x	
6	x		x	x		x
7		x				
8					x	
9			x		x	
10			x			
Pass/Fail status	P	P	F	P	P	P

4.1.5 Market Vulnerability

The market vulnerability metric is used to represent the percentage of the market in which a component is vulnerable. Software exhibits different behaviors in the diverse clients on which it is executed. Consider, for instance, a new software version deployed on three different clients (A, B, and C), which correspond to different revenue rates for the software developer, viz., 22%, 47%, and 31% respectively. In this example, should

component *x* fail only on A, its market vulnerability is 0.22. If, however, it fails on both A and C, its market vulnerability is 0.53.

As all experiments conducted by the study used an Android platform, its market vulnerability was computed from the market share of each device on an Android platform as presented in Section 4.3.2. This metric expresses the vulnerability of a component across devices according to market distribution (ANDROID. . . , 2015), and the greater the market share of a given device, the greater the market vulnerability of a component should it fail on such a device or one with similar features. Accordingly, the computation of market vulnerability entails identifying which component is associated with each failed test cases in each device. Section 4.3.2 presents an automated means to collect this information, followed by an example.

4.2 Model Formulation

In addition to the metrics described above, several others were considered for use in formulating the SCOUT model. As analysis indicated that some had strong positive correlations, they were not kept. Accordingly, a correlation analysis was undertaken for the remaining metrics that could be considered for a model focused on selecting components for unit testing. These variables were cyclomatic complexity, cost of unit testing, expected number of bugs, cost of future maintenance, frequency of calls, fault risk, and market vulnerability. Their correlations are provided in Table 4.2.

Table 4.2: *Metrics Correlation*

	Cyclomatic Complexity	Cost of Unit Testing	Expected Number of Bugs	Cost of Future Maintenance	Frequency of Calls	Fault Risk	Market Vulnerability
Cyclomatic Complexity	1.00						
Cost of Unit Testing	0.71	1.00					
Expected Number of Bugs	0.71	0.92	1.00				
Cost of Future Maintenance	0.57	0.89	0.77	1.00			
Frequency of Calls	-0.06	-0.03	-0.05	-0.01	1.00		
Fault Risk	-0.37	-0.18	-0.23	-0.09	0.08	1.00	
Market Vulnerability	-0.33	-0.15	-0.20	-0.04	0.15	0.76	1.00

To compute the benefit of a subset of selected components, the following metrics were retained: cost of future maintenance, frequency of calls, fault risk, and market vulnerability as shown in Table 4.2. These variables were chosen because they do not have strong correlations with others, except in the case of fault risk and market vulnerability (0.76). Despite this correlation, both metrics were retained since in cases in which a component is associated with failed test cases, fault risk will invariably increase. It cannot

be guaranteed, however, that the same will hold for market vulnerability because failed test cases could occur in devices with low market vulnerability.

As confirmed in interviews with software-test practitioners and as noted in Harman, Jia e Zhang (2015), software testers are unlikely to be concerned solely with a single test objective. Bearing this and the importance of the metrics discussed in the previous section in mind, the formulation of the multi-objective CSP, its objective functions and its variables, and how they are combined into a unique optimization process will be described.

The multi-objective problem consists of two objective functions, with five variables in the first and one in the second. The objective of the optimization process is to select components for unit testing that simultaneously optimize both objective functions, as described below:

$$\max b = \sum_{i=1}^{N} \left(\frac{cfm_i + rf_i + fp_i + v_i}{4}\right) \cdot x_i \tag{4-3}$$

where cfm_i, rf_i, fp_i, v_i are, respectively the normalized value of: cost of future maintenance, risk of fault, frequency of calls, and market vulnerability of the component i. x_i receives value 1 if the component i is selected, and 0 otherwise.

The benefit is computed using the average among the four variables because the definition of weight for each variable is specific to each problem. Thus, even if it adversely impacts the advantages of SCOUT over the baseline approaches compared in the experimental evaluation, these weights are excluded from the scope of this thesis.

The second objective function is formulated as follows:

$$\min c = \sum_{i=1}^{N} c_i \cdot x_i \tag{4-4}$$

where c_i is the cost to develop unit testing activities for the component i.

Note that a conflict arises between the first and second objective functions once their natures tend to become inversely proportional, i.e., the higher the benefit, the higher the cost. This characteristic enables us to investigate and use multi-objective evolutionary approaches, as presented in Chapter 5.

A description of the automation processes used in this study to extract, process, store, and manipulate data required for CSP optimization follows.

4.3 Automation

Karhu et al. (2009) define automated software testing as the automation of software testing activities including the development and execution of test scripts, verification of testing requirements, and use of automated testing tools. The following sections describe strategies used to automate the use of SCOUT on an Android platform.

4.3.1 Static Metrics

As noted in Section 4.1.2, the value of t is represented by Halstead effort, which represents the amount of effort required to to understand a unit as measured in seconds. The variable to estimate the number of bugs liable to occur in a particular piece of code, *bugs*, is represented by the Halstead bugs metric E in Table 4.4. Both variables were derived from (JHAWK, 2016). The value of t was also used to estimate the time required per component to develop unit testing activities and, consequently, to compute the constraint of the problem as given by the percentage of the total time required for all components.

The Halstead measures are based on four scalar numbers derived directly from a program's source code, as shown in Table 4.3.

Table 4.3: *Four scalar numbers used to compute Halstead effort.*

n1	=	number of distinct operators
n2	=	number of distinct operands
N1	=	total number of operators
N2	=	total number of operands

Table 4.4: *Five derived Halstead measures.*

Measure	Symbol	Formula
Program length	N	N= N1 + N2
Program vocabulary	n	n= n1 + n2
Volume	V	V= N * (LOG2 n)
Difficulty	D	D= (n1/2) * (N2/n2)
Effort	E	E= D * V

Further information regarding these metrics can be found in (JHAWK, 2016).

4.3.2 Dynamic Metrics

This section describes how the dynamic metrics used in the SCOUT model (frequency of calls, fault risk, and market vulnerability) were computed automatically.

Frequency of Calls

Before collecting the dynamic metrics used by SCOUT, the software needs to be prepared according to the platform and technologies used to build it. The collection of

the frequency of calls metric in Android apps is performed in two steps: enabling method profiling and parsing the execution results. There are two ways to enable method profiling on Android (*android.os.Debug* class): $startMethodTracing()$ and $stopMethodTracing()$, which are responsible for starting and stopping the generation of profiling, respectively. The method $startMethodTracing()$ is used to start logging trace files and can be accomplished by inserting the code in the Android app (e.g., the $onCreate()$ method). To stop tracing, the code $stopMethodTracing()$ can be invoked (e.g., by the $onDestroy()$ method). This process is presented in greater detail in (ANDROID. . . , 2016).

Consequently, a trace file will be generated in the Android device or emulator that provides detailed metrics regarding a method, such as the number of calls, execution time, and time spent executing the method. Then, this file is parsed using the traceview tool shipped with Android SDK and a tool we developed to convert smali code[1] into Java code. As a result, a set of detailed data for all methods executed in runtime is generated, as exemplified in Table 4.5.

Table 4.5: *Frequency of Calls after profiling.*

Package	Class	Method	Parameters	Type Return	Number of Calls	Frequency of Calls
com.bottleworks.dailymoney.ui	DesktopActivity	initPasswordProtection	()	void	1	0.05
com.bottleworks.dailymoney.ui	DesktopActivity	initialContent	()	void	11	0.55
com.bottleworks.dailymoney.ui	TestsDesktop	<init >	(Activity)	void	1	0.05
com.bottleworks.dailymoney.ui	TestsDesktop	TestsDesktop	()	void	1	0.05
com.bottleworks.dailymoney.data	SQLiteDataProvider	newAccount	(String,Account)	void	10	0.50
com.bottleworks.dailymoney.context	Contexts	cleanDataProvider	(Context)	void	2	0.1
com.bottleworks.commons.util	CalendarHelper	toDayEnd	(Calendar)	Date	6	0.3
com.bottleworks.dailymoney.data	SQLiteDataProvider	findAccount	(String)	Account	20	1.00
com.bottleworks.commons.util	CalendarHelper	toDayStart	(Calendar)	Date	5	0.25
com.bottleworks.commons.util	CalendarHelper	monthEndDate	(Date)	Date	2	0.10
com.bottleworks.commons.util	CalendarHelper	absMonthEndDate	(Date)	Date	2	0.10
com.bottleworks.commons.util	CalendarHelper	monthStartDate	(Date)	Date	7	0.35
com.bottleworks.commons.util	CalendarHelper	absMonthStartDate	(Date)	Date	2	0.10
com.bottleworks.dailymoney.ui	ReportsDesktop	<init >	(Activity)	void	1	0.05
com.bottleworks.dailymoney.ui	DesktopActivity	initialTab	()	void	1	0.05

In this example, the rows show the consolidated data for each component of the Daily Money App at the method level. As can be seen in the eighth row, findCount was the method most frequently invoked by the user (20 times), Thus, the frequency of calls for this method is 1, and the frequency of calls for the other methods is computed as the value proportional to the most frequently executed method (in this case, findCount). For example, the frequency of calls for the method initialContent executed 11 times is 0.55, i.e., 11/20.

Fault Risk

One of SCOUT's principal advantages is the use of dynamic metrics and market information that come from user interactions. These interactions may be performed by the

[1]The smali code is an intermediate representation of the Dalvik bytecode. Its format is more convenient for static analysis than the original format of the Dalvik bytecode (ZHENG et al., 2012).

testing team or by users themselves. The idea is that all dynamic metrics can be collected automatically from user interactions.

As Android apps were used to validate SCOUT, the Barista tool was used to generate Android UI test cases from user interactions. The test cases are written in Espresso API (Espresso, 2015). Further details regarding Barista may be found in (CHOUDHARY, 2015a).

The test suite generated by Barista was then executed across devices to compute the fault risk and market vulnerability for each component, as described below.

Market Vulnerability

Android fragmentation is a problem that has long concerned software developers. According to the documentation available at the time this study was written, Android versions are distributed among 22 API's, with screen size classified under four categories (small, normal, large, and extra large) and six densities (ldpi, mdpi, hdpi, xhdpi, xxhdpi, and xxxhdpi). There are also nine different sizes of memory. Accordingly, taking just these factors into consideration yields the possibility of 4, 752 different device configurations.

The market vulnerability metric was created to express the vulnerability of a component across devices according to market distribution (ANDROID..., 2015). In sum, the greater the market penetration of a given device configuration, the greater the probability that failures on apps running on such device will affect many users. When a suite of UI test cases written in Espresso API comes from user interactions, test cases are run on a set of devices and execution reports are generated using Spoon 1.1 (Spoon, 2015). The sequential extraction of market vulnerability for a component can be expressed as follows:

- For each device, a list of test cases with failures is compiled.
- For each device with failures, its minimum and maximum associated market is computed.
- The api market (*am*) is calculated as the sum of the API market percentage of these devices (see (ANDROID..., 2015));
- The screen market (*sm*) is calculated as the sum of the screen size and density market percentage of these devices (see (ANDROID..., 2015);
- The minimum market vulnerability can be expressed as the maximum value between *am* and *sm*.
- The maximum market vulnerability can be expressed as the sum of *am* and *sm*.
- The medium market vulnerability is the average between minimum and maximum market vulnerability.

Thus, if a component has a failed execution, the devices in which the execution failed are identified, and the method market vulnerability is defined as the median market vulnerability.

Table 4.6 presents an example in which one component failed in three devices (D3, D5, and D10). In this case, the market vulnerability is 73.65% (the average of 59.6% and 87.7%). The data used to compute the market vulnerability are in Tables 4.7 and 4.8.

Table 4.6: *Example of method market vulnerability.*

Device	API	Screen Size	Density	API Market	Screen / Density Market
D3	21	Xlarge	xhdpi	15.9%	0.7%
D5	19	Normal	xhdpi	39.2%	20.9%
D10	18	Normal	mdpi	4.5%	6.5%
			Total	59.6%	28.1%

4.3.3 Device Selection

As previously noted, the Android operating system presents a particular challenge: fragmentation. Android has a large number of versions and thousands of devices that use its platform. Fragmentation is regarded as both a strength and a weakness. On the one hand, Android provides users with a broad spectrum of alternatives. On the other hand, the wide diversity of devices creates a design and testing minefield with hundreds of screen sizes, hardware features, OS versions, input gestures, and just about every other testing scenario one could imagine. Thus developers and testers alike need to make sure that their apps will run properly on the devices they are working with. The task of checking the compatibility of an Android app is generally manual and thus expensive as there are no tools available to completely automate the process. Accordingly, selecting which devices to use to fragmentation issues is an additional problem since dynamic metrics such as fault risk and market vulnerability are dependent on cross-device execution.

Android's official website provides monthly data on the most active devices in the Android and Google Play markets, including such characteristics as version, screen size, and density. Table 4.7 shows the relative number of devices running a given Android version, and Table 4.8 shows the relative number of devices that have a particular screen configuration, defined by a combination of screen size and density, during a 7-day period ending on September 2, 2015 (ANDROID. . . , 2015).

According to Kuhn, Kacker e Lei (2010), pairwise ensures that all possible pairs of parameter values are covered by at least one combination. This approach is known for its two-way combinations. In Kuhn e Reilly (2002), it is suggested that more than 95%

Table 4.7: *Distribution of versions on Android platform.*

Version	Codename	API	Distribution
2.2	Froyo	8	0.2%
2.3.3 - 2.3.7	Gingerbread	10	4.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	3.7%
4.1.x	Jelly Bean	16	12.1%
4.2.x		17	15.2%
4.3		18	4.5%
4.4	KitKat	19	39.2%
5.0	Lollipop	21	15.9%
5.1		22	5.1%

Table 4.8: *Market share on Android platform.*

Total	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	3.3%						3.3%
Normal		6.5%	0.1%	41%	20.9%	15.6%	84.1%
Large	0.3%	4.8%	2.3%	0.6%	0.6%		8.6%
Xlarge		3.0%		0.3%	0.7%		4.0%
Total	3.6%	14.3%	2.4%	41.9%	22.2%	15.6%	

of errors in the studied software would be detected by test cases that cover all four-way combinations of values.

To automate this task, the adoption of Orthogonal Array Technique (OATS) to generate a small number of configurations with a high level of coverage is proposed.

Few tools are available to explore OATS in a practical way. The solution was implemented in addition to Automated Combinatorial Testing for Software (ACTS) (YU et al., 2013), which was developed by the National Institute of Standards and Technology (NIST, 2016). By way of illustration, Table 4.9 presents the configuration generated by ACTS using an OATS algorithm based on the following parameters: API version, screen size, and density. The table also presents the minimum and maximum expected market for each configuration, according to the data in Tables 4.7 and 4.8. Constraints were added to enable the algorithm to represent screen configurations not being used by any device on the market, e.g., Small/mdpi.

Section 5 provides a view of the impact of the vulnerability of an app on the Android market based on the execution of user interactions across devices.

4.4 Optimization Process

Once all required metrics are collected, an algorithm is applied to solve the component selection problem. The algorithms used in the present study can be divided in two groups: single- and multi-objective. The following chapter will describe in greater detail the algorithms applied in CSP and the input for both single- and multi-objective algorithms.

Table 4.9: *Configurations suggested by OATS.*

Setup#	API	Screen	Density	Width	Height	dpi	Display	Real Device	Market			
									API	Screen	Min	Max
1	10	Large	ldpi	480	800	120	7.80	Motorola (XT317)	7.80%	0.50%	7.80%	8.30%
2	10	XLarge	mdpi	1280	800	160	9.40	Samsung Tablet SGH	7.80%	3.50%	7.80%	11.30%
3	10	Normal	hdpi	800	480	240	3.90	Nexus One	7.80%	38.70%	38.70%	46.50%
4	10	Large	xhdpi	1200	1920	320	7.10	Nexus 7	7.80%	0.60%	7.80%	8.40%
5	10	Normal	xxhdpi	1920	1080	480	4.60	Google Nexus 5	7.80%	15.80%	15.80%	23.60%
6	15	Small	ldpi	320	240	120	3.30	LG Optimus L3 II	6.70%	4.60%	6.70%	11.30%
7	15	Normal	mdpi	480	320	160	3.60	LG-C800G	6.70%	8.40%	8.40%	15.10%
8	15	XLarge	hdpi	1920	1200	240	9.40	Sony Xperia Tablet Z	6.70%	0.30%	6.70%	7.00%
9	15	Large	xhdpi	1200	1920	320	7.10	Nexus 7	6.70%	0.60%	6.70%	7.30%
10	15	Normal	xxhdpi	1920	1080	480	4.60	Google Nexus 5	6.70%	15.80%	15.80%	22.50%
11	16	Small	ldpi	320	240	120	3.30	LG Optimus L3 II	19.20%	4.60%	19.20%	23.80%
12	16	Large	mdpi	1152	720	160	8.50	Samsung Galaxy S4	19.20%	5.10%	19.20%	24.30%
13	16	XLarge	hdpi	1920	1200	240	9.40	Sony Xperia Tablet Z	19.20%	0.30%	19.20%	19.50%
14	16	Normal	xhdpi	1280	720	320	4.60	Google Galaxy Nexus	19.20%	18.90%	19.20%	38.10%
15	16	Normal	xxhdpi	1920	1080	480	4.60	Google Nexus 5	19.20%	15.80%	19.20%	35.00%
16	17	Small	ldpi	320	240	120	3.30	LG Optimus L3 II	20.30%	4.60%	20.30%	24.90%
17	17	XLarge	mdpi	1280	800	160	9.40	Samsung Tablet SGH	20.30%	3.50%	20.30%	23.80%
18	17	Large	hdpi	1280	720	240	6.10	SAMSUNG-SGH-I527	20.30%	0.60%	20.30%	20.90%
19	17	XLarge	xhdpi	2560	1600	320	9.40	Nexus 10	20.30%	0.60%	20.30%	20.90%
20	17	Normal	xxhdpi	1920	1080	480	4.60	Google Nexus 5	20.30%	15.80%	20.30%	36.10%
21	18	Small	ldpi	320	240	120	3.30	LG Optimus L3 II	6.50%	4.60%	6.50%	11.10%
22	18	Normal	mdpi	480	320	160	3.60	LG-C800G	6.50%	8.40%	8.40%	14.90%
23	18	Large	hdpi	1280	720	240	6.10	SAMSUNG-SGH-I527	6.50%	0.60%	6.50%	7.10%
24	18	XLarge	xhdpi	2560	1600	320	9.40	Nexus 10	6.50%	0.60%	6.50%	7.10%
25	18	Normal	xxhdpi	1920	1080	480	4.60	Google Nexus 5	6.50%	15.80%	15.80%	22.30%
26	19	Small	ldpi	320	240	120	3.30	LG Optimus L3 II	39.10%	4.60%	39.10%	43.70%
27	19	Normal	mdpi	480	320	160	3.60	LG-C800G	39.10%	8.40%	39.10%	47.50%
28	19	Large	hdpi	1280	720	240	6.10	SAMSUNG-SGH-I527	39.10%	0.60%	39.10%	39.70%
29	19	XLarge	xhdpi	2560	1600	320	9.40	Nexus 10	39.10%	0.60%	39.10%	39.70%
30	19	Normal	xxhdpi	1920	1080	480	4.60	Google Nexus 5	39.10%	15.80%	39.10%	54.90%
31	21	Small	ldpi	320	240	120	3.30	LG Optimus L3 II	0.10%	4.60%	4.60%	4.70%
32	21	Normal	mdpi	480	320	160	3.60	LG-C800G	0.10%	8.40%	8.40%	8.50%
33	21	Large	hdpi	1280	720	240	6.10	SAMSUNG-SGH-I527	0.10%	0.60%	0.60%	0.70%
34	21	XLarge	xhdpi	2560	1600	320	9.40	Nexus 10	0.10%	0.60%	0.60%	0.70%
35	21	Normal	xxhdpi	1920	1080	480	4.60	Google Nexus 5	0.10%	15.80%	15.80%	15.90%
36	10	Small	ldpi	320	240	120	3.30	LG Optimus L3 II	7.80%	4.60%	7.80%	12.40%

CHAPTER 5

Evaluation

This section describes the experimental design for evaluating SCOUT. To assess its utility, nine Android apps were were used. The evaluation, which compares the efficiency and efficacy of a set of algorithms, is based on a user study. The study, conducted under diverse conditions, addressed the following research questions:

RQ1 - Which solver is most appropriate for use in a situation in which cost and benefit have equal weight of importance for the specialist?

RQ2 - What is the impact of using SCOUT in contexts in which:

[**RQ2.1**] - cost and benefit have the same weight of importance for the specialist?

[**RQ2.2**] - product quality has a higher priority than testing cost?

[**RQ2.3**] - testing cost has a higher priority than product quality?

RQ3 - How effective is SCOUT in selecting the most significant components in terms of market relevance?

The remainder of the chapter describes the study's subjects, user study, and evaluative findings.

5.1 Subjects

As noted in Section 1.1, the Android platform was used to validate SCOUT. The market for Android is expanding but the operating system presents a particular challenge: fragmentation, as it is run in numerous versions on countless devices in all shapes and sizes. Nine real-world Android apps available in F-DROID (2016) were selected for use in the study in light of their popularity, high number and range of installations according to Google Play Store (2016), categories, complexity, and sizes. As open-source apps, they are accessible to all researchers. To prune the search space and thereby reduce the complexity of optimization, only methods with a frequency of execution greater than zero (as it is not productive to generate test cases for methods that are seldom, if ever,

used), and a cyclomatic complexity greater than two were used. The pruned search space is depicted in Figure 5.1, and the remaining number of methods under different time constraints (1%, 5%, 10%, and 20% of the total available time for unit testing activities) in Figure 5.2.

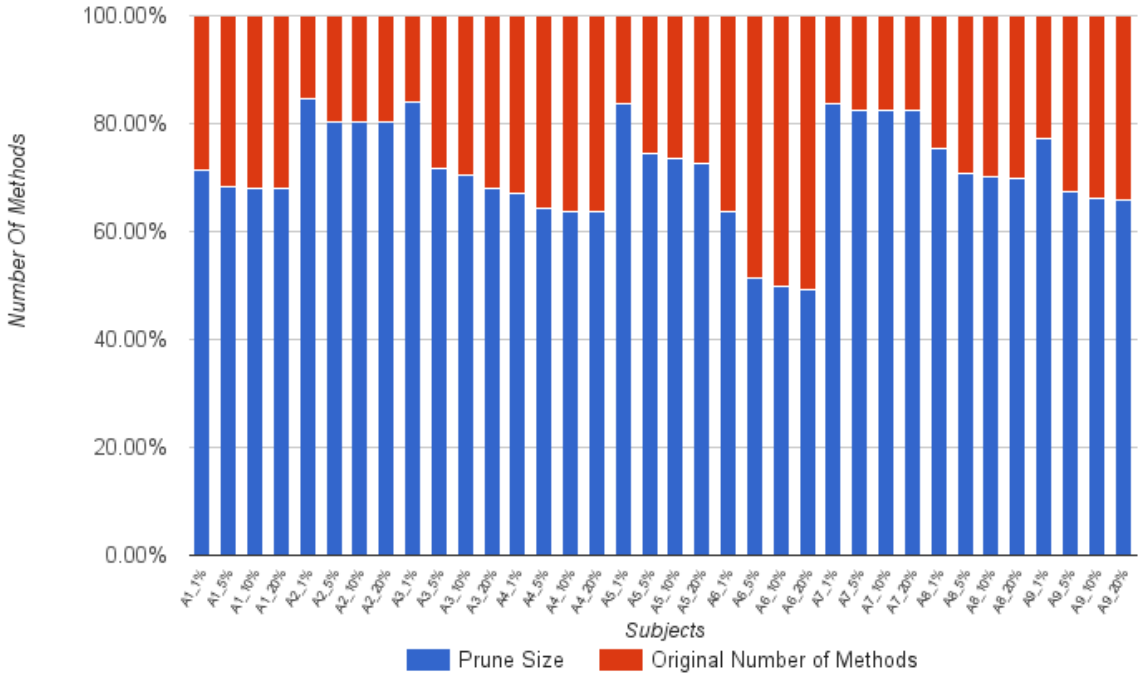

Figure 5.1: *Prune size in the subjects for each time constraint.*

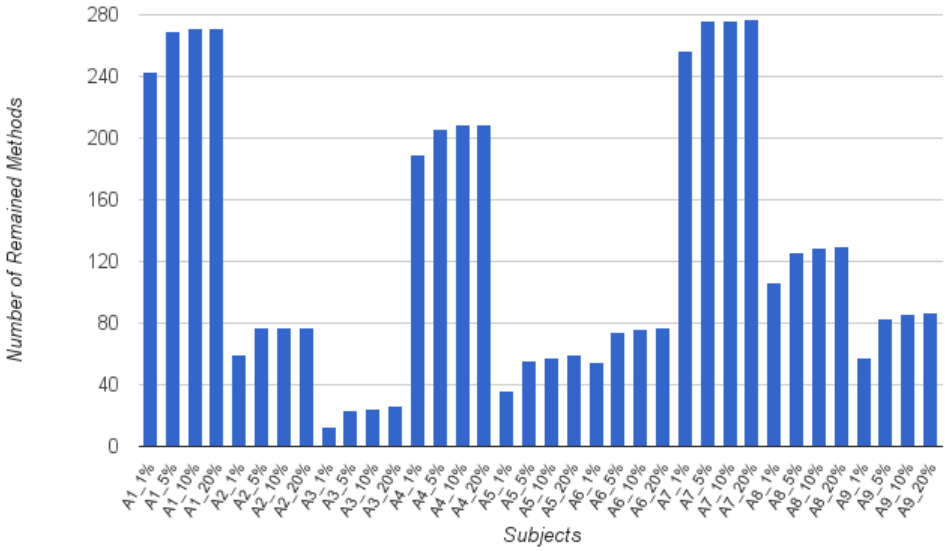

Figure 5.2: *Number of methods after pruning the search space.*

Table 5.1: *Description of experimental subjects.*

Subject Name	APP Name	Category	Min Installs	Max Installs	Original Number of Methods	Number of Methods (After Pruning)	Number of Test Cases
A1	Daily Money	Finance	500,000	1,000,000	849	271	28
A2	Alarm Klock	Tools	500,000	1,000,000	393	79	30
A3	Simple C25K	Health & Fitness	50,000	10,000	81	27	27
A4	EP Mobile	Medical	50,000	100,000	575	209	30
A5	BeeCount Knitting Counter	Productivity	10,000	50,000	220	60	30
A6	Bodhi Timer	Lifestyle	10,000	50,000	152	77	30
A7	andFHEM	Personalization	10,000	50,000	1577	277	15
A8	Xmp Mod Player	Music & Audio	10,000	50,000	432	130	15
A9	World Clock & Weather Widget	Travel & Local	50,000	100,000	254	87	15
Total	–	–	1,190,000	2,410,000	4,483	1,217	220

5.2 User Study

To avoid study bias in defining application use, 17 graduate students at four universities in three countries and seven software professionals were requested to write natural language test cases (NLTCs) by defining actions and assertions, yielding 220 NLTCs. An example is shown in Appendix B. Subsequently, UI test cases in Espresso API (Espresso, 2015) corresponding to these NLTCs were generated using Barista (CHOUDHARY, 2015a), enabling the app's posterior instrumentation and profiling to collect all dynamic metrics, using LG G FLEX running API level 19. To compute Android market vulnerability, seven devices were used [LG G Flex (D1), Motorola Moto X (D2), HTC One M8 (D3), Sony Xperia Z3 (D4), Samsung Galaxy S5 (D5), Nexus 5 (D6), and LG G3 (D7)].

5.3 Experimental Design

Considering that every method requires time t to have its unit testing developed and that an app has n methods, the total time required to create unit testing for the entire application is given by the product of t times n. Given the dynamic Android market, the time available for testing activities is relatively brief. Several software application developers were interviewed and asked the testing time span needed for a new release. Their responses indicate that generally requires not no more than 20% of the total time required to test the entire app. Accordingly, the time constraints used in the study were defined as the following percentages of the total time required to test the entire app: 1%, 5%, 10%, and 20%. Due to the random nature of the evolutionary approaches, each test was conducted 30 times for the evolutionary algorithms (GA, SPEA_II, NSGA_II, and NSGA_III) and the average among the best results was used for comparison in accordance with the parameters customarily cited in the literature for evolutionary algorithms: $population\_size = 200$, $number\_maximum\_of\_evaluations = 200,000$, $crossover\_rate = 0.85$, $mutation\_rate = 0.01$, and tournament as the selection operator.

5.4 Analysis of RQ1

To answer RQ1 (Which solver is most appropriate for use in a situation in which cost and benefit have equal weight of importance for the specialist?), a comparison was made using seven algorithms as baselines.

Random (R): Random technique is widely used as a basic baseline in Search Based Software Engineering for comparisons in testing and, in some cases, can generate potential solutions at a low computational cost.

Constructivist Heuristic (CH): Also known as the greedy algorithm, this technique compiles the set of units for testing incrementally, determining the next unit that will provide the best cost-benefit value, with cost corresponding to the time a unit requires for testing and benefit defined by the objective functions described in Section 4.2.

Gurobi (G): A free academic licensed Gurobi tool (OPTIMIZATION et al., 2015) was used as a deterministic technique in the ur baseline. Gurobi provides a mixed-integer programming solver with an extensive panoply of additional methods, such as branch variable selection.

Genetic Algorithm (GA): This evolutionary meta-heuristic is based on Darwin's theory of natural selection. It was run as a mono-objective solver, according the same weight to each objective function. The fitness function is defined as the average of the fitness of the five objective functions.

Multi-objective Algorithms: Two of the most frequently used multi-objective meta-heuristics algorithms: SPEA_II (ZITZLER et al., 2001), and NSGA_II (DEB et al., 2002) were used. Recent studies (ISHIBUCHI; AKEDO; NOJIMA, 2014; YUAN; XU; WANG, 2014) have suggested that SPEA_II as NSGA_II degrade severely when the number of objectives exceeds four. As SCOUT's formulation involves five variables, NSGA_III was also experimented.

As some algorithms are capable of optimizing only one objective function, while others can optimize several, their formulation of objective functions differ. For the single-objective solvers R, CH, G, and GA, the fitness of each solution is computed as follows:

$$f = (wb \cdot b) - (wc \cdot c) \tag{5-1}$$

where:
wb: is the weight of importance of the benefit;
wc: is the weight of importance of the cost.

In these single-objective algorithms, the benefit of selecting a component is computed as the ratio between its cost (c) and benefit (b), i.e., b/c (see Section 4.2).

As SPEA_II, NSGA_II, and NSGA_III are multi-objective algorithms, the optimization process included the two objective functions in Section 4.2. As the algorithm was run 30 times for each test, the fitness value used for these solvers in the comparison was the average of the results of all executions. For each execution, the solution with the highest value of $(wb \cdot b) - (wc \cdot c)$ in the Pareto Front was selected.

To answer RQ1, wb = 1 and wc = 1 was used.

A comparison among these algorithms of two characteristics was elaborated: efficiency as measured by the variable $time$, which represents the time in seconds that each algorithm takes to find a solution, and efficacy as measured by analysis of the $fitness$ and *residual* (percentage of available time unused by the algorithm) of each solver.

According to the efficiency comparison, CH was the most efficient algorithm and NSGA_III the least. As can be seen in Table 5.2, NSGA_II and NSGA_III took on average 4.93 seconds and 7.93 seconds, respectively, to find their best solutions, while Gurobi took just 0.03 seconds. While efficiency is an important criterion in selecting an algorithm, the study's analysis also took into consideration the context of mobile app testing. As planning for unit testing does not mandate a time faster than that required by NSGA_III to find its solution, the efficiency of all algorithms can be considered as adequate adding weight to other considerations such as efficacy.

Table 5.2: *Baseline efficiency.*

Constraint	R	CH	G	GA	SPEA_II	NSGA_II	NSGA_III
1%	0.71	0.01	0.02	10.71	23.16	4.27	7.37
5%	0.82	0.01	0.03	15.34	22.60	4.94	8.42
10%	0.85	0.01	0.03	14.44	22.32	5.74	7.95
20%	1.03	0.01	0.04	12.04	22.76	4.76	7.98
Average	0.85	0.01	0.03	13.13	22.71	4.93	7.93

As Gurobi attained the best efficacy among the single-objective algorithms, its efficacy was used as a comparative reference with the other baseline algorithms, including the multi-objective ones. The fitness used to compare considered for the comparison with the latter was the average fitness found in 30 executions. For the Randomly algorithm, the best fitness found in 200,000 executions was used (the same number used for GA and the multi-objective algorithms) .

Table 5.3 compares Gurobi's efficacy with the other baselines. The single-objective algorithms R and GA were outperformed consistently. Although CH attained the second best efficiency among all baseline algorithms and had the same efficacy as G in 75% of the testing situations, in highly restrictive ones (1%), its efficacy was inferior to the three multi-objective algorithms.

Among multi-objective algorithms, NSGA_II had the best efficacy, with equivalence to Guorbi in 86.11% of the testing situations, followed by SPEA_II with 72.22%, and NSGA_III with 69.44%, which was designed to work with four or more objectives. Although there are five objective functions in SCOUT's formulation, NSGA_III was used

with only two: maximizing the benefit and minimizing the cost. For this reason, we were unable to explore all its benefits. As stated in Chapter 6, future could use NSGA_III to determine its effectiveness in evaluating all of SCOUT's objective functions.

Although the multi-objective algorithms used in this study are based on an evolutionary approach subject to randomness and although their average fitness was less than Gurobi, they attained the maximum fitness a reasonable number of times among their 30 executions, as can be seen in Table 5.3. Accordingly, some advantages in using multi-objective algorithms' evolutionary approach follow.

Table 5.3: *Gurobi efficacy against the others baselines.*

Subject	Constraint	Gurobi fitness against:						Number of times the maximum fitness was found		
		R	CH	GA	SPEA_II	NSGA_II	NSGA_III	SPEA_II	NSGA_II	NSGA_III
A1	1%	25.43%	2.39%	2.78%	0.11%	0.02%	0.11%	6	19	13
	5%	33.71%	0.00%	2.06%	0.11%	0.09%	0.20%	7	8	3
	10%	32.62%	0.00%	1.66%	0.09%	0.00%	0.03%	8	30	15
	20%	27.78%	0.00%	19.56%	0.07%	0.00%	0.05%	10	30	9
A2	1%	2.08%	1.41%	0.00%	0.00%	0.00%	0.00%	30	30	30
	5%	8.19%	0.54%	3.55%	0.05%	0.00%	0.00%	30	30	28
	10%	9.38%	0.00%	17.37%	0.03%	0.00%	0.00%	17	30	29
	20%	9.97%	0.00%	11.39%	0.00%	0.00%	0.02%	28	30	23
A3	1%	3.47%	5.48%	0.10%	0.00%	0.00%	0.00%	30	30	30
	5%	2.16%	0.00%	3.80%	0.00%	0.00%	0.00%	30	30	30
	10%	2.52%	0.00%	4.73%	0.00%	0.00%	0.00%	30	30	30
	20%	0.00%	0.00%	2.73%	0.00%	0.00%	0.00%	30	30	30
A4	1%	15.40%	8.54%	4.12%	0.00%	0.04%	0.12%	28	25	16
	5%	32.66%	3.13%	4.04%	0.09%	0.02%	0.23%	13	25	8
	10%	30.30%	0.00%	13.71%	0.06%	0.01%	0.09%	11	17	9
	20%	28.15%	0.00%	19.69%	0.02%	0.00%	0.00%	20	30	26
A5	1%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	30	30	30
	5%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	30	30	30
	10%	1.95%	0.00%	0.22%	0.00%	0.00%	0.00%	30	30	30
	20%	8.76%	0.00%	9.61%	0.00%	0.00%	0.01%	30	30	27
A6	1%	0.95%	0.00%	0.00%	0.00%	0.00%	0.00%	29	30	30
	5%	3.92%	0.00%	0.00%	0.11%	0.00%	0.00%	27	30	30
	10%	4.72%	0.00%	3.37%	0.00%	0.00%	0.00%	30	30	30
	20%	6.59%	0.00%	12.04%	0.01%	0.00%	0.00%	25	30	26
A7	1%	0.89%	0.00%	0.76%	0.00%	0.00%	0.00%	30	30	30
	5%	1.52%	0.00%	2.10%	0.00%	0.00%	0.00%	30	30	26
	10%	3.95%	0.00%	6.00%	0.00%	0.00%	0.00%	30	30	26
	20%	6.76%	0.00%	15.49%	0.00%	0.00%	0.02%	30	30	15
A8	1%	12.40%	1.79%	0.25%	0.00%	0.00%	0.00%	30	30	30
	5%	22.31%	0.00%	0.08%	0.00%	0.00%	0.01%	30	30	26
	10%	15.33%	0.00%	10.39%	0.00%	0.00%	0.01%	30	30	28
	20%	12.23%	0.00%	21.54%	0.00%	0.00%	0.00%	30	30	26
A9	1%	2.08%	4.79%	4.85%	0.00%	0.00%	0.00%	30	30	30
	5%	9.92%	4.27%	11.72%	0.00%	0.00%	0.00%	30	30	30
	10%	12.11%	0.00%	14.06%	0.00%	0.00%	0.00%	30	30	30
	20%	11.33%	0.00%	16.79%	0.00%	0.00%	0.00%	30	30	30

Accordingly, as there is no time pressure regarding the duration of optimization process in selecting components for unit testing, NSGA_II can be executed a reasonable number of times in such way we can select its best solution, once that the efficacy is more important than the algorithm efficiency (of course under plausible time).

Table 5.4: *Average residual for each scenario of constraint.*

Constraint	R	CH	GA	G	SPEA_II	NSGA_II	NSGA_III
1%	10.33%	13.47%	11.48%	17.39%	17.43%	17.36%	17.48%
5%	3.91%	13.12%	8.61%	14.48%	15.07%	14.35%	14.91%
10%	2.42%	35.29%	10.61%	35.29%	34.83%	35.09%	35.22%
20%	6.40%	55.86%	10.26%	55.86%	56.11%	55.86%	56.00%

Efficiency: As stated previously, although the multi-objective solvers have take more time to deliver a set of components for unit testing, these times are within those required in real-life practice and are, accordingly, sufficient to justify their consideration.

Comparing Single- and Multi-objective Algorithms: In comparing a specific heuristic such as Gurobi with a meta-heuristic such as NSGA_II, two key differences are noteworthy: their input and output. Gurobi works as a single-objective solver, which means it can only optimize one objective function in each execution. As the study has two functions to satisfy simultaneously, viz., maximizing the benefit and minimizing the cost of unit testing, a single objective computing a special function is defined as the difference between benefit and cost $(b - c)$. Thus before Gurobi's optimization, this function is computed for it. On the other hand, the multi-objective solvers can optimize two or more objective functions simultaneously. Thus, the study used as input for these kind of solvers two arrays of n elements: the first representing the benefit and the second, the cost of unit testing.

Gurobi's output comprises a set of units for testing that maximize fitness and do not violate constraints. The multi-objective solvers deliver one or more solutions, i.e., feasible components for testing, each with its own characteristics. This set of solutions is known as a Pareto Front and affords the software specialist the option of deciding which best meets the specific needs.

Flexibility: Gurobi, a commercial algorithm, was used not as a meta-heuristic in the study but as a heuristic designed to solve a Knapsack problem. It uses a set of closed proprietary optimization resources to achieve this result. With evolutionary algorithms openly known in the literature and many implementations available in cost-free frameworks, there is the flexibility to adapt these algorithms to specific contexts.

5.5 Analysis of RQ2

To answer RQ2 (What is the impact of using SCOUT in various contexts?), component selections made in the following strategies was examined.

Strategy 1 (S1). The primary goal of Strategy 1 is to prioritize for selection for unit testing those components with a high anticipated number of bugs. This strategy was selected as representative of a defect prediction model based on static metrics. Halstead Bugs implemented by JHawk tool (JHAWK, 2016) was used as the metric to represent the anticipated number of bugs. The benefit (b) is computed as follows:

$$b = \sum_{i=1}^{n} e_i \cdot x_i \tag{5-2}$$

where e_i is the normalized number of expected bugs of the component i, and x_i receives value 1 if the component i is selected, and 0 otherwise.

Strategy 2 (S2). To represent a fault localization model based on dynamic metrics, the primary goal of Strategy 2 is to prioritized for selection for unit testing prioritizing those components with a high fault risk. The metric adopted here was based on the coefficient of suspiciousness implemented by Tarantula tool (JONES; HARROLD; STASKO, 2002). The benefit (b) is computed as follows:

$$b = \sum_{i=1}^{n} r_i \cdot x_i \tag{5-3}$$

where r_i is the normalized risk of fault (suspiciousness) of the component i.

Strategy 3 (S3). Strategy 3 represents SCOUT, whose benefit (b) is computed as follows:

$$b = \sum_{i=1}^{n} (\frac{cfm_i + rf_i + fp_i + v_i}{4}) \cdot x_i \tag{5-4}$$

where cfm_i, rf_i, fp_i, v_i are, respectively, the normalized value of cost of future maintenance, risk of fault, frequency of method calls, and market vulnerability of the component i.

To compare these three strategies, 63 distinct scenarios of prioritization were constructed to simulate the broad spectrum of diverse realities present in the software industry and to measure the impact of using S1, S2, and S3 in these scenarios. The scenarios were based on six criteria: defect proneness (DP), fault risk (RF), market vulnerability (MV), frequency of profiling (P), cyclomatic complexity (CC), and cost of future maintenance (CFM). In the first scenario (S01), the priority is selecting components with high rate of defect proneness rate for unit testing. In the second (S02), it is selecting components with high degree of suspiciousness. This rule is followed until the sixth scenario, according to the other criteria. From the seventh scenario, an an arrangement incorporating all criteria combinations among them is generated, as presented in Table 5.5. For instance, in the eighth scenario, the prioritized components are defined as those with a high rate of defect proneness and vulnerability. The normalized value of the criteria considered were used to construct the scenarios.

As a result of optimization, each strategy generates a list of components for unit testing. The efficacy of each strategy in a scenario is measured according to that attained in Algorithm 1. The fitness of each component in the list is computed according to the scenario, and the efficacy is the sum of the fitness of all components.

Table 5.5: *Criteria used to construct scenarios.*

Scenario	DP	RF	MV	P	CC	CFM
S01	x					
S02		x				
S03			x			
S04				x		
S05					x	
S06						x
S07	x	x				
S08	x		x			
S09	x			x		
S10	x				x	
S11	x					x
S12		x	x			
S13		x		x		
S14		x			x	
S15		x				x
S16			x	x		
S17			x		x	
S18			x			x
S19				x	x	
S20				x		x
S21					x	x
S22	x	x	x			
S23	x	x		x		
S24	x	x			x	
S25	x	x				x
S26	x		x	x		
S27	x		x		x	
S28	x		x			x
S29	x			x	x	
S30	x			x		x
S31	x				x	x
S32		x	x	x		
S33		x	x		x	
S34		x	x			x
S35		x		x	x	
S36		x		x		x
S37		x			x	x
S38			x	x	x	
S39			x	x		x
S40			x		x	x
S41				x	x	x
S42	x	x	x	x		
S43	x	x	x		x	
S44	x	x	x			x
S45	x	x		x	x	
S46	x	x		x		x
S47	x	x			x	x
S48	x		x	x	x	
S49	x		x	x		x
S50	x		x		x	x
S51	x			x	x	x
S52		x	x	x	x	
S53		x	x	x		x
S54		x	x		x	x
S55		x		x	x	x
S56			x	x	x	x
S57	x	x	x	x	x	
S58	x	x	x	x		x
S59	x	x	x		x	x
S60	x	x		x	x	x
S61	x		x	x	x	x
S62		x	x	x	x	x
S63	x	x	x	x	x	x

The goal in RQ2.1, RQ2.2, and RQ2.3 is to simulate real-life contexts in the software industry. As previously noted, in RQ2.1, the idea is determine the impact of

Algorithm 1: Evaluation of solution of a strategy.

```
Input  : LA: List of selected components of a strategy
Input  : wb: weight of benefit
Input  : wc: weight of cost
Output: f: Fitness of a solution in a scenario
1 begin
2     foreach i ∈ LA do
3         f ← f + ((b_i · wb) − (c_i · wc))
4 return F
```

using SCOUT in a context in which cost and benefit have the same weight of importance, whereas in RQ2.2, its impact is determined in a context in which product quality has a higher priority than testing cost. Finally in RQ2.3, the context is one in which time constraints for unit testing prioritize test cost over product quality. To evaluate the behavior of the strategies under the priorities stipulated in RQ2.1, RQ2.2, and RQ2.3, the final fitness was computed as shown in Algorithm 1, assuming the weights for cost and benefit provided in Table 5.6. according to

Table 5.6: *Weights for cost and benefit in RQ2.*

Weight	Research Question	Cost	Benefit
WS1	RQ2.1	50.00%	50.00%
WS2	RQ2.2	20.00%	80.00%
WS3	RQ2.3	20.00%	80.00%

Based on Algorithm 1, the fitness of each solution delivered by each strategy is computed for all 63 scenarios. Figure 5.3, compares strategies S3 and S1 under weights WS1, WS2, and WS3. Each line in the graph represents the relative fitness of S3, the SCOUT strategy, compared with that of S1, which prioritizes the selection of the most defect prone components for testing.

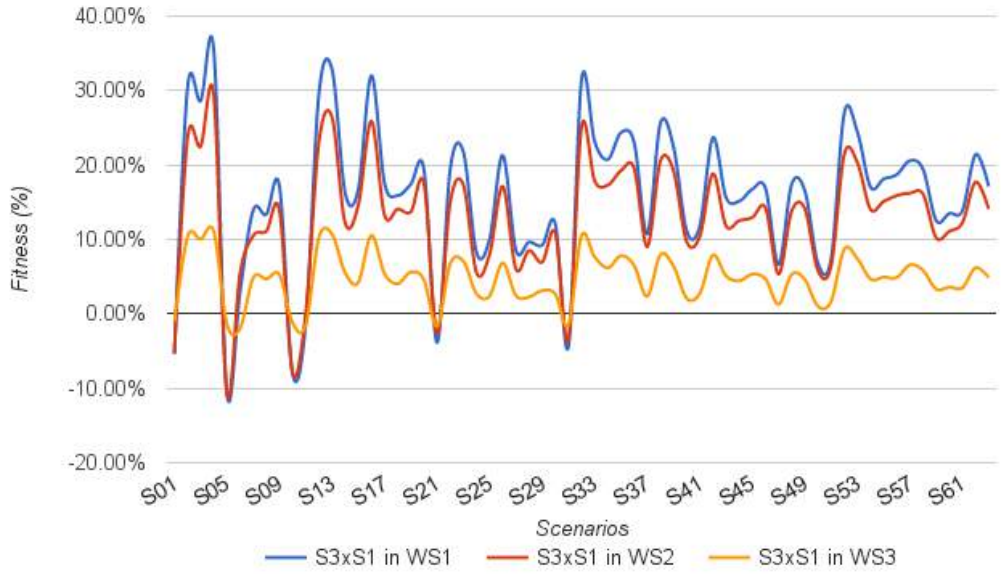

Figure 5.3: *Fitness comparison S3/S1 in all 63 scenarios.*

S3 attained its best results in scenarios in which cost and where benefit and cost have the same priority (WS1), exceeding S1's fitness in 90.48% of the scenarios, with an

average 15.82% above S1's, with a standard deviation of 9.90%. S3's fitness also exceeded S1's in 92.06% of scenarios in which product quality was prioritized over testing cost (WS2), with an average 12.87% higher than S1's, with a standard deviation of 8.07%. In the context that prioritizing testing cost over product quality (WS3), S3 's fitness exceed that of S1 in 88.89% of the scenarios, with an average 4.75% above S1's, with a standard deviation of 3.31%. As can be seen in the Table 5.7, the fitness of S1 only exceeded that of S3 in scenarios which prioritized defect proneness, as presented in the Table 4.2.

Table 5.7: *Scenarios in which S3's fitness was exceeded by S1's.*

Comparison	Strategy 3 x Strategy 1			Strategy 3 x Strategy 2		
Research Question	RQ2.1	RQ2.2	RQ2.3	RQ2.1	RQ2.2	RQ2.3
Scenarios where S3 was overcome	S01, S05, S10, S11, S21, and S31	S01, S05, S10, S21, and S31	S01, S05, S06, S10, S11, S21, and S31	S02, S07, S14, S15, S24, S25, S37, and S47	S02, S07, S12, S14, and S33	S02, S07, S12, S14, S24, and S33

Regarding to the comparison between S3 and S2, Figure 5.4 depicts the percentage of scenarios in which S3 exceeded S2 in terms of fitness. As can be seen in Table 5.8, S3's fitness exceeded those of S2 in 89.95% of scenarios, averaging RQ2.1, RQ2.2, and RQ2.3. As can be seen in Table 5.7, the scenarios in which S2's fitness exceeded S3's were those that prioritized fault risk in selecting components for testing.

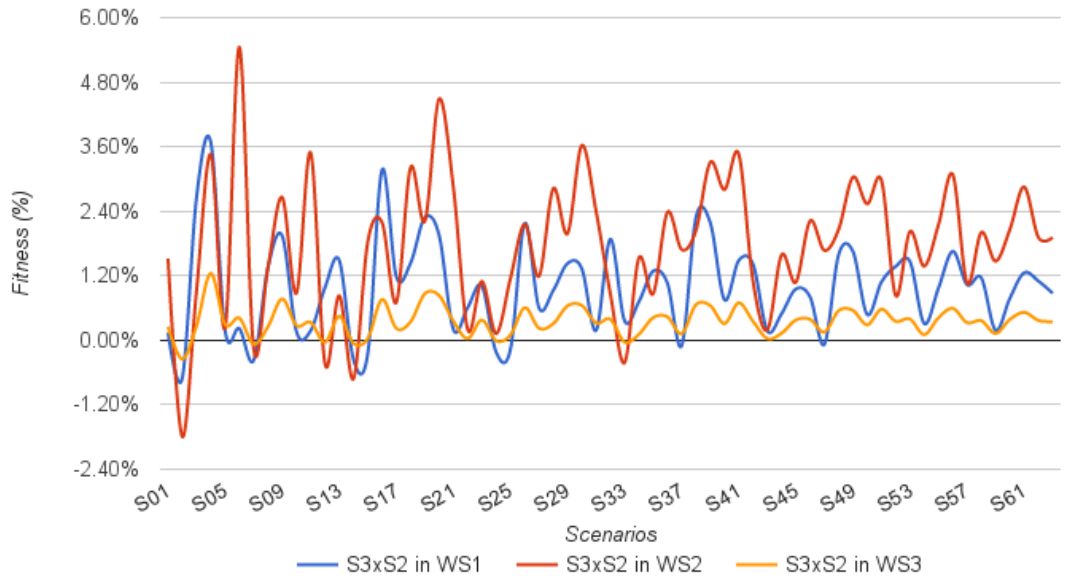

Figure 5.4: *Fitness comparison S3/S2 in all 63 scenarios.*

These findings confirm the hypothesis that the the SCOUT strategy has a greater overall capacity in different real-world contexts, as represented by RQ2.1, RQ2.2, RQ2.3, and in distinct scenarios (S01 to S63), as shown in Table 5.8. Although S3 outperformed S1 and S2 in all three contexts (WS1, WS2, and WS3), it attained a low average in both WS1 and WS3 with average values of 0.98% and 0.35%, respectively, in its comparison

Table 5.8: *Performance of S1, S2, and S3 in RQ2.*

Comparison	Weight	Count	Avg.	SD
S3 x S1	WS1	90.48%	15.82%	9.90%
S3 x S2	WS1	87.30%	0.98%	0.88%
S3 x S1	WS2	92.06%	12.87%	8.07%
S3 x S2	WS2	92.06%	1.77%	1.28%
S3 x S1	WS3	88.89%	4.75%	3.31%
S3 x S2	WS3	90.48%	0.35%	0.27%

with S2. In both these contexts, S3's impact decreases when cost is weighed equal with or greater than benefit per Table 5.6.

The study sought to evaluate the efficacy of each strategy under the different time constraints shown in Table 5.9. While S3 outperformed S1 and S2 under all constraints for WS1, WS2, and WS3, when a relaxation of constraint of 5% to 10% occurred, its average fitness increased. Surprisingly, however, Table 5.9 indicates that S3's fitness decreases in its comparison with of S2 under a constraints of 5% and 10% in WS2.

Table 5.9: *Strategy performance under various time constraints.*

Research Question	Constraint	Count		AVG		Std Dev	
		S3xS1	S3xS2	S3xS1	S3xS2	S3xS1	S3xS2
WS1	1%	88.89%	79.37%	14.435%	1.149%	8.354%	1.144%
	5%	90.48%	85.71%	17.597%	1.583%	10.041%	1.502%
	10%	90.48%	87.30%	14.954%	0.624%	9.735%	0.690%
	20%	90.48%	90.48%	16.313%	0.581%	11.825%	0.599%
WS2	1%	90.48%	71.43%	13.766%	1.473%	8.168%	2.142%
	5%	90.48%	93.65%	13.950%	1.703%	7.908%	1.248%
	10%	90.48%	98.41%	13.242%	1.176%	8.525%	0.602%
	20%	92.06%	74.60%	10.539%	2.736%	8.770%	3.304%
WS3	1%	88.89%	96.83%	4.781%	0.403%	3.328%	0.251%
	5%	88.89%	80.95%	4.831%	0.251%	3.433%	0.282%
	10%	88.89%	87.30%	4.773%	0.387%	3.346%	0.355%
	20%	88.89%	93.65%	4.611%	0.363%	3.154%	0.214%

To understand this phenomenon, consider the data for subject A4 as presented in Table 5.10.

As can be seen in Table 5.10, the unique method selected by S2 under a constraint of 10% that had not been selected under that of 5% was method 2. Methods 3 and 4 were selected by S3 under a constraint of 5%, but not under that of 10%, while methods 7 and 12 were selected by S3 under a constraint of 10% but not under one of 5%. With the constraint relaxed from 5% to 10% methods with a fault risk greater than zero that were not selected under a constraint of 5% became available for selected under one of 10%. These methods had a greater effect on S2's fitness, where RF is the sole variable to compute benefit than on S3's, where it is only one among four (RF, P, CFM, and MV) that compute benefit. Thus, when using these methods with a fault risk greater than zero, S2's fitness tends to increase more than S3's, especially in scenarios whose criteria are directly correlated to RF.

Methods with high cyclomatic complexity but null frequency of profile (P) or those that were not used in failed test cases limit selection for S2 and S3 when the

Table 5.10: *Analysis of subject A4 in WS2.*

Selected Artifacts under 5%		Selected Artifacts under 10%		Method	RF	P	MV	CFM	Cost	Constraint of 5%				Constraint of 10%			
S2	S3	S2	S3							Benefit S3	Fitness S3	Benefit S2	Fitness S2	Benefit S3	Fitness S3	Benefit S2	Fitness S2
x	x	x	x	0	0.0725	0.5510	0.0769	0.0047	0.0427	0.1763	0.1335	0.0725	0.0298	0.1325	0.0897	0.0494	0.0067
x	x	x	x	1	0.0888	0.0941	0.0769	0.0000	0.0454	0.0649	0.0196	0.0888	0.0434	0.0429	-0.0025	0.0619	0.0166
	x	x	x	2	0.0725	0.0588	0.0769	0.0052	0.0421	0.0534	0.0113	0.0725	0.0304	0.0343	-0.0078	0.0496	0.0075
x	x	x		3	0.0888	0.0471	0.0769	0.0193	0.0355	0.0580	0.0225	0.0888	0.0533	0.0393	0.0038	0.0639	0.0284
x	x	x		4	0.0888	0.0471	0.0769	0.0142	0.0384	0.0567	0.0184	0.0888	0.0504	0.0377	-0.0007	0.0633	0.0249
x	x	x	x	5	0.0888	0.0471	0.0769	0.0003	0.0447	0.0533	0.0086	0.0888	0.0441	0.0337	-0.0110	0.0621	0.0174
x	x	x	x	6	0.0725	0.0294	0.0769	0.0028	0.0433	0.0454	0.0021	0.0725	0.0292	0.0277	-0.0156	0.0493	0.0060
x		x	x	7	0.0725	0.0294	0.0769	0.0022	0.0435	0.0453	0.0017	0.0725	0.0290	0.0275	-0.0160	0.0493	0.0058
x	x	x	x	8	0.0888	0.0235	0.0769	0.0012	0.0442	0.0476	0.0034	0.0888	0.0446	0.0292	-0.0149	0.0622	0.0180
	x		x	9	0.0888	0.0235	0.0769	0.5091	0.0054	0.1746	0.1692	0.0888	0.0833	0.1386	0.1332	0.0699	0.0645
x	x	x	x	10	0.0888	0.0235	0.0769	0.0002	0.0450	0.0474	0.0024	0.0888	0.0438	0.0289	-0.0161	0.0620	0.0170
x	x	x	x	11	0.0888	0.0235	0.0769	0.0001	0.0452	0.0473	0.0021	0.0888	0.0436	0.0288	-0.0164	0.0620	0.0168
			x	12	0.0000	0.0020	0.0769	0.0000	0.0453	0.0197	-0.0256	0.0000	-0.0453	0.0067	-0.0386	-0.0091	-0.0544
				14	0.0000	0.0000	0.0000	0.2774	0.0000	0.0693	0.0693	0.0000	0.0000	0.0555	0.0555	0.0000	0.0000
				15	0.0000	0.0000	0.0000	0.0155	0.0399	0.0039	-0.0360	0.0000	-0.0399	-0.0049	-0.0447	-0.0080	-0.0478
				16	0.0000	0.0000	0.0000	0.0030	0.0431	0.0007	-0.0424	0.0000	-0.0431	-0.0080	-0.0512	-0.0086	-0.0517
				17	0.0000	0.0000	0.0000	0.0422	0.0299	0.0106	-0.0193	0.0000	-0.0299	0.0025	-0.0274	-0.0060	-0.0359
				18	0.0000	0.0000	0.0000	0.0011	0.0443	0.0003	-0.0440	0.0000	-0.0443	-0.0086	-0.0529	-0.0089	-0.0531
				19	0.0000	0.0000	0.0000	0.0206	0.0357	0.0052	-0.0305	0.0000	-0.0357	-0.0030	-0.0387	-0.0071	-0.0428
				20	0.0000	0.0000	0.0000	0.0333	0.0367	0.0083	-0.0284	0.0000	-0.0367	-0.0007	-0.0374	-0.0073	-0.0441
				21	0.0000	0.0000	0.0000	0.0152	0.0374	0.0038	-0.0336	0.0000	-0.0374	-0.0044	-0.0418	-0.0075	-0.0449
				22	0.0000	0.0000	0.0000	0.0037	0.0428	0.0009	-0.0419	0.0000	-0.0428	-0.0078	-0.0506	-0.0086	-0.0514
				23	0.0000	0.0000	0.0000	0.0142	0.0404	0.0036	-0.0368	0.0000	-0.0404	-0.0052	-0.0456	-0.0081	-0.0485
				24	0.0000	0.0000	0.0000	0.0073	0.0419	0.0018	-0.0401	0.0000	-0.0419	-0.0069	-0.0489	-0.0084	-0.0503
				25	0.0000	0.0000	0.0000	0.0070	0.0422	0.0017	-0.0404	0.0000	-0.0422	-0.0070	-0.0492	-0.0084	-0.0506
				26	0.0000	0.0000	0.0000	0.0002	0.0452	0.0000	-0.0451	0.0000	-0.0452	-0.0090	-0.0542	-0.0090	-0.0542

frequency of profile, fault risk, and market vulnerability are directly correlated to benefit. These methods, e.g., Methods 15 through 26 in Table 5.10, are highly restrictive as the fitness values for those components in WS2 tend to be negative and thus not interesting for selectors, in particular, ly S2 and S3.

Comparing strategies applied in WS2 under 10% and 20% finds a decreased number of scenarios in which S3 outperformed S2 (98.41% and 74.60%, respectively), with average fitness increasing from 1.176% to 2.736%.

Thus the use of SCOUT (S3) in scenarios with different priorities reveals its significant superiority over S1 and S2 both in terms of the number of simulated scenarios with diverse priorities and in average fitness, as shown in Tables 5.8 and 5.11.

Table 5.11: *Advantages of S3 under different constraints.*

Constraint	Count		Avg		St Dev	
	S3xS1	S3xS2	S3xS1	S3xS2	S3xS1	S3xS2
(1%)	89.42%	82.54%	10.994%	1.008%	6.617%	1.179%
(5%)	89.95%	86.77%	12.126%	1.179%	7.127%	1.011%
(10%)	89.95%	91.01%	10.990%	0.729%	7.202%	0.549%
(20%)	90.48%	86.24%	10.488%	1.227%	7.916%	1.372%

Thus in response to RQ2, SCOUT presents a positive effect with an overall advantage over the other strategies.

5.6 Analysis of RQ3

The study now turns to RQ3, perhaps the most significant research question of the three: How effective is SCOUT in selecting the most significant components in terms of market relevance? Fewer anticipated bugs may have higher market vulnerability,

especially on an Android platform. To test this hypothesis, simulated bug scenarios were constructed and the capacity of all strategies to minimize market vulnerability in them was tested.

Determining which components would be designated as containing bugs was based on six criteria: defect proneness, fault risk, market vulnerability, profiling frequency, cyclomatic complexity, and cost of future maintenance. In the first bug scenario (BS-01), components with a high rate of defect proneness were marked as containing a bug,; whereas in the the second (BS-02), it was components with a high risk of fault. This rule was followed until BS-06 according to the other criteria. For the subsequent scenarios the same strategy used to generate Table 5.5 was adopted. The composition of all bug scenarios with their criteria is presented in Table 5.12.

Table 5.12: *Composition of bug scenarios.*

Bug Scenarios	DP	RF	MV	P	CC	CFM
BS-01	x					
BS-02		x				
BS-03			x			
BS-04				x		
BS-05					x	
BS-06						x
BS-07	x	x				
BS-08	x		x			
.....	.....	.....	.....	.....	.....	.....
BS-60	x	x		x	x	x
BS-61	x		x	x	x	x
BS-62		x	x	x	x	x
BS-63	x	x	x	x	x	x

To illustrate the creation of the bug scenarios, consider BS-01. To create a list of components marked as containing bugs in this scenario, each Android app is iterated over its components in decreasing order of defect proneness, deeming each as containing bugs until the sum of the required time for testing is less than the total time available for testing.

Examine the components marked as containing bugs in Table 5.13 and their market vulnerability as presented in ue Table 5.14. It can be seen that S1 exhibited the worst performance in this scenario because it failed to select component 76, which had the highest market vulnerability of among all components marked as containing errors. Note that S3 attained the best result as component 112, which it missed, had the lowest market vulnerability.

Table 5.13: *Components marked as containing errors.*

Components	Not selected by S1	Not selected by S2	Not selected by S3
18,76,87,112	76	18	112

The market vulnerability for each strategy was computed based on the sum of market vulnerability metric for those components marked with bugs that the strategy

Table 5.14: *Market vulnerability of components marked with bugs.*

Component	Market Vulnerability
18	18%
76	27%
87	36%
112	12%

failed to select. As shown in Figure 5.5, S3 had a lower market vulnerability than S1 in all scenarios under weights WS1, WS2, and WS3.

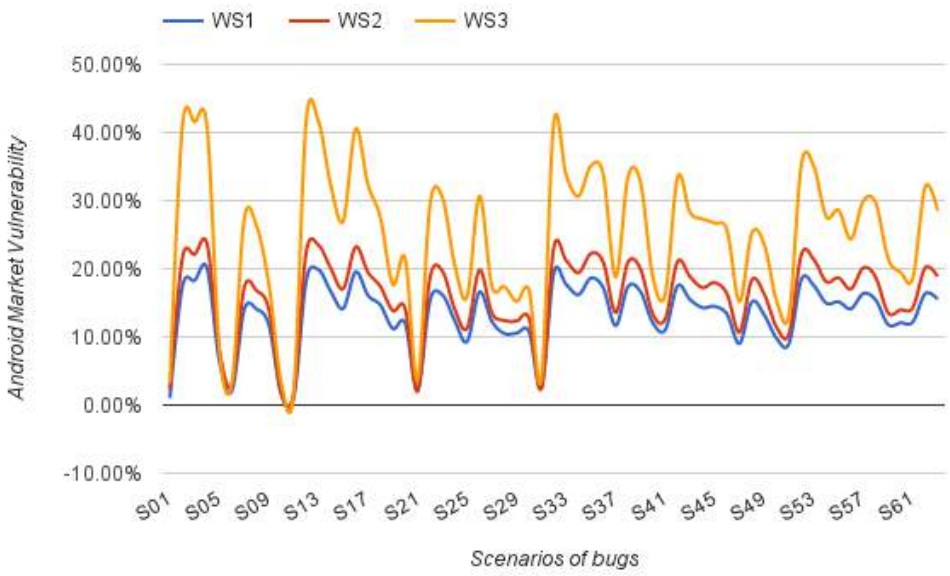

Figure 5.5: *Market vulnerability comparison S1/S3.*

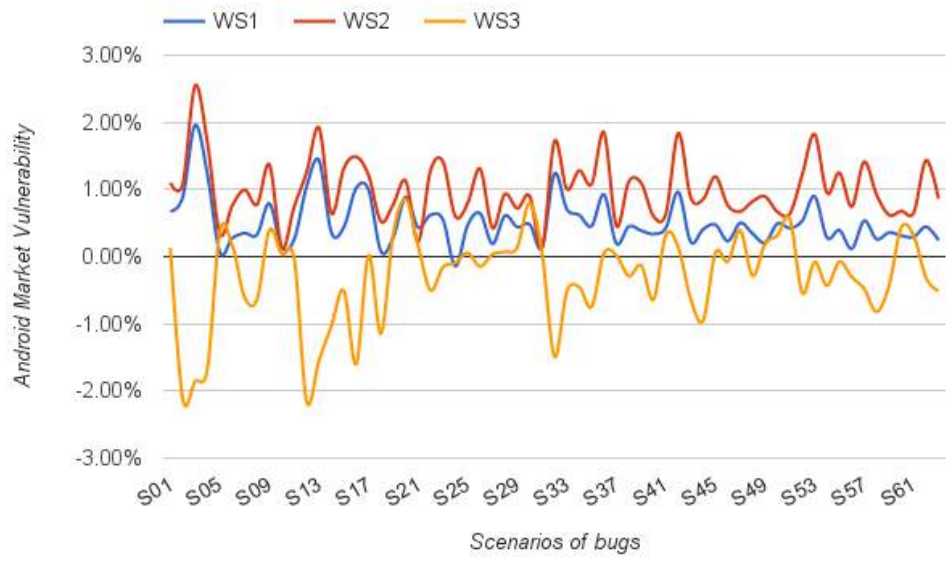

Figure 5.6: *Market vulnerability comparison S2/S3.*

As can be seen in Table 5.15, S1 and S2 had greater vulnerability than S3 in all scenarios save under WS3, demonstrating the advantage of using the SCOUT strategy to

select components for unit testing to minimize market vulnerability. In the case of WS3, the sole exception, S2's market vulnerability average was just 0.31% less than S3's, while the number of scenarios in which its market vulnerability exceeded S3's was 41.27%.

Table 5.15: *Market vulnerability in scenarios of bugs.*

Weight Scenario	Percentage of Scenarios		Average		St Dev	
	S1 >S3	S2 >S3	S1 >S3	S2 >S3	S1 >S3	S2 >S3
WS1	100.00%	98.41%	13.29%	0.52%	4.83%	0.36%
WS2	100.00%	100.00%	15.95%	1.00%	5.84%	0.46%
WS3	100.00%	41.27%	24.70%	-0.31%	10.69%	0.67%
Average	100.00%	79.89%	17.98%	0.40%	7.12%	0.50%

To address the question why S3 had higher average market vulnerability than S2 under WS3, the effects of using S3 under different time contraints for unit testing were examined, as reported in Table 5.16.

Table 5.16: *Market vulnerability under various time constraints.*

Constraint	WS1		WS2		WS3	
	S1 - S3	S2 - S3	S1 - S3	S2 - S3	S1 - S3	S2 - S3
(1%)	10.57%	0.48%	-0.01%	0.01%	15.37%	0.28%
(5%)	15.12%	1.75%	13.46%	1.08%	24.34%	0.32%
(10%)	13.56%	-0.47%	15.24%	1.75%	28.36%	0.25%
(20%)	13.94%	0.31%	18.18%	0.99%	30.75%	-2.10%

As can be seen in Table 5.16, under a time constraint of 20%, S3 has higher market vulnerability than S2. In this scenario, under WS3, the capacity of S3 to increase benefit is low since there are few components with positive fitness available for selection. On the other hand, if there is an available component and its normalized fault risk exceeds zero, S2's benefit tends to be greater than S3's, as stated in the section 5.5.

In general, S2 performed better than S1. This arises from the strong correlation between the fault risk and market vulnerability metrics (0.76 per Table 4.2), which were retained in the SCOUT formulation for the reasons set forth in Section 4.2. As confirmed by the study's findings, SCOUT should be recommended to select components for unit testing to minimize Android market vulnerability. The sole in which S3 had a higher market vulnerability than S2 was when time available for unit testing reached 20% and cost was prioritized over benefit (WS3).

5.7 Threats to Validity

As with any empirical study, there are potential threats to validity. To minimize these threats, the following steps were taken and cautions noted:

Android apps used in the study were limited to nine. To ensure external validity and provide generalized results, applications were selected to maximize coverage of distinct classes (popularity, high number of installations, complexity, size, and others).

Although some frameworks (MOEA, JMetal, and others) were used as references to implement the baseline algorithms, these implementations are subject to failure, as are the algorithms, scripts, and technologies used to manipulate data and consolidate the results.

As a repository of bugs serviceable for the study was not located, simulated bug scenarios were constructed as previously noted. It is, however, possible that some scenarios may not faithfully reflect the reality they represent. To minimize this, a broadly diverse spectrum of scenarios were devised. .

As the number of devises used in the study was limited to seven, a greater number of devices or emulators could generate different results. To mitigate this risk, Orthogonal Array Testing was used to enhance the selection of devices and emulators with different characteristics were also used.

CHAPTER 6

Conclusion

The lack of resources for testing activities is a very present reality in the software development context. This makes testers have to choose a subset of components in the midst of endless possibilities. This makes testers and managers have to choose a subset of components in the midst of endless possibilities. As presented in our section of motivation 1.1 and also in related work Section 3, some strategies for the selection of these components use static metrics, or dynamic metrics, or still business information to support decision-making process in the selection, but none of them provides these metrics combined in an automated way as our approach.

In this work, we presented a novel method called SCOUT to select components for unit testing. Unlike others, the proposed method simultaneously takes into consideration many objectives to assist software testers in deciding which units they should test in a limited available time while satisfies five important metrics, and two objective functions. These metrics are: cost of future maintenance (from static analysis); frequency of method calls, and risk of fault (from dynamic analysis); market vulnerability (from Android market); and the cost of unit testing in terms of time (see Section 4).

In order to validate SCOUT, in the Section 5 a set of experiments were performed with nine-top Android apps and the results show that it can be useful and effective in practice. We performed these experiments seeking to answer the following research questions:

RQ1 - Which solver is more appropriated to be used in a scenario where benefit and cost have the same weight of importance for the specialist?

RQ2 - What is the impact of using SCOUT in scenarios of different priorities? In contexts:

[**RQ2.1**] - where benefit and cost have the same weight of importance for the specialist.

[**RQ2.2**] - where the specialist prioritizes a high quality of the product instead of a low cost testing strategy.

[**RQ2.3**] - which requires low cost of testing strategy in detriment of quality.

RQ3 - What is the efficacy of SCOUT in selecting more important components in terms of their market relevance?

To answer RQ1 seven algorithms/techniques were analyzed to solve this multi-objective problem: Randomly approach (R), Constructivist Heuristic (CH), Genetic Algorithm (GA), SPEA_II, NSGA_II, NSGA_III, and a heuristic implemented by a comercial tool called Gurobi. We compared both its effectiveness and efficiency, and the results indicate some benefits in using NSGA_II as solver for the multi-objective component selection problem, althouth in general Gurobi had the best efficacy and efficiency 5.4.

In order to answer RQ2 we also investigated how SCOUT works in different scenarios: (1) different importance for the specialist; (2) when the benefit and cost have different weights for the specialist. Finally, to answer RQ3 we investigated how does SCOUT work in simulated scenarios of bugs. The results confirmed that SCOUT is able to reduce the market vulnerability, and also it is generally more suitable to select components for unit tetsing than defect prediction models, and fault localization models.

Besides we formulate a novel multiobjective method that considers important variables for optimizing the selection of components for Android unit testing, there are others main contributions of our work as follows:

1. A comparison analysis of both efficacy and efficiency among three strategies and seven solvers to address the problem;
2. A compiled database containing metrics and algorithms to replicate the experiments done in this research, and also to be a novel benchmark for the problem of components selection for Unit testing;
3. A strategy for reducing the numbers of devices to test the market vulnerability, based on Orthogonal Array Technique;
4. A paper entitled “A Parallel Genetic Algorithm to Coevolution of the Strategic Evolutionary Parameters” published in the International Conference on Artificial Inteligence (ICAI’13), Las vegas/USA;
5. A paper entitled “Prioritization of Artifacts for Unit Testing Using Genetic Algorithm Multi-objective Non Pareto” published in the International Conference on Software Engineering Research and Practice (SERP’14), Las vegas/USA;
6. A paper entitled “From Manual Android Tests to Automated and Platform Independent Test Scripts” submited to International Conference on Automated Software Engineering (ASE/2016), Singapore.

In addition, as stated in the recommendation letter in Appendix A, as result of this collaboration at Checkdroid/Gerogia Tech we had:

1. An initial prototype of Capture/Replay tool called Android Mirror Tool (AMT) generating Input Tests written in Espresso API (FREITAS, 2015);

2. A tool for generating automated UI test cases in Espresso API called Barista (CHOUDHARY, 2015a);
3. A paper in submission with Georgia Tech researchers (by the time this thesis was written).

For future works, we suggest the possibility of performing a more extensive case study, increasing the number subjects, the number of Android apps, and the number of devices, and the size of our user study to confirm our initial results. We suggest to investigate how the many-objective formulation proposed behaves when each of its objective functions has different weights. Also investigate how SCOUT reduces the search space for the selection of test cases problem. Another interesting future research is to combine the selection of artifacts for testing with automated test generation techniques to evaluate the efficacy and efficiency of these test data generators on detecting faults on the selected components. Also, once the evolutionary algorithms are openly known in the literature, and there are also many implementations available in cost-free frameworks, there is the opportunity and flexibility for extending these algorithms for the CSP context.

Finally, it will be interesting to investigate the use of the SCOUT in the context of Desktop and Web applications, confronting the set of static and dynamic metrics more adequate to be used by SCOUT on each context.

Bibliography

ABREU, R. et al. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, Elsevier, v. 82, n. 11, p. 1780–1792, 2009.

AMALFITANO, D. et al. Using GUI Ripping for Automated Testing of Android Applications. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2012.

AMMANN, P.; OFFUTT, J. *Introduction to software testing*. [S.l.]: Cambridge University Press, 2008.

ANAND, S. et al. Automated Concolic Testing of Smartphone Apps. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2012.

ANDROID Market. 2015. `http://goo.gl/3UoYaG`.

ANDROID Profilling. 2016. `http://developer.android.com/tools/debugging/debugging-tracing.html`.

ANDROID Screencast. 2015. `https://code.google.com/p/androidscreencast/`.

ASSUNÇÃO, W. K. G. et al. A multi-objective optimization approach for the integration and test order problem. *Information Sciences*, Elsevier, v. 267, p. 119–139, 2014.

ASTELS, D. *Test driven development: A practical guide*. [S.l.]: Prentice Hall Professional Technical Reference, 2003.

AZIM, T.; NEAMTIU, I. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. [S.l.: s.n.], 2013.

BATE, I.; KHAN, U. Wcet analysis of modern processors using multi-criteria optimisation. *Empirical Software Engineering*, Springer, v. 16, n. 1, p. 5–28, 2011.

BOEHM, B. W. Value-based software engineering: Overview and agenda. In: *Value-based software engineering*. [S.l.]: Springer, 2006. p. 3–14.

BRIAND, L.; LABICHE, Y.; CHEN, K. A multi-objective genetic algorithm to rank state-based test cases. In: *Search Based Software Engineering*. [S.l.]: Springer, 2013. p. 66–80.

Calabash. 2015. `https://github.com/calabash/calabash-android`.

CHANG, C. K. Changing face of software engineering. *IEEE Software*, v. 11, n. 1, p. 4–5, 1994.

CHANG, C. K. et al. Spmnet: a formal methodology for software management. In: IEEE. *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*. [S.l.], 1994. p. 57.

CHANG, C. K. et al. Software project management net: a new methodology on software management. In: IEEE. *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*. [S.l.], 1998. p. 534–539.

CHEN, M. Y. et al. Pinpoint: Problem determination in large, dynamic internet services. In: IEEE. *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. [S.l.], 2002. p. 595–604.

CHIKOFSKY, E. J.; CROSS, J. H. et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, IEEE, v. 7, n. 1, p. 13–17, 1990.

CHOI, W.; NECULA, G.; SEN, K. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. [S.l.: s.n.], 2013.

CHOUDHARY, S. R. *Barista: Making Espresso Testing for Android*. 2015. Project Web Page. Available at: `https://checkdroid.com/barista/`. Accessed on: 01/20/2016.

CHOUDHARY, S. R. *Checkdroid Company*. 2015. Project Web Page. Available at: `https://checkdroid.com`. Accessed on: 01/20/2016.

CHOUDHARY, S. R.; GORLA, A.; ORSO, A. Automated test input generation for android: Are we there yet? *arXiv preprint arXiv:1503.07217*, 2015.

COHN, M. *Succeeding with agile: software development using Scrum*. [S.l.]: Pearson Education, 2010.

COMMITTEE, S. E. S. et al. Ieee standard for software maintenance. *IEEE Std*, p. 1219–1998, 1998.

Cucumber. 2015. `https://cukes.info/`.

CZERWONKA, J. et al. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In: IEEE. *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. [S.l.], 2011. p. 357–366.

DEB, K. et al. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, IEEE, v. 6, n. 2, p. 182–197, 2002.

DELAMARO, M. E. et al. Introdução ao teste de software. In: ____. [S.l.]: Campus, 2007. cap. Teste de Mutação, p. 77–118.

DEMILLO, R. A. *Software Testing and Evaluation*. [S.l.]: The Benjamin/Cummings Publishing Company Inc„ 1987.

DMR. *Androi Statistic*. 2016. Project Web Page. http://expandedramblings.com/index.php/android-statistics. Accessed on: 01/20/2016.

DOLADO, J. J. A validation of the component-based method for software size estimation. *Software Engineering, IEEE Transactions on*, IEEE, v. 26, n. 10, p. 1006–1021, 2000.

DURILLO, J. J. et al. A study of the bi-objective next release problem. *Empirical Software Engineering*, Springer, v. 16, n. 1, p. 29–60, 2011.

ELBERZHAGER, F.; BAUER, T. From assumptions to context-specific knowledge in the area of combined static and dynamic quality assurance. In: IEEE. *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. [S.l.], 2012. p. 298–301.

ELBERZHAGER, F.; ESCHBACH, R.; MÜNCH, J. Using inspection results for prioritizing test activities. In: *21st International Symposium on Software Reliability Engineering, Supplemental Proceedings*. [S.l.: s.n.], 2010. p. 263–272.

ELBERZHAGER, F. et al. Guiding testing activities by predicting defect-prone parts using product and inspection metrics. In: IEEE. *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. [S.l.], 2012. p. 406–413.

ELBERZHAGER, F. et al. Focusing testing by using inspection and product metrics. *International Journal of Software Engineering and Knowledge Engineering*, World Scientific, v. 23, n. 04, p. 433–462, 2013.

ELBERZHAGER, F.; MÜNCH, J. Using early quality assurance metrics to focus testing activities. *arXiv preprint arXiv:1312.1043*, 2013.

ELBERZHAGER, F.; MÜNCH, J.; ASSMANN, D. Analyzing the relationships between inspections and testing to provide a software testing focus. *Information and Software Technology*, Elsevier, v. 56, n. 7, p. 793–806, 2014.

ELBERZHAGER, F.; MÜNCH, J.; NHA, V. T. N. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information and Software Technology*, Elsevier, v. 54, n. 1, p. 1–15, 2012.

ELBERZHAGER, F. et al. Optimizing cost and quality by integrating inspection and test processes. In: ACM. *Proceedings of the 2011 International Conference on Software and Systems Process*. [S.l.], 2011. p. 3–12.

ELBERZHAGER, F. et al. Reducing test effort: A systematic mapping study on existing approaches. *Information and Software Technology*, Elsevier, 2012.

Espresso. 2015. https://goo.gl/N2bT8j.

F-DROID. *F-Droid | Free and Open Source Android App Repository*. 2016. Project Web Page. Available at: https://f-droid.org. Accessed on: 01/20/2016.

FOWLER, M. *TestPyramid*. 2012. Web Page. Available at: http://goo.gl/VbrNqF. Accessed on: 05/15/2015.

FREITAS, E. N. de A. *Android Mirror Tool*. 2015. Project Web Page. Available at: https://www.youtube.com/watch?v=VSh78LQdDHY. Accessed on: 01/20/2016.

GAO, J. et al. Mobile application testing: a tutorial. *Computer*, IEEE, n. 2, p. 46–55, 2014.

GOMEZ, L. et al. Reran: Timing-and touch-sensitive record and replay for android. In: IEEE. *Software Engineering (ICSE), 2013 35th International Conference on*. [S.l.], 2013. p. 72–81.

GOOGLE. *UI Testing - Android Developers*. 2015. Project Web Page. Available at: https://goo.gl/s06AuG/. Accessed on: 01/20/2016.

Google Play Store. *Google Play*. 2016. Project Web Page. Available at: https://play.google.com/store. Accessed on: 01/20/2016.

HAO, S. et al. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. [S.l.: s.n.], 2014.

HARMAN, M.; JIA, Y.; ZHANG, Y. Achievements, open problems and challenges for search based software testing. In: IEEE. *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. [S.l.], 2015. p. 1–12.

HARMAN, M.; JONES, B. F. Search-based software engineering. *Information and Software Technology*, Elsevier, v. 43, n. 14, p. 833–839, 2001.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, ACM, v. 45, n. 1, p. 11, 2012.

HARMAN, M. et al. Search–based approaches to the component selection and prioritization problem. In: ACM. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. [S.l.], 2006. p. 1951–1952.

HARMAN YUE JIA, Y. Z. M. Achievements, open problems and challenges for search based software testing. *International Conference Software Testing - ICST*, 2015.

HASSAN, A. E.; HOLT, R. C. The top ten list: Dynamic fault prediction. In: IEEE. *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.* [S.l.], 2005. p. 263–272.

HOWDEN, W. E. Functional program testing. *Software Engineering, IEEE Transactions on*, IEEE, n. 2, p. 162–169, 1980.

HOWDEN, W. E. *Functional Program Testing and Analysis*. New York, NY: McGrall-Hill, 1987. (Software Engineering and Technology).

IBM. *IBM Rational Test RealTime 8.0.0*. 2016. Project Web Page. Available at: https://www-01.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/rwizmetricsdiagram.htm?lang=en. Accessed on: 01/20/2016.

IDC. *International Data Corporation*. 2016. Project Web Page. Available at: http://www.idc.com. Accessed on: 01/20/2016.

ISHIBUCHI, H.; AKEDO, N.; NOJIMA, Y. Behavior of multi-objective evolutionary algorithms on many-objective knapsack problems. IEEE, 2014.

JHA, P. et al. Optimal testing resource allocation during module testing considering cost, testing effort and reliability. *Computers & Industrial Engineering*, Elsevier, v. 57, n. 3, p. 1122–1130, 2009.

JHAWK. *JHawk Tool*. 2016. Project Web Page. Available at: http://www.virtualmachinery.com. Accessed on: 01/20/2016.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of test information to assist fault localization. In: ACM. *Proceedings of the 24th international conference on Software engineering*. [S.l.], 2002. p. 467–477.

JUNIT. *JUnit*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.junit.org/>.

KAPUR, P. et al. Optimal testing resource allocation for modular software considering cost, testing effort and reliability using genetic algorithm. *International Journal of Reliability, Quality and Safety Engineering*, World Scientific, v. 16, n. 06, p. 495–508, 2009.

KARHU, K. et al. Empirical observations on software testing automation. In: IEEE. *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. [S.l.], 2009. p. 201–209.

KIPER, J. D.; FEATHER, M. S.; RICHARDSON, J. Optimizing the v&v process for critical systems. In: ACM. *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. [S.l.], 2007. p. 1139–1139.

KUHN, D. R.; KACKER, R. N.; LEI, Y. Practical combinatorial testing. *NIST Special Publication*, Citeseer, v. 800, n. 142, p. 142, 2010.

KUHN, D. R.; REILLY, M. J. An investigation of the applicability of design of experiments to software testing. In: IEEE. *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*. [S.l.], 2002. p. 91–95.

KUHN, D. R.; WALLACE, D. R.; GALLO, J. A. Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, IEEE, v. 30, n. 6, p. 418–421, 2004.

LI, Q. Using additive multiple-objective value functions for value-based software testing prioritization. *University of Southern California Computer Science Department*, 2009.

LI, Q.; BOEHM, B. Improving scenario testing process by adding value-based prioritization: an industrial case study. In: ACM. *Proceedings of the 2013 International Conference on Software and System Process*. [S.l.], 2013. p. 78–87.

LIN, Y.-D. et al. Improving the accuracy of automated gui testing for embedded systems. *Software, IEEE*, IEEE, v. 31, n. 1, p. 39–45, 2014.

LIN, Y.-D. et al. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *Software Engineering, IEEE Transactions on*, v. 40, n. 10, p. 957–970, Oct 2014. ISSN 0098-5589.

LIU, C. H. et al. Capture-replay testing for android applications. In: IEEE. *Computer, Consumer and Control (IS3C), 2014 International Symposium on*. [S.l.], 2014. p. 1129–1132.

MACHADO, P.; CAMPOS, J.; ABREU, R. Mzoltar: automatic debugging of android applications. In: ACM. *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile*. [S.l.], 2013. p. 9–16.

MACHIRY, A.; TAHILIANI, R.; NAIK, M. Dynodroid: An Input Generation System for Android Apps. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. [S.l.: s.n.], 2013.

MAHMOOD, R.; MIRZAEI, N.; MALEK, S. EvoDroid: Segmented Evolutionary Testing of Android Apps. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2014.

MANDL, R. Orthogonal latin squares: an application of experiment design to compiler testing. *Communications of the ACM*, ACM, v. 28, n. 10, p. 1054–1058, 1985.

MIRARAB, S.; AKHLAGHI, S.; TAHVILDARI, L. Size-constrained regression test case selection using multicriteria optimization. *Software Engineering, IEEE Transactions on*, IEEE, v. 38, n. 4, p. 936–956, 2012.

MYERS, G. J. *The Art of Software Testing*. [S.l.]: Wiley, New York, 1979.

MYERS, G. J. et al. *The Art of Software Testing*. [S.l.]: Wiley, New York, 2004.

NIST. *National Institute of Standards and Technology*. 2016. Project Web Page. Available at: `http://csrc.nist.gov/groups/SNS/acts/index.html`. Accessed on: 01/20/2016.

OPTIMIZATION, G. et al. Gurobi optimizer reference manual. *URL: http://www. gurobi. com*, 2015.

PAPADIMITRIOU, C. H.; STEIGLITZ, K. *Combinatorial optimization: algorithms and complexity*. [S.l.]: Courier Corporation, 1998.

PRESSMAN, R. S. *Software Engineering – A Practitioner's Approach*. 6. ed. [S.l.]: McGraw-Hill, 2005.

PUGH, K. *Lean-Agile Acceptance Test-Driven-Development*. [S.l.]: Pearson Education, 2010.

RAY, M.; MOHAPATRA, D. P. Code-based prioritization: a pre-testing effort to minimize post-release failures. *Innovations in Systems and Software Engineering*, Springer, v. 8, n. 4, p. 279–292, 2012.

REN, J. *Search Based Software Project Management*. Tese (PhD Thesis) — Department of Computer Science, University College London, London, UK, maio 2013. Available em: http://www0.cs.ucl.ac.uk/staff/mharman/jian-phd.pdf. Accessed in: 01/01/2016.

Sauce Labs. *Appium*. 2015. Project Web Page. Available at: http://appium.io. Accessed on: 01/20/2016.

Selendroid. 2015. http://selendroid.io/.

SHAH, G.; SHAH, P.; MUCHHALA, R. Software testing automation using appium. 2014.

SHELBURG, J.; KESSENTINI, M.; TAURITZ, D. R. Regression testing for model transformations: A multi-objective approach. In: *Search Based Software Engineering*. [S.l.]: Springer, 2013. p. 209–223.

SHI, A. et al. Balancing trade-offs in test-suite reduction. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2014. p. 246–256.

SHIHAB, E. et al. Prioritizing the creation of unit tests in legacy software systems. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 41, n. 10, p. 1027–1048, set. 2011. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/spe.1053>.

SHUKLA, K. Neuro-genetic prediction of software development effort. *Information and Software Technology*, Elsevier, v. 42, n. 10, p. 701–713, 2000.

Spoon. 2015. http://square.github.io/spoon.

STATISTA. *The Statistics Portal*. 2016. Project Web Page. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed on: 01/20/2016.

TILLMANN, N.; HALLEUX, J. de; XIE, T. Parameterized unit testing: Theory and practice. In: IEEE. *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*. [S.l.], 2010. v. 2, p. 483–484.

TURING, A. Checking a large routine. In: MIT PRESS. *The early British computer conferences*. [S.l.], 1989. p. 70–72.

UI/APPLICATION Exerciser Monkey. 2015. http://developer.android.com/tools/help/monkey.html.

VINCENZI, A. M. R. *Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação*. Tese (Tese de Doutoramento) — Instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, SP, maio 2004. Available em: http://www.teses.usp.br/teses/disponiveis/55/55134/tde-17082004-122037. Acesso em: 21/10/2004.

VINCENZI, A. M. R. et al. Ii pernambuco school on software engineering: Software testing. In: ____. 1. ed. New York, NY: Springer Berlin Heidelberg, 2010. (Lecture Notes in Computer Science, v. 6153), Incs Functional, Control and Data Flow, and Mutation Testing: Theory and Practice, p. 18–58. Disponível on-line: http://www.springer.com/computer/swe/book/978-3-642-14334-2.

WALLACE, D. R.; KUHN, D. R. Failure modes in medical device software: an analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering*, World Scientific, v. 8, n. 04, p. 351–371, 2001.

WANG, Z.; TANG, K.; YAO, X. Multi-objective approaches to optimal testing resource allocation in modular software systems. *Reliability, IEEE Transactions on*, IEEE, v. 59, n. 3, p. 563–575, 2010.

WEYUKER, E.; OSTRAND, T.; BELL, R. Using static analysis to determine where to focus dynamic testing effort. In: *Proceedings of the IEEE Second International Workshop on Dynamic Analysis*. [S.l.: s.n.], 2004. p. 1–8.

YANG, W.; PRASAD, M. R.; XIE, T. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. [S.l.: s.n.], 2013.

YEH, T.; CHANG, T.-H.; MILLER, R. C. Sikuli: using gui screenshots for search and automation. In: ACM. *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. [S.l.], 2009. p. 183–192.

YOO, S.; HARMAN, M. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, Elsevier, v. 83, n. 4, p. 689–701, 2010.

YU, L. et al. Acts: A combinatorial test generation tool. In: IEEE. *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. [S.l.], 2013. p. 370–375.

YUAN, Y.; XU, H.; WANG, B. An improved nsga-iii procedure for evolutionary many-objective optimization. In: ACM. *Proceedings of the 2014 conference on Genetic and evolutionary computation*. [S.l.], 2014. p. 661–668.

ZADGAONKAR, H. *Robotium Automated Testing for Android*. [S.l.]: Packt Publishing Ltd, 2013.

ZHANG, Y. *Multi-Objective Search-based Requirements Selection and Optimisation*. Tese (Doutorado) — University of London, 2010.

ZHANG, Y.; HARMAN, M.; LIM, S. L. Empirical evaluation of search based requirements interaction management. *Information and Software Technology*, Elsevier, v. 55, n. 1, p. 126–152, 2013.

ZHENG, C. et al. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In: ACM. *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. [S.l.], 2012. p. 93–104.

ZHU, M. H. Y.; PERI, R.; REDDI, V. J. Mosaic: Cross-platform user-interaction record and replay for the fragmented android ecosystem. 2015.

ZITZLER, E. et al. *SPEA2: Improving the strength Pareto evolutionary algorithm*. [S.l.]: Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische Informatik und Kommunikationsnetze (TIK), 2001.

APPENDIX A

Checkdroid Letter

CheckDroid @ Georgia Tech

266 First Dr. • Atlanta, GA 30308 • Phone: 404-345-6993
E-Mail: shauvik@checkdroid.com Web: http://checkdroid.com

Date: June 1, 2015

To: Whomsoever it may concern
Subject: Recommendation for Mr. Eduardo Noronha de Andrade Freitas's towards the successful completion of his visit to Georgia Tech

I am pleased to write a letter of recommendation for Mr. Eduardo Noronha de Andrade Freitas for the completion of his visit at Georgia Tech. During his visit, Eduardo worked closely with me on the CheckDroid project for commercializing research ideas from our lab into a potential start up. As one can imagine, a startup environment is pretty challenging: requiring oneself to come up with innovative solutions and iterate quickly based on user feedback. Eduardo helped in the building of our initial prototype, which we showed to potential users of our technology and received important feedback. His help was crucial in generating stronger discussions with users and within my research group, thereby shaping this project as it is today.

In addition to CheckDroid, Eduardo was also involved in a collaborative research project for test input generation of Android. Under this project, he interacted with several study participants and performed most of the experimentation for the project. The data from his experimentation formed the basis for the evaluation section of the paper.

Thus, I strongly recommend Eduardo for the successful completion of his program requirements from his visit to Georgia Tech. I believe that he will show excellence in his future projects and towards completing his PhD.

If you would like to discuss this further, please feel free to contact me.

Sincerely,

Dr. Shauvik Roy Choudhary
Instructor, Georgia Tech
Entrepreneurial Lead, CheckDroid

APPENDIX B

Natural Language Test Case (NLTC)

Android App: Daily Money

1 - Click back button
2 - Click “Add Detail” icon
3 - Click “From” drop down list
4 - Click “Asset - Cash” text label
5 - Click “To” drop down list
6 - Click “Party” text label
7 - Click “Calendar” button at the middle-right
8 - Click “-” button to select “Jan 12 2015”
9 - Click “Done” button
10 - Click the edit text under “Money” text label
11 - Type “200” in the edit text
12 - Close keyboard (bottom-left key in the keyboard)
13 - Click the edit text under “Note” text label
14 - Type “Test” in the edit text
15 - Close keyboard (bottom-left key in the keyboard)
16 - Click “Create” button
17 - Click “From” drop down list
18 - Click “A” text label
19 - Click “To” drop down list
20 - Click “Expense - Other expense” text label
21 - Click the edit text under “Money” text label
22 - Type “50” in the edit text
23 - Close keyboard (bottom-left key in the keyboard)
24 - Click “Create(1)” button
25 - Assert text label of bottom-left button is equal to “Create(2)”
26 - Click back button

27 - Assert text label at bottom center position is equal to “Monthly : $250”

28 - Click “Reports” tab

29 - Click “Monthly balance” icon

30 - Assert text label on the right of “Asset” text label is equal to “$-250”