Pós-Graduação em Ciência da Computação

CLAUDIO JOSÉ ANTUNES SALGUEIRO MAGALHÃES

HSP: A Hybrid Selection and Prioritisation of Regression Test Cases based on Information Retrieval and Code Coverage applied on an Industrial Case Study

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

CLAUDIO JOSÉ ANTUNES SALGUEIRO MAGALHÃES

HSP: A Hybrid Selection and Prioritisation of Regression Test Cases based on Information Retrieval and Code Coverage applied on an Industrial Case Study

Dissertação apresentada ao Programa de Pós-Graduação em Ciências da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciências da Computação.

Área de Concentração: Seleção e priorização de casos de teste.

Orientador: Prof. Alexandre Cabral Mota
Coorientadora: Prof. Flávia Almeida Barros

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

M188h Magalhães, Claudio José Antunes Salgueiro
HSP: a hybrid selection and prioritisation of regression test cases based on information retrieval and code coverage applied on an industrial case study / Claudio José Antunes Salgueiro Magalhães. – 2019.
70 f.: il., fig., tab.

Orientador: Alexandre Cabral Mota.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências.

1. Ciência da computação. 2. Teste de regressão. 3. Análise estática. I. Mota, Alexandre Cabral (orientador). II. Título.

004 CDD (23. ed.) UFPE- MEI 2019-054

CLAUDIO JOSÉ ANTUNES SALGUEIRO MAGALHÃES

HSP: A Hybrid Selection and Prioritisation of Regression Test Cases based on Information Retrieval and Code Coverage applied on an Industrial Case Study

Dissertação apresentada ao Programa de Pós-Graduação em Ciências da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciências da Computação.

Aprovado em: 15/02/2019.

BANCA EXAMINADORA

Prof. Dr. Juliano Manabu Iyoda
Centro de Informática/UFPE

Prof Dr. Lucas Albertins de Lima
Departamento de Computação/UFRPE

Prof. Dr. Alexandre Cabral Mota
Centro de Informática/UFPE
(Orientador)

I dedicate this work to everyone that believed me.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my parents for all support, teachings, trust in me and for always investing in my education.

I would also like to thank all my friends from my neighbourhood and college that listened to me, advised me, laughed with me and chit-chatted with me during my journey. I also thank every friend that passed through my life.

A special thanks to my professor Alexandre Mota who guided me to get here. Without his support, and novel ideas, the path would have been harder. I am grateful to the professor Flávia Barros who counselled me to find the ideal solution. Furthermore, I am thankful to all professors that I have met in my life.

In addition, I would like to thank my research partners: João Luiz, Filipe Arruda, Lucas Perrusi, Thais Pina, Cloves Lima and Chaina Oliveira. They were essential to mature the strategy and discuss about innovation in tests. I also thank Vírginia, Marlom, Daniel, João Veras and Rodolfo that worked in the tool deployment on the Motorola environment from CIn/Motorola project. I am grateful for the constant cooperation, support, feedback and incentive by Alice Arashiro, Viviana Toledo, Eliot Maia, Lucas Heredia, Luis Momente and André Bazante from Motorola.

Finally, I am thankful to the CNPQ and CIn/Motorola project, that invested in education and research. I am also lucky and grateful for the CIn-UFPE to provide excellent support and infrastructure for all students and professors.

ABSTRACT

The usual way to guarantee quality of software products is via testing. This dissertation presents a novel strategy for selection and prioritisation of Test Cases (TC) for Regression testing. In the lack of code artifacts from where to derive Test Plans, this work uses information conveyed by textual documents maintained by Industry, such as Change Requests. The proposed process is based on Information Retrieval techniques combined with indirect code coverage measures to select and prioritise TCs. The aim is to provide a high coverage Test Plan which would maximise the number of bugs found. This process was implemented as a prototype tool which was used in a case study with our industrial partner (Motorola Mobility). Experiment results revealed that the combined strategy provides better results than the use of information retrieval and code coverage separately. Yet, it is worth to mention that any of these automated options performed better than the original manual process deployed by our industrial partner to create test plans.

Keywords: Test cases selection and prioritisation. Regression testing. Static analysis. Information retrieval. Code coverage.

RESUMO

A maneira usual de garantir a qualidade dos produtos de software é através de testes. Este trabalho apresenta uma nova estratégia para seleção e priorização de Casos de Teste (TC) para testes de regressão. Na falta de artefatos de código de onde Planos de Teste são derivados, este trabalho usa informações transmitidas por documentos textuais mantidos pela Indústria, como Solicitações de Mudança (CR). O processo proposto é baseado em técnicas de recuperação de informações combinadas com medidas de cobertura de código para selecionar e priorizar os TCs. O objetivo é fornecer um Plano de Teste de alta cobertura que maximize o número de falhas encontradas. Esse processo foi implementado como uma ferramenta protótipo que foi usada em um estudo de caso com nosso parceiro industrial (Motorola Mobility). Os resultados dos experimentos revelaram que a estratégia combinada fornece melhores resultados do que o uso de recuperação de informações e cobertura de código de forma independente. No entanto, vale a pena mencionar que qualquer uma dessas opções automatizadas teve um desempenho melhor do que o processo manual originalmente realizado por nosso parceiro industrial para criar planos de teste.

Palavras-chaves: Seleção e priorização de casos de teste. Teste de regressão. Análise estática. Recuperação de informação. Cobertura de código.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ADB	Android Debug Bridge
APFD	Average Percentage of Faults Detected
ATP	AutoTestPlan
ATC	AutoTestCoverage
CFG	Control Flow Graph
CR	Change Request
FDL	Fault Detection Loss
HSP	Hybrid Selection and Prioritisation
IR	Information Retrieval
MVC	Model-View-Controller
NLP	Natural Language Processing
TC	Test Case
TF-IDF	Term Frequency-Inverse Document Frequency
TP	Test Plan
VSM	Vector Space Model
SUT	System Under Test
SW	Software
RN	Release Note
RTS	Regression Test Selection

CONTENTS

1 INTRODUCTION

The usual way to guarantee quality of software products is via testing. However, this is known to be a very expensive task [Myers, 2004]. Although automated testing is an essential factor of success in projects development, unfortunately it is not always possible or economic viable to adopt this practice in industrial environments. This way, several companies rely on manual or semi-automated testing — as frequently seen in the smart phones industry, for instance. In this context, it is not always feasible to test the entire code of every new SW release, due to time and costs constraints. Thus, an appropriate test plan is crucial for the success of any Regression testing campaign, aiming to maximise the number of bugs found.

Yet, automation is a key solution for this bottleneck. However, several industrial environments still rely on manual testing processes based on textual test cases. Clearly, this context does not favour the development and use of automated solutions for test plans creation. Nevertheless, when it is not possible to apply traditional solutions for test case selection and prioritisation solely in terms of programming code, text based solutions may be adopted [Birkeland, 2010].

This is precisely the context of the research work presented here, which is developed within a long-term collaboration with an industrial partner. Due to intellectual property restrictions, the research team has restricted access to source code. Note that this context is frequently observed in collaborative partnerships between Academic institutions and Industry, thus motivating the investigation of alternative solutions for testing. Yet, besides access restrictions, the current test cases are, in their majority, freely written natural language texts (in English).

1.1 MOTIVATION

Test Architects need to select test cases since it is unfeasible to retest all of them, mainly because they are manual ones. Therefore, they have to select test cases that improve the test campaign. Besides the test selection, these test cases must be prioritised in an order that increases the agility to find failures. Furthermore, it is important to define a reliable process that ever obtains the same result for the same inputs.

For example, Test Architects have to select a subset from a set of test cases where the testers have to execute them. This set of test cases is unfeasible to execute completely, thus it is necessary to select a subset of test cases. However, sometimes there are no efforts to execute all this subset, then the prioritisation put the most important test cases first.

Creating a process to select and prioritise test cases automatically does not just standardise the test campaign but makes it faster and better than the selection made by the

test architect.

1.2 PROPOSAL

In this light, we developed a novel strategy for Regression Test plans creation which combines Information Retrieval (IR) techniques [Baeza-Yates and Ribeiro-Neto, 1999] and indirectly obtained code coverage information. The proposed Hybrid Selection and Prioritisation (HSP) process receives as input CRs related to a particular SW application under test, and creates two intermediate test plans (IR and code coverage) which are finally merged and prioritised based on IR and code coverage measures. Note that, in our context, test plans are ordered lists of textual test cases.

The IR based strategy relies on a database of previously indexed textual Test Cases (TCs) (named as *TCs Index base*), which is independent from the Software (SW) application under test — i.e., this base indexes all TCs available from the general TCs repository (see Section 3.1.1). Test Cases are retrieved from the Index base using as queries relevant keywords extracted from the input CRs. The retrieved TCs are merged and ranked, originating the IR based Test plan (see Section 3.2.1) [Magalhães et al., 2016a].

Since the above strategy relies on a general TCs Index base, the resulting Test plans are usually long, and may contain TCs which are irrelevant to the current SW application under test. This fact motivated the search for a more precise strategy to Test cases selection. In order to improve relevance and precision, we developed a strategy to Test plans creation based on code coverage.

The strategy based on code coverage relies on the *TCs Trace base* (Section 3.1.2), with information about the log of previously executed TCs (TC + trace). We create code coverage information from a test case execution indirect measurement, that means, it is gathered simultaneously during the test execution. The coverage based Test plan contains all TCs in the Trace base whose execution trace match any code fragment present in the current CRs (Section 3.2.3).

Note that, unlike the general TCs Index base, the Trace base can only contain TCs related to the current SW application under test. This way, the Test plans created using this strategy are expected to be more precise and relevant to the ongoing Regression testing campaign than the IR based Test plan. However, these Test plans tend to be very short, since they are limited to the already executed TCs. The characteristics of the Test plans obtained using a single selection strategy justify the need to combine both strategies in the HSP process.

As mentioned above, the Trace base can only contain TCs related to the current SW application under test. This way, this base must be recreated for every new SW application (see Section 3.1.2). This base is updated by the execution monitoring tool whenever a Test plan is executed (Section 3.2.2) within the same testing campaign.

The Test plans obtained by the two different strategies are merged and prioritised based on the code coverage information. A detailed presentation of the HSP is given in Section 3.2. The output Test plan is submitted to the test architect. As mentioned above, all execution runs update the Trace base, including new code coverage information.

The implemented prototype was used to conduct a case study with our industrial partner (Motorola Mobility). Experiments results revealed that the combined strategy provides better results than the use of Information Retrieval and code coverage independently. Yet, it is worth to mention that any of these automated options perform better than the previous manual process deployed by our industrial partner to create test plans.

1.3 CONTRIBUTIONS

The main contributions of this work are:

- a hybrid strategy to select and prioritise test cases based on information retrieval and code coverage;
- a web tool where the hybrid strategy was implemented
- an empirical comparison between selection based on information retrieval, code coverage and both.

1.4 ORGANIZATION OF THE DISSERTATION

This document is organized into another 5 further chapters, as follows.

- **Background:** this chapter presents the definitions and concepts related to the software testing, information retrieval and code coverage;
- **Development:** this chapter details the HSP process and implementation, which performs several different selection and prioritisation strategies; also it shows some setup details;
- **Tool:** this chapter explains the tool architecture, shows its usability and also presents its screens.
- **Case Study:** this chapter explains the performed experiments, shows the obtained results and compares the proposed strategies against the architect and random selections;
- **Conclusion:** this chapter aims to conclude this document with an overall appreciation of the developed work and the achieved results, also pointing at future work to be developed. In addition, it aims to show some of the related work found in the available literature.

2 BACKGROUND

This chapter aims to introduce some important concepts and definitions related to Software Engineering and Information Retrieval. These will serve to better understand the rest of this work.

2.1 SOFTWARE PROCESS

The software process is a set of actions that are executed for creating a software product. The objective is to deliver the software with the expected quality for the sponsor and the user at the right time [Pressman, 2009]. This process is adaptable for each environment or project. Thus, it establishes just a set of framework activities that can be used from small to large projects. According to Pressman [2009], they are:

- **Communication:** It is to gather the requirements that define the software features;
- **Planning:** It is the resource, risk, schedule and activities management.
- **Modelling:** It is the creation of templates and documents to design the software
- **Construction:** It develops the software code, either for maintainability, and tests it to find out possible faults;
- **Deployment:** the software is delivered to the stakeholders and evaluated by them.

The framework activities are widely used in iterative progress. This means that, all the activities are applied for each project iteration. In its turn, each project iteration produces a deliverable software version [Pressman, 2009]. Therefore, the software evolves constantly and is under testing and maintenance (detailed in Section 2.2 and Section 2.3) in each iteration.

2.2 SOFTWARE MAINTENANCE

Software maintenance is one of the software construction activities because there is no distinction between development and maintenance. Software maintenance activities are those that change a system [Sommerville, 2010]. There are three kinds of changes:

- **Fault repairs:** this is done when failures are found in the software;
- **Environmental adaptation:** this is done when the system environment (such as hardware or operational system) changes;

- **Functionality addition:** this is done when requirements change to meet new customer demands.

These activities can be performed before (pre-delivery) or after (post-delivery) the deployment phase of the software. Due to the evolution of the source code and the correction of faults, the software is always changing. Therefore, it requires a good software maintenance process to ensure its integrity [Society, 2014].

This work uses two very important SW artifacts for software maintenance: Release Notes and Change Requests. These are briefly described in what follows.

2.2.1 Release Notes

Each new software product release usually brings an associated RN. RNs are textual documents that provide high-level descriptions of enhancements and new features integrated into the delivered version of the system. Depending on the level of detail, RNs may contain information about CRs incorporated in the current release. RNs can be manually or automatically generated [Moreno et al., 2014]. Figure 1 brings a sample template of a RN.

An analysis performed in Moreno et al. [2014] revealed that the most frequent items included in Release Notes are related to fixed bugs (present in 90% of Release Notes), followed by information about new (with 46%) and modified (with 40%) features and components. However, it is worth to point out that RNs only list the most important issues about a software release, being aimed to provide quick and general information, as reported in [Abebe et al., 2016].

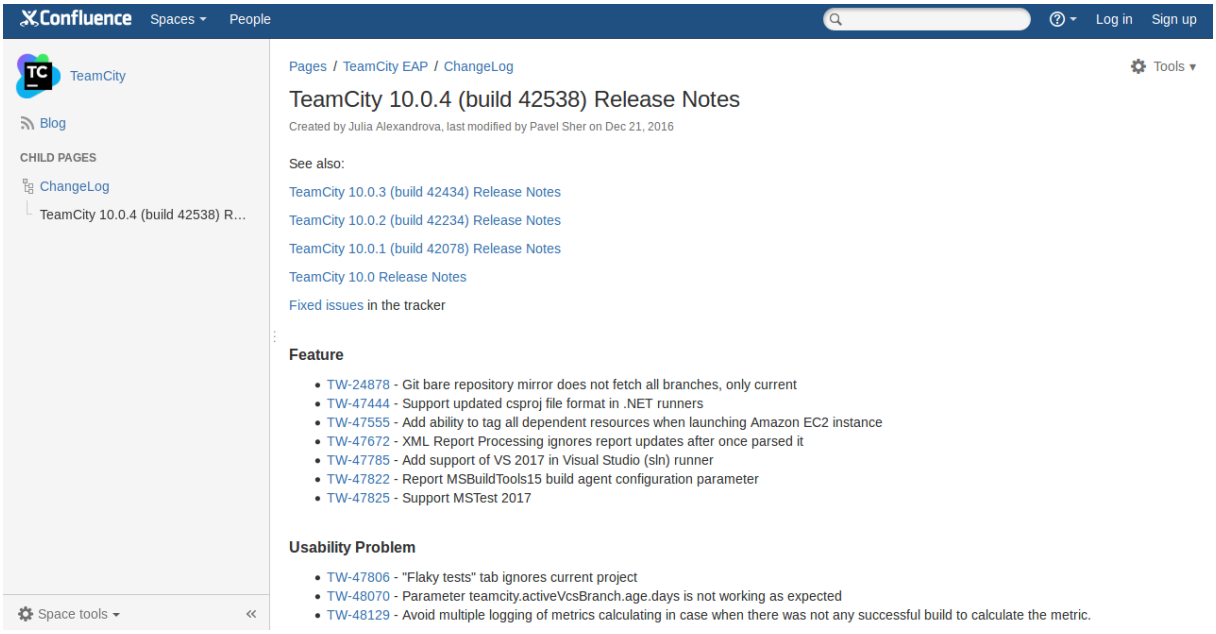


Figure 1 – Generic Release Notes template

2.2.2 Change Request

Change Requests (CRs) are also textual documents whose aim is to describe a defect to be fixed or an enhancement to the software system. Some tracking tools, such as Mantis [Mantis, 2019], Bugzilla [Bugzilla, 2019] and Redmine [Redmine, 2019] are used to support the CR management. They enable stakeholders to deal with several activities related to CRs, such as registering, assigning and tracking.

CRs may be created (opened) by developers, testers, or even by special groups of users. In general, testers and developers use change requests as a way of exchanging information. While testers use CRs to report failures found in a System Under Test (SUT), developers use them primarily as input to determine what and where to modify the source-code, and as output to report to testers what has been modified in the new source-code release. Figure 2 brings a sample template of a CR.

A Release Note is thus composed of several Change Requests as well as new features integrated into the delivered version of the system.

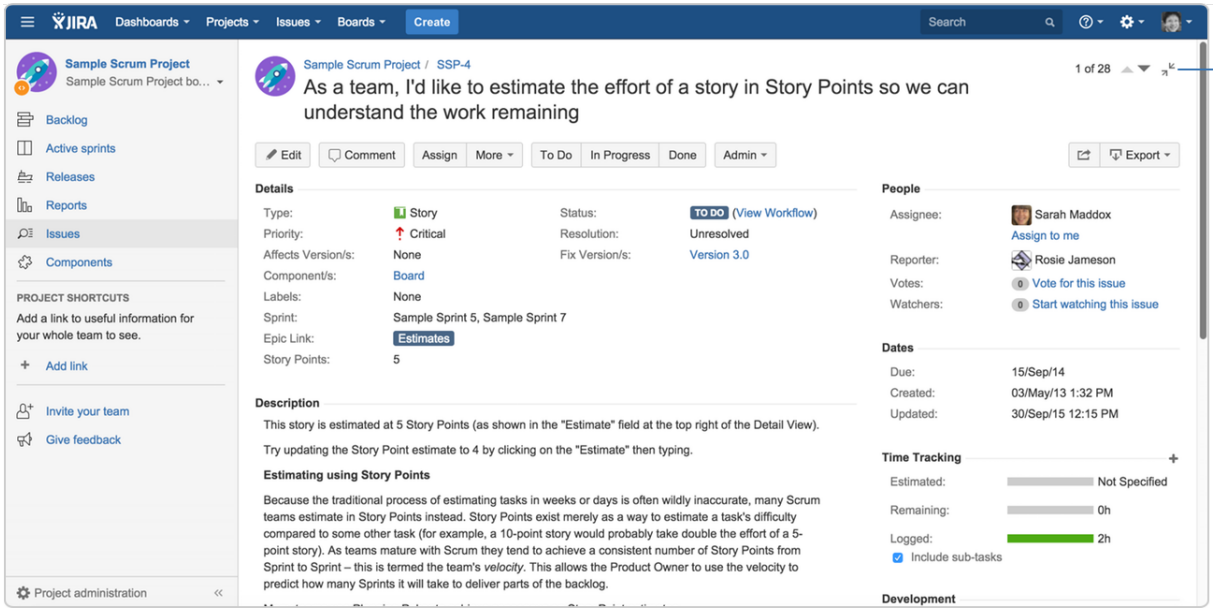


Figure 2 – Generic CR template

2.3 SOFTWARE TESTING

Software testing is an activity that ensures the software functionality and quality are in an acceptable level and is also a fault revealer [Society, 2014]. Therefore, there are two main purposes of testing: verifying if the implemented software corresponds to its requirement and if the software behaviour is incorrect [Sommerville, 2010]. Furthermore, following Dijkstra [1972], tests can only indicate the presence of bugs in a SUT and not their absence.

Software Testing aims to ensure that the software corresponds to the specification and user need. Verification is the activity to check if the software satisfies the requirements, while Validation is the activity to assure that the requirements are correct and corresponds to the customers' expectations [Sommerville, 2010].

The test process is usually done by a tester team that works together with the development team. However, sometimes the tester team works separately [Society, 2014].

2.3.1 Test levels

Testing is an activity that can be performed during all the development process. For each stage of development they target a specific goal. According to Sommerville [2010] and Society [2014], the test levels are:

- **Unit Testing**: It aims to verify each source code unity, such as methods or objects, to ensure its expected behavior. Usually, it helps to find faults in the beginning of the software life cycle.
- **Component or Integration Testing**: It aims to check interactions among software components (modules). It focuses on testing the interface among the modules.
- **System Testing**: It aims to ensure the expected behavior of the entire software. Therefore it is focused on testing the interactions between the users and the system. It is also appropriate to check non-functional requirements.

2.3.2 Objectives of Testing

In general, testing activities can aim various objectives. Asserting objectives help to evaluate and control the test process. Moreover, testing can be proposed to verify either functional or non-functional properties. It is important to note that the objectives can vary in different test levels or test techniques [Society, 2014].

According to Society [2014] some of testing objectives are:

- **Acceptance Testing:** It assures that the software satisfies its acceptance criteria, usually the criteria are the customer requirements;
- **Performance Testing:** It is focused on checking whether performance is satisfiable and corresponds to its specification - for instance, if the data is received in less than 10 seconds;
- **Regression Testing:** It verifies whether the most recent modification to a system has introduced faults or it still conforms to the requirements;
- **Security Testing:** It aims to guarantee the confidentiality, integrity, and availability of the systems and its data;

- **Stress Testing:** It exercises the software in its maximum capacity.

Regression testing is going to be detailed in what follows because it is the focus of this work.

2.3.2.1 Regression Testing

During a software development process, the original code is modified several times — modules are added, removed or modified. Whenever a new SW version is released, it is of main importance to execute Regression tests to verify whether the previously existing features are still properly performed. Due to its nature of detecting bugs caused by changes, regression testing should consider all test levels and cover both functional and nonfunctional tests [Spillner et al., 2014]. Yet, as they are frequently repeated, test cases used in regression campaigns need to be carefully selected to minimize the effort of testing while maximizing the coverage of the modifications reported in the release note [Spillner et al., 2014].

To attend the above demands, regression test plans may contain a large set of test cases. However, it is not always feasible to execute large test plans, due to time constraints. Thus, it is very important to define criteria which lead to the selection of the most relevant TCs to reveal failures on the new SW version. For instance, we may choose to select TCs related to the modified areas in the code, since in general these areas are more unstable than the unmodified (and already tested) areas. As such, this criterion should increase the probability of selecting TCs able to find bugs. However, the work of Rothermel and Harrold [1996] shows that this is not a rule (see Section 2.3.3.2.1 for further discussion).

Regression tests can be classified in terms of the following aspects:

- **Test Case Selection:** this technique seeks to select a set of test cases relevant to the modified area of the SUT, that is, it is *modification-aware*. If the test case exercises an area modified in the SUT then it should be selected to compose the test suite [Yoo and Harman, 2012]. The formal definition follows [Rothermel and Harrold, 1996]:

  Definition 1. *Let P and P' be a program and its modified version, respectively, and T a test suite for P.*

  Problem: Find a subset of T, named T', such that T' becomes the test suite for P'.

- **Test Suite Minimization:** its goal is to reduce the number of test cases in a test suite by removing redundant tests. This means that when different tests exercise the same part of the SUT, just one of them should remain in the test suite. Also, this technique is called Test Suite Reduction [Yoo and Harman, 2012]. The formal definition follows [Rothermel et al., 2002]:

Definition 2. *Let T be a test suite, $\{r_1,...,r_n\}$ a set of test requirements, which must be satisfied to provide the desired "adequate" testing of the program, and subsets of T, $T_1,...,T_n$, associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i. Problem: Find a representative set T' of test cases from T that satisfies all r_is.*

- **Test Suite Prioritisation:** its goal is sorting a test suite to maximize a criterion, such as the rate of fault detection or the rate of code coverage, as fast as possible [Yoo and Harman, 2012]. The formal definition follows [Rothermel and Harrold, 1996]:

  Definition 3. *Let T be a test suite, PT a set of permutations of T, and f: $PT \rightarrow \mathbb{R}$ a function from PT to real numbers.*

  Problem: Find $T' \in PT$ such that $\forall T'': PT \mid T'' \neq T' \bullet f(T') \geq f(T'')$

Although we have presented the definition of test suite minimization, this work will not consider it.

2.3.3 Testing techniques

Testing techniques define which aspects and requirements of the software will be evaluated, and the depth of the analysis to be performed. We now present the two most common test techniques in the software industry: black-box testing (focus on requirements) and white-box testing (focus on code).

2.3.3.1 Black-Box Testing

According to Myers [2004], black-box testing is the act of checking whether a software product satisfies its requirements. However, the test must also perform unforeseen situations. Different from white-box, the black-box testing does not use the source code.

This technique allows inferring input conditions to exercise the requirements. Therefore, it is a technique related to selecting a test set that exercises the most significant software paths and finds the largest number of possible failures [Myers, 2004].

Black-box testing is accomplished during the system and integration testing stages and it aims the acceptance, beta, regression and functional testing objectives [Nidhra, 2012]. It is also complementary to the white-box testing technique, finding different kinds of failures. Thus, according to Pressman [2009], its main errors found are:

- incorrect or missing function;
- interface errors;
- errors in data structures or external database access;

- behavior or performance errors;
- initialization and termination errors

2.3.3.2 White-Box Testing

The source code is essential for defining white-box testing, because it is based on the internal code structure [Myers, 2004]. Thus, it can test aspects of the system that cannot be stated in the specification.

White-box testing is accomplished during the unit and integration testing stages and it aims the structural and regression testing objectives [Nidhra, 2012]. According to Pressman [2009], this technique helps the test architect to:

- Guarantee that all independent paths within a module have been exercised at least once
- Exercise all logical decisions on their true and false sides;
- Execute all loops at their boundaries and within their operational bounds;
- Exercise internal data structures to ensure their validity

In general, this technique uses a program representation known as a Control Flow Graph (CFG). Summarising, the CFG is a graph notation that represents any code fragment, as shown in Figure 3, whether it represents a simple function or a complex system.

Node	Statement
(1)	`while(x<100){`
(2)	`if (a[x] % 2 == 0) {`
(3)	`parity = 0;`
	`}`
	`else {`
(4)	`parity = 1;`
(5)	`}`
(6)	`switch(parity){`
	`case 0:`
(7)	`println( "a[" + i + "] is even");`
	`case 1:`
(8)	`println( "a[" + i + "] is odd");`
	`default:`
(9)	`println( "Unexpected error");`
	`}`
(10)	`x++;`
	`}`
(11)	`p = true;`

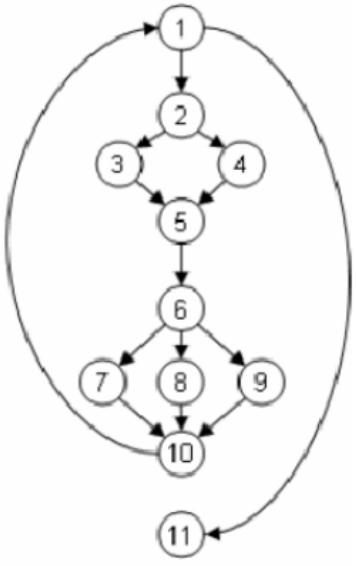


Figure 3 – Generic CFG example

2.3.3.2.1 Code Coverage Analysis

When white-box testing is applied, a metric known as code coverage is usually employed to characterize the degree to which some region of the source code of an application has been executed after a test campaign [Horváth et al., 2015]. It is a percentage, calculated by the division $\frac{Code_{exec}}{Code_{region}}$, exhibiting the amount of the source code that was executed ($Code_{exec}$) with respect to the total amount of source code that should be exercised ($Code_{region}$).

The higher the code coverage, the larger is the amount of source code exercised during a testing execution with respect to the source code region of interest. Although literature [Schwartz and Hetzel, 2016] has already pointed out that high code coverage does not necessarily means high bug detection, this measure suggests a lower chance of escaping undetected bugs compared to an application with lower code coverage. As such, code coverage used as a relative measure serves as a good indicator of a good test campaign.

Several different metrics can be used to calculate code coverage. For object-oriented applications, one can measure classes, methods, statements, conditionals, etc. This list is in the fine-grained direction; i.e., class coverage is less accurate than conditionals coverage. In this paper, we focus on coverage of methods, which has shown to be satisfactory to guide the selection of test cases based on code information provided by CRs, to complement selection based solely on IR from textual documents.

In general, code coverage is performed by instrumenting the source code of the application and then obtaining coverage data from the instrumentation part. There are several tools that provide such information, for instance, JaCoCo[1]. However, in some contexts, researchers are not allowed to perform instrumentation, since testing is not being conducted at development level. In such cases, it is only possible to perform a monitoring process which does not need to modify already compiled applications [Kell et al., 2012a]. For Android, we use ddmlib[2] tracers to listen to methods calls while performing a test case execution.

2.4 INFORMATION RETRIEVAL

Information Retrieval (IR) applications are being increasingly used in software engineering systems [Manning et al., 2008]. IR focuses on indexing and search: given a massive collection of information items (e.g., documents, images, videos), IR attempts to select the most relevant items based on a user query [Baeza-Yates and Ribeiro-Neto, 1999].

As the present work deals only with textual documents, we will concentrate on text indexing and retrieval. Note that building IR systems to deal with image, video and other

[1] http://www.eclemma.org/jacoco
[2] https://mvnrepository.com/artifact/com.android.tools.ddms/ddmlib

media involve different processing techniques which are out of the scope of the present work.

Usually, IR search engines count on two separate processes: (1) *Indexing* — Creation of the documents Index base; and (2) *Querying* and *Retrieval* — querying the Index base for data retrieval.

2.4.1 Creation of the documents Index base

When dealing with huge quantities of documents, it is clear that sequential search would not be appropriate. Thus, the first step to build an IR search engine is to index the corpus of documents in order to facilitate retrieval. Ordinary IR systems are based on an index structure in which words are used as keys to index the documents where they occur. Due to its nature, this structure is named as Inverted index or Inverted file.

The words used to index the documents constitute the Index base *Vocabulary*, which is obtained from the input corpus based on three main processes: tokenization, stopwords removal, and stemming (detailed below). Note that there are other preprocessing procedures which can be adopted (such as, use of n-grams and thesaurus) [Baeza-Yates and Ribeiro-Neto, 1999], however they have been less frequently used.

The *Tokenization* phase aims to identify the words/terms appearing in the input text. Usually, it considers blank and punctuation as separators. Following, duplications and irrelevant words to index the documents are eliminated. Usually, we consider as irrelevant words which are too frequent/infrequent in the corpus, or carry no semantic meaning (e.g., prepositions and conjunctions — usually named as *stopwords*). The vocabulary may still undergo a *stemming* process, which reduces each word to its base form. This process improves term matching among query and documents, since two inflected or derived words may be mapped onto the same stem/base form (e.g., *frequently* and *infrequent* will be reduced to *frequent*). Yet, this process usually reduces the vocabulary size.

The resulting set of words (Vocabulary) will then be used to index the documents in the corpus, originating the Index base. The aim is to allow for faster retrieval (see Section 3.1.1 for details).

2.4.2 Querying the Index base for data retrieval:

In this phase, the search engine receives as input keyword queries and returns a list of documents considered relevant to answer the current query. Ideally, this list should the ranked (particularly for large bases, which return a long list of documents).

However, not all search engines are able to rank the retrieved list of documents. This feature will dependent upon the underlying IR model used to build the search engine. Boolean models, for instance, are not able to rank the retrieved results, since they only classify documents between relevant or irrelevant. Thus, the response list will maintain the order in which the documents appear in the Index base. More sophisticated models, such

as Vector Space and Latent Semantics, count on ranking functions to order the retrieved documents according to each query [Baeza-Yates and Ribeiro-Neto, 1999].

Finally, the search engine can be evaluated by measuring the quality of its output list with respect to each input query. The ordinary measures are precision and recall.

2.4.3 IR underlying models

Given a set of textual documents and a user information need represented as a set of words, or more generally as a free text, the IR problem is to retrieve all documents which are relevant to the user.

The IR representation model may be given by a tuple [D, Q, $\mathfrak{F}$, R(q_i, d_j)] [Baeza-Yates and Ribeiro-Neto, 1999], where:

- D is a set that represents a collection of documents;
- Q is a set that represents the user's information needs, usually represented by keyword queries q_i;
- $\mathfrak{F}$ is the framework for modeling the IR process, such as Boolean model, Vector space, Probabilistic model;
- R(q_i, d_j) is the ranking function that orders the retrieved documents d_j according to queries q_i.

In this work we focus on the Vector Space Model (VSM), which is a classic and widely used model. It is more adequate than the Boolean model since it can return ranked list of documents including also documents that partially coincide with the query [Baeza-Yates and Ribeiro-Neto, 1999].

In this model, documents and queries are represented as vectors in a multi-dimensional space in which each axis corresponds to a word in the base Vocabulary. Similarity between the query vector and the vectors representing each document in the space aims to retrieve the most relevant documents to the query. Here, relevance is measured by the cosine of the angle between the vector representing the query and the vector representing a document. The smaller the angle between the two vectors, the higher the cosine value.

In ordinary VSM, the degree of similarity of the document d_j in relation to the query q is given by the correlation between the vectors $\vec{d_j}$ and $\vec{q}$. The $\vec{d_j}$ and $\vec{q}$ are weighted vectors ($w_{i,j}$ and $w_{i,q}$) associated with the term-query pair (k_i,q), with $w_{i,q} \geq 0$. This correlation can be quantified, for instance, by the cosine of the angle between the two vectors as follows: [Baeza-Yates and Ribeiro-Neto, 1999]

$$sim(d_j, q) = \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}| \times |\vec{q}|} = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{t}^{i=1} w_{i,q}^2}}$$

We have been using Solr[3] that is an information retrieval web system open-source provided by Apache. The Solr is a high-performance web-server built using Lucene. The Lucene is an open-source text-processing framework that implements these information retrieval techniques McCandless et al. [2010].

However, Solr uses by default the Lucene's model which have refined the VSM, where query-boost(q) is a boost that users can specify at search time to each query, doc-len-norm(d) is a different document length normalization factor which normalizes to a vector equal to or larger than the unit vector and doc-boost(d) is a boost that users can specify that certain documents are more important than others. Therefore, the score follows [Foundation, 2018]:

$$score(d_j, q) = query\text{-}boost(q) \cdot \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}| \times |\vec{q}|} \cdot doc\text{-}len\text{-}norm(d_j) \cdot doc\text{-}boost(d_j)$$

[3] http://lucene.apache.org/solr/

3 STRATEGY

This chapter aims to detail how the information retrieval and the code coverage was combined to create the hybrid process, called HSP. In Section 3.1 some necessary information to understand the HSP is presented. In Section 3.2, the HSP process is completely explained.

3.1 HSP: OVERVIEW

This section brings an overview of the HSP process for automatic TC selection and prioritisation for Regression Test Plans. As it can be seen in Figure 4, the HSP creates Test plans based on input CRs related to a SW application under test and accesses two databases: the Index base and the Trace base.

The strategy is going to be explained by two main processes: the Initial Setup run, which is executed once per each SUT; and the general Hybrid process, executed for all subsequent releases of the same SUT. It is important to notice that the setup run does not exist in the implementation code because it is just a phase when the trace base is empty for the SUT. Since the trace base is empty it is impossible to select test cases by code coverage.

In the setup run, the TCs are retrieved from a large indexed database of textual TCs (see Section 3.1.1). The queries to the TC repository are created based on keywords extracted from the input CRs. Each CR originates one query, which retrieves a list of TCs ordered by relevance.

The output lists are merged and ordered according to a combination of relevance criteria (Section 3.2.1). Following, redundancies are eliminated. This process was implemented as an information retrieval system, the ATP (Figure 6). The final TCs list is delivered to the test architect, who will create a Test Plan using the top n TCs, according to the available time to execute the current testing campaign.

The Test Plan execution is monitored by an implemented Android monitoring tool, which records the log of each executed TC (trace) (see Section 3.2.2). This information is persisted in a database which associates each TC to its trace. The Trace database is used by the general hybrid process, described below.

The hybrid process receives as input the CRs related to a new release of the SUT, delivering a Test Plan created on the basis of selection and prioritisation of TCs (Section 3.2.4). Initially, the ATP tool is used to retrieve new TCs based on the current input CRs. Following, it selects the TCs in the Trace base which matches code fragment associated with the input CRs. The available code coverage information is used to select and prioritise the TCs in the output list. The output TCs list is delivered to the test architect,

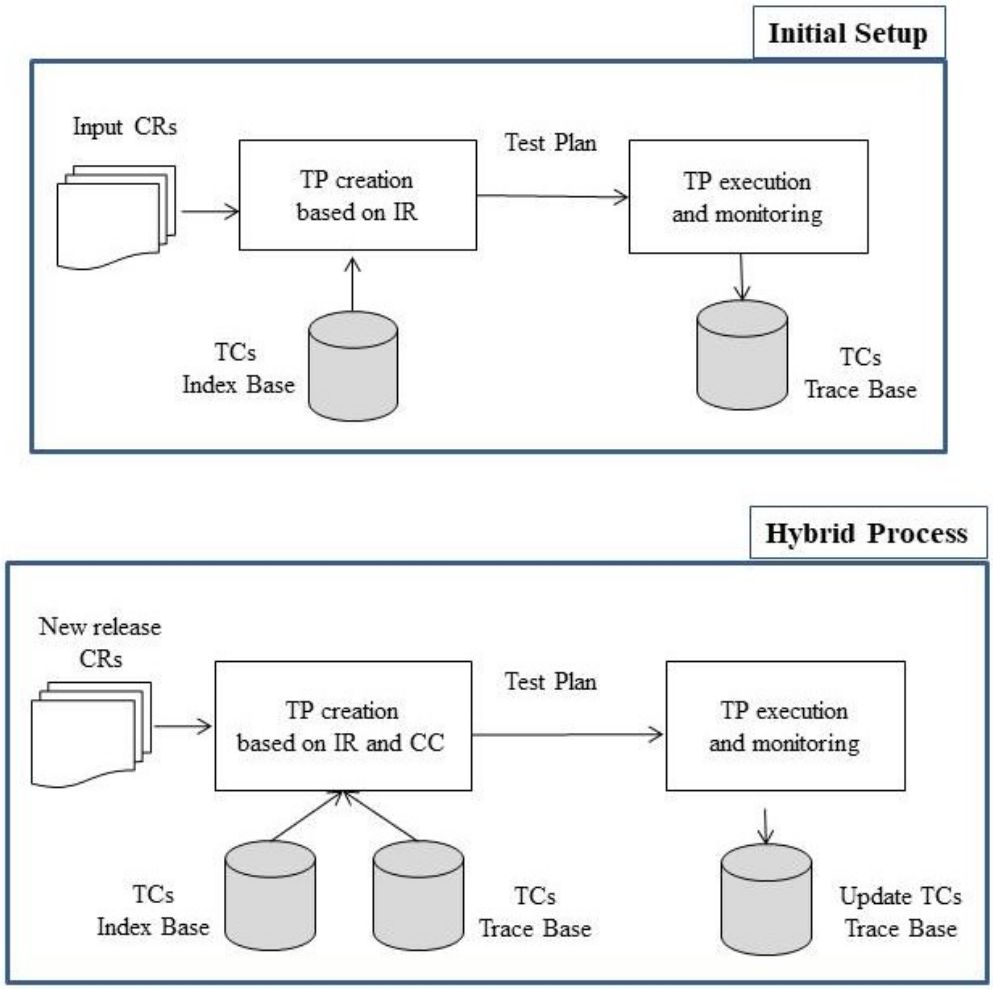


Figure 4 – Overview of the Proposed Process

who will guide the execution of the Test Plan in the same way as described for the initial setup run. All execution runs update the Trace database, including new code coverage information.

Details of each module will be presented in the subsequent sections: Section 3.2.1 presents the strategy for TCs retrieval and merging based on Information Retrieval; Section 3.2.2 shows the code coverage monitoring process; and Section 3.2.4 details the general hybrid process for Test Plan creation.

3.1.1 TCs Index Base Creation and Update

This section details the creation and update of the TC Index base, which is used by the HSP process to select and prioritise Test Cases based on IR techniques, as detailed in Section 3.2.1.

Index bases are data structures created to facilitate the retrieval of text documents based on keyword queries (Section 2.4). As already mentioned, in this structure each document is indexed by the words/terms occurring in its text. Whenever a query is

submitted to the search engine, all documents containing the words in the query are retrieved.

In our context, the corpus of documents to be indexed consists of a Master Plan with textual TCs descriptions. Each TC in the Master plan represents a document to be indexed. The Master plan is directly obtained from our partner's central TC database management system, and contains all available TCs related to the SW application under test.

The creation and update of the TC Index base is supported by a local indexing/search engine implemented using an Apache Lucene open-source IR software library[1]. We use the Solr Apache[2], which implements a modified VSM (see Section 2.4) where each dimension corresponds to one word/term in the vocabulary of the Index base.

The Index base Vocabulary is automatically obtained by the indexing/search engine, which selects the relevant words/terms present in the TCs descriptions. It is worth to mention that the user is able to edit the stopwords list, in order to add or remove any word or term of interest. Regarding the stemming process, the Solr already provides a stemmer (based on the Porter algorithm)[3], which can be activated when desired through the Solr settings.

This base is automatically updated on a daily basis, by consulting the central TC database using as input the current SW version under test. Note that the update of the Index base is an independent process, which is previous to its use by the Selection and Prioritisation process.

3.1.2 TCs Trace Base Creation and Update

After the Test Plan (TP) creation, the testers will execute each test case script in order to find failures and get the trace.

The TP execution is monitored by an implemented Android monitoring tool (Section 3.2.2.2), which records the log of each executed TC. This information is persisted in the TCs Trace base, which associates each TC to its trace (i.e., the methods exercised by the TC). As mentioned above, this information will guide the TC selection and prioritisation process based on code coverage.

Different from the Index base, the Trace base is not a general information base for any SW under test. Instead, it is closely related to the SW application being tested. This way, this base must be recreated for each new SW application which will undergo a Regression testing campaign. Yet, this base is only updated when a new TP is executed. The execution monitoring tool will provide new associations (TC - trace) to be included in the TCs Traces (see details in Section 3.2.2).

[1] https://lucene.apache.org/
[2] http://lucene.apache.org/solr/
[3] Porter algorithm: http://snowball.tartarus.org/algorithms/porter/stemmer.html

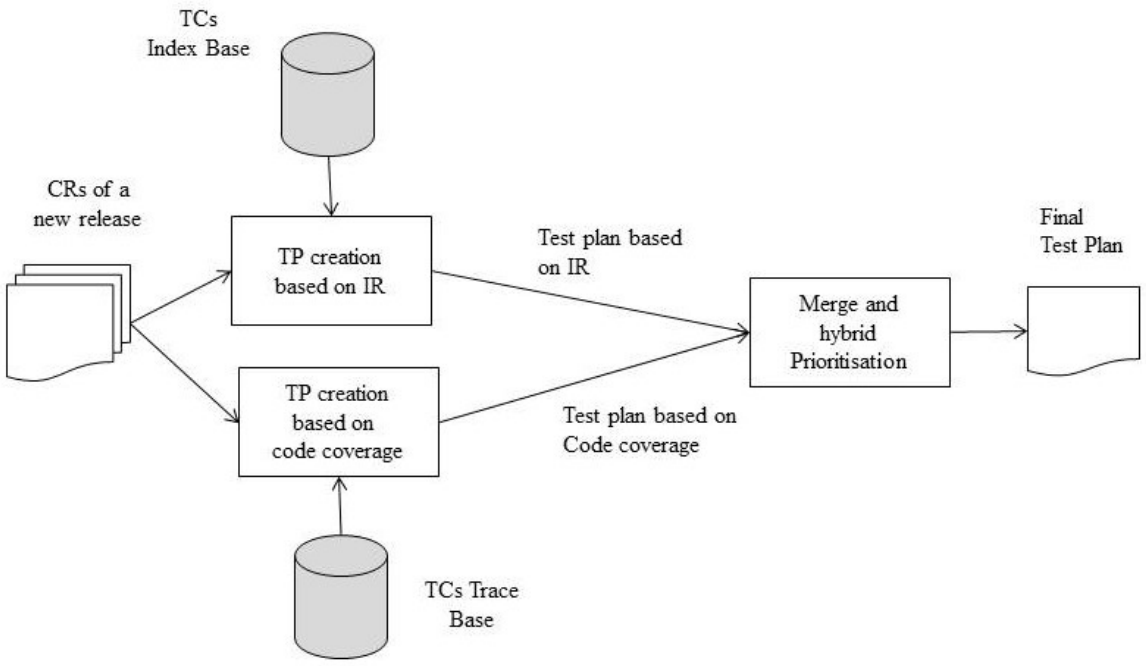
3.2 HSP: A HYBRID STRATEGY FOR TCS SELECTION AND PRIORITISATION

According to the related literature, regression Test Cases selection is more precisely performed using code related artifacts (modified source code and implemented test cases) [Di Nardo et al., 2015, Gligoric et al., 2015, Öqvist et al., 2016]. However, we also identified works which use Information Retrieval techniques for TC selection [Kwon et al., 2014, Saha et al., 2015].

In this light, we envisioned a way to benefit from both directions, creating a hybrid process based on IR and traceability information. The HSP process receives as input CRs related to a new SW release, and delivers a final Test plan based on hybrid selection and prioritisation. This process counts on three main phases (Figure 5):

- Test Plan creation based on Information retrieval (Section 3.2.1);
- Test Plan creation based on code coverage information (Section 3.2.3);
- Merge and prioritisation of the Test plans created by these two previous phases (Section 3.2.4).

Figure 5 – Hybrid process (selection and priorisation)



The final Test Plan will be executed and monitored (see Section 3.2.2). All execution runs update the TCs Trace database, including new code coverage information to be used by future runs.

The following sections provide details about the above listed phases. Although the Test plan execution and monitoring only occurs after the HSP process resumes, it will be

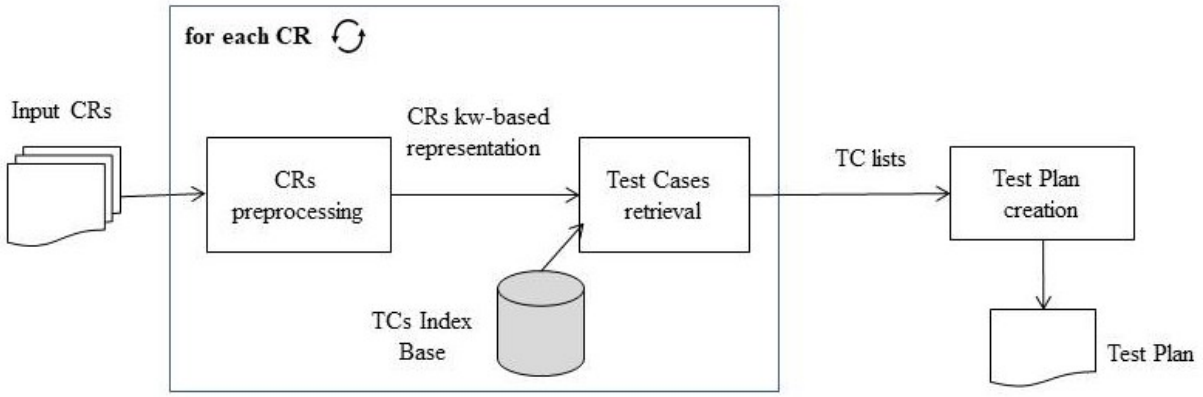


Figure 6 – TP creation based on Information Retrieval

necessary to detail the monitoring process before presenting the Test plan creation based on code coverage, which depends upon the information provided by the monitoring phase.

3.2.1 Test Plan creation using Information Retrieval

This phase receives the input CRs and returns a Test Plan. This process counts on three main phases, detailed in Figure 6. Initially, the input CRs are preprocessed, originating a keyword based representations for each CR. The obtained representations will be used in the second phase to build keyword queries to retrieve TCs from the TC index base (see Section 3.1.1). Finally, the third phase merges and prioritises the TC lists returned by the second phase.

Aiming to respect Software Engineering principles of modularity (providing extensibility and easy maintenance), the implemented prototype counts on three main processing modules, which correspond to the three phases mentioned above. The prototype was implemented using Django Bootstrap[4], a high-level Python Web framework, bearing an Model-View-Controller (MVC) architecture. Figure 7 shows the prototype's interface screen.

The three processing phases (modules) will be detailed in what follows.

3.2.1.1 CRs preprocessing.

This phase is responsible for the creation of the CRs keyword based representations. The corresponding implemented module receives the input CRs and delivers their keyword based representations, counting on three steps:

- **Step 1 (Keywords extraction)**: The relevant keywords that represent the input CR are automatically extracted from previously defined fields in the CR template -

[4] http://www.djangoproject.com/

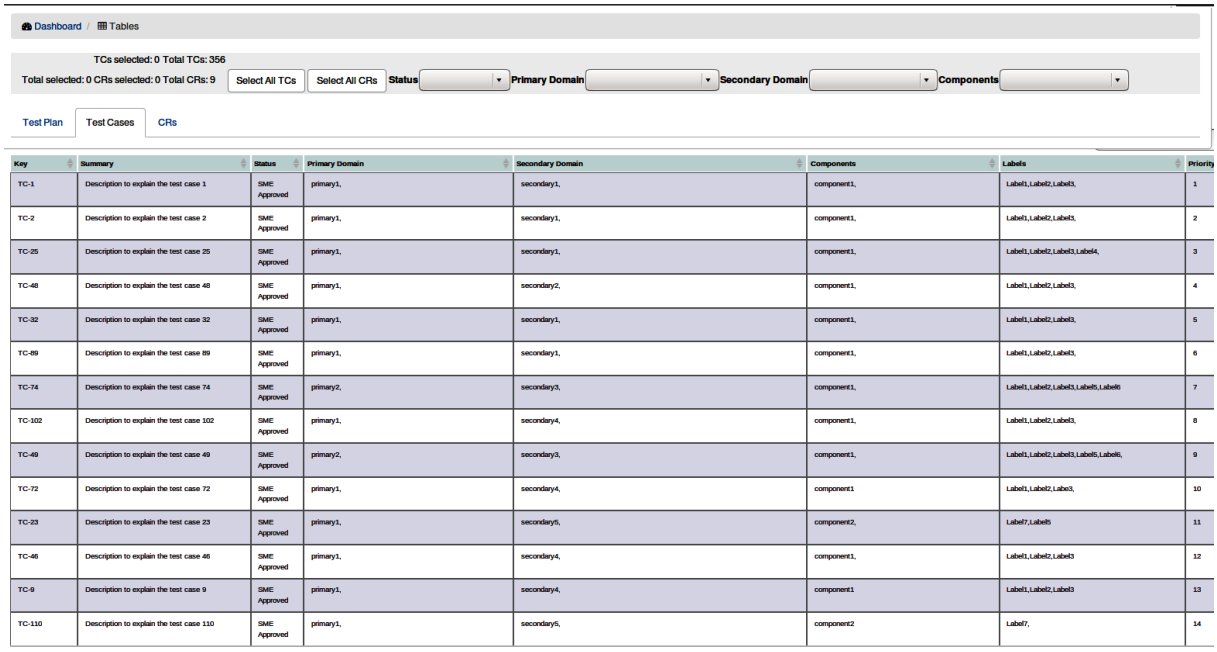


Figure 7 – The prototype's interface screen.

the ones with meaningful information for the task (e.g., title, product component, problem description). These fields were chosen with the help of a test architect. This step identifies each of the targeted fields and extracts its content, including the extracted words to the keywords list being created.

- **Step 2 (Stopwords elimination)**: Each CR representation is then filtered through the elimination of stopwords. These are words which are too frequent in the TC repository (and thus will not help to identify a particular test), or which are not relevant in the current application. Note that we must use the same list of stopwords used to create and update the Index base (Section 3.1.1).

- **Step 3 (Stemming)**: The resulting list of keywords may also undergo a stemming process, to reduce inflected or derived words to their word stem/base form (for instance, messages → message). This process favors a higher number of matches between the query and the indexed documents. As a consequence a larger number of TCs may be retrieved from the index base, originating longer Test Plans. Therefore, it should be optional and only used when necessary. Remind that this process should only be executed when it has also been executed in the generation of the TCs Index base (Section 3.1.1).

3.2.1.2 Test Cases Retrieval.

This phase receives as input the CRs keyword representations and delivers one query per CR. Following, each query is used to retrieve TCs from the Index base. The corresponding

implemented module counts on two steps:

- **Step 1 (Queries creation)**: This step creates one query per each input CR representation (a bag of words which may contain redundancies). Before duplications are eliminated, Keywords with more than one occurrence are positioned at the beginning of the query. This way, they will be more relevant to the retrieval process (mainly to the automatic ranking process performed by the search engine).

- **Step 2 (Queries processing)**: each query is individually submitted to the search engine, retrieving a list of TCs related to the corresponding CR. These lists are automatically ordered by the search engine according to the relevance of each TC to the current query (see Section 3.1.1). The obtained lists of TCs are given as input to Phase 3, described below.

3.2.1.3 Test Plan Creation.

This phase receives as input the ordered lists of TCs retrieved by the search engine (one list per query), and merges these lists, delivering the final Test Plan. Note that different queries (representing different CRs) may retrieve the same TC from the Index base. The existing duplications is eliminated. However, we understand that TCs retrieved by more than one query tend to be more relevant for the testing campaign as a whole. So, the duplicated TCs will be positioned at the top of the final ordered Test Plan.

The merging strategy developed in this work takes into account the position of the TC in the ordered list plus the number of times the same TC was retrieved by different queries. Section 3.2.1.3 and Figure 9 illustrate the merging strategy currently under use, considering as example a Regression test campaign based on 3 CRs. This strategy counts on 2 steps (regardless the number of input CRs):

- **Step 1 (Input lists merging)**: The input lists are initially merged alternating, one by one, the TCs from each input list (Section 3.2.1.3). The idea is to prioritise TCs well ranked by the search engine for each query. This way, the higher ranked TCs will be placed on the top of the Test Plan list, regardless the CR from which they were retrieved..

- **Step 2 (Duplications elimination)**: This step aims to eliminate eventual TCs duplication in the Test Plan. As mentioned before, the number of occurrences of each TC will influence on its final rank. Considering the current example (a test campaign based on 3 CRs), each TC can appear in the merged list 1, 2 or 3 times. TCs appearing tree times are positioned on the top of the list, followed by the TCs with two and then with one occurrence (see Figure 9).

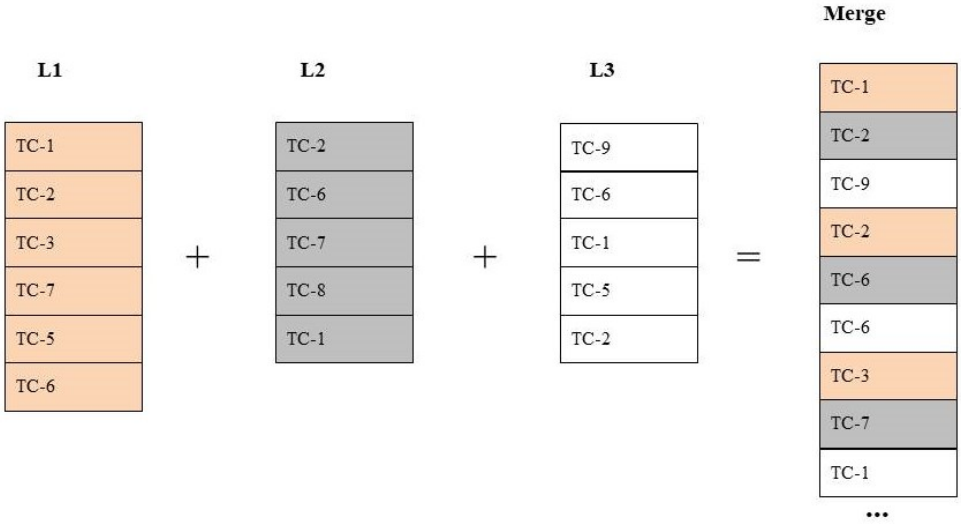


Figure 8 – Initial TC merged list

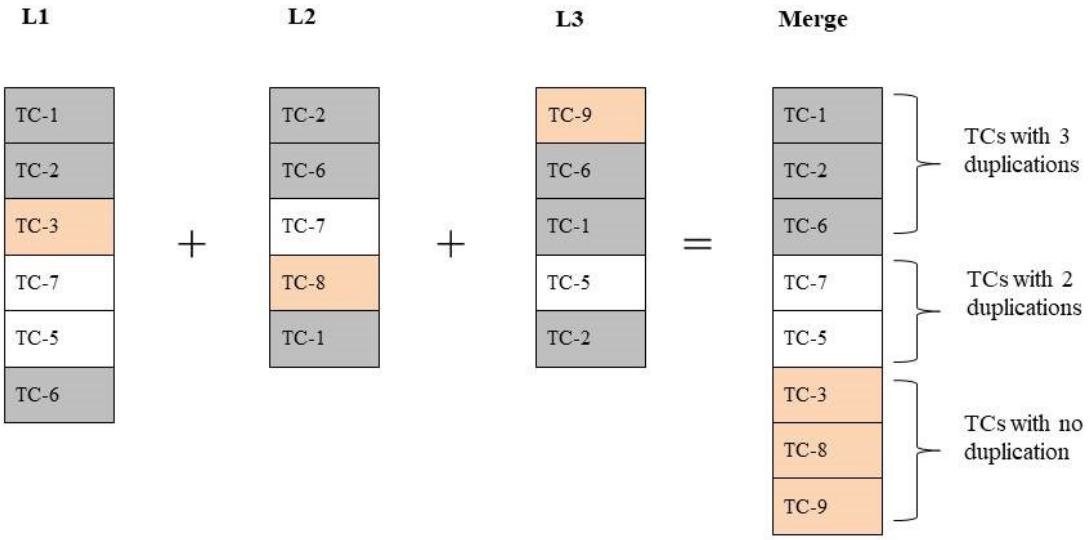


Figure 9 – TCs duplication elimination and reordering

The final merged list will constitute the Test Plan to be executed in the Regression test. The following section presents the monitoring strategy developed to obtain the TCs traces.

3.2.2 Test plan Execution and Monitoring

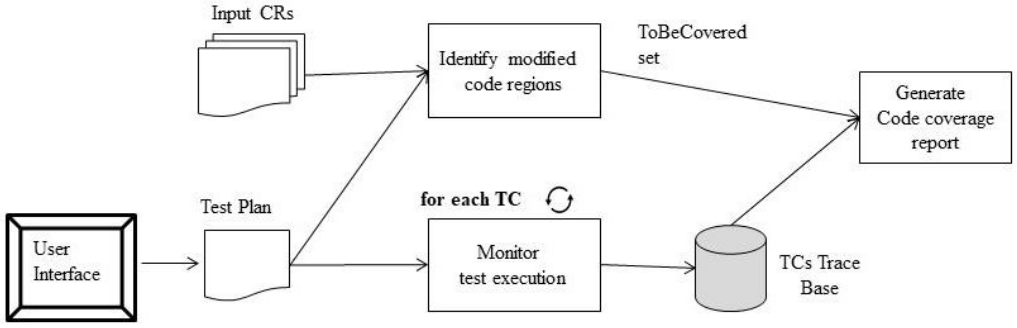
This section details the automatic monitoring of the TCs manual execution, which aims to provide a code coverage analysis of the Test Plan regarding the SW release under test. According to de Andrade Neto [2018], Perrusi [2018], this process is divided into three main phases, detailed in what follows (Figure 10):

- Identify the modified code regions among two different SW versions, which are

precisely the code segments that need to be exercised (tested) by the Regression campaign;

- Monitor the execution of the current TCs using the SW release under test, in order to create the Traces base.
- Generate the code coverage report.

Figure 10 – Monitoring Test Plans execution



The Test Plan execution is monitored by an implemented Android monitoring tool which records the log (trace) of each executed TC [de Andrade Neto, 2018, Perrusi, 2018]. This information is persisted in a database which associates each TC to its trace. The Trace database is used by Hybrid Selection and Prioritisation (HSP) described in Section 3.2.4.

As our industrial partner does not allow us to perform code instrumentation [Kell et al., 2012b], we could not use traditional code coverage tools (such as JaCoCo [Yang et al., 2006]). In this context, it was necessary to adopt an alternative way to obtain code coverage: dynamically monitoring code execution using profiling techniques [Kell et al., 2012a].

Since the implemented prototype is based on profiling techniques, it cannot be appropriately applied to time-sensitive tests, because the application may suffer from time performance degradation due to the monitoring process. However, this aspect did not impact the conducted experiments presented in this work (since they consist of regression testing) de Andrade Neto [2018], Perrusi [2018].

The implemented processing modules are detailed below.

3.2.2.1 Identifying the modified code regions

All modifications in the source code must be reported in the corresponding CR, which also brings a link to the corresponding modified code (stored at Gerrit repository[5]). From

[5] https://www.gerritcodereview.com/

these source code fragments, it is possible to identify which methods must be exercised by a Test Plan.

Although we are not allowed to instrument code, it is possible to access the application source code in the Gerrit repository. Thus, we can collect all the modified methods mentioned in the input CRs. This is indeed the hardest part of our analysis, since it is not allowed to identifying file differences (as provided by Git diffs). We use static analysis [Neamtiu et al., 2005] to detect new and modified methods, without taking into account source code line locations, spacing, etc. (see Algorithm 1). We keep these methods in a set named *ToBeCovered*.

Algorithm 1, named *ToBeCovered − CollectingMethodsModifications*, was implemented as the main procedure of this processing module.

Algorithm 1 ToBeCovered - Collecting Methods Modifications
Input: $TargetApp$ and $alfa$ version and *beta* version of App
Output: Set of modified Methods $ToBeCovered$

```
1:  function GetToBeCovered
2:      ToBeCovered ← ∅
3:      Repo ← UpdateRepoToVersion(TargetApp, beta)
4:      AllModifs ← GetRepoDiff(Repo, alfa, beta)
5:      JavaModif ← FilterJavaDiffs(AllModifs)
6:      JavaFileList ← GetChangedJavaFilesDirectories(JavaModif)
7:      MethodsInfo ← GetMethodsInfo(JavaFileList)
8:      For each JM : JavaModif do
9:          ChangedMethod ← GetModifMethods(MethodsInfo[JM.Dir], JM)
10:         if ChangedMethod ≠ ∅ then
11:             ToBeCovered ← ToBeCovered ∪ ChangedMethod
12:         end if
13:     end for
14:     return ToBeCovered
15: end function
```

Algorithm 1 implements a function named GetToBeCovered, which receives as input the application name, *TargetApp*, and two different software versions, *alfa* and *beta*, where *beta* is more recent than *alfa*. The result is a set of modified methods from *alfa* to *beta* versions, called *ToBeCovered*. These are the methods which should be covered in a Regression test.

Initially, the function goes through a setup phase. The *ToBeCovered* is initialized as an empty set, and *Repo* receives the reference to the repository in version *beta*, which is provided by function UpdateRepoToVersion. The variable *AllModifs* receives a list of all modified files from *alfa* to *beta* versions, provided by function GetRepoDiff using *Repo* and both versions. However, as we need only Java files modifications, filterJavaDiffs is used to instantiate *JavaModif* with all Java modifications from the source code in Gerrit.

Following, *JavaFileList* receives the paths of all modified Java files using GetChangedJavaFilesdirectories, such that it is now possible to navigate through those paths and collect information about Methods. Navigation through file paths is conducted by the GetMethodsInfo function, which keeps in the *MethodsInfo* a list of each method declaration, its first and last line, its return type and which parameters the method receives.

Finally, it is necessary to cross information between the *JavaModif* and *MethodsInfo* list to check which method in the application the java code modifications belongs. For each Java Modifications *JM* from *JavaModif* and the *MethodsInfo*, the GetModifMethods function selects only the modified methods, which are then added to the *ToBeCovered* initial set. This way, the algorithm obtains a more specific list of methods that were modified between the two SW versions.

3.2.2.2 Monitoring test executions

As seen above, the GetToBeCovered function is able to identify a collection of source code methods that needs to be exercised in the Regression test, and thus should guide the Test Plan creation. However, it is important to remind that we only count on textual Test Cases, which do not carry information about source code. Thus, the obtained methods cannot be directly used to identify which TCs in the database are more closely related to those methods.

To fill in this gap, it is necessary to monitor the execution of each TC in the Test Plan created using only Information Retrieval (Section 3.2.1), in order to obtain the sequences of method calls. To track all methods called during a test execution, we use the Android Library *ddmlib*[6], which provides Android background information about runtime execution.

The monitoring process counts on four main steps:

1. The device is connected to a computer;

2. The Android Debug Bridge (ADB) is started, and receives as a parameter the application package to be monitored. Note that it is necessary to indicate all packages related to the application, and these packages must be visible to the ADB, so that the right trace to these packages can be tracked.

3. To start log trace files, the method $StartMethodTracer$ is called. This method enables the profiling on the specific packages.

4. The log files (sequences of method calls—$Full_{seq}$—involving methods inside or outside the modified region; that is, methods in or out the $ToBeCovered$ set) from the

[6] https://android.googlesource.com/platform/tools/base/+/tools_r22/ddmlib/src/main/java/com/android/ddmlib

above step are persisted. There will be one log file associated to each executed TC registering the tuple (TC, OS, device, package) where the TC is the Test Case (TC), the OS is the operational system, the device is the smartphone under test and the package is the application under test. Since we need to know which areas of the application were exercised by each TC. This information will guide a more precise selection of TCs to create Test Plans (see Section 3.2.4).

Based on the monitoring results, it is possible to create an association between each TC (TC_k, where $k \in TestPlan_{Index}$) and the exercised methods, such as:

$$TC_k \mapsto m_1(\cdot); m_2(\cdot); \ldots ; m_T(\cdot)$$

These associations are persisted in the Trace base, which will be used by the Hybrid process to select and prioritise Test Cases based on code coverage information (Section 3.2.4).

3.2.2.3 Generating the Code Coverage Report

Code coverage may be reported at the Test Plan level, or at the CRs level, which is a more fine grained measure.

- **Test Plan Coverage**. The log files provided by the previous module are filtered based on the identified modified regions (that is, we compute the new set $Modif_{seq}$). Assume that $Seq \downarrow Set$ yields a new sequence resulting from the sequence Seq by preserving only the elements in the set Set.

$$Modif_{seq} = Full_{seq} \downarrow ToBeCovered \tag{3.1}$$

  The Test Plan code coverage is given by the ratio between the number of elements in $Modif_{seq}$ and the set $ToBeCovered$, for each test case, as follows:

$$TP\_Coverage = \frac{\#Modif_{seq}}{\#ToBeCovered} \tag{3.2}$$

  The overall coverage must take into account the full methods names (package.class.method) appearing in more than one test trace sequence. Avoiding counting the same coverage method numerous times.

- **CR Coverage**. The CR coverage is obtained in a similar way as in the previous case. However, instead of using all methods in $ToBeCovered$ set, we consider only the methods modified by the present CR.

$$CR\_Coverage = \frac{\#Modif_{seq}}{\#CR\_ToBeCovered} \tag{3.3}$$

  The CR coverage shows in more detail which areas have been exercised and what remains uncovered after the execution of a Test Plan.

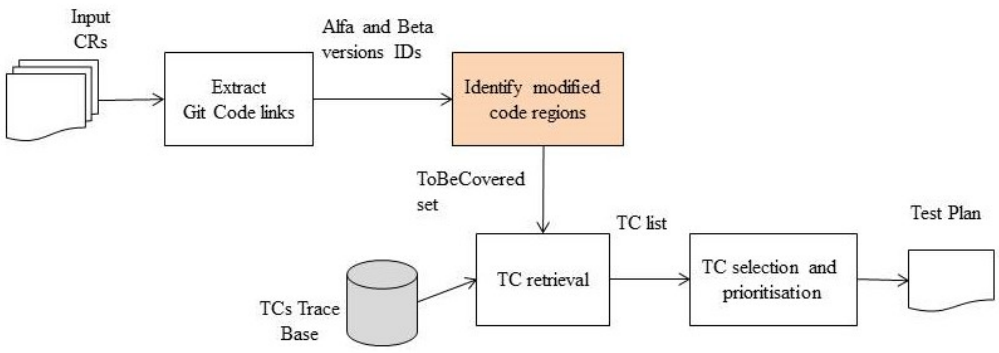
3.2.3 Test Plan creation based on Code Coverage

This phase receives as input CRs related to a new version of the same SW product whose initial version has been submitted to the Setup Process (Section 3.1), and uses the Trace base to perform the selection of the related test cases. Note that the Trace base can only use information related to several versions of a single product. It would be misleading to use code coverage information about a SW product to select TCs related to a different one since each product is singular (otherwise, it would just be a version of a previous SW).

Figure 11 depicts the overall process of Test Plans creation based on code coverage, which counts on four steps:

- Extract Git code links from the input CRs;
- Identify modified code regions (Section 3.2.2.1);
- Retrieve TCs from Trace base;
- Select and prioritise TC based on code coverage.

Figure 11 – TP creation based on Code Coverage



Two different selection strategies have been implemented: Selection using total coverage (CCS_t), and Selection using additional greedy (ATC_g). Note that the three initial steps are independent from the selection strategy adopted, being performed in the same way in both cases. The following sections will provide details on these steps.

3.2.3.1 Selection using total coverage (CCS_t)

Based on the coverage information obtained by the execution monitoring procedure (Section 3.2.2), it is possible to create a relationship between each input CR (which contains a subset of modified methods—i.e., a subset of $ToBeCovered$), and the corresponding subset of test cases (i.e., those TCs that, when executed, exercise exactly these methods). These relationships are stored at the Trace base, as already mentioned.

Initially, it is necessary to create the *ToBeCovered* set. This is accomplished by traversing all CRs and collecting the methods that were modified in the current SW version to the previous version. Following, the TCs are selected based on the *ToBeCovered* set and the information available from the Trace base.

Algorithm 2 Selection of test cases using coverage data

Input: Two lists: modified methods (*ToBeCovered*) and test cases (*tests*)
Output: A list of test cases to compose a test plan (*plan*)

```
1: function SelectPlanFromCode
2:     plan ← []
3:     For each tc ∈ tests do
4:         if tc.methods ∩ ToBeCovered ≠ ∅ then
5:             plan.append(tc)
6:         end if
7:     end for
8:     return sortedByCoverage(plan)
9: end function
```

The selection strategy described in Algorithm 2 is implemented by traversing TCs that could run in the SUT ($tc \in tests$) and then checking whether a test case *tc* has associated covered methods belonging to the *ToBeCovered* set (when the intersection becomes not empty). If this happens, the TC will be included in the Test Plan (*plan.append*(*test*)).

Finally, the TCs in the Test Plan are prioritised (*sortedByCoverage*(*plan*)) based on the individual coverage rate of each TC with respect to the most recent coverage data available from the Trace base.

Remind that the obtained CC-based Test plan will still be merged to the IR-based Test plan already created by Phase 1 of the hybrid process (Figure 5).

3.2.3.2 Selection using additional greedy (CCS_g)

The strategy described here (Algorithm 3) selects test cases based on a global coverage measure, instead of considering the TC individual coverage used by the previous strategy. The algorithm selects the TCs which increase the overall coverage of the Test Plan (i.e., those TCs that exercise the methods in the set *ToBeCovered*, not yet covered by another TC already in the Test Plan).

This algorithm receives as input the *ToBeCovered* and *tests* sets, and returns a list named *greedy_selection*. This list is initially empty. The variable *frontier* is initialised with the set *tests*. The *additional_methods* set, initially empty, will hold all methods exercised by the tests in *greedy_selection*.

After the variables initialisation, the algorithm performs a loop which stops when there is no additional test to consider. Within the loop, the relative coverage is calculated. The variables *maximum_additional* and *additional_tc* are updated when there is coverage to consider and a higher coverage rate can be achieved, respectively. If this holds, the

Algorithm 3 Selection of test cases using additional greedy

Input: Two lists: modified methods (TBC) and all test cases ($tests$)
Output: A list of test cases to compose a test plan ($plan$)

```
function GreedyAdditionalCoverage
    greedy_selection ← []
    frontier ← tests
    additional_methods ← ∅
    repeat
        maximum_additional ← 0
        additional_tc ← None
        For each tc ∈ frontier do
            coverage ← #((tc.methods ∪ additional_methods) ∩ TBC)/#TBC
            if coverage > maximum_additional then
                maximum_additional ← coverage
                additional_tc ← tc
            end if
        end for
        if additional_tc ≠ None then
            greedy_selection.append(additional_tc)
            additional_methods ← additional_methods ∪ additional_tc.methods
            frontier.remove(additional_tc)
        end if
    until additional_tc = None
    return greedy_selection
end function
```

TC under analysis is added to the *greedy_selection*, the TC corresponding methods are added to the *additional_methods* and the $frontier$ is update by removing this TC from it.

3.2.4 Test Plans Merge and Hybrid Prioritisation

This is the last phase of the HSP process. It comprehends the merge and prioritisation of the test plans created in sections 3.2.1 and 3.2.3.

Although selection by code coverage is more precise than selection by information retrieval, there are two main threats if we consider only the selection by code coverage:

1. Tests that cannot be monitored, due to restrictions of the Android monitor (such as restart, unplug the USB cable, recharge), will never be selected, since there will be no coverage data provided by Algorithm 2 or Algorithm 3.

2. Tests never selected before (or new) cannot be related to source code methods in the CRs because they had not been traced.

To prevent these problems, we propose two hybrid strategies: Merge and Priotitisation using Code Coverage Ranking (MPCCR) and Merge and Priotitisation using Information

Retrieval Ranking (MPIRR). Figure 12 illustrates both hybrid strategies. Note that the selection by code coverage can be a result of the selection using additional greedy ($\mathbf{CCS}_g$) or the selection using total coverage ($\mathbf{CCS}_t$). In this case we add the subscript to indicate which case is being considered in what follows.

3.2.4.1 Merge and Priotitisation using Code Coverage Ranking (MPCCR)

This prioritization assign importance to the code coverage ranking. Therefore, the code coverage selection order is maintained and it will be complemented by the information retrieval selection.

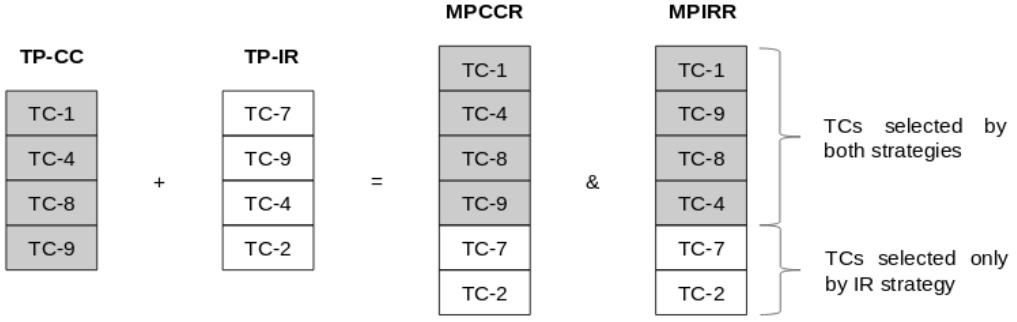
In this merge and prioritisation we perform what follows. Consider that a selection based on code coverage produces the following list of test cases $\langle TC_1, TC_4, TC_8, TC_9\rangle$, and the list $\langle TC_7, TC_9, TC_4, TC_2\rangle$ by information retrieval. The final list will be $\langle TC_1, TC_4, TC_8, TC_9, TC_7, TC_2\rangle$, where TC_4 and TC_9 maintain the same position as in the initial code coverage selection. As said previously, we can have MPCCR_t (total coverage) and MPCCR_g (additional greedy).

3.2.4.2 Merge and Priotitisation using Information Retrieval Ranking (MPIRR)

This prioritization assign importance to the information retrieval ranking. Therefore, the code coverage selection order is altered by similar information retrieval selection test cases and it will be complemented by the remaining information retrieval test cases. However, the code coverage selection is at the top of the plan because it is directly related to the modified source code while the information retrieval selection is related to the test case keywords.

In this merge and prioritisation we perform what follows. Consider the same lists as before. Using code coverage we have $\langle TC_1, TC_4, TC_8, TC_9\rangle$, and using information retrieval we have $\langle TC_7, TC_9, TC_4, TC_2\rangle$. The final list then becomes $\langle TC_1, TC_9, TC_8, TC_4, TC_7, TC_2\rangle$ where TC_4 and TC_9 follows the information retrieval ordering. As said previously, we can have MPIRR_t (total coverage) and MPIRR_g (additional greedy).

Figure 12 – Hybrid prioritisation



4 IMPLEMENTATION

This chapter aims to explain how the tool was implemented and how it works. It is worth recalling that the implementation presented in this chapter was focused in the smartphone industry, because of our industrial partner, but it can be generalised to any other context as well.

4.1 DEVELOPMENT

The HSP process was implemented in a tool called ATP. The ATP was implemented in a web environment so that architects in different sites could create their test plans in an easy way. In addition, web tools are easier to maintain and update.

This tool was developed using Django[1], a high-level Python Web framework, bearing an MVC architecture that are widely used in web development. It was also used the Solr[2] system for indexing and retrieving the test cases and a MySQL[3] database to store the data.

4.1.1 Architecture

The ATP tool was developed seeking to use current best practices. Therefore, it works on a Docker platform, which assists the portability of the tool. Docker enables the packaging of an application or entire environment within a container, hence it becomes easily portable for any other Docker Host [Docker, 2019].

This tool has 4 containers in the docker, where each container contains an application. They are:

- **MySQL:** It is the container that contains the database responsible for storing the tool data;
- **Django:** It is the container that contains the Django code responsible for selecting and prioritizing test cases;
- **Solr:** It is the container that contains the Solr responsible for indexing and retrieving test cases;
- **Nginx:** It is the container that contains the Nginx[4] responsible for web serving the tool.

The ATP architecture is shown in Figure 13.

[1] http://www.djangoproject.com
[2] http://lucene.apache.org/solr/
[3] https://www.mysql.com/
[4] https://www.nginx.com/

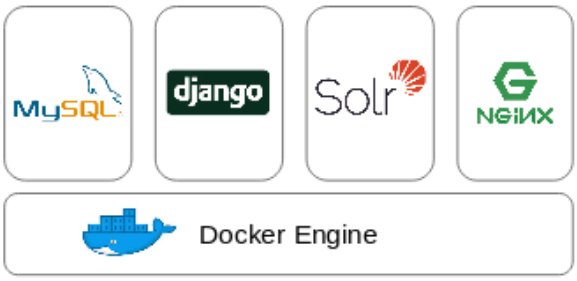


Figure 13 – ATP architecture

The MVC architecture is widely used for separating the user interface, application data and its functionalities [Jacyntho et al., 2002]. The model contains the application data and processing logic. The view is the user interface that presents the content to the end user. The controller manages the requests flows to views and models [Pressman, 2009]. A representation of the MVC architecture is shown in Figure 14.

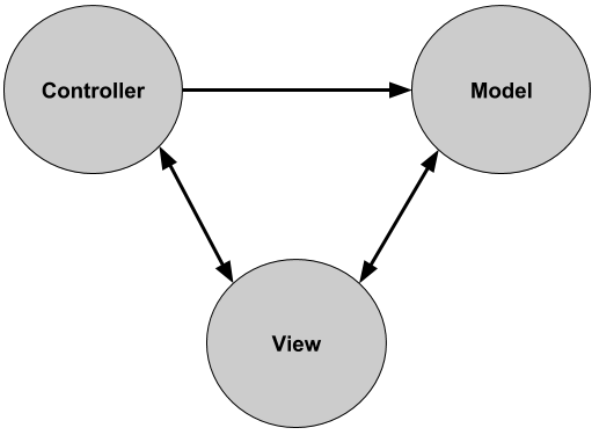


Figure 14 – MVC architecture

4.2 TOOL

This section presents the ATP tool screens, and its usage. It also brings the ATC main screen.

4.2.1 Test Case Searching

For searching test cases it is necessary to select which release notes will include in the test plan, the project which this test plan will be part of and the chosen device for this test plan (shown in Figure 15).

When the user asks the tool to search for TCs, the tool traverses the release notes reading each CR content to get its keywords and code changes. It also maps its component to the test management tool component (detailed in Section 4.2.3). Then, the CRs

contents are used as queries for retrieving the test cases following the process explained in Section 3.2.

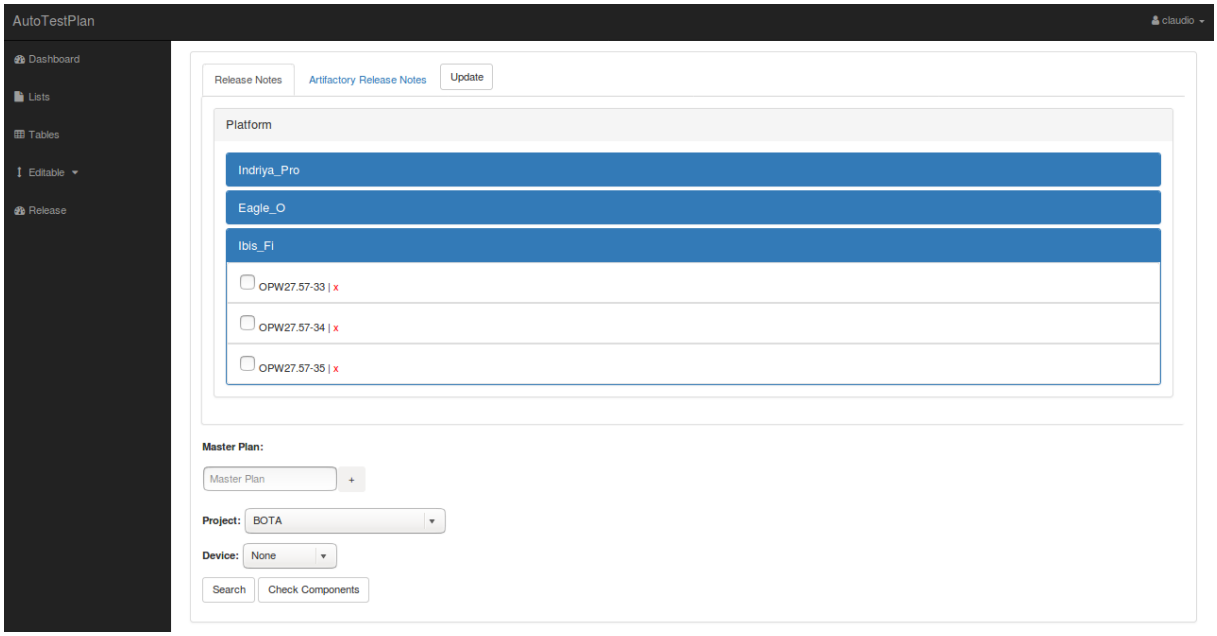


Figure 15 – Test Case Searching Screen

4.2.2 Test Plan Creation

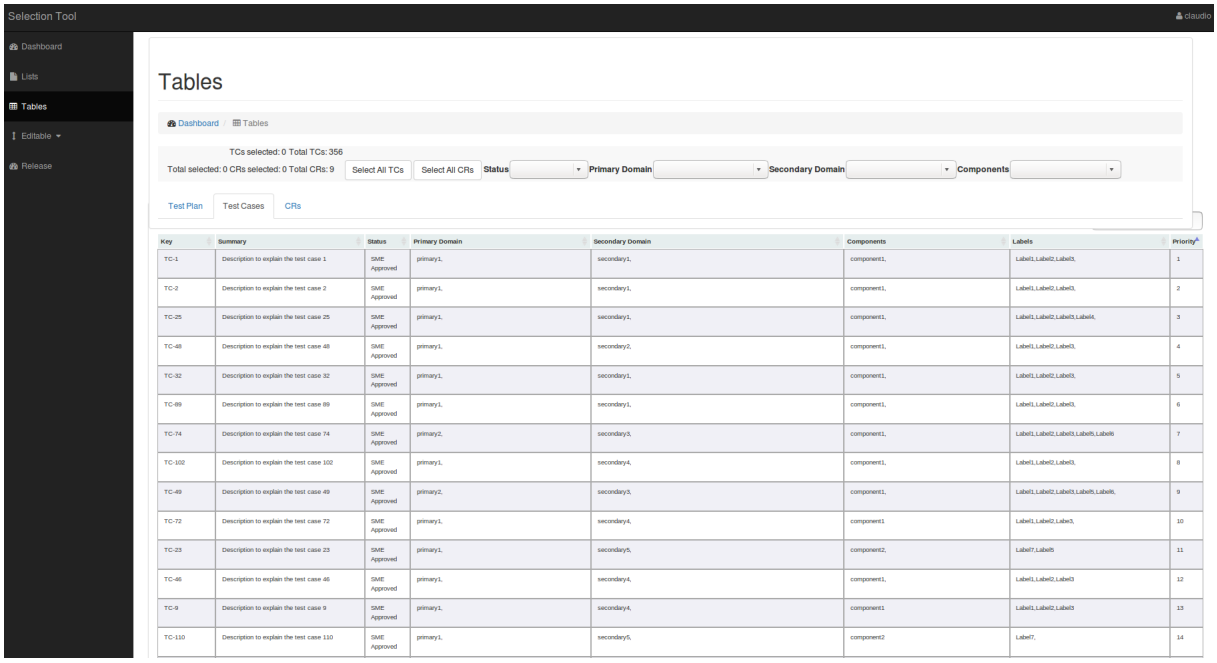


Figure 16 – Test Cases Screen

When the most appropriate test cases for the CRs in the RN are selected, they are presented in a prioritised way, as shown Figure 16. At this point, the architect chooses the amount of test in the tool suggestion list that is possible to exercise, due to the limited human resources for executing manual test cases. It is important to note that it is impossible to select test cases that are not in the suggested test plan.

After that, the user has to fulfill the test plan form with information such as title, purpose, software, regression level, initial and final date (as shows Figure 17). After fulfilling these elements, the test plan is created in the ATP and it is also exported to the test management tool.

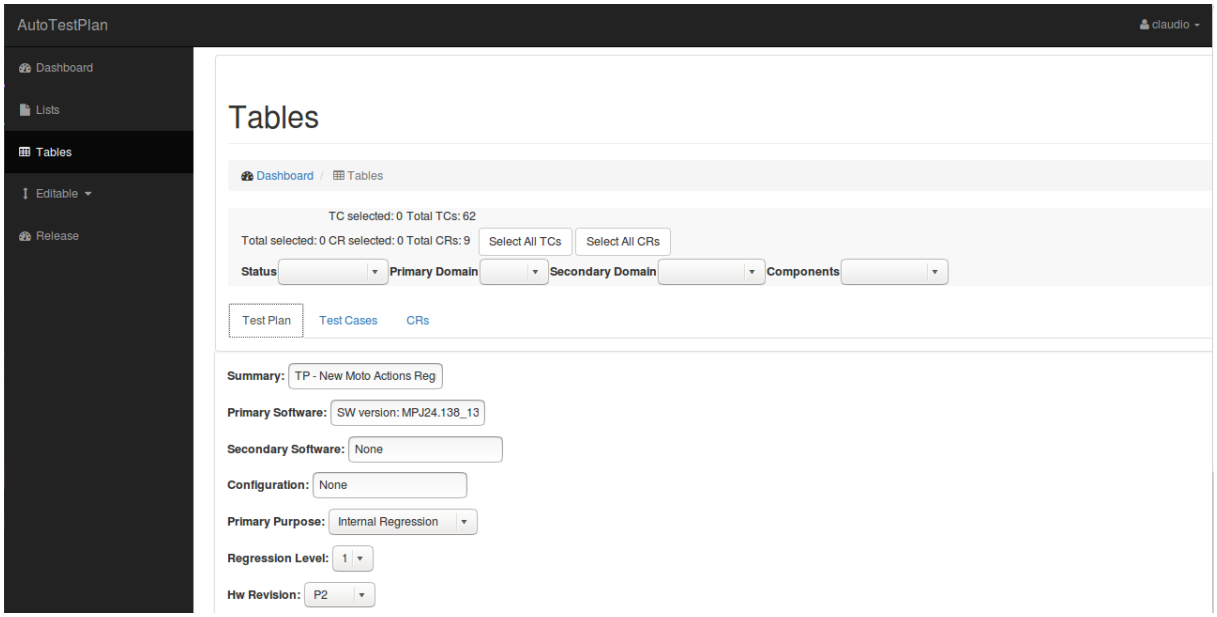


Figure 17 – Test Plan Screen

4.2.3 Mapping Components

The system creates a mapping from the component of the CR management tool to the component of the test management tool. This mapping turns the information retrieval part of ATP more precise by firstly focusing on the specific components in question.

This mapping is a relationship between a CR component and a test component. For creating such a mapping, the user has to select which CR component corresponds to one or many test components and for which project it belongs to (as shown Figure 18). The mapping varies for each project.

4.2.4 Release Notes Updating

Aiming to help the user to avoid manually upload release notes documents, the ATP tool has a web crawler that finds release notes for a specific product registered in the system.

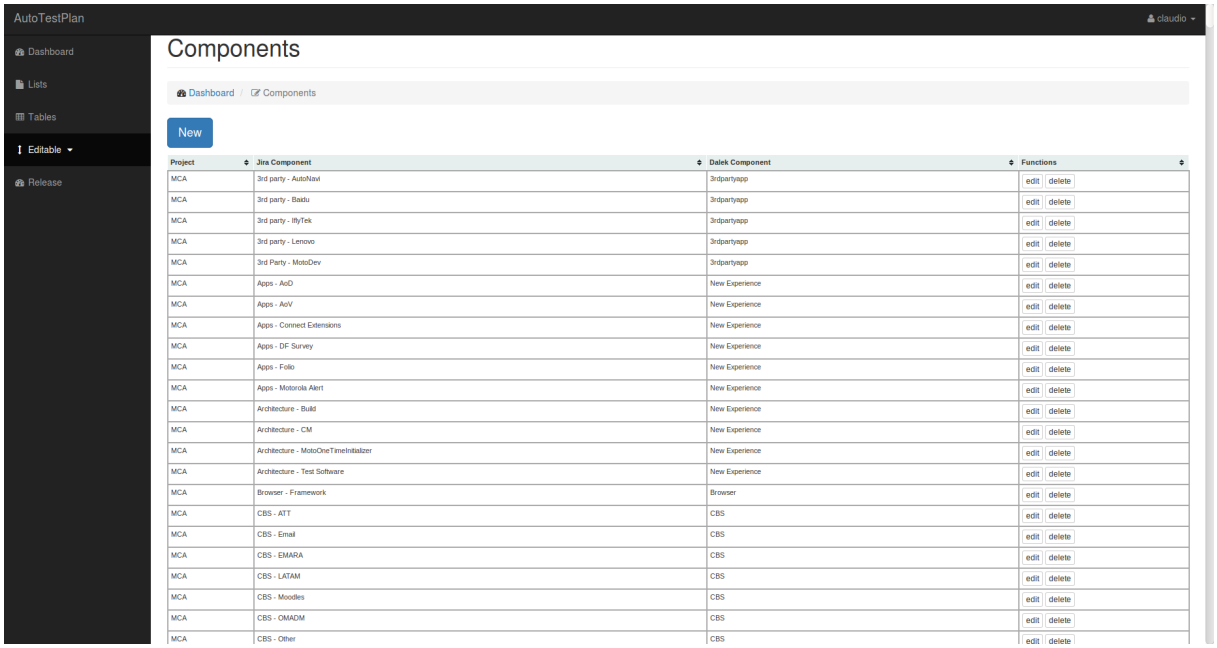


Figure 18 – Components Screen

A web crawler is a software that sends requests to other servers for downloading the pages automatically. This means that it navigates through URLs, searching and collecting usefull information in those pages Baeza-Yates and Ribeiro-Neto [1999]. This functionality is not mandatory for the operation of the tool because the user can upload the corresponding information manually. However, this feature saves time.

When the the user registers the link corresponding to the product repository, the tool access the repository every day, searching for the newest release notes. Then, it saves the link, reads its content and shows it as detailed in Section 4.2.1.

4.2.5 ATC: Monitoring Test Case

The AutoTestCoverage (ATC) [de Andrade Neto, 2018, Perrusi, 2018] is the tool that monitors test cases execution, aiming to get the test case trace (sequence of methods calls) as explained in more detail in Section 3.2.2. As shown in Figure 20, the tool lists all the test cases of a test plan given by the user. The tester, for each test case, press "play" to start recording the trace and press "stop" at the end of the test case. The tool stores a local copy of each trace file which will be sent to the trace base (see Section 3.1.2) by pressing the "send data" button. To record a trace, it is necessary that the device is connected to the computer (by USB or Wi-Fi).

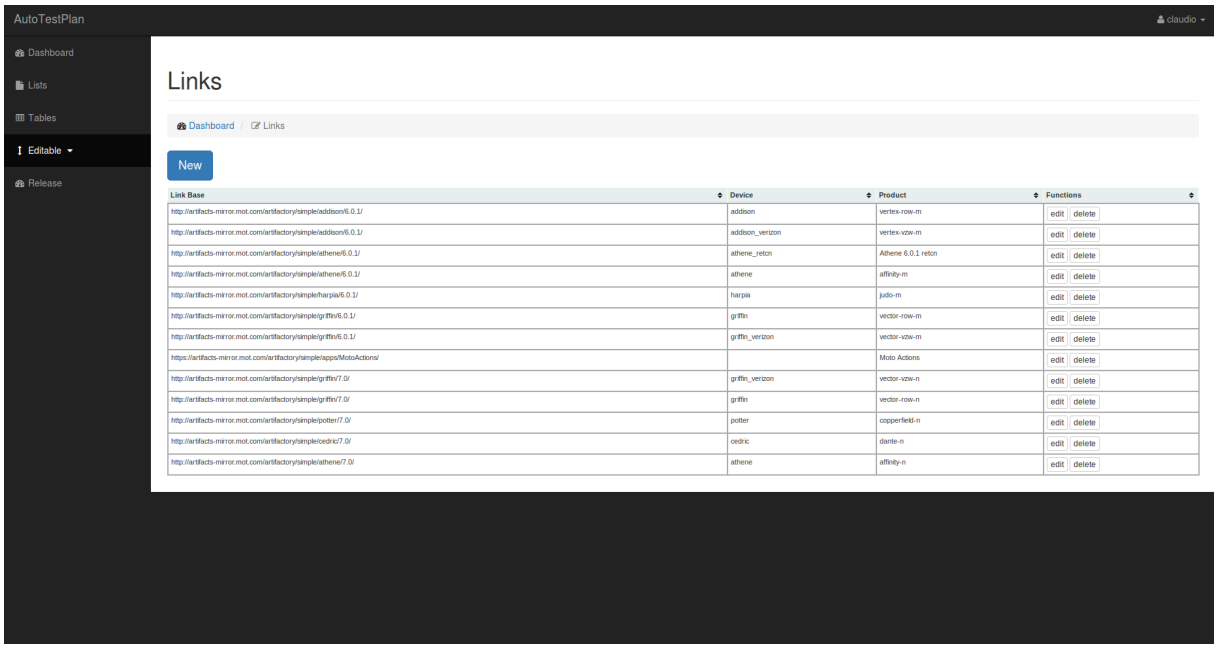


Figure 19 – RNs updating screen

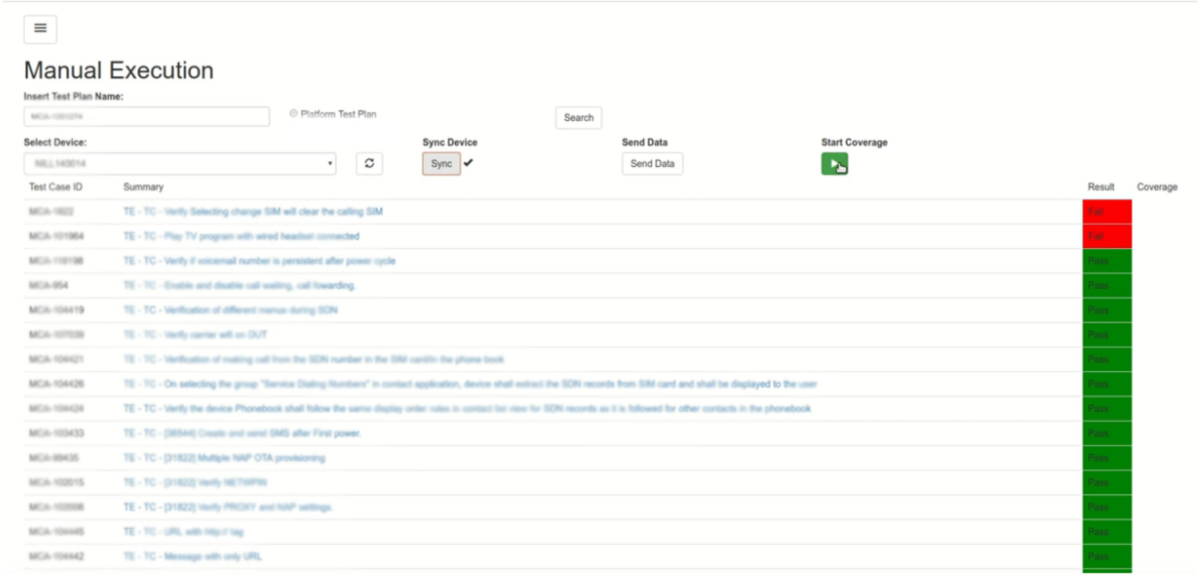


Figure 20 – ATC main screen

5 CASE STUDY

This chapter presents an empirical evaluation of the selection and prioritisation strategies presented in this work.

5.1 EMPIRICAL EVALUATION

For the evaluation we consider the test plan creation using only Information Retrieval (IR), described in Section 3.2.1, and the test plan creation using only code coverage (CCS), described in Section 3.2.3. In this case we also consider the selection by total coverage (CCS_t) and the selection by additional greedy (CCS_g). Finally, we consider our main proposal, the test plan merge and hybrid prioritisation by preserving the code coverage ranking (MPCCR) and by preserving the information retrieval ranking (MPIRR).

As baselines, we use a random selection as well as selection by human architects. Due to operational difficulties to perform an experiment in a real industrial context, we could only access a specific application and its source code as a reader. This application is associated with triggering actions in the smart phone from gestures of its users. We considered two consecutive test executions, without and with code coverage data (respectively, Setup run and Hybrid run).

In what follows, we present some Research Questions (RQ) that we intend to answer in this work.

1. **RQs related to the Setup run:**
    - **RQ1:** Considering random, architects and IR based selections, which one does exhibit the highest coverage of modified regions when
        a) All tests are executed?
        b) The available operational capacity is executed?
    - **RQ2:** Among IR based, random and architects selections, which one does detect the highest number of failures when
        a) All tests are executed?
        b) The available operational capacity is executed?
    - **RQ3:** What is the correlation between each TC code coverage and its position in the ranked list provided by the IR based selection when
        a) All tests are executed?
        b) The available operational capacity is executed?
    - **RQ4:** Regarding each CR, what is the coverage reached by architects and IR based selections when

a) All tests are executed?

b) The available operational capacity is executed?

2. **RQs related to the Hybrid run:**

- **RQ5:** Considering IR based, code coverage based, hybrid, random and architects selections, which one does exhibit the highest coverage of modified regions when
  a) All tests are executed?
  b) The available operational capacity is executed?
- **RQ6:** Regarding IR based, code coverage based, hybrid, random and architects selections, which one does detect the highest number of failures when
  a) All tests are executed?
  b) The available operational capacity is executed?
- **RQ7:** What is the correlation between each TC code coverage and its position in the different ranked lists provided by IR based, code coverage based and Hybrid selection when
  a) All tests are executed?
  b) The available operational capacity is executed?
- **RQ8:** What is the relation between the coverage estimated for the Test plan (using previous data) and the real monitored coverage?
- **RQ9:** Regarding each CR, what is the coverage reached by IR based, code coverage based, hybrid and architects selections when
  a) All tests are executed?
  b) The available operational capacity is executed?

The experiment results will be detailed below. We adopted the following terminology:

- **#TCs** - number of test cases selected to compound the test plan;
- **Coverage** - Percentage of code coverage obtained by the execution of the test plan;
- **Failures** - number of failures found;

5.1.1 HSP without Code Coverage (Setup run)

This section presents the experiments results regarding TC selection based solely on IR. As mentioned in Section 3.1, the setup run is simply a hybrid run state where the trace database is empty. Thus the selection and prioritisation is governed by information retrieval.

Although this experiment concerns IR based selection, note that all experiments results are given in terms of coverage metrics. It is possible to obtain these values because we know the set of methods that should be covered by the selected test cases. This way, we can calculate the code coverage obtained by execution runs using the different strategies considered up to this point (i.e., IR based, random and architect's selection).

5.1.1.1 (RQ1) Comparison of selection strategies with respect to coverage

In the Setup run, 184 test cases were selected and prioritised using the Information retrieval strategy (Section 3.2.1) from a base with 5000 TC. The manual execution of this Test plan was monitored, having obtained a code coverage of 49.38%, as seen in Table 1. In turn, the architect selected 88 test cases, which 85% also appeared in the automatically created Test Plan. The architect's selection obtained a coverage of 44.44%. Considering just code coverage it was a hard effort increasing 96 TCs to increase 5% of code coverage. Nevertheless, the number of failures founds should be taken into account.

It is worth to mention that the architect is allowed to selected a higher number of TCs, since there is no fixed upper bound for Test Plans size. However, the architect tends to select a number of TCs which is feasible to be manually executed within the available time.

We also considered 100 random re-orderings of the 184 test cases, where we assume that the tests are independent (i.e., the execution order does not influence the coverage obtained) and we used the same IR based coverage data due to the time to execute all those. Thus, it exhibited the same coverage result of the original 184 selection.

As long as the test architect was not able to select a larger number of TCs due to available human workforce to execute the selected tests, we considered the plan shrunk to the same amount of the TCs selected by the architect (88 test cases) in a way to better compare the different selection strategies. Table 1 shows that the IR shrunk, which is the top 88 TCs in the IR based list (with 184 TCs), obtained a coverage of 35.80%. Finally, the Randoms shrunk, which is the top 88 TCs in the random selection (with 184 TCs) for each random, varies from 35.80% to 49.38% coverage. These runs originated the box plot in Figure 21.

The IR selection coverage decreased in the shrunk case, this can be an indication that the prioritisation is not good when related to the code coverage because the first tests cases did not have the biggest coverage. However, it does not mean that the IR prioritisation is bad because it is prioritised through keywords that are not directly related to the code coverage.

5.1.1.2 (RQ2) Comparison of selection strategies with respect to failures found

Concerning the number of failures found, the initial list of 184 TCs selected and ranked by the IR module (and the 100 random rearrangements) obtained the same result: 20

Strategy	#TCs	Coverage
IR/Random	184	49.38%
Architect	88	44.44%
IR Shrunk	88	35.80%
Random Shrunk	88	35.80% to 49.38%

Table 1 – Setup run - Coverage

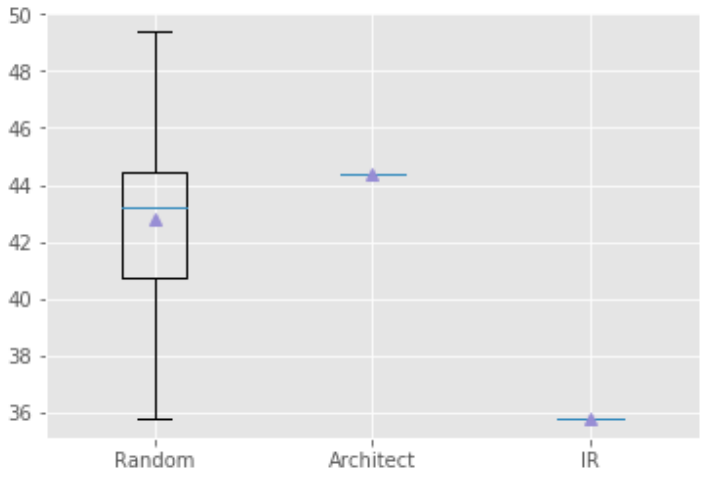


Figure 21 – Setup run - Coverage - Shrunk

failures identified. On the other hand, the architect's selection found only 6 failures where are included in the 20 failures found by IR based selection (see Table 2).

Considering the limit of 88 TCs per Test plan, the IR based selection found 12 failures whereas the architect's selection found only 6 failures where 5 is similar to the IR based one. The failures identified by the random selections varied from 5 to 14. See Figure 22 for the the box plot related to these results.

Although the slight increase of only 5% of code coverage in the normal one (see Section 5.1.1.1), the number of failures found increased significantly by 70%. This indicates that the additional TCs from IR based selection is important for the test campaign. Besides, in the IR selection shrunk the number of failures found increased 50% even with a lower coverage than the architect selection. This can indicate that failures were found in areas considered stable, that means, the SW has not been entirely tested and failures found corresponds to older software version.

5.1.1.3 (RQ3) Analysis of correlation between prioritisation of test cases and coverage

The correlation was calculated using the Spearman method. the objective is to correlate the position in the test cases ranking with their code coverage value. According to the Spearman method, there is correlation when the values are close to 1 or -1, on the other hand, when the value is close to 0 there are no correlation.

Strategy	#TCs	Failures found
IR/Random	184	20
Architect	88	6
IR Shrunk	88	12
Random Shrunk	88	5 to 14

Table 2 – Setup run - Failures found

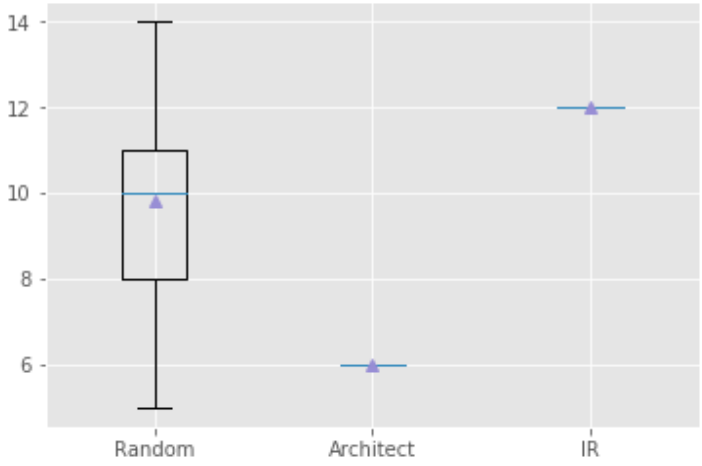


Figure 22 – Setup run - Failures Found - Shrunk

We did not identify a correlation between coverage and the ranking provided by the different selection strategies (IR based, random and architect's selection) (see Figure 23) even when we consider the selection of 88 test cases (the operationally viable execution) (see Figure 24). Thus, in principle, we may be executing non-related test cases. However, as the IR based selection matches keywords its ranking is not directly related to code coverage.

This is the reason that the code coverage showed in section 5.1.1.1 was worst in the shrunk plan. As there is no correlation between the code coverage and the test case ranking in the IR based selection the prioritisation is not ordered by code coverage, thus when the test plan is shrunk important TCs which cover the modified source code is not chosen.

5.1.1.4 (RQ4) CR Coverage Analysis

Figure 25 presents CR coverage for full test plans created by IR selection strategy and by the architect. This figure reveals that the IR selection slightly improved coverage by upgrading one CR from 0% of coverage to the class of $< 50\%$ of coverage.

On the other hand, when we consider only the 88 top test cases from the IR selection, Figure 26 shows that this strategy achieved the same coverage in terms of CRs than the architect's selection.

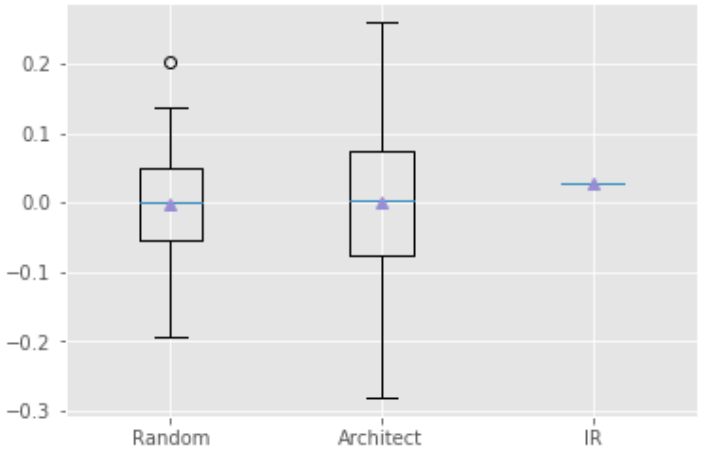


Figure 23 – Setup run - Correlation - Normal

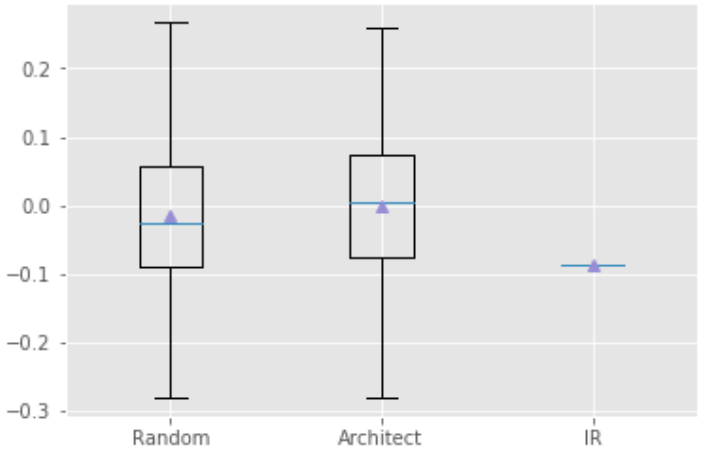


Figure 24 – Setup run - Correlation - Shrunk

It is important to note that 3 CRs was not covered and almost 50% of the CRs was not entirely covered. This indicates that part of the source code was not tested and these area are considered stable. Besides, failures may be escaping because of the lack of TCs for testing these areas.

5.1.2 Full HSP (IR and CC merge and prioritisation)

The Hybrid run executes selection and prioritisation based on IR and code coverage information, as already detailed. The code coverage of each TC executed in the Setup run (in terms of which methods each TC exercises) is used in this run to allow a selection based on code coverage.

This investigation aimed to determine which of the three selection strategies (information retrieval solely, code coverage solely or some combination of both) is the most appropriate to our context. As the coverage information used in this run was obtained from the Setup run, a question of interest is whether the coverage estimate for the Test

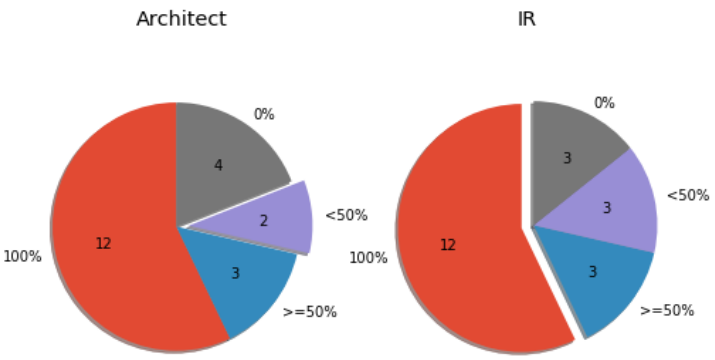


Figure 25 – Setup run - CR Coverage - Normal

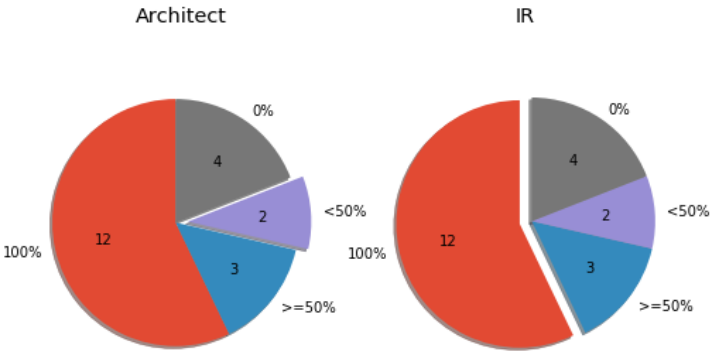


Figure 26 – Setup run - CR Coverage - Shrunk

plan (using previous data) is close to the real monitored coverage (**RQ8**).

While in the Setup run only 81 methods were modified (*ToBeCovered*) between the versions under test, in the Hybrid run 1,119 modified methods were identified. This have occurred because the setup run was run using 4 RNs while the hybrid was run using 10 RNs, thus the hybrid run have considered more SW modification.

5.1.2.1 (RQ5) Comparison of selection strategies with respect to coverage

Table 3 presents the number of TCs selected by the IR based strategy (302 tests), CCS_t strategy using just the coverage information from Setup Run (184 tests), $MPCCR_t$ and $MPIRR_t$ as a combination of the previous two selections (296 tests), $MPCCR_g$ and $MPIRR_g$ as a combination of the CCS_g and IR based selections (296 tests), the CCS_g strategy solely (24 Tests), the architect's selection was the same 88 TCs of Setup Run, and Random selection (the same 296 TCs from the Hybrid selection using different ranking orderings). Those selections selected 95 TCs more than the architect selection at minimum, except the CCS_g, for improving in 6% of code coverage. It is a hard execution effort, however it can be important to find failures.

Strategy	#TCs	Coverage	Failures found
IR	302	47.08%	13
CCS_t	183	47.08%	11
$MPCCR_t$	296	47.08%	13
$MPIRR_t$	296	47.08%	13
$MPCCR_g$	296	47.08%	13
$MPIRR_g$	296	47.08%	13
Random	296	47.08%	13
Architect	88	41.69%	2
CCS_g	24	36.48%	0

Table 3 – Hybrid run

Table 3 presents the code coverage values obtained by executing each of these Test plans. The architect's selection reached 41.69%, while the CCS_g obtained 36.48% coverage rate. All other selection strategies obtained a higher rate of 47.08% code coverage.

When considering only the top 88 TCs in each Test plan, Figure 27 shows that $MPIRR_t$ exhibited the best coverage rate, followed by the median of the random selection, next are CCS_t and $MPCCR_t$ that are tied, after that are the architect's selection, $MPCCR_g$, $MPIRR_g$ and finally the IR selections. Thus, it is the first evidence that the $MPIRR_t$ has a good prioritisation, although it needs to be confirmed by the correlation and failures found.

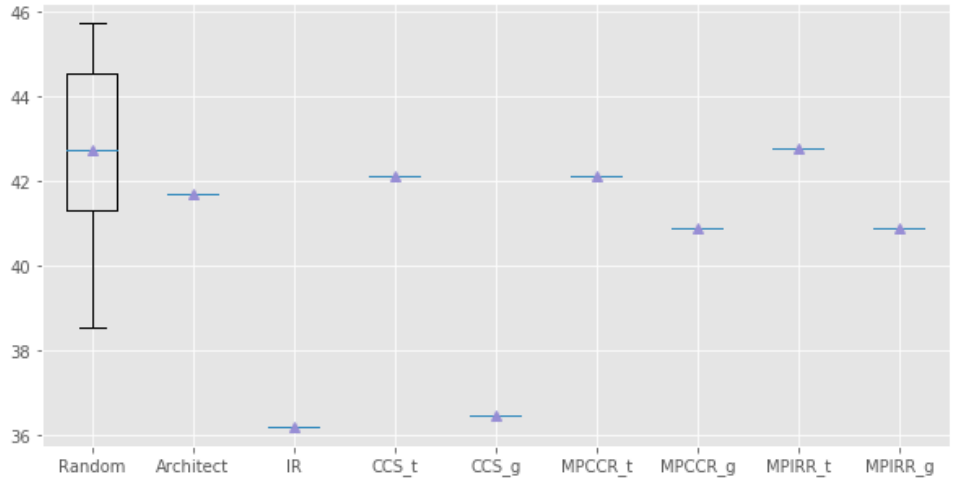


Figure 27 – Hybrid run - Coverage - Shrunk

5.1.2.2 (RQ6) Comparison of selection strategies with respect to failures found

Table 3 shows that the architect's selection identified 2 failures, CCS_g did not find failure and the CCS_t found 11 failures while the other Test plans found 13 failures when all TCs selected were executed.

When considering only the top 88 test cases of each Test plan, Figure 28 shows that the $MPIRR_t$ found 9 failures, the IR found 4 failures, the architect's selection found 2 failures and the CCS_g did not find failures while the others found 3 failures. The random selection varied from 1 to 11, but it found 6 failures on average. Therefore, the $MPIRR_t$ has the highest number of failures found, on average. Confirming the evidence which was presented before.

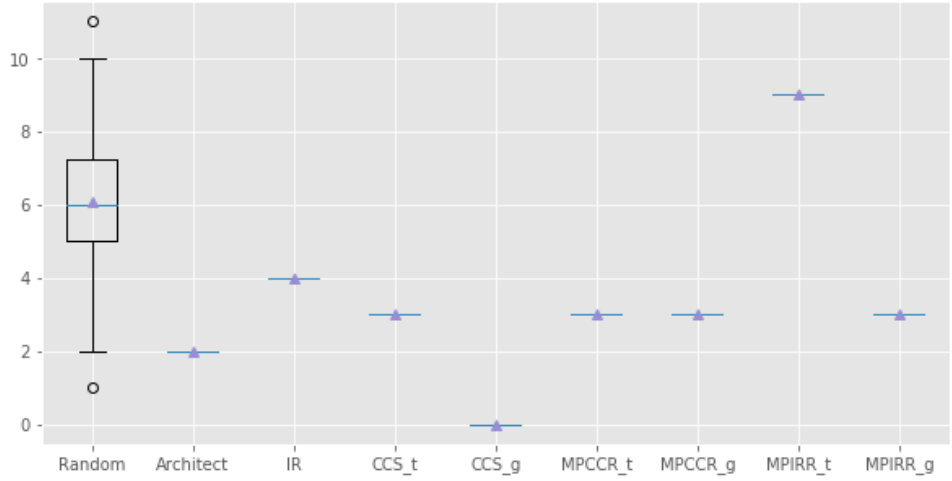


Figure 28 – Hybrid run - Failures Found - Shrunk

5.1.2.3 (RQ7) Correlation analysis between prioritisation of test cases and coverage

Analyzing again the relation between Test plans rankings and coverage, it is possible to observe in Figure 29 that the correlation values are almost equivalent for architect's plan, random and IR selections, $MPCCR_g$, $MPIRR_g$ and CCS_g (which is slightly better) and all those are close to 0 that means there is no correlation. For the CCS_t, $MPCCR_t$ and $MPIRR_t$ orderings, a negative correlation was obtained, which means that the test cases in the lowest ranking values (on the top of the list) exhibited the highest coverage rates. Thus, they are indeed correctly prioritised.

When only the top 88 test cases are considered, the correlation values of the CCS_t and $MPCCR_t$ are the best ones, followed by the $MPCCR_g$ and $MPIRR_g$ (see Figure 30). The $MPIRR_t$, CCS_g, random, architect and IR selection have no correlation. The $MPIRR_t$ might have turned point because the test case order is mixed in the prioritisation since the test cases are not ordered only by code coverage.

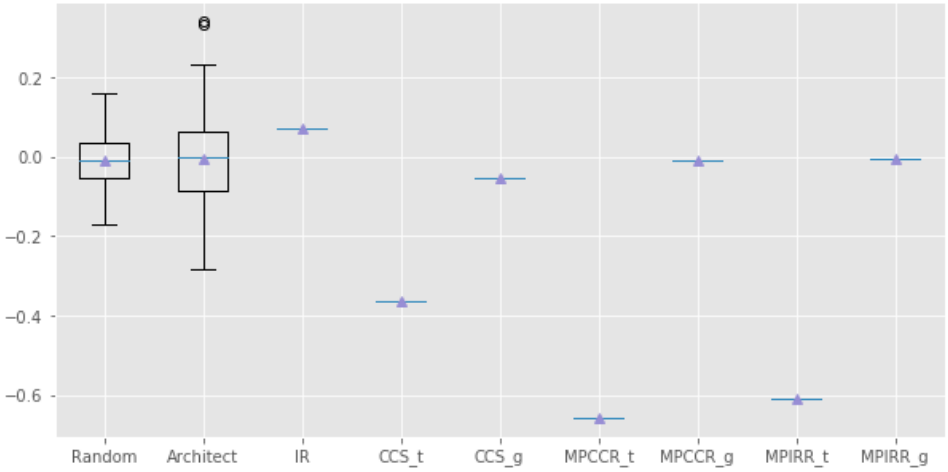


Figure 29 – Hybrid run - Correlation - Normal

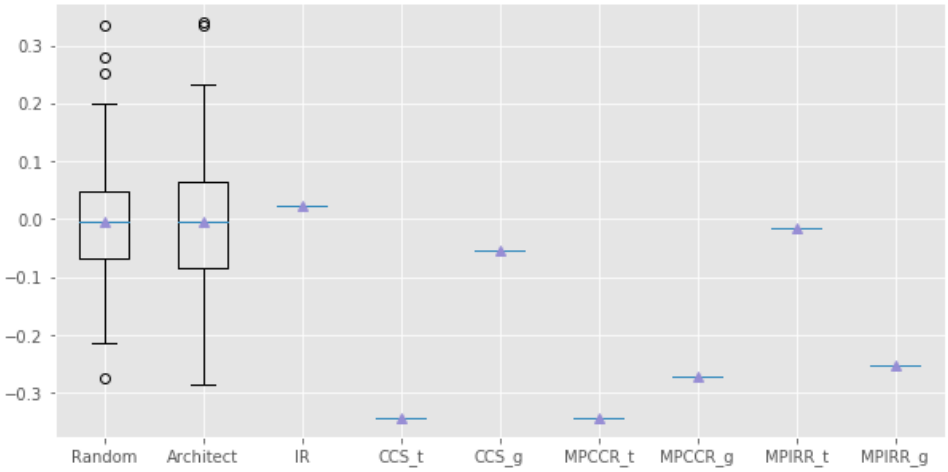


Figure 30 – Hybrid run - Correlation - Shrunk

5.1.2.4 (RQ8) Correlation analysis between the estimate coverage and the real coverage

The estimated coverage using data from the previous regression campaign (Setup Run) for the subsequent test campaign was 36.84%. The execution of these test cases was monitored. This Test plan reached 47.08% code coverage, and 13 failures were found.

Analysing the obtained methods, we can conclude that the estimated coverage was conservative at 92.45% of the methods that were expected to be covered. This means that almost 100% of the methods that were predicted to be covered were indeed covered. Another important result is that the difference from coverage expected and actual coverage was small. The predicted methods correspond to 72.33% of the total methods that were covered.

Thus, it indicates that the use of code coverage data to select test cases can be used to predict the covered methods.

5.1.2.5 (RQ9) CRs Coverage Analysis

Figure 31 shows that the IR, CC_t and Hybrid (all those hybrid) selections improved CR code coverage when compared to the architect's selection, by promoting 3 CRs with 0% to others classes. However, the CC_g selection performed worst than the architect's one.

When considering the 88 initial test cases, Figure 32 shows that the IR, $MPCCR_g$ and $MPIRR_g$ selection decreased the CR code coverage number of a 100% group while the CCS_t and $MPCCR_t$ have been more conservative despite their loss. Summarising, the $MPIRR_t$ was the best in CR coverage criteria because it was still performing better than the architect and the other selections.

It is important to note that 10 CRs was not covered and almost 75% of the CRs was not entirely covered. This indicates again that part of the source code was not tested and these area are considered stable. Besides, failures may be escaping because of the lack of TCs for testing these areas.

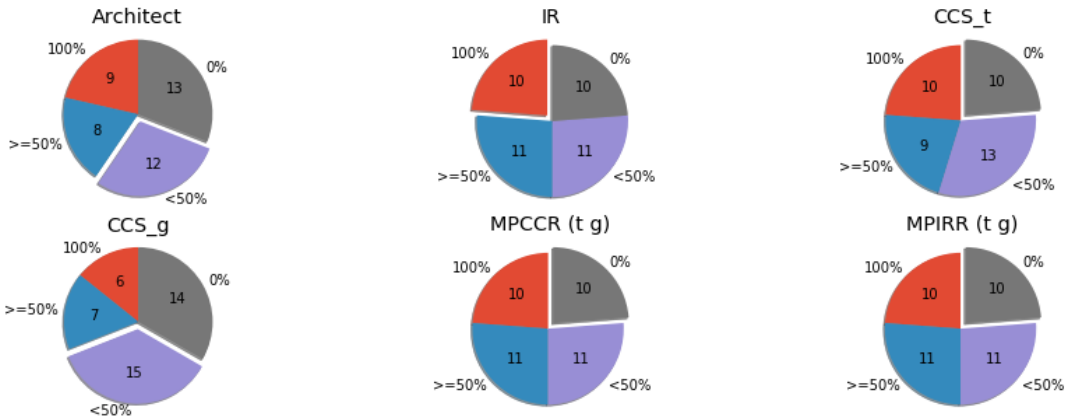


Figure 31 – Hybrid run - CR Coverage - Normal

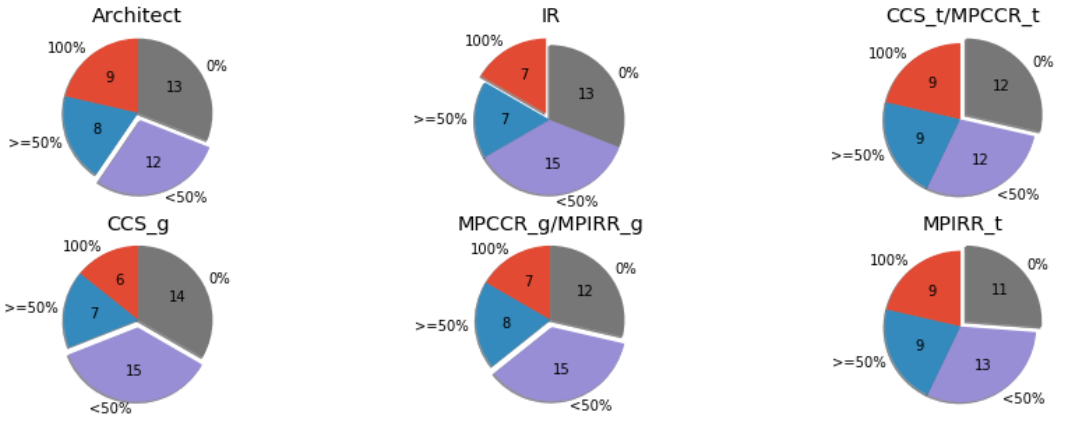


Figure 32 – Hybrid run - CR Coverage - Shrunk

6 CONCLUSION

This work presented an approach to the selection and prioritisation of test cases based on Information Retrieval and code coverage techniques. We conducted an empirical evaluation on a real case study provided by our industrial partner (Motorola Mobility) aiming to evaluate which combination of information retrieval and code coverage data better fits this industrial context.

From Section 5.1, we can indicate that the HSP process performs better than the architect's selection. This is an important finding, since Industry immediately detects the cost reduction obtained with automation, but does not envision the quality improvement in the overall testing process.

As seen in the results of the experiments, the IR selection performs nicely when all TCs in the Test plan can be executed. When only the top-list TCs can be executed it still shows good performance concerning failures found, however, it reaches less code coverage than the architect's selection. This was our main reason to consider code coverage, aiming to improve our results. As seen, the hybrid process (namely, MPIRR_t) obtained very promising results, showing that the combination of information retrieval and code coverage represents the best balance between code coverage and bugs found among all combinations evaluated.

Apart from the previous important conclusion, we also observed that the coverage measures of all proposed combinations are around 50% of the areas that must be exercised. This is softened by our industrial partner by using exploratory testing in an attempt to improve such a coverage. Although this is out of scope of the present work, we also measured the coverage of exploratory testing campaigns, and the improvement was just marginal. Thus, it is worth investing in selection processes and test case creation

6.1 RELATED WORK

Selection and prioritisation of test cases are research areas of great interest, specially concerning Regression Testing [Rothermel and Harrold, 1996]. We can identify in the related literature several works focusing on these areas. However, those works treat these areas separately, whereas our work proposes a hybrid solution which integrates both areas. This research topic associated with Information Retrieval can be found in the works Kwon et al. [2014], Ledru et al. [2012], Nguyen et al. [2011], Saha et al. [2015] and code coverage in Di Nardo et al. [2015], Gligoric et al. [2015], Öqvist et al. [2016].

The work of Kwon et al. [2014] proposed a prioritisation method based on code information. The method combines traditional code coverage scores with the scores reported by IR using a linear regression model to fit the best weights for the two scores. Similar to

the work Kwon et al. [2014], our work also uses the Change Request (CR) description to retrieve test cases and we also use code coverage. However, the test cases in our work are only textual documents while they index the test case trace. Another difference is that we propose TCs selection followed by prioritisation. The work Kwon et al. [2014] uses the rank provided by Lucene's model as the mainly prioritisation criterion, while we use the same rank to prioritise the test cases per CR on IR selections, and two approaches to prioritise by code coverage (Total and Additional Coverage). Yet, the work Kwon et al. [2014] does not consider an industrial context.

Similarly to our approach, the work Nguyen et al. [2011] proposed a test case prioritisation method to Web services compositions based on CR descriptions. However, Information Retrieval is used in Nguyen et al. [2011] to match the terms found in the CRs with test cases' traces, while we use textual test cases. In Nguyen et al. [2011] the TF-IDF measure is used to prioritise the test campaign, whereas in our work TF-IDF is used as one of the prioritisation strategies, however preceded by test cases selection. Similar to Nguyen et al. [2011], we also use code coverage, however we combine code coverage with information retrieval in the Hybrid approach.

Concerning the use of textual TCs without code coverage, we highlight the work Ledru et al. [2012]. Several algorithms (e.g. Cartesian, Levenshtein, Hamming and Manhattan) were used to calculate the string distances between each test case on a testing campaign, in order to prioritise them. The closest test cases must be executed first, and the most distant TCs should be executed afterwards. According to the work Ledru et al. [2012], both documents CR and TC are textual artifacts, however they prioritise test cases through similarity, that means, similar test cases stay on the top while we prioritise the test cases based on IR and Code Coverage. Different from the above cited work, our work proposes an approach to select and prioritise test cases.

An approach for Regression test prioritisation, named REPiR, was proposed in the work Saha et al. [2015]. Given two versions of a software, REPiR uses the source code difference as a query. The test case code information, as well as classes or method names are used to match the query and rank the test case. In our work, we match textual CRs with textual test cases, and CRs code difference with test case traces. The work by Saha et al. [2015] uses a modified version of BM25 model that fits better their environment, whereas we use the Solr score (detailed in Section 2.4) combined with test case frequency and code coverage to fit our own rank. Yet, we propose an approach to select and prioritise test cases which is different from the work reported in Saha et al. [2015].

The work reported in Di Nardo et al. [2015] compared coverage-based Regression testing techniques. Both fine-grained and coarse-grained coverage criteria were considered, as well as different techniques for selection, prioritisation and minimization of automated test cases. Coverage and fault detection measures were used to evaluate the test campaign. Differently, our work only counts on manual test cases. Yet, we could not instrumentate

the code, thus being restricted to compute coverage in terms of class and methods calls (see Section 3.2.2.3).

A Regression testing selection tool, named Ekstazi, was proposed in the work Gligoric et al. [2015] to adopt the Regression Test Selection (RTS) in industry. Ekstazi uses files dependencies to select the test cases that affect the modified files, while we select test cases based on their traces combined with information retrieval. The Ekstazi tool extracts a set of files that were accessed on the test execution by dynamic analysis from the instrumented code, obtaining coverage of class and methods calls. In a similar way, we obtained dynamic traces and the same grained measures. Automated test cases were used for execution, whereas we work with manual test cases. However, similarly to their work, our work use both coverage and fault detection measures to evaluate a testing campaign.

A Regression testing selection technique based on static analysis was presented in the work Öqvist et al. [2016]. This technique, named *Extraction-Based RTS*, extracts file dependencies from the test using the control flow graph, and creates a dependency graph that relates files and test cases. Then, it is possible to select tests that exercise modified files. On our work, we created the Hybrid selection process that uses the test cases traceability to select test cases that exercise modified areas, and also complement the Test plan with test cases selected through IR. The work Öqvist et al. [2016] used automated test cases, while we used manual test cases. The work reported in Öqvist et al. [2016] does not evaluate coverage, only performance.

6.2 THREATS TO VALIDITY

According to Wohlin et al. [2012], it is almost impossible to avoid all threats. Thus, we have identified some threats to validity and tried to mitigate possible problems that they may cause in the experiments.

One problem is that the test should have executed automatically to guarantee the independence of the test execution. However, the set test used could not be automated. Therefore, the test must be executed in an identical way to get the same code coverage.

Concerning the code coverage, as smartphones are multiple threads the methods sequence may vary in the execution log. Nevertheless, it does not matter, because the algorithms use the set of methods. Thus, the order is not relevant.

Another limitation is that tests for measuring performance cannot be monitored, but for the company performance tests are done in another test phase different from regression.

6.3 FUTURE WORK

As first future work, we intend to monitor the use of the proposed combinations in our industrial partner to identify which combination is indeed the best one in the long run. That is, we intend to conduct continuous experiments during the software cycle life to

check if the results obtained in these experiments will be kept. As well as, we intend to do controlled experiments in order to prove the HSP efficacy.

Another future work concerns applying test suite reduction regarding similarity, using both natural language as well as source code, evaluating them through Fault Detection Loss (FDL) [Singh and Shree, 2018]. The motivation comes from the fact that, in general, the testing teams are not able to execute all TCs selected by the combinations presented here. Furthermore, as the selections are greater than those of test architect, the similarity can be an efficient way to reduce the number of test cases in a test campaign taking into account the code coverage and the test meaning (text).

Yet, we also consider investigating prioritisation strategies reported in Magalhães et al. [2016b], regarding coverage and bugs found. We have presented in this work ways to prioritise test cases through information retrieval or code coverage. However, using Z3, we can define a mathematical model to prioritise the test plan using the historical test data as input and evaluate its performance using Average Percentage of Faults Detected (APFD) [Elbaum et al., 2000].

Upon the coverage rate obtained in this work, we conclude that is also relevant to propose strategies/approaches to create and suggest new test cases or scenarios to increase the coverage and possibly failures found in test campaigns.

About creating new test cases, one future work is to identify areas that need further testing [Rolfsnes et al., 2017] and to create new tests through test step switching. This will not create completely new tests, nevertheless, it could lead to finding new failures through exercising new paths. Another possibility is to be able to convert a CR to a requirement or a use case using standardized CR writing through a controlled language that can be reused by TaRGeT [Ferreira et al., 2010] to generate new test cases. Since that it is possible to create test cases derived from the description of use cases or requirements [Sarmiento et al., 2014, Verma and Beg, 2013].

Last but not least, it would be possible to identify the area of code that was not exercised during the test campaign. Using this information together with Reis and Mota [2018] it would be possible to have the navigation that would reach the non-exercised area. In order to complement the test and ensure consistency the work Sampaio and Arruda [2016] helps to solve the dependencies of the test and ensure that it could run. The advantage of this approach is that it can serve both for automated testing cases and manual testing cases.

REFERENCES

Surafel Lemma Abebe, Nasir Ali, and Ahmed E. Hassan. An empirical study of software release notes. *Empirical Software Engineering*, 21(3):1107–1142, 2016. ISSN 1573-7616. doi: 10.1007/s10664-015-9377-5.

Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 020139829X.

Jørn Ola Birkeland. From a timebox tangle to a more flexible flow. In *Lecture Notes in Business Information Processing*, pages 325–334. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-13054-0_35. URL https://doi.org/10.1007/978-3-642-13054-0_35.

Bugzilla. http://www.bugzilla.org/, 2019. (accessed Jan 11, 2019).

João Luiz de Andrade Neto. Autotestcoveragec: Uma ferramenta para obtenção de cobertura de código para componentes android sem uso de instrumentação, dec 2018. http://www.cin.ufpe.br/~tg/2018-2/TG_CC/tg_jlan.pdf.

Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability*, 25(4):371–396, 2015. ISSN 1099-1689. doi: 10.1002/stvr.1572. URL http://dx.doi.org/10.1002/stvr.1572. stvr.1572.

Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on Structured Programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972. ISBN 0-12-200550-3. URL http://dl.acm.org/citation.cfm?id=1243380.1243381.

Docker. Docker engine. https://www.docker.com/, 2019. (accessed Jan 20, 2019).

Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, 25(5):102–112, August 2000. ISSN 0163-5948. doi: 10.1145/347636.348910. URL http://doi.acm.org/10.1145/347636.348910.

Felype Ferreira, Laıs Neves, Michelle Silva, and Paulo Borba. Target: a model based product line testing tool. *Tools Session of CBSoft*, 2010.

Apache Software Foundation. TFIDFSimilarity. http://lucene.apache.org/core/7_3_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html, 2018. [Online; accessed 06-Dec-2018].

Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771784. URL http://doi.acm.org/10.1145/2771783.2771784.

Ferenc Horváth, Béla Vancsics, László Vidács, Árpád Beszédes, Dávid Tengeri, Tamás Gergely, and Tibor Gyimóthy. Test suite evaluation using code coverage based metrics. In *14th Symposium on Programming Languages and Software Tools, SPLST 2015*. CEUR-WS, 2015.

Mark Douglas Jacyntho, Daniel Schwabe, and Gustavo Rossi. A software architecture for structuring complex web applications. *J. Web Eng.*, 1(1):37–60, October 2002. ISSN 1540-9589. URL http://dl.acm.org/citation.cfm?id=2011098.2011104.

Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. The jvm is not observable enough (and what to do about it). In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 33–38, New York, NY, USA, 2012a. ACM. ISBN 978-1-4503-1633-0. doi: 10.1145/2414740.2414747. URL http://doi.acm.org/10.1145/2414740.2414747.

Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. The jvm is not observable enough (and what to do about it). In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 33–38, New York, NY, USA, 2012b. ACM. ISBN 978-1-4503-1633-0. doi: 10.1145/2414740.2414747. URL http://doi.acm.org/10.1145/2414740.2414747.

J. H. Kwon, I. Y. Ko, G. Rothermel, and M. Staats. Test case prioritization based on information retrieval concepts. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 19–26, Dec 2014. doi: 10.1109/APSEC.2014.12.

Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, Mar 2012. ISSN 1573-7535. doi: 10.1007/s10515-011-0093-0. URL https://doi.org/10.1007/s10515-011-0093-0.

Cláudio Magalhães, Alexandre Mota, Flávia Barros, and Eliot Maia. Automatic selection of test cases for regression testing. In *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing (SAST), Maringá, Brasil*, pages 1–8. ACM, 2016a. doi: 10.1145/2993288.2993299.

Cláudio Magalhães, Alexandre Mota, and Eliot Maia. Automatically finding hidden industrial criteria used in test selection. In *28th International Conference on Software*

Engineering and Knowledge Engineering, SEKE'16, San Francisco, USA, pages 1–4, 2016b. doi: 10.18293/SEKE2016-198.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

Mantis. Mantis bug tracker. http://www.mantisbt.org/, 2019. (accessed Jan 11, 2019).

Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 484–495, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635870.

Glenford J. Myers. *The Art of Software Testing, Second Edition*. Wiley, 2004. ISBN 9780471469124. URL https://www.amazon.com/Art-Software-Testing-Second/dp/0471469122?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0471469122.

Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083143. URL http://doi.acm.org/10.1145/1082983.1083143.

C. D. Nguyen, A. Marchetto, and P. Tonella. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *2011 IEEE International Conference on Web Services*, pages 636–643, July 2011. doi: 10.1109/ICWS.2011.75.

Srinivas Nidhra. Black box and white box testing techniques - a literature review. *International Journal of Embedded Systems and Applications*, 2(2):29–50, jun 2012. doi: 10.5121/ijesa.2012.2204. URL https://doi.org/10.5121/ijesa.2012.2204.

Jesper Öqvist, Görel Hedin, and Boris Magnusson. Extraction-based regression test selection. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, pages 5:1–5:10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4135-6. doi: 10.1145/2972206.2972224. URL http://doi.acm.org/10.1145/2972206.2972224.

Lucas Bezerra Perrusi. Autotestcoveragep: Uma ferramenta para cobertura de testes de integração no contexto android sem uso de código-fonte, dec 2018. http://www.cin.ufpe.br/~tg/2018-2/TG_CC/tg_lbp.pdf.

Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2009. ISBN 0073375977. URL https://www.amazon.com/Software-Engineering-Practitioners-Roger-Pressman/dp/0073375977?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0073375977.

Redmine. Flexible project management web application. http://www.redmine.org/, 2019. (accessed Jan 11, 2019).

Jacinto Reis and Alexandre Mota. Aiding exploratory testing with pruned gui models. *Information Processing Letters*, 133:49 – 55, 2018. ISSN 0020-0190. doi: https://doi.org/10.1016/j.ipl.2018.01.008. URL http://www.sciencedirect.com/science/article/pii/S0020019018300176.

Thomas Rolfsnes, Leon Moonen, and David Binkley. Predicting relevance of change recommendations. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 694–705, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-2684-9. URL http://dl.acm.org/citation.cfm?id=3155562.3155649.

Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, August 1996. ISSN 0098-5589. doi: 10.1109/32.536955. URL http://dx.doi.org/10.1109/32.536955.

Gregg Rothermel, Harrold Mary Jean, von Ronne Jeffery, and Hong Christie. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002. doi: 10.1002/stvr.256. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.256.

R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 268–279, May 2015. doi: 10.1109/ICSE.2015.47.

Augusto Sampaio and Filipe Arruda. Formal testing from natural language in an industrial context. In Leila Ribeiro and Thierry Lecomte, editors, *Formal Methods: Foundations and Applications*, pages 21–38, Cham, 2016. Springer International Publishing. ISBN 978-3-319-49815-7.

E. Sarmiento, J. C. S. d. P. Leite, and E. Almentero. C amp;l: Generating model based test cases from natural language requirements descriptions. In *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, pages 32–38, Aug 2014. doi: 10.1109/RET.2014.6908677.

Amanda Schwartz and Michael Hetzel. The impact of fault type on the relationship between code coverage and fault detection. In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, pages 29–35, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4151-6. doi: 10.1145/2896921.2896926. URL http://doi.acm.org/10.1145/2896921.2896926.

Shilpi Singh and Raj Shree. A new similarity-based greedy approach for generating effective test suite. *International Journal of Intelligent Engineering and Systems*, 11(6): 1–10, dec 2018. doi: 10.22266/ijies2018.1231.01. URL https://doi.org/10.22266/ijies2018.1231.01.

IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, 2014. ISBN 9780769551661. URL https://www.amazon.com/Guide-Software-Engineering-Knowledge-SWEBOK/dp/0769551661?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0769551661.

Ian Sommerville. *Software Engineering (9th Edition)*. Pearson, 2010. ISBN 0137035152. URL https://www.amazon.com/Software-Engineering-9th-Ian-Sommerville/dp/0137035152?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0137035152.

Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook Computing. Rocky Nook, fourth edition edition, 2014. ISBN 1937538427,9781937538422.

R. P. Verma and M. R. Beg. Generation of test cases from software requirements using natural language processing. In *2013 6th International Conference on Emerging Trends in Engineering and Technology*, pages 140–147, Dec 2013. doi: 10.1109/ICETET.2013.45.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-29044-2. URL https://doi.org/10.1007/978-3-642-29044-2.

Qian Yang, Jingjing Li, and David M. Weiss. A survey of coverage based testing tools. *Comput. J.*, 52:589–597, 2006.

S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012. ISSN 0960-0833. doi: 10.1002/stv.430. URL http://dx.doi.org/10.1002/stv.430.