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

Claudio Magalhães, João Andrade, Lucas Perrusi, Alexandre Mota, Flávia Barros, Eliot Maia

PII: S0164-1212(19)30204-3
DOI: https://doi.org/10.1016/j.jss.2019.110430
Reference: JSS 110430

To appear in: *The Journal of Systems & Software*

Received date: 14 December 2018
Revised date: 24 August 2019
Accepted date: 24 September 2019

Highlights

- A hybrid strategy to select and prioritise test cases based on information retrieval and code coverage;
- A code coverage analysis without instrumentation;
- An empirical comparison between selection based on information retrieval, code coverage and both.

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

Claudio Magalhães[1], João Andrade[1], Lucas Perrusi[1], Alexandre Mota[1], Flávia Barros[1], and Eliot Maia[2]

[1] Centro de Informática (CIn), Universidade Federal de Pernambuco (UFPE), Brazil
{cjasm, jlan, lbp, fab, acm}@cin.ufpe.br
[2] Motorola Mobility LLC
eliot@motorola.com

Abstract. The usual way to guarantee quality of software products is via testing. This paper 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). Experiments results revealed that the combined strategy provides better results than the use of information retrieval and code coverage independently. Yet, it is worth mentioning that any of these automated options performed better than the previous 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

1 Introduction

The usual way to guarantee quality of software products is via testing. However, this is known to be a very expensive task. Although automated testing is an essential factor of success in projects development, unfortunately it is not always possible to adopt this practice in industrial environments. This way, several companies still 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 software (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 scenery 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 [4].

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 scenery 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).

In this light, we developed a novel strategy for Regression Test plans creation which combines Information Retrieval (IR) techniques [3] and indirectly obtained code coverage information. The proposed Hybrid Selection and Prioritisation (HSP) process receives as input Change Requests (CRs) related to a particular SW application under test, and creates two different test plans 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 *T*est *C*ases (named as *TCs Index base*), which is independent from the SW application under test — i.e., this base indexes all TCs available from the general TCs repository (see Section 3.1). 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 4.1) [14].

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 3.2), with information about the log of previously executed TCs (TC + trace). We create code coverage information from an indirect measurement. The coverage based Test plan contains all TCs in the Trace base whose execution trace match any code fragment present in the current input CRs (Section 4.3).

Note that, different from 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 3.2). This base is updated by the execution monitoring tool whenever a Test plan is executed (Section 4.2) 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 4. The output Test plan will be 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 mentioning that any of these automated options perform better than the previous manual process deployed by our industrial partner to create test plans.

The main contributions of this work are:

– a hybrid strategy to select and prioritise test cases based on information retrieval and code coverage;
– a code coverage analysis without instrumentation;
– an empirical comparison between selection based on information retrieval, code coverage and both.

This paper is organized into another 7 further sections, as follows. Section 2 briefly presents definitions and concepts closely related to the present work; Section 3 brings an overview of the proposed work, together with a description of the test case index base creation and update; Section 4, in turn, details the Hybrid process, which performs several different selection and prioritisation strategies; Section 5 brings a detailed description of the performed experiments and the obtained results; Section 6 aims to show some of the related work found in the available literature; and Section 7 concludes this document with an overall appreciation of the developed work and the achieved results, also pointing at future works to be developed.

2 Background

This section aims to introduce some important concepts and definitions related to Software Engineering and Information Retrieval, which are the central areas of the work presented in this paper.

2.1 Software Maintenance

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

Release Notes Each new software product release usually brings an associated Release Note (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 (either bug fixes or new features incorporated in the current SW release). RNs can be manually or automatically generated [19].

An analysis performed in [19] 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 pointing out that RNs only list the most important issues about a software release, being aimed to provide quick and general information, as reported in [1].

Change Request Change Requests (CR) 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 [17], Bugzilla [5] and Redmine [23] 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. Each closed CR has a field with the committed code where the related source-code modification was submitted. Figure 1 brings a sample template of a CR.[3]

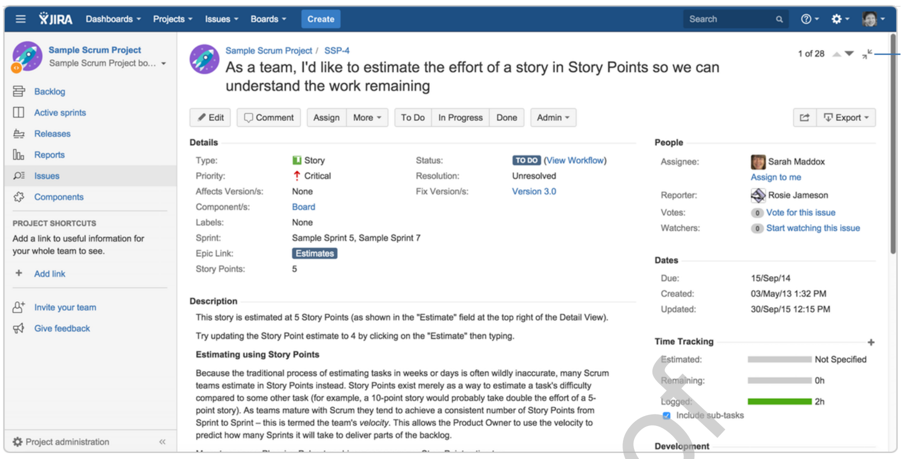
A Release Note is thus composed of several Change Requests as well as new features integrated into the delivered version of the system.

2.2 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 [27]. 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 [27].

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

[3] http://www.softwaretestinggenius.com/software-change-request-form-and-its-sample-template



Sample Scrum Project / SSP-4

As a team, I'd like to estimate the effort of a story in Story Points so we can understand the work remaining

Details

Type:	Story	Status:	TO DO (View Workflow)
Priority:	Critical	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	Version 3.0
Component/s:	Board		
Labels:	None		
Sprint:	Sample Sprint 5, Sample Sprint 7		
Epic Link:	Estimates		
Story Points:	5		

Description

This story is estimated at 5 Story Points (as shown in the "Estimate" field at the top right of the Detail View).

Try updating the Story Point estimate to 4 by clicking on the "Estimate" then typing.

Estimating using Story Points

Because the traditional process of estimating tasks in weeks or days is often wildly inaccurate, many Scrum teams estimate in Story Points instead. Story Points exist merely as a way to estimate a task's difficulty compared to some other task (for example, a 10-point story would probably take double the effort of a 5-point story). As teams mature with Scrum they tend to achieve a consistent number of Story Points from Sprint to Sprint – this is termed the team's *velocity*. This allows the Product Owner to use the velocity to predict how many Sprints it will take to deliver parts of the backlog.

People

Assignee:	Sarah Maddox Assign to me
Reporter:	Rosie Jameson
Votes:	0 Vote for this issue
Watchers:	0 Start watching this issue

Dates

Due:	15/Sep/14
Created:	03/May/13 1:32 PM
Updated:	30/Sep/15 12:15 PM

Time Tracking

Estimated:	Not Specified
Remaining:	0h
Logged:	2h

✓ Include sub-tasks

Development

Fig. 1. Generic CR template

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 [24] shows that this is not a rule (see Section 2.3 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 case 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 [29]. The formal definition follows [24]:

  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 [29]. The formal definition follows [8]:

  Definition 2. *Let T be a test suite, $\{r_1,\ldots,r_n\}$ a set of test requirements, which must be satisfied to provide the desired "adequate" testing of the program, and subsets of T, $T_1,\ldots,T_n$, one 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 [29]. The formal definition follows [24]:

  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 | 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 Code Coverage Analysis

Code coverage is a metric used to characterize the degree to which some region of the source code of an application has been executed after a test campaign [9]. 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 [26] has already pointed out that high code coverage does not necessarily mean high bug detection. However, it is not possible to state anything about the uncovered area [27]. 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., exercised. 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[4]. 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 [10]. For Android, we use ddmlib[5] tracers to listen methods calls while performing a test case execution.

2.4 Information Retrieval

Information retrieval (IR) applications are being increasingly used in software engineering systems [16]. IR focuses on indexing and search: given a massive

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

collection of information items (e.g., documents, images, videos), IR attempts to select the most relevant items based on a user query [3].

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 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.

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) [3], 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 for details).

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 be 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 [3].

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.

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)] [3], 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, 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 [3].

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 this model, the degree of similarity of the document d_j in relation to the query q is given by the correlation between the vectors $\boldsymbol{d_j}$ and $\boldsymbol{q}$. The $\boldsymbol{d_j}$ and $\boldsymbol{q}$ 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[3] $sim(d_j, q) = \frac{\boldsymbol{d_j} \cdot \boldsymbol{q}}{|\boldsymbol{d_j}| \times |\boldsymbol{q}|} = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{t}^{i=1} w_{i,q}^2}}$

Note that other similarity functions may be used in this context. However, cosine is the most traditional one.

3 HSP: Overview

This section brings an overview of the HSP process for automatic TC selection and prioritisation for Regression Test Plans and Table 1 shows the meaning of the words abbreviation. As it can be seen in Figure 2, 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.

Abbreviation	Name
CR	Change Request
TC	Test Case
TP	Test Plan
SUT	System Under Test
ATP	AutoTestPlan tool
IR	Information Retrieval
CCS	Code Coverage Selection
CCS_g	CCS using additional greedy algorithm
CCS_t	CCS using total coverage algorithm
MPCCR	Code Coverage Merge Prioritisation
MPCCR_g	MPCCR using additional greedy algorithm
MPCCR_t	MPCCR using total coverage algorithm
MPIRR	Information Retrieval merge Prioritisation
MPIRR_g	MPIRR using additional greedy algorithm
MPIRR_t	MPIRR using total coverage algorithm
HSP	Hybrid Selection and Prioritisation

Table 1. Abbreviations Meaning

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 SW. 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). 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 4.1). Following, redundancies are eliminated. This process was implemented as an information retrieval system, the AutoTestPlan - ATP (Figure 4). 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 4.2). 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

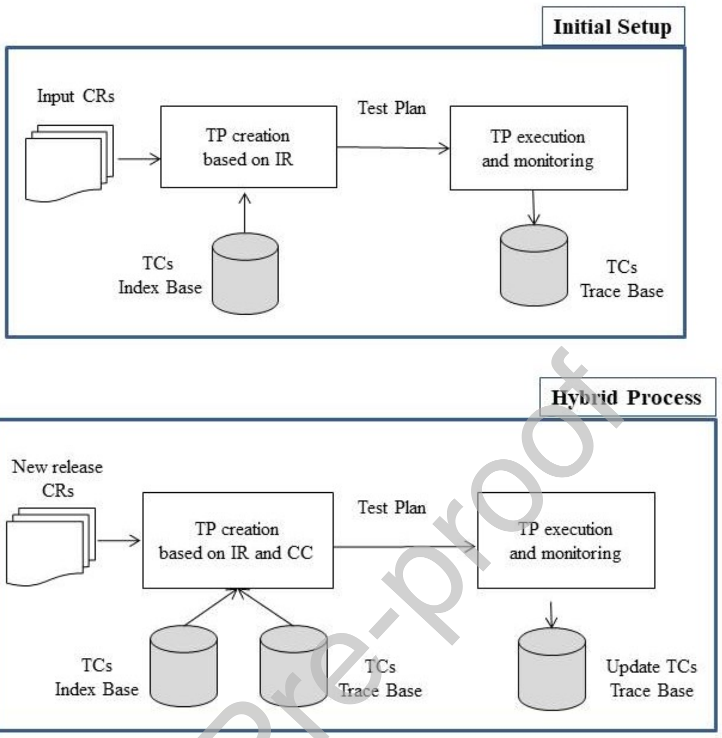


Fig. 2. Overview of the Proposed Process

of TCs (Section 4.4). 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 match code fragment present in 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, 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 4.1 presents the strategy for TCs retrieval and merging based on Information Retrieval; Section 4.2 shows the code coverage monitoring process; and Section 4.4 details the general hybrid process for Test Plan creation.

3.1 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 4.1.

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 textual document, containing words in English, to be indexed. The Master plan is directly obtained from tour 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[6]. We use the Solr Apache[7], which implements a modified Vector Space Model (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 mentioning 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)[8], which can be activated when desired.

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.

3.2 TCs Trace Base Creation and Update

After the TP creation, the testers will execute manually each test case in order to find failures and get the trace. The majority of test cases are not automatized because they involve human gestures (e.g twist the wrist) or other features that are hard to automate.

The Test plan execution is monitored by an implemented Android monitoring tool (Section 4.2), 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 update when a new Test plan is executed. The execution monitoring tool will provide

[6] https://lucene.apache.org/
[7] http://lucene.apache.org/solr/
[8] Porter algorithm: http://snowball.tartarus.org/algorithms/porter/stemmer.html

new associations (TC - trace) to be included in the TCs Traces (see details in Section 4.2).

4 HSP: A Hybrid Strategy for TCs Selection and Prioritisation

According to the related literature, Regression TC selection is more precisely performed using code related artifacts (modified source code and implemented test cases) [6,7,22]. However, we also identified works which use Information Retrieval techniques for TC selection [12,25].

In this light, we envisioned a way to benefit from both directions, creating a hybrid process based on IR and traceability information. This integration is necessary, as justified below.

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. Figure 3 is the detailed view of the Hybrid process presented in Figure 2. This process counts on three main phases:

- Test Plan creation based on Information retrieval (Section 4.1);
- Test Plan creation based on code coverage information (Section 4.3);
- Merge and prioritisation of the Test plans created by these two previous phases (Section 4.4).

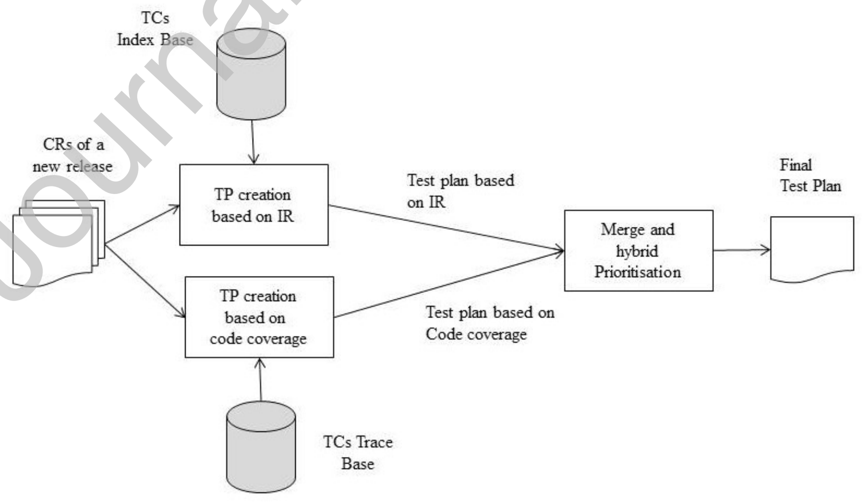


Fig. 3. Hybrid process (selection and prioritisation)

The final Test Plan will be executed and monitored (see Section 4.2). 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 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.

4.1 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 below in Figure 4. Figure 4 is the detailed process of the initial setup in Figure 2 as well as the superior flow in Figure 3. Initially, the input CRs are preprocessed, originating a keyword based representations for each CR. The obtained representations will be used in Phase 2 to build keyword queries to retrieve textual TCs from the TC index base (see Section 3.1). Finally, Phase 3 merges and prioritises the TC lists returned by Phase 2.

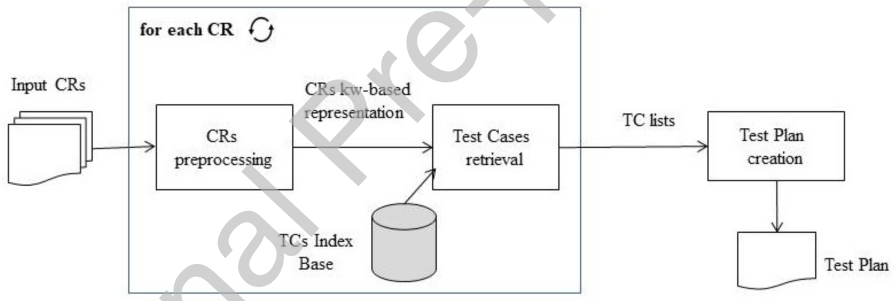


Fig. 4. TP creation based on Information Retrieval

Aiming to respect Software Engineering principles of modularity (to provide for 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[9], a high-level Python Web framework, bearing an MVC (Model-View-Controller) architecture. Figure 6 shows the prototype's interface initial screen.

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

CRs preprocessing. This phase is responsible for the creation of the CRs keyword based representations. The corresponding implemented module receives

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

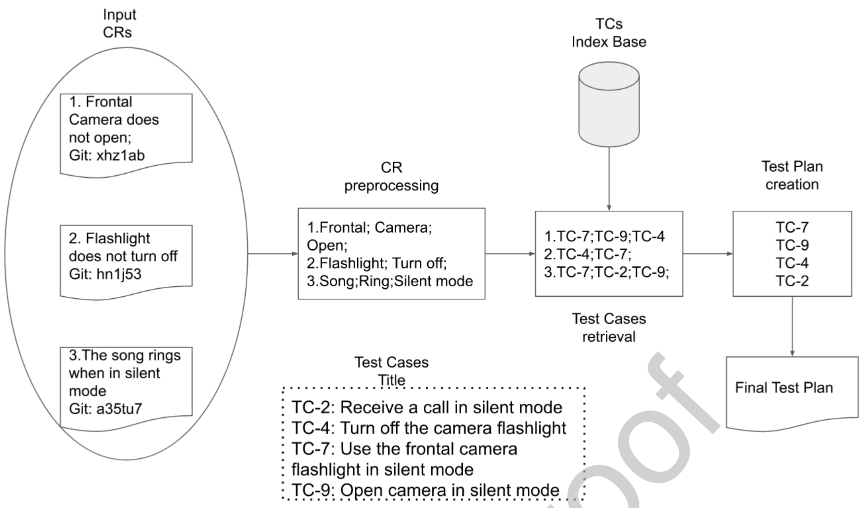
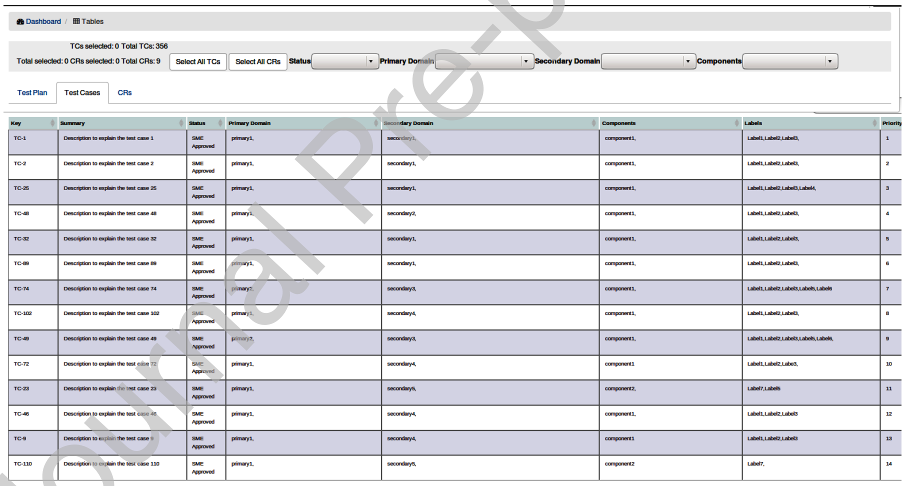


Fig. 5. TP creation based on Information Retrieval example



Dashboard / Tables

TCs selected: 0 Total TCs: 356

Total selected: 0 CRs selected: 0 Total CRs: 9 | Select All TCs | Select All CRs | Status | Primary Domain | Secondary Domain | Components

Test Plan | Test Cases | CRs

Key	Summary	Status	Primary Domain	Secondary Domain	Components	Labels	Priority
TC-1	Description to explain the test case 1	SME Approved	primary1,	secondary1,	component1,	Label1,Label2,Label3,	1
TC-2	Description to explain the test case 2	SME Approved	primary1,	secondary1,	component1,	Label1,Label2,Label3,	2
TC-25	Description to explain the test case 25	SME Approved	primary1,	secondary1,	component1,	Label1,Label2,Label3,Label4,	3
TC-48	Description to explain the test case 48	SME Approved	primary1,	secondary2,	component1,	Label1,Label2,Label3,	4
TC-32	Description to explain the test case 32	SME Approved	primary1,	secondary1,	component1,	Label1,Label2,Label3,	5
TC-89	Description to explain the test case 89	SME Approved	primary1,	secondary1,	component1,	Label1,Label2,Label3,	6
TC-74	Description to explain the test case 74	SME Approved	primary2,	secondary3,	component1,	Label1,Label2,Label3,Label5,Label6	7
TC-102	Description to explain the test case 102	SME Approved	primary1,	secondary4,	component1,	Label1,Label2,Label3,	8
TC-49	Description to explain the test case 49	SME Approved	primary2,	secondary3,	component1,	Label1,Label2,Label3,Label5,Label6,	9
TC-72	Description to explain the test case 72	SME Approved	primary1,	secondary4,	component1	Label1,Label2,Labe3,	10
TC-23	Description to explain the test case 23	SME Approved	primary1,	secondary5,	component2,	Label7,Label5	11
TC-46	Description to explain the test case 46	SME Approved	primary1,	secondary4,	component1,	Label1,Label2,Label3	12
TC-9	Description to explain the test case 9	SME Approved	primary1,	secondary4,	component1	Label1,Label2,Label3	13
TC-110	Description to explain the test case 110	SME Approved	primary1,	secondary5,	component2	Label7,	14

Fig. 6. The tool interface.

the input CRs and delivers their keyword based representations, counting on three steps:

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

For instance, Figure 5 provides a hypothetical example where three CRs are used as an input with the following descriptions: 1. Frontal Camera does not open, 2. Flashlight does not turn off, 3. The song rings when in silent mode. Thus, keywords and stopwords are extracted creating a CR keyword-based representation like that: 1.⟨Frontal, Camera, Open⟩ 2.⟨Flashlight, Turn off⟩ 3.⟨Song, Ring, Silent mode⟩. These data are used afterwards to retrieve test cases.

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 textual TCs from the Index base. As we have mentioned in Section 3.1, the textual TCs descriptions are indexed in order to be able to retrieve them. 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. The search engine compares the query text (CR's keyword text) to the TCs description text. These lists are automatically ordered by the search engine according to the relevance of each TC to the current query (see Section 3.1). The obtained lists of TCs are given as input to Phase 3, described below.

This can be exemplified using again Figure 5 where each CR keyword-based representation, as presented before, is applied as a query to the information retrieval system to recover a list of test cases: 1.⟨ TC-7, TC-9, TC-4⟩, 2.⟨ TC-4, TC-7⟩, 3.⟨ TC-7, TC-2, TC-9⟩. These test cases are selected because their descriptions are: TC-2. Receive a call in silent mode, TC-4. Turn off the camera flashlight, TC-7. Use the frontal camera flashlight in silent mode, TC-9. Open camera in silent mode.

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 must be 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. Figure 7 and Figure 8 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 (Figure 7. 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 which retrieved them.
– **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 apperar 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 8).

Finally, in Figure 5 each list of test cases is merged into a single one. By considering the frequency of each test case in all lists, we get: test case TC-7 has 3 occurrences, test case TC-9 and TC-4 both have 2 occurrences and test case TC-2 has 1 occurrence. Then, the final test plan suggested is ⟨ TC-7, TC-9, TC-4, TC-2⟩.

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.

4.2 Test plan Execution and Monitoring

This section details the test case execution and 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. The test cases in a test

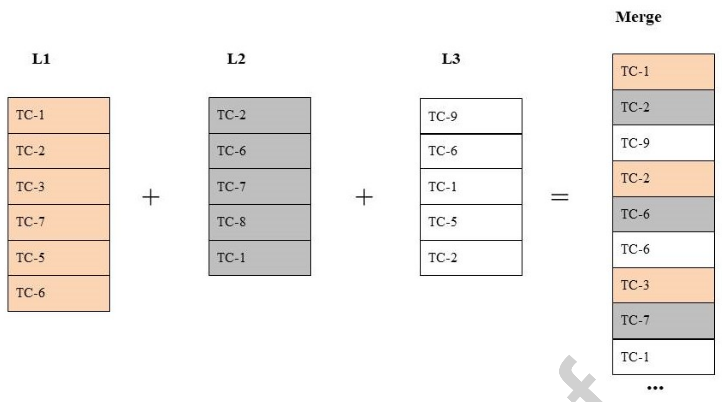


Fig. 7. Initial TC merged list

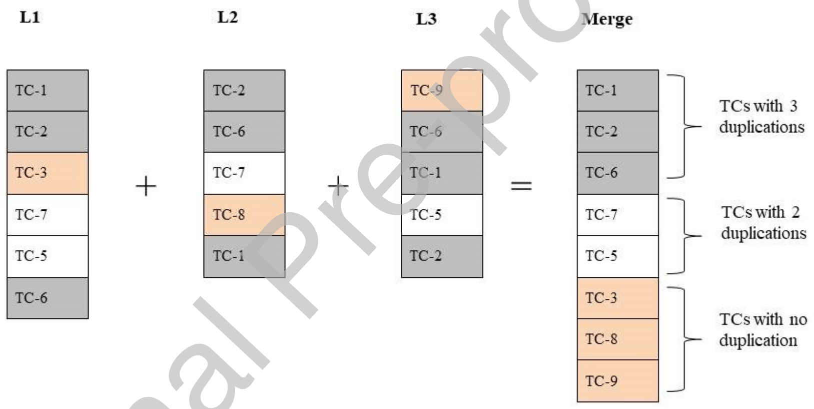


Fig. 8. TCs duplication elimination and reordering

plan are executed manually because there are only a few automated test cases in the test database. Our industrial partner works with smartphones where it is complicated to automate test cases mainly because some features involve human gestures. However, our partner is working in ways to automate the execution of test cases either by capture and replay, programming techniques or using robotic solutions [2,18].

Figure 9 represents the last box about the initial setup as well as the hybrid process in Figure 2, and also details the monitoring process. The monitoring process is divided into three main phases, detailed in what follows:

– 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.

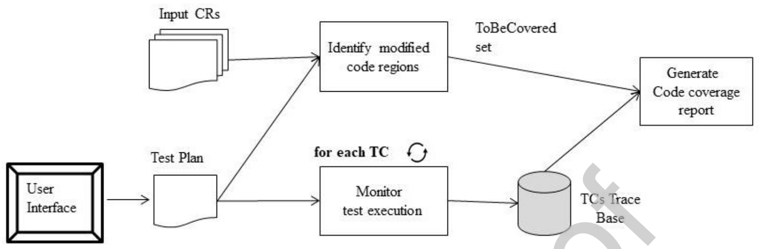


Fig. 9. Monitoring Test Plans execution

The Test Plan execution is monitored by an implemented Android monitoring tool which records the log of each executed TC (trace). 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 4.4.

As our industrial partner does not allow us to perform code instrumentation [11], we could not use traditional code coverage tools (such as JaCoCo [28]). In this context, it was necessary to adopt an alternative way to obtain code coverage: dynamically monitoring code execution using profiling techniques [10].

Since the implemented prototype is based on profiling techniques, it cannot be appropriately applied to time-sensitive tests, because the application may suffer time performance degradation due to the monitoring process. However, this aspect did not impact the conducted experiments presented in this paper (since they consist of functional testing).

The implemented processing modules are detailed below.

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[10]). From 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 restricted to identifying file differences (as provided by Git diffs). We use static analysis [20] to detect new and modified

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

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 $ToBeCovered$

```
function GetToBeCovered
    $ToBeCovered \leftarrow \emptyset$
    $Repo \leftarrow$ UpdateRepoToVersion($TargetApp$, $beta$)
    $AllModifs \leftarrow$ GetRepoDiff($Repo$, $alfa$, $beta$)
    $JavaModif \leftarrow$ FilterJavaDiffs($AllModifs$)
    $JavaFileList \leftarrow$ GetChangedJavaFilesDirectories($JavaModif$)
    $MethodsInfo \leftarrow$ GetMethodsInfo($JavaFileList$)
    for each $JM : JavaModif$ do
        $ChangedMethod \leftarrow$ GetModifMethods($MethodsInfo[JM.Dir], JM$)
        if $ChangedMethod \neq \emptyset$ then
            $ToBeCovered \leftarrow ToBeCovered \cup ChangedMethod$
        end if
    end for
    return $ToBeCovered$
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 if the modified method is in the application. 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 among the two SW versions.

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 4.1), in order to obtain the sequences of method calls. To track all methods called during a test execution, we use the Android Library *ddmlib*[11], 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 in or outside the modified region; that is, methods in or out the $ToBeCovered$ set) from the above step are persisted. There will be one log file associated to each executed TC, 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 4.4).

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 4.4).

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

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.

$$Modif_{seq} = Full_{seq} \downarrow ToBeCovered \tag{1}$$

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:

$$TP\_Coverage = \frac{\#Modif_{seq}}{\#ToBeCovered} \tag{2}$$

The overall coverage must take into account the methods names appearing in more than one test trace sequence. Avoiding counting the same coverage method numerous time.

– **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.

$$CR\_Coverage = \frac{\#Modif_{seq}}{\#CR\_ToBeCovered} \tag{3}$$

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

4.3 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), and uses the Trace base to perform the selection of the related test cases. Note that the Trace base can only contain information related to (several versions of) a single product. It would be misleading to use code coverage information about an 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 10 details the bottom flow in Figure 3. It 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 4.2);
– Test cases retrieval from Trace base;
– TC selection and prioritisation based on code coverage.

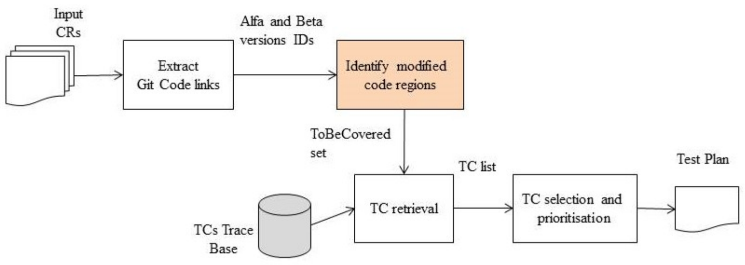


Fig. 10. TP creation based on Code Coverage

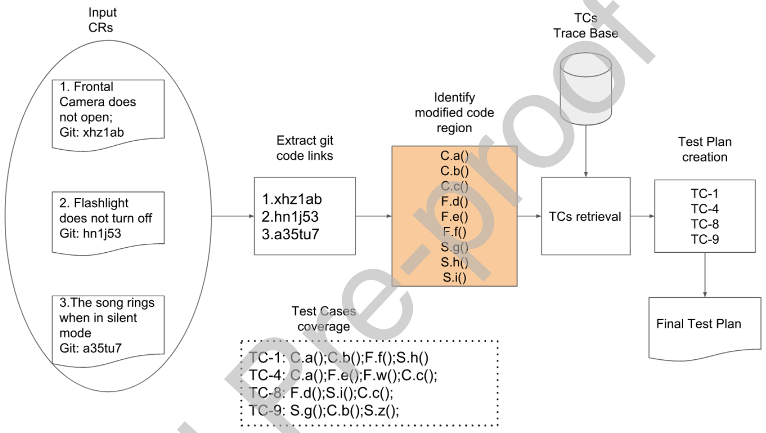


Fig. 11. TP creation based on Code Coverage example

For instance, Figure 11 considers the same three CRs as already presented in Figure 5. But this time our strategy uses the Git code links associated with each CR: 1. xhz1ab, 2. hn1j53, 3. a35tu7. These git code links are sent to the coverage tool which identifies and retrieves the modified source code (in terms of method names, which we call the *ToBeCovered* set): {C.a(), C.b(), C.c(), F.d(), F.e(), F.f(), S.g(), S.h(), S.i()}. The *ToBeCovered* set is used to select and prioritise test cases. In parallel to that, all test cases of a previous test plan have their coverage monitored and stored in the TC trace base as traces of method names (called/executed). In our example, the test cases traces of the previous run are: TC-1:⟨ C.a();C.b();F.f();S.h()⟩, TC-4:⟨ C.a();F.e();F.w();C.c()⟩, TC-8:⟨ F.d();S.i();C.c() ⟩, TC-9:⟨ S.g();C.b();S.z() ⟩. Thus, for example, as C.a() (a method name from the *ToBeCovered* set) belongs to the traces of test cases TC-1 and TC-4, these test cases are selected.

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.

Initial Steps

– Extract Git code links from the input CRs;
– Identify modified code regions (Section 4.2);
– Test cases retrieval from Trace base;

Selection using total coverage ($\mathbf{CCS_t}$) Based on the coverage information obtained by the execution monitoring procedure (Section 4.2), 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 all test cases ($tests$)
Output: A list of test cases to compose a test plan ($plan$)

```
function SelectPlanFromCode
    $plan \leftarrow []$
    for each $tc \in tests$ do
        if $tc.methods \cap ToBeCovered \neq \emptyset$ then
            $plan.append(tc)$
        end if
    end for
    return $sortedByCoverage(plan)$
end function
```

The selection strategy described in Algorithm 2 is implemented by traversing all TCs ($tc \in tests$) 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 3).

Selection using additional greedy ($\mathbf{CCS}_g$) 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 emptied. 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*.

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 $GreedyAdditionalCoverage$
        $greedy\_selection \leftarrow []$
        $frontier \leftarrow tests$
        $additional\_methods \leftarrow \emptyset$
        repeat
            $maximum\_additional \leftarrow 0$
            $additional\_tc \leftarrow None$
            for each $tc \in frontier$ do
                $coverage \leftarrow \#((tc.methods \cup additional\_methods) \cap TBC)/\#TBC$
                if $coverage > maximum\_additional$ then
                    $maximum\_additional \leftarrow coverage$
                    $additional\_tc \leftarrow tc$
                end if
            end for
            if $additional\_tc \neq None$ then
                $greedy\_selection.append(additional\_tc)$
                $additional\_methods \leftarrow additional\_methods \cup additional\_tc.methods$
                $frontier.remove(additional\_tc)$
            end if
        until $additional\_tc = None$
        return $greedy\_selection$
    end function
```

After the variables initialization, 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 update when there is coverage to consider and a higher coverage rate can be achieved, respectively. If this holds, the 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.

Figure 11 also illustrates that the selected test cases are finally prioritised using their frequency of occurrence from each test case trace. In this scenario, test case TC-1 has 4 methods belonging to the *ToBeCovered*, test cases TC-4 and TC-8 have 3 methods and test case TC-9 has 2 methods. Then, the final test plan suggested is ⟨TC-1, TC-4, TC-8, TC-9⟩. It is worth recalling that CCS$_g$ selection technique could be used as well. However, we have demonstrated only CCS$_t$ in this example.

4.4 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 4.1 and 4.3. This means that it is the last box in Figure 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, will never be selected, since there will be no coverage data provided by Algorithm 2 or Algorithm 3.
2. Tests never selected before cannot be related to source code methods in the CRs because they had not 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 (**CCS**$_g$) or the selection using total coverage (**CCS**$_t$). In this case we add the subscript to indicate which case is being considered in what follows.

Merge and Priotitisation using Code Coverage Ranking (MPCCR) This prioritisation assigns importance to the code coverage ranking. Therefore, the code coverage selection order is maintained and 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).

The MPCCR ranking was implemented according to Algorithm 4 where it receives two test plans: *planFromIR* that is the test plan based on information retrieval and *planFromCC* that is the test plan based on code coverage. The function *MergePlansBasedOnCC* merges these two test plans into a single one called *mergedPlan*. The function *MergePlansBasedOnCC* traverses all test cases in the *planFromCC* and adds them into *mergedPlan*,

Algorithm 4 MPCCR

Input: Two lists: Test Plan from IR (*planFromIR*) and Test Plan from CC (*planFromCC*)
Output: A merged test plan (*mergedPlan*)

```
function MergePlansBasedOnCC
    $mergedPlan \leftarrow []$
    for each $tc \in PlanFromCC$ do
        $mergedPlan.append(tc)$
    end for
    for each $tc \in PlanFromIR$ do
        if $tc \notin mergedPlan$ then
            $mergedPlan.append(tc)$
        end if
    end for
    $UpdateRankingIndex(mergedPlan)$
    return $mergedPlan$
end function
```

then it traverses all the test cases in the *planFromIR* and adds them into *mergedPlan*, only those that are not already in the *mergedPlan*. Finally, the function *UpdateRankingIndex* assigns a ranking value to these test cases.

Merge and Priotitisation using Information Retrieval Ranking (MPIRR)
This prioritisation assigns importance to the information retrieval ranking. Therefore, the code coverage selection is at the top of the plan because it is directly related to the modified source code and its order is changed by similar test cases selected by information retrieval. Then, it is complemented by the remaining test cases selected by information retrieval.

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).

The MPIRR ranking is implemented according to Algorithm 5 where it receives two test plans: *planFromIR* that is the test plan based on information retrieval and *planFromCC* that is the test plan based on code coverage. The function *MergePlansBasedOnIR* merges these two test plans into a single one called *mergedPlan*. Firstly, the function traverses all the test cases in the *planFromCC* and adds them into *mergedPlan* if they are not in *planFromIR*. Otherwise, it adds *None* into *mergedPlan* to preserve the index to another test case and adds the test cases into *auxiliary* the test case from *planFromIR* because of the IR index. Next, it sorts the list *auxiliary* by the test cases ranking index using the function *sortByRankingIndex*. Thus these test cases will

Algorithm 5 MPIRR

Input: Two lists: Test Plan from IR ($PlanFromIR$) and Test Plan from CC ($PlanFromCC$)
Output: A merged test plan ($mergedPlan$)

```
function MergePlansBasedOnIR
    $mergedPlan \leftarrow []$
    $auxiliary \leftarrow []$
    for each $tc \in PlanFromCC$ do
        if $tc \notin PlanFromIR$ then
            $mergedPlan.append(tc)$
        else
            $mergedPlan.append(None)$
            $auxiliary.append(getTestCaseFromList(tc, PlanFromIR))$
        end if
    end for
    $sortByRankingIndex(auxiliary)$
    for each $tc \in mergedPlan$ do
        if $tc = None$ then
            $tc \leftarrow auxiliary.pop()$
        end if
    end for
    for each $tc \in PlanFromIR$ do
        if $tc \notin mergedPlan$ then
            $mergedPlan.append(tc)$
        end if
    end for
    $UpdateRankingIndex(mergedPlan)$
    return $mergedPlan$
end function
```

follow the IR order. After this, it traverses the *mergedPlan* to fill the empty spaces with the *auxiliary* test cases. Then it traverses all the test cases in the *planFromIR* and adds them into *mergedPlan*, only those that are not already in the list *mergedPlan*. Finally, the function *UpdateRankingIndex* assigns a ranking value to these test cases.

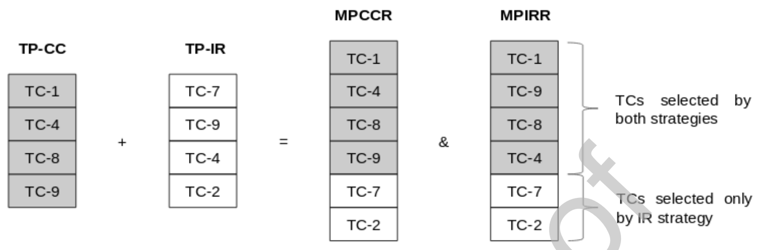


Fig. 12. Hybrid prioritisation

5 Empirical Evaluation

This section presents an empirical evaluation of the selections and prioritisations strategies presented in this work. We consider the test plan creation using only information retrieval (IR), described in Section 4.1, the test plan creation using only code coverage (CCS), described in Section 4.3. 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, with and without 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. For the next RQs, we evaluate the results considering the entire technique selection. This means that we compare all test cases selected by the techniques (IR, CCS, HSP) with all test cases selected by the test architects. Also, we compare the selections according to the operational team capacity to execute the test plans; we call them as Shrunk Plans. Sometimes, the team does not have enough time to execute all test cases selected by the technique. Then it is necessary to shrink the capacity to execute. For us, the operational capacity is the same amount of TCs that is selected by the test architect since they select the max amount of TC that can be executed.

RQs related to the Setup run

- **RQ1:** Considering random, architects and TC selection solely based on IR, which one does exhibit the highest coverage of modified regions when
  1. All tests are executed?
  2. The available operational capacity is executed?
- **RQ2:** Among IR based, random and architects selections, which one does detect the highest number of failures when
  1. All tests are executed?
  2. 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
  1. All tests are executed?
  2. The available operational capacity is executed?
- **RQ4:** Regarding each CR, what is the coverage reached by the Test plan when
  1. All tests are executed?
  2. The available operational capacity is executed?

RQs related to the Hybrid run

- **RQ5:** Considering TC selection solely based on IR, solely based on code coverage, hybrid, random and architects selections, which one does exhibit the highest coverage of modified regions when
  1. All tests are executed?
  2. The available operational capacity is executed?
- **RQ6:** Among TC selection solely based on IR, solely based on code coverage, hybrid, random and architects selections, which one does detect the highest number of failures when
  1. All tests are executed?
  2. 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?
  1. All tests are executed?
  2. The available operational capacity is executed?
- **RQ8:** What is the relation between the the coverage estimate for the Test plan (using previous data) and the real monitored coverage?
- **RQ9:** Regarding each CR, what is the coverage reached by the Test plan when
  1. All tests are executed?
  2. The available operational capacity is executed?

Moreover, we separate the RQs in categories because we understand that it is easier to analyse the evaluation separately. Although, all elements are considered in the interpretation. The experiments 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 HSP without Code Coverage (Setup run)

This section presents the experiments results regarding TC selection based solely on Information Retrieval (IR). As mentioned in Section 3, 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.

Despite 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).

(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 4.1). The manual execution of this Test plan was monitored, having obtained a code coverage of 49.38%, as seen in Table 2. The architect, on the other hand, selected 88 test cases, where 85% of these 88 test cases also appeared in the automatically created Test Plan. The architect's selection obtained a coverage of 44.44%. Thus, it is not reasonable (particularly in a real scenario) to increase the test effort by 96 test cases (184-88) and have just a gain of 5% in code coverage. However, these new 96 test cases found 14 bugs (20 - 6) with respect to the test architect and thus this is meaningful.

It is worth mentioning that the architect is allowed to select 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, assuming that these tests are independent (i.e., the execution order does not influence the coverage obtained). We used the same IR based coverage data due to the time to execute all those. These randomised lists 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 2 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 13

The IR selection coverage decreased in the shrunk case. This can be an indication that the prioritisation is not well related to code coverage. As the literature [26] already showed that, code coverage and bug finding are not directly related. But this decrease in coverage cannot be assumed to be absolutely bad because the IR selection is related to the test cases text. Therefore, it needs to verify the other criteria.

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 2. Setup run - Coverage

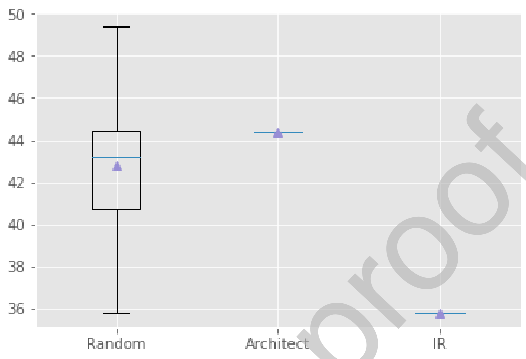


Fig. 13. Setup run - Coverage - shrunk

(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 failures identified. On the other hand, the architect's selection found only 6 failures whereas IR has found 20 failures (see Table 3).

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 14 for the the box plot related to these results.

About the failures found, all of them were real failures found in the system. As we cannot instrument or change the source code we cannot seed any failure. From the 20 failures found in the IR selection, 15 failures were unique while the IR shrunk found 7 unique failures. In the Architect selection, the 6 failures found were different among them.

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

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

Table 3. Setup run - Failures found

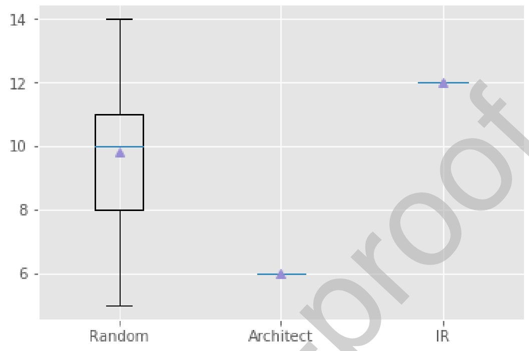


Fig. 14. Setup run - Failures Found - Shrunk

(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, and no correlation when the value is close to 0.

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 15) even when we consider the selection of 88 test cases (the operationally viable execution) (see Figure 16). 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 before 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) are not chosen.

(RQ4) CR Coverage Analysis: Figure 17 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.

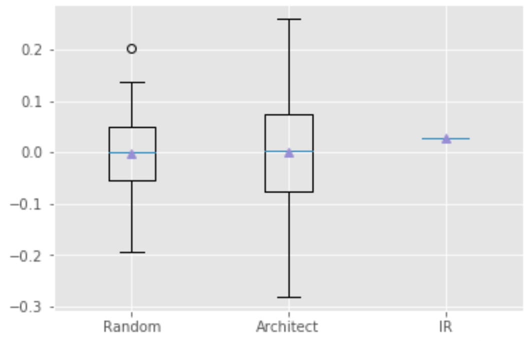


Fig. 15. Setup run - Correlation - Normal

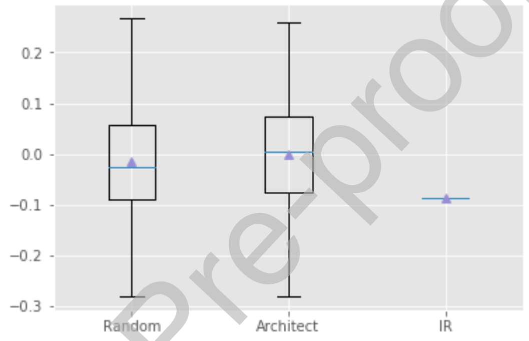


Fig. 16. Setup run - Correlation - Shrunk

On the other hand, when we consider only the 88 top test cases from the IR selection, Figure 18 shows that this strategy achieved the same coverage in terms of CRs than the architect's selection.

It is important to note that 3 CRs reached 0% of code coverage and almost 50% of the CRs were not entirely covered. This indicates that part of the source code was not tested and these areas are considered stable. Besides, failures may be escaping because of the lack of TCs for testing these areas.

5.2 Full HSP (IR and CC merge and prioritisation)

The Hybrid run executes selection and prioritisation based on IR and code trace 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

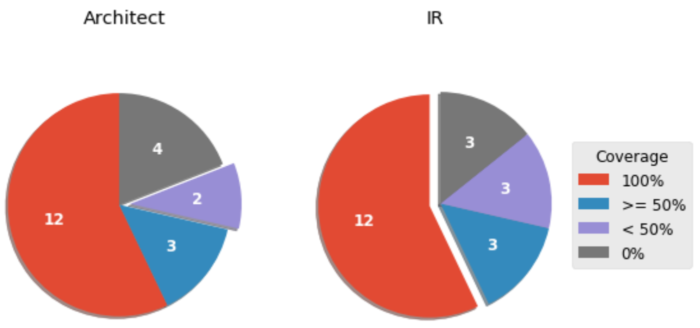


Fig. 17. Setup run - CR Coverage - Normal

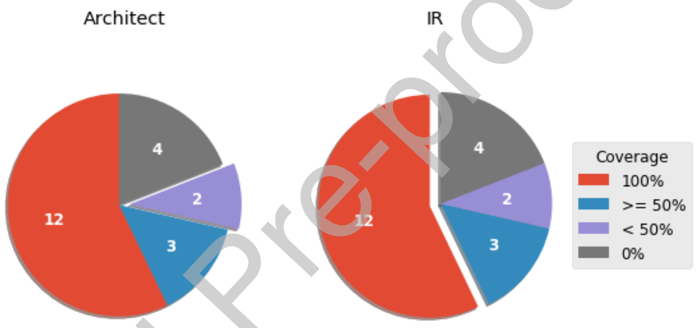


Fig. 18. Setup run - CR Coverage - Shrunk

this run was obtained from the Setup run, a question of interest is whether the coverage estimate for the Test 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 has occurred because the setup run was run using 4 RNs while the hybrid was run using 10 RNs. Thus the hybrid run has considered more SW modifications.

(RQ5) Comparison of selection strategies with respect to coverage: Table 4 presents the number of TCs selected by the IR based strategy (302 tests), CCS$_t$ strategy using just the coverage information from Setup Run (183 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).

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 4. Hybrid run

Table 4 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 19 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.

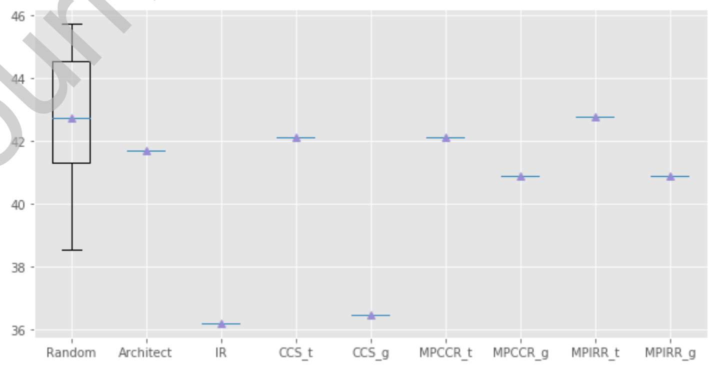


Fig. 19. Hybrid run - Coverage - Shrunk

(RQ6) Comparison of selection strategies with respect to failures found: Table 4 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 20 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. About the failures, those failures were real and not seeded, as said before. From the 13 failure found in the hybrid selections, IR and CCS_t, 7 were unique failures. In the Architect selection, the 2 failures found were different between them. In the selections shrunk, the MPCCR_t, MPCCR_t and MPIRR_g selections found 3 unique failures, while the IR selection found 4 unique failures and the MPIRR_t found 6 unique failures. Confirming the evidence which was presented before.

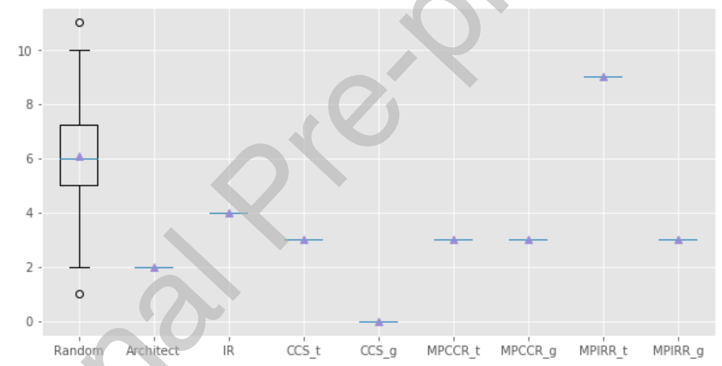


Fig. 20. Hybrid run - Failures Found - Shrunk

(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 21 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 22). 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.

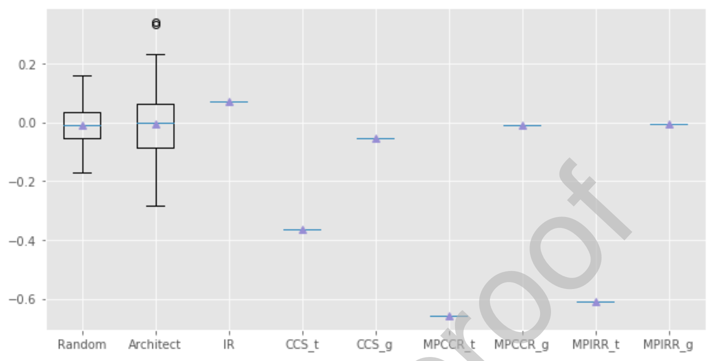


Fig. 21. Hybrid run - Correlation - Normal

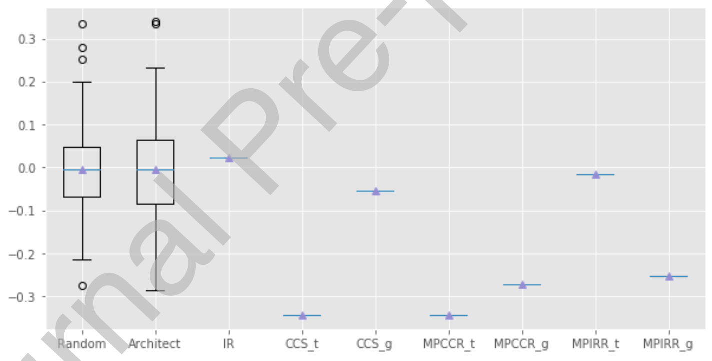


Fig. 22. Hybrid run - Correlation - Shrunk

(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.

(RQ9) CRs Coverage Analysis: Figure 23 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 24 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 reached 0% of code covered and almost 75% of the CRs were not fully covered. This indicates again that part of the source code was not tested and these areas are considered stable. Besides, failures may be escaping because of the lack of TCs for testing these areas.

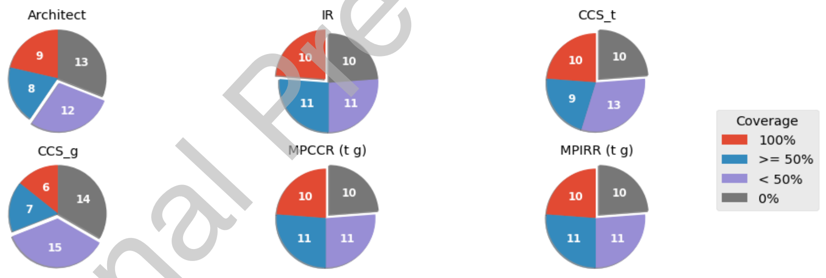


Fig. 23. Hybrid run - CR Coverage - Normal

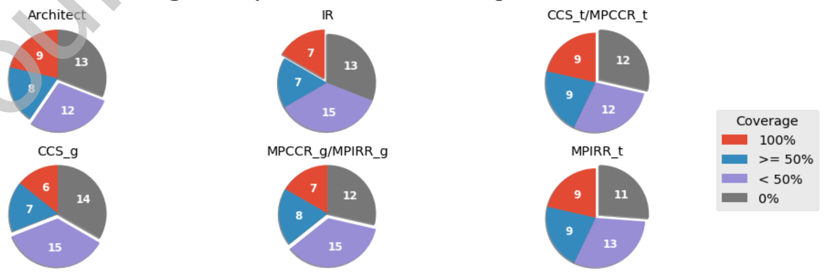


Fig. 24. Hybrid run - CR Coverage - Shrunk

6 Related Work

Selection and prioritisation of test cases are research areas of great interest, specially concerning Regression testing [24]. 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 [12,13,21,25] and code coverage in [6,7,22].

The work of [12] 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 [12], 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 (not source code). Another difference is that we propose TCs selection followed by prioritisation. The work [12] uses the rank provide 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 [12] does not consider an industrial context.

Similarly to our approach, the work [21] proposed a test case prioritisation method to Web services compositions based on CR descriptions. However, Information Retrieval is used in [21] to match the terms found in the CRs with test cases' traces, while we use textual test cases. in [21] 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 [21], 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 [13]. 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 [13], 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 [25]. 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 [25] uses a modified version of BM25 model that fits better their environment, whereas we use the Solr score (detailed in 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 [25].

The work reported in [6] 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 4.2).

A Regression testing selection tool, named Ekstazi, was proposed in the work [7] to adopt the RTS (Regression Test Selection) 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 [22]. 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 [22] used automated test cases, while we used manual test cases. The work reported in [22] does not evaluate coverage, only performance.

7 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, 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, where the CCS selects TCs straight related to the modified source code while the IR helps to improve test cases ranking and update the trace database. Although the IR selection results as presented in this paper, the IR selection was used by our partner in a simple and easy way [14]. It is easier to apply IR in our context than code coverage because it does not need cover test by test, which is not usual for the testers. The IR selection just needs the text of test cases and CRs, as they are already available. However, code coverage impacts the test case execution, as long as the testers may monitor each single test case execution.

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. Some CRs were not covered in those test suites. This may happen because there are new features' CRs or the test cases in the database are not enough (although the architects try to create new test cases in this case as well) to cover the source code modification. In both cases, it is important to create new test cases. 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.

As first future work, we intend to monitor the use of the proposed combinations by our industrial partner, in order to identify which combination is indeed the best one in the long run. That is, we intend to conduct continuous experiments. We also intend to do experiments to compare the HSP with other RTS and RTP techniques and to create plans individually per CR instead of a set.

Another future work concerns in applying test suite reduction regarding similarity, using both natural language as well as source code. 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.

Yet, we also consider investigating prioritisation strategies reported in [15], in terms of coverage and bugs found. This is an interesting future work to pursue as well.

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

Acknowledgment

We would like to thank Alice Arashiro, Viviana Toledo, Eliot Maia, and Lucas Heredia from Motorola Mobility, as well as Virginia Viana from the collaboration between CIn and Motorola Mobility. This work was supported by the research cooperation project between Motorola Mobility (a Lenovo Company) and CIn-UFPE.

References

1. Surafel Lemma Abebe, Nasir Ali, and Ahmed E. Hassan. An empirical study of software release notes. *Empirical Software Engineering*, 21(3):1107–1142, 2016.
2. Filipe Arruda, Augusto Sampaio, and Flávia A. Barros. Capture & replay with text-based reuse and framework agnosticism. In *The 28th International Conference on Software Engineering and Knowledge Engineering, SEKE 2016, Redwood City, San Francisco Bay, USA, July 1-3, 2016.*, pages 420–425, 2016.
3. Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
4. Jørn Ola Birkeland. *From a Timebox Tangle to a More Flexible Flow*, pages 325–334. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
5. Bugzilla. http://www.bugzilla.org/. (accessed Dec 01, 2016).
6. 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. stvr.1572.
7. 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.
8. Rothermel Gregg, Harrold Mary Jean, von Ronne Jeffery, and Hong Christie. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249.
9. 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.
10. 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, 2012. ACM.
11. 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, 2012. ACM.
12. 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.
13. Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, Mar 2012.
14. 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, 2016.

15. 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, 2016.
16. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
17. Mantis. Mantis bug tracker. http://www.mantisbt.org/. (accessed Dec 01, 2016).
18. Ke Mao, Mark Harman, and Yue Jia. Robotic testing of mobile apps for truly black-box automation. *IEEE Software*, 34(2):11–16, mar 2017.
19. 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.
20. 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.
21. 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.
22. 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.
23. Redmine. Flexible project management web application. http://www.redmine.org/. (accessed Dec 01, 2016).
24. Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, August 1996.
25. 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.
26. 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.
27. 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.
28. Qian Yang, Jingjing Li, and David M. Weiss. A survey of coverage based testing tools. *Comput. J.*, 52:589–597, 2006.
29. S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

Biography

Claudio Magalhes

He holds a degree in Information Systems from the Federal University of Pernambuco (2016) and a Master's Degree in Computer Science from the Federal University of Pernambuco (2019). Currently, He is a Doctors student in Computer Science from the Federal University of Pernambuco. His research interests includes Software Engineering, Software Testing, Natural Language Processing, Artificial Intelligence.

Joo Luiz
He is a Bachelor's student in Computer Science from the Federal University of Pernambuco since 2013. His research interests are Software Engineering, Software Testing and Smartphones (Android).

Lucas Perrusi

He is a Bachelor's student in Computer Science from the Federal University of Pernambuco since 2013. His research interests are Software Engineering and Software Testing.

Alexandre Cabral Mota

Alexandre Mota is professor at Center of Informatics (UFPE) and holds a researcher scholarship from CNPq since 2012. His interests include CSP, testing and model checking, theorem proving, formal modelling and refinement. From 2005 to 2009 he was a researcher in a project between CIn and Motorola Mobility. From 2011 to 2014 he was a deputy (Brazilian side) of the COMPASS Project (FP7). Since 2006 he has been collaborating with Embraer, focusing on safety assessment and model-based testing. Now he also coordinates a new research project initiative between CIn and Motorola Mobility towards smartphone testing that started since 2015.