UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ISABELLY SANTOS CAVALCANTE

TEST CASE PRIORITIZATION: A CASE STUDY IN THE EVOLUTION OF A REAL SYSTEM

CAMPINA GRANDE - PB
2020

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Test Case Prioritization: A Case Study in the Evolution of a Real System

Isabelly Santos Cavalcante

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Testes em Software

Dr. Tiago Lima Massoni (Orientador)
Dra. Patrícia Duarte de Lima Machado (Orientadora)

Campina Grande, Paraíba, Brasil

C376t Cavalcante, Isabelly Santos.
Test case prioritization : a case study in the Evolution of a real system / Isabelly Santos Cavalcante. - Campina Grande, 2020.
55 f. : il. Color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2020.
"Orientação: Prof. Dr. Tiago Lima Massoni, Profa. Dra. Patrícia Duarte de Lima Machado.
Referências.

1. Engenharia de Software. 2. Teste de Regressão. 3. Priorização de Casos de Teste. 4. Prioritization Techniques. 5. Test Case Prioritization. 6. Regression Testing. I. Massoni, Tiago Lima. II. Machado, Patrícia Duarte de Lima. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELO BIBLIOTECÁRIO GUSTAVO DINIZ DO NASCIMENTO CRB-15/515

TEST CASE PRIORITIZATION: A CASE STUDY IN THE EVOLUTION OF A REAL SYSTEM

ISABELLY SANTOS CAVALCANTE

DISSERTAÇÃO APROVADA EM 03/02/2020

TIAGO LIMA MASSONI, Dr., UFCG
Orientador(a)

PATRICIA DUARTE DE LIMA MACHADO, PhD, UFCG
Orientador(a)

EVERTON LEANDRO GALDINO ALVES, Dr., UFCG
Examinador(a)

JULIANO MANABU IYODA, PhD, UFPE
Examinador(a)

CAMPINA GRANDE - PB

Resumo

A manutenção de software pode introduzir novos bugs em um sistema. Por isso, é crucial verificar os impactos causados no restante do código. Uma metodologia amplamente usada para validar alterações no programa é o teste de regressão. Ele consiste em executar os testes antigos na nova versão do sistema para verificar se não houve alterações no comportamento. No entanto, esse processo pode ser caro e demorado. A Priorização de Casos de Teste (PCT) é uma das estratégias desenvolvidas para ajudar a resolver esse problema. Ela reordena o conjunto de testes com base em um determinado critério, para que os casos de teste prioritários sejam executados primeiro. Existem muitas técnicas de PCT disponíveis na literatura. E, ainda assim, algumas equipes de desenvolvedores ainda encontram suas próprias maneiras de passar pelos custos da regressão. Neste trabalho, realizamos um estudo de caso em um grande sistema em evolução para avaliar e comparar a eficácia de seis técnicas de priorização, que usam informação de cobertura, com o que ocorre na prática durante o desenvolvimento de um software. Os desenvolvedores deste sistema realizam testes em dois momentos: antes de integrar o código, onde é necessário executar o conjunto inteiro, e durante o desenvolvimento, onde geralmente escolhem os testes a serem executados para economizar tempo. Nosso objetivo neste trabalho é analisar o possível uso das técnicas de PCT durante o processo de desenvolvimento e integração do código nesse sistema. Existem apenas alguns estudos de caso industriais envolvendo faltas reais e avaliando técnicas de PCT que utilizam informações de cobertura em um sistema com integração contínua. Nossos resultados mostraram que: 1) as técnicas baseadas em modificação são as mais eficazes na detecção de faltas e na redução da taxa de dispersão de testes falhos, entre as técnicas de PCT avaliadas. 2) No cenário em que todos os testes precisam ser executados, todas as técnicas de PCT apresentaram um melhor resultado que a ordem original. 3) Embora as técnicas PCT tenham melhores resultados do que a ordem original quando todos os testes precisam ser executados, elas não foram mais eficazes na detecção de faltas do que a seleção do desenvolvedor quando as restrições de tempo são maiores. Eles geralmente selecionam o mesmo número, ou menos, de testes com falha usando o mesmo tempo de execução. 4) A seleção que o desenvolvedor faz é talvez mais eficaz, porque ele quase sempre seleciona testes alterados/adicionados, que são os que geralmente falham, enquanto as técnicas não

priorizam esses testes. Testes adicionados ou alterados sempre devem ser considerados ao se propor uma nova técnica de priorização.

Abstract

The software maintenance can introduce new bugs in the system. Therefore, it is crucial to check the impacts caused on the rest of the code. One widely used methodology for validating program changes is regression testing. It consists of running the old tests on the new version of the system to check that there have been no changes in behavior. However, this process can be expensive and time-consuming. The Test Case Prioritization (TCP) is one of the strategies developed to help solve this problem. It rearranges the test suite based on a given criterion so that priority test cases are performed first. There are many TCP techniques available in the literature. And still, some developer's teams still find their own ways of going through the regression costs. In this work, we conducted a case study on a large evolving system to evaluate and compare the effectiveness of six prioritization techniques, which use coverage information, with what happens in practice during software development. The developers of this system perform tests at two moments: before integrating the code, where it is required to run the entire suite, and during development, where they usually choose the tests to run to save time. Our objective in this work is to analyze the possible use of TCP techniques during the process of developing and integrating the code in this system. There are only a few industrial case studies involving real faults and evaluating TCP techniques that use coverage information in a system with a continuous integration environment. Our results showed that: 1) the modification-based techniques are the most effective in fault detection and reduction of the spread rate of failed tests, among the evaluated TCP techniques. 2) In the scenario where all the tests need to be performed, all the TCP techniques presented a better result than the original order. 3) Although TCP techniques had better results than the original order when all the tests need to be run, they were not more effective at fault detection than developer selection when time constraints are higher. They usually select the same number, or less, of failed tests using the same execution time. 4) The developer selection is maybe more effective because he almost always selects changed/added tests, which are the ones that usually fail, whereas the techniques do not prioritize these tests. Added or altered tests should always be considered when proposing a new prioritization technique.

Acknowledgements

First of all, I would like to thank *God* for always giving me strength and patience when I needed it, for allowing me to complete this stage of my studies and for the blessings poured out on my life.

I am immensely grateful to my husband *Matheus* for literally everything. For being always present, patient and helpful. For all the love, care, and encouragement. For always helping me when I was lost. For every night of company. For cheering and believing in me more than myself. For being my best friend, my mentor and love of my life.

To my sister *Carol*, one of the most important people to me, for all the love, affection, conversations, and moments of relaxation.

To my father *Assis* and all my family for always being by my side, providing love, encouragement, and support. For always understanding that I was not able to visit them as I wanted. I am especially gratefully to my aunt *Marcia* for always believing in me.

I also thank my mentors *Tiago* and *Patricia*, for guiding me on this journey and for all these years of teaching, discussion, trust, patience, review, and availability. For always being excellent teachers.

To my close friends, specially *Ivyna* and *Adysia*, I am very grateful for all the patience over these years and for never left my side. To my colleagues and friends of UFCG, who have helped me throughout this journey, you have certainly made these years easier. I especially thank my course mates *Wesley, Lucas, Julio*, and *Marcos* for having gone all this way together. To *Julie* for friendship, snacks, and company, especially this past year.

My sincere thanks to all members *ePol* for making our day to day work enjoyable and for being the best dream team. Thanks for sharing so much knowledge with me.

To teachers and employees of *COPIN* and *SPLab* for the great help, patience, and competence. Special thanks to Professor *Everton* for introducing me to the testing world as a graduate student and always answering my questions. To *Lilian* and *Marilene* for all the help and talk these years. And *Paloma* and *Lyana* for all help with the processes and questions.

Lastly, I thank *CNPq* for the financial support provided for this work.

Contents

1 Introduction **1**

2 Background **4**

2.1 Test case prioritization 4

2.1.1 Studied techniques 5

2.1.2 Metrics 7

2.2 Continuous integration 10

3 Study Methodology **12**

3.1 Study definition 12

3.1.1 Research questions 12

3.2 Study context 13

3.2.1 The changes and their workflow 14

3.3 Experimental Procedure 15

3.3.1 Experimental units 15

3.3.2 Instrumentation and data collection 16

3.3.3 Applying prioritization techniques 19

4 Results **25**

4.1 Answers to research questions 1 to 4 25

4.1.1 RQ1: Do the chosen TCP techniques improve the system test suite fault detection rate? 25

4.1.2 RQ2: Are the chosen TCP techniques equally efficient according to the APFD metric? 29

4.1.3 RQ3: Do the chosen TCP techniques decrease the spread of failed test cases in the system test suite? 30
4.1.4 RQ4: Are the chosen TCP techniques equally efficient according to the F-Spreading metric? 33
4.2 Answers to research questions 5 and 6 34
4.2.1 RQ5: Was the developer able to select all tests that detected the fault? 34
4.2.2 RQ6: Are time-constrained prioritization techniques more efficient than manual selection in the development environment? 35

5 Discussion **38**
5.1 Prioritization tradeoffs 38
5.2 Coverage Processing 40
5.3 Project-related issues 41
5.4 Practice of Testing 43
5.5 Threats to validity 44

6 Related Work **46**
6.1 Prioritization techniques and their evaluation 46
6.2 Empirical studies 48

7 Conclusion **51**

List of Figures

2.1 Graph representing the APFD calculation of T_1. 8
2.2 Graph representing the APFD calculation of T_2. 9
2.3 Example of the CI process. 10

3.1 System change development flow. 15
3.2 Scenarios where the data collection script occurred. 18
3.3 Data flow after script call. 18
3.4 Extraction steps of the data coverage information. 22
3.5 Script executing the TCP techniques. 24

4.1 Boxplots with the APFD distribution per technique. 26
4.2 Confidence interval for the APFD median values of each technique. 27
4.3 Boxplots with the F-Spreading distribution per technique. 30
4.4 Confidence interval for the F-Spreading median values of each technique. . 31
4.5 Comparison, per experimental unit, of the number of failed tests that the most efficient technique detected utilizing the time it took the developer to perform her selection, with the number of tests the developer detected, and with the total number of tests that had to be detected. 36
4.6 Comparison between the execution time of the developer-selected tests, and the time that the most effective technique of each respective unit, took to perform the same amount of failed tests. 37

5.1 Scenario using the coverage memory solution. 42

List of Tables

2.1 Test suite and list of faults exposed by the same. 8

3.1 Summary of the discarded tests runs. 20

4.1 Shapiro-Wilk Normality Test for APFD metric. 27

4.2 Conover test (p-value) for APFD metric. In gray cells, the pairs with population difference. 28

4.3 Cliff Delta measurement result for APFD metric. In gray cells, the pairs with the largest effects. 29

4.4 Shapiro-Wilk Normality test for F-Spreading metric. In gray cells, the techniques that follow a non-normally distribution. 32

4.5 Conover test (p-value) for F-Spreading metric. 32

4.6 Cliff's Delta measure results for F-Spreading metric. 33

5.1 Average processing time of each technique. In gray, the technique with the shortest time of processing. 40

Chapter 1

Introduction

The modern world could not exist without software [22]. It is present in many products, services, and structures around the world. However, no matter how well designed and tested the software was before it was delivered, it will eventually be modified [18; 26]. According to Lu et al. [11], during software maintenance, a system continuously evolves due to various reasons, e.g., adding new features, fixing bugs, or improving efficiency and maintainability.

The maintenance can introduce new bugs in the system, so it is crucial to verify the impacts caused on the rest of the code. One widely used methodology for validating program changes is regression testing [18]. It consists of running the old tests on the new version of the system to check that areas supposedly not affected by the updates kept the same behavior as before. However, this process can be expensive and time-consuming [3; 11; 17; 24]. According to some works, running an entire test suite may take weeks [16]. And as the software evolves, its test suite also tends to grow, and because of that, it will need more resources over time. Many strategies have been developed to help to solving this problem, one of them is Test Case Prioritization (TCP).

This strategy aims to reorganize a test suite to improve the achievement of specific testing goals (e.g., a faster rate of fault detection) [2]. In other words, for fast fault detection, it sorts the test suite and puts the test cases that detect the fault in the first positions so that they can be executed earlier than the rest. With a good prioritized test suite, developers can run only the top test cases, or the number of tests which is feasible due to resource limitations, without losing much of the testing potential [3]. Many prioritization techniques have been proposed and evaluated in the literature [11; 17; 18; 24; 26]. In 2012, Singh et al. [21] conducted

a systematic review and found 106 prioritization techniques evaluated in 65 papers by that year. And, recent studies show that this number continues to grow. In the work of Mukherjee et al. [14], a survey about different TCP approaches, they reviewed 90 scholarly articles ranging from 2001 to 2018.

Even with all these techniques available in the literature, there are still teams of developers who find their own ways of going through regression costs. An example of this is the system that motivated our study. There, developers began to manually select test cases that would validate their changes to avoid waiting for the entire test suite to run. When all tests were required to run, it took an average of 28 minutes to execute[1]. Waiting for this time in a Continuous Integration (CI) environment, considering scale issues, is unsustainable.

As further discussed in Chapter 6, there are only a few industrial case studies involving real faults and evaluating TCP techniques that use coverage information in a system with CI environment. In this work, we conducted a case study on a large evolving system to evaluate and compare the effectiveness of six prioritization techniques, which use coverage information, with what happens in practice during the software development of this system. In our study, we used real faults inserted accidentally by the developers during maintenance.

In summary, the main contributions of this work are as follows:

- A case study on a real evolving system, used by the Brazilian Federal Police, that has approximately 260 KLoC in Java and 4,000 test cases in its regression test suite. The case study used information collected about maintenance changes made by 20 developers.

- An empirical study that compared and evaluated six prioritization techniques that use coverage information using 43 versions of maintenance changes with real faults.

- A discussion about the prioritization tradeoffs and the coverage processing problem in CI environments.

- A website with the implementations in Java of all strategies used in this work and the scripts used during the experiment to help to extract coverage information per test using Jacoco.

[1]Time calculated from the test executions made during the experiment.

The results show that, in the scenario where it is necessary to perform all tests, the modification-based techniques, Echelon and Echelon Time [23], are the most effective techniques for fault detection and reduction of the spread rate of failed tests among the evaluated TCP techniques. When we compared them, considering the cost-benefit of implementation, the result is still the same. Also, all TCP techniques presented better results than the original order used in the CI environment. We believe this is an incentive to research solutions that allow collecting coverage information in a CI environment, where test suites continuously arrive as developers submit commits, and where the time available to perform this analysis is limited [7; 10].

Another significant result is that, although the TCP techniques had better results than the original order, they could not be more effective than the developer selection in the scenario where the time constraints to perform tests are higher. They usually select the same number of failed tests or less using the same time available to execution. Developer selection is maybe more effective because he almost always includes changed/added tests, which are the tests that usually fails, whereas the techniques do not prioritize these tests. And even these tests are not considered part of the regression tests by definition, they will still be run by the developers along with the rest of the tests. Therefore, if they are part of the executed suit, they should always be considered when proposing a new prioritization technique.

We organize the rest of this work as follows. Chapter 2 presents the key concepts needed for our research and describes the techniques and metrics used in the experiment. Chapter 6 describes related work, followed by Chapter 3 that presents the research questions and the case study performed. Chapter 4 and 5 shows a summary of our experimental findings and a discussion about them, respectively. And finally, Chapter 7 presents our conclusions.

Chapter 2

Background

This chapter defines the main concepts needed for our research. It also describes the techniques and metrics used in our study.

2.1 Test case prioritization

The test case prioritization aims to reorganize a test suite to improve the achievement of specific testing goals (e.g., a faster rate of fault detection) [2]. In other words, it orders a test suite, so that higher-priority test cases run earlier than lower-priority ones. Rothermel et al. [17] have formally defined the TCP problem as follows:

Given: **T**, a test suite; PT, the set of permutations of T; and **f**, a function from PT to the real numbers.

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

According to this definition, *PT* represents the set of all possible orders of *T*, whereas *f* is a function that returns the best possible value, given some criterion, when some order is applied. For example, if a team wants to increase the fault detection rate of their test suite, the function *f* returns the fault detection rate of each ordered suite. The higher the value of this rate, the better the fault detection.

In the literature, there are several approaches designed to solve the prioritization problem. Singh et al. [21] classified the 106 techniques founded in their work into eight categories: coverage-based approaches, modification, faults, requirements, history, genetics, composite approaches (when using more than one approach at a time) and others (approaches used by

only one technique in the literature). However, in practice, the most commonly explored category is the coverage-based, follow by the requirement-based in second place [14].

We choose to apply only TCP techniques in our experiment because they can be applied in the two test environments of the system chosen for our case study. If we tried to use a selection technique in the scenario in which there is no need to execute the whole suite, it would be necessary to introduce two different techniques in the process (one for each environment). With prioritization techniques, we were able to interfere in both scenarios and obtain results changing as little as possible of the case study process.

2.1.1 Studied techniques

Most existing test prioritization techniques guide their prioritization process based on coverage information, which refers to whether any structural unit is covered by a test [11]. In our study, we choose to use only this type of techniques because of their simplicity and overall satisfactory results [2], and also because they do not need previous information about the system or the test suite to implement.

We adopted six different strategies for prioritizing test suites: Total-stmt, Add-stmt, Genetic, ARTMaxMin, Echelon, and Echelon Time. In addition to these techniques, we use as a baseline the current strategy used in the system to order the test cases. The presentation of the strategies used in the study follows:

- **Total-stmt**: Total Statement Coverage (Total-stmt) technique is a greedy algorithm. It works on the principle that the element with the maximum weight is taken first, followed by the element with the second-highest weight, and so on until complete the solution [9]. In the case of the Total-stmt technique, the weight is the total number of statements covered by the test cases, meaning that it schedules test cases according to their statements covered.

- **Add-stmt**: Additional Statement Coverage (Add-stmt) works similar to Total. However, it uses feedback about coverage so far obtained by selected tests to focus on statements not yet covered.

- **Genetic**: This strategy is an application of search-based software engineering. Typically, this type of test prioritization takes all the permutations as candidate solutions

and utilizes some heuristics to guide the searching process [11]. The fitness function of this strategy is based on the coverage of tests to find the optimal solution [9].

- **ARTMaxMin**: In 2009, Jiang et al. [8] proposed a family of test case prioritization techniques based on the concept of Adaptive Random Testing (ART)[1]. ARTMaxMin was the technique with the best results from the family. It first selects a test case with the highest coverage. And then, the remaining tests are reordered based on a function *f* that finds the longest minimum distance of a test compared to the previously selected ones [25].

- **Echelon**: This technique, proposed by Srivastava et al. [23], works similar to Total and Add-stmt in regards to instruction prioritization. However, it uses coverage information about the modified code. It first identifies the changed blocks between two versions and then schedules the test cases according to their total number of statements covered inside the changed block [3].

- **Echelon Time**: It is an extension of the Echelon technique that differs by considering the execution time of the tests in the prioritization. If two tests provide the same coverage of impacted blocks, the one with the shortest time should be selected; the rest of the prioritizing algorithm remains unchanged [23].

- **Original**: It is the original sequence of test cases provided by the developers. It uses the file-system order, the default configuration to order the tests in Maven[2]. We consider this strategy as the control treatment[3].

Total and Add-stmt techniques were proposed by Rothermel et al. in 1999 and they are some of the most commonly used techniques in TCP works [2]. We obtained their implementations and the implementation of Genetic and ARTMaxMin techniques from Lu et al.'s work [11]. The implementation of Echelon and its extension, Echelon Time, were developed

[1] A concept proposed to replace random testing for test case generation. The basic idea of ART is to spread the distribution of test cases as evenly as possible across the input domain [8].

[2] https://maven.apache.org/

[3] We do not use an optimal order as a baseline because our main objective is to compare the techniques with what occurs in development in practice.

by us using the algorithm present in Srivastava et al.'s work [23]. We implemented them using the same language as all other implementation techniques, Java.

In this work, we choose to apply the techniques using only the granularity of the statement due to the dependence with the *Jacoco* framework. As we wanted to modify as little as possible to integrate the techniques in the system, we decided to continue using the *Jacoco* framework, since the system already used it to obtain coverage information. However, it was challenging to get this information per test case. As some works [11] show that different coverage criteria present similar results, we applied efforts only to obtain coverage by the statement.

2.1.2 Metrics

To evaluate the TCP techniques and to help us to answer our research questions, we focus on two metrics: APFD and F-Spreading.

2.1.2.1 The Average Percentage of Faults Detected (APFD)

The APFD is the most dominant metric for addressing prioritization [8; 14; 18; 25]. It calculates the weighted average percentage of faults detected over the life of a test suite [16]. In other words, it measures how fast a test suite detects faults. Its values range from 0 to 100, the higher the values the faster the fault detection rate.

The APFD metric was formally defined in the work of Elbaum et al. [6] as: Consider *T* as a test suite with *n* test cases, *F* as a set of *m* faults revealed by *T* and TF_i as the position of the first test case in the reordered *T'* suite that reveals the fault *i*. The value of the APFD of *T'* is given by the Equation 2.1

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \tag{2.1}$$

Example

To illustrate this metric consider any program that contains a suite with five test cases, from A to E, and also contains eight faults detected by these test cases. Table 2.1 shows this relationship.

Tests/Faults	1	2	3	4	5	6	7	8
A	x			x				
B	x			x	x		x	
C	x	x	x	x	x	x	x	
D					x			
E			x				x	x

Table 2.1: Test suite and list of faults exposed by the same.

Assuming that a first execution order T_1 for these test cases is **A-B-C-D-E**. After executing the test case **A**, two of eight faults are detected. Thus, 25% of the faults have been reveled after 0.2 of the test suite has been used. After executing the test case **B**, two more faults are detected. Thus, 50% of the faults have been detected after 0.4 of the test suite has been used. This relationship of the percentage of detected faults versus the percentage of tests run can be followed in Figure 2.1.

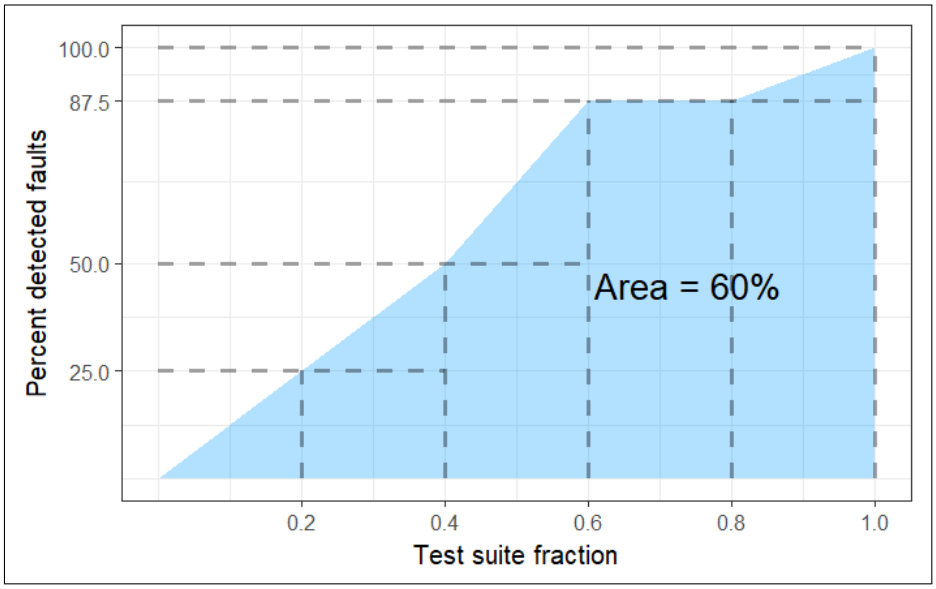

Figure 2.1: Graph representing the APFD calculation of T_1.

Now suppose a second order T_2 for the test cases being **C-E-A-B-D**. After executing the test case **C**, seven faults will are detected, which represents 87.5% of total faults. Because of that, in this second scenario, the detection curve will rise faster than the first scenario, as

shown in Figure 2.2.

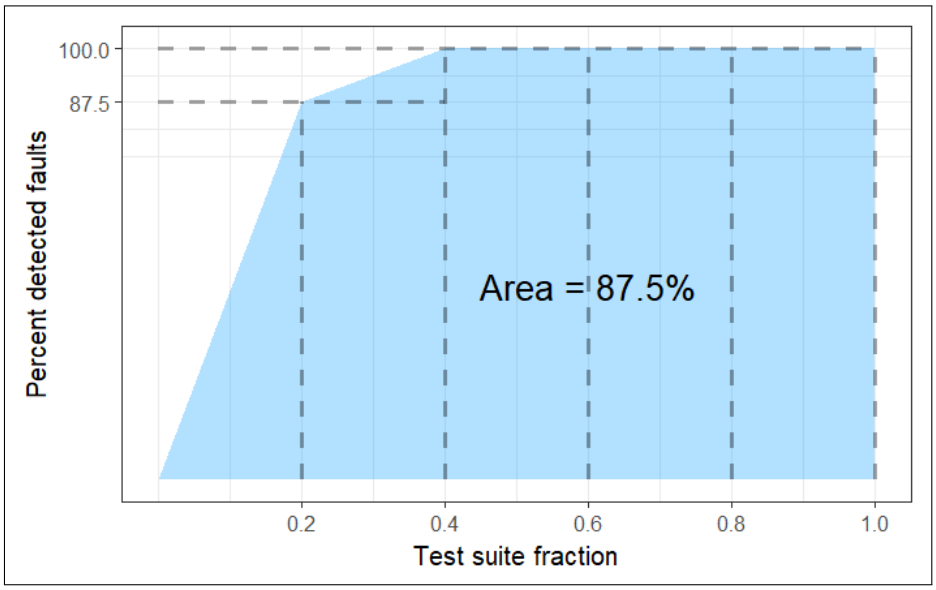

Figure 2.2: Graph representing the APFD calculation of T_2.

Finally, by comparing the APFD values of the two scenarios, we can conclude that T_2 is faster at fault detection than T_1 with 87.5% and 60%, respectively.

2.1.2.2 F-spreading

The F-spreading, proposed by Alves et al. [2], is a rate that measures how the failing test cases are spread in a prioritized test suite. According to the authors, when failing test cases are close, it can help a tester to find and locate the fault. The F-spreading values range from 0 to 1, the higher the value, the more spread the behavioral revealing test cases are. Thus, a good prioritization should generate prioritized suites with low F-spreading.

The Equation 2.2 formalizes the metric, where n is the number of test cases of the test suite; m is the number of failing test cases; TF is a sequence containing the positions of failing test cases; and TF_i is the position of the i^{th} failing test case in the prioritized suite.

$$F\text{-}Spreading = (\sum_{i=2}^{m} TF_i - TF_{i-1}) * \frac{1}{n}, \tag{2.2}$$

Example

For instance, suppose a program that contains a suite T with 100 test cases, from which five fail due to one fault. Assuming that there are two execution orders T1, and T2 for this

suite and the failing test cases appear in the following positions T1:{1, 20, 50, 85, 88}, and T2:{5, 10, 11, 15, 40}. The F-spreading values for T1 and T2 are 0.87 and 0.35, respectively. So, using T2 is better to help a tester find and locate the fault.

2.2 Continuous integration

According to Fowler [1], continuous integration is a software development practice where members of a team integrate their work frequently. Each integration is verified by an automated build that uses the regression test suite to detect integration errors as quickly as possible. Figure 2.3 shows an example of the CI process. When a developer sends her code changes to the main repository, the CI server identifies this new commit and starts a new job. It builds the code, regression tests are executed and statistical analysis are extracted. In the end, everything is reported to the manager and developers team.

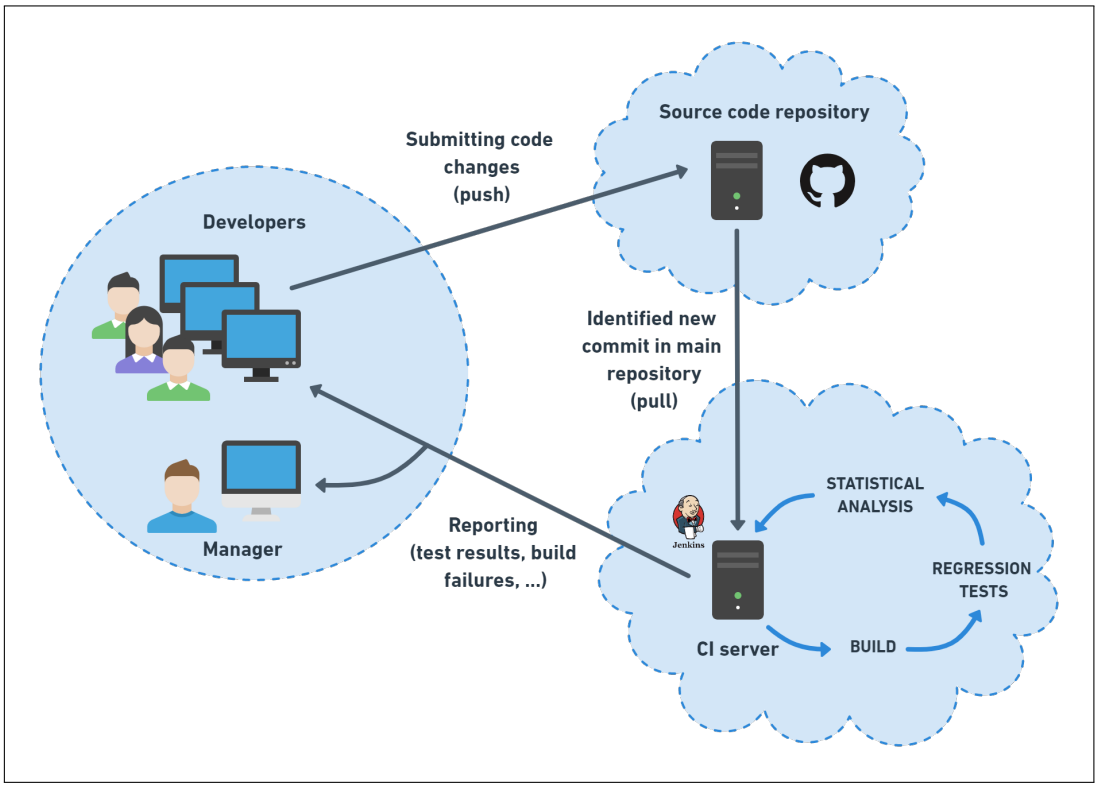

Figure 2.3: Example of the CI process.

Companies like Google, Amazon, Facebook, and Microsoft have adopted CI and its ability to better match the speed and scale of their development efforts [10]. To be able to put

this process into practice, tools such as versioning and build generation and automation are essential. The first is important because it supports development in a number of ways, such as keeping track of software developments, allowing developers to work in parallel on the same file, and enabling rapid and effective code reversal. CI servers work by generating automatic builds, doing static code analysis, and performing regression testing.

Chapter 3

Study Methodology

This chapter describes the questions that guided our research, the study and experiment performed.

3.1 Study definition

The purpose of this study is to **compare the effectiveness of the TCP techniques** with a focus on their possible use in the development process, from the point of view of the software developer in the context of a large web system in the evolution phase.

3.1.1 Research questions

In this work, we are interested in the following research questions:

RQ1: Do the chosen TCP techniques improve the system test suite fault detection rate?

RQ2: Are the chosen TCP techniques equally efficient according to the APFD metric?

RQ3: Do the chosen TCP techniques decrease the spread of failed test cases in the system test suite?

RQ4: Are the chosen TCP techniques equally efficient according to the F-Spreading metric?

From *RQ1* to *RQ4*, we want to investigate the effectiveness of prioritization techniques when applied to a large evolving web system. The focus of those is to **compare the performance of the prioritized tests with the order of the tests used in practice during the**

system CI process.

RQ5: Was the developer able to select all tests that detected the fault?

RQ6: Are time-constrained prioritization techniques more efficient than manual selection in the development environment?

In *RQ5* and *RQ6*, we want to analyze the developers' test selection used to validate their changes in the development environment. These RQs' focus is to see **if developers can include in their selection all failed tests** from the regression suite and compare the **effectiveness of that selection with a selection made from a prioritized list**. Our goal with the last question is to see if it is worth replacing developer selection for a time-constrained selection based on a prioritized list obtained from a TCP technique.

Prioritized list selection will work by taking the tests from the top positions while the sum of their time execution does not exceed the total time execution of the developer selection.

3.2 Study context

For the study, we use a web system called ePol. Its objective is to computerize Brazilian Federal Police inquiries, allowing a fast record, access, and maintenance of the data inserted in the system. The policemen are already using ePol since 2016, and we estimate that until the end of 2020, the number of users will be up to 11,000.

The ePol back-end has approximately 260 Java KLoC and over 4,000 test cases in its regression test suite. It's developers team had 20 engineers when the experiment started in February 2018 and kept with this average until March of 2019.

To test the ePol back-end, the developers used the TestNG[1] and Arquillian[2] frameworks. And to collect the coverage information, they used the Jacoco[3] framework. This is important to mention because there is no tool to quickly extract coverage information per test with Jacoco. Because of this, we had to create a script using the Jacoco source code to extract the statement coverage information the way the techniques needed it. This script can be seen on our website[4].

[1] https://testng.org/doc/

[2] http://arquillian.org/

[3] https://www.eclemma.org/jacoco/

[4] https://sites.google.com/view/isabellycavalcante-research/

Since the implantation, the ePol has undergone many evolutionary and adaptive changes. We observed that when developers maintained software, they tended to make code modifications in small parts. As a result, they often needed to run regression tests to validate each part. However, running the entire regression test suite took an average of 28 minutes. Waiting even for a few minutes in an agile development environment like this can make it impossible to perform all regression tests whenever the developer needs it. As a result of the lack of time to run the entire test suite during changes, the developers began manually choosing which tests to execute to validate their changes. They only run all regression tests at the code review stage as it was mandatory.

3.2.1 The changes and their workflow

The changes made in the ePol system include functionality addition, bug fixes, and refactoring. We can classify a change into one of these types through descriptions added by the developers in the system version file after completing the change.

The development flow of these changes is very similar, as shown in Figure 3.1. For example, to correct a bug, the developer first makes a copy of the system code (or, in Git terminology, clone the repository) in her machine (1). After that, she begins the analysis phase (2), where she identifies the causes of the bug and verifies the impact of the fix on the system. Then, she begins to change the source code (3).

During this step, she may choose to select and run some tests to ensure that the rest of the system has not changed its behavior (4). When the bug fix is finished, the modified code is submitted for review (5). At this stage, if the reviewer encounters code issues that need to be resolved, the developer will return to step 3 to correct them. At the end of the review phase, the developer or the reviewer must run the entire regression test suite at least once (6). If one or more test fails, the developer returns to step 3 to find and correct the problem. However, if all tests pass, and the reviewer gives his permission to submit the code (7), it will be sent to the main repository (8).

It is not mandatory for the developer to always follow this sequence. She may decide to analyze the impact of the change (step 2) before copying the code to her machine (step 1), or she may decide not to run tests in the development phase (step 4). However, steps 3, 5 and 6 must always happen in all scenarios because if a change was made, it must be reviewed and

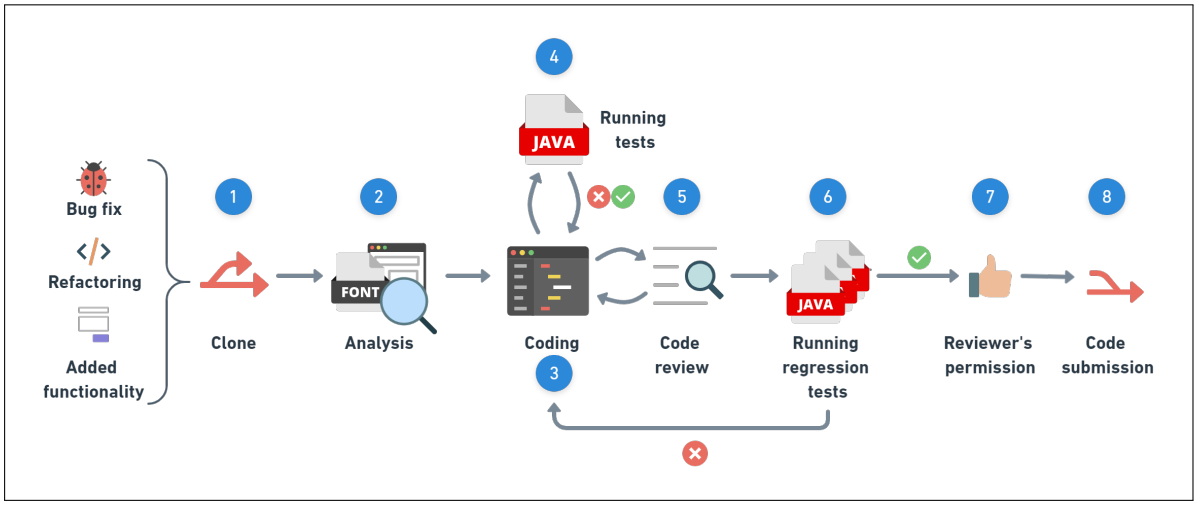

Figure 3.1: System change development flow.

submitted to the regression tests suite at least once before being sent.

3.3 Experimental Procedure

The experiment execution involved the following steps:

- Data collection about test executions;
- Extraction of the information necessary to run the techniques, such as statement coverage and execution time;
- Execution of the prioritization techniques;

However, before describing the experiment, we define in the next section, the experimental units used.

3.3.1 Experimental units

The experimental unit of this study consists of one or more test suites that, when executed, detected a real system fault, which means that our experimental unit is composed of the test suit, the modified code, and the bugs present in the code.

In this work, we consider as fault one or more changes in the code that cause some tests to fail. For example, assuming a developer changed two files, and after running the test suite,

he notices five tests failed. At this point, the developer begins to investigate and fix one by one of the failed tests. He does this until all the tests pass again. For us, the fault is all what made the five tests fail. It does not matter if the fault was caused by a problem on one line in one file or multiple lines in different files. That way, our experimental unit always has just one fault.

3.3.2 Instrumentation and data collection

The first step of our experiment was to collect data about system test runs. To do this, we had to understand the ePol testing phase.

3.3.2.1 The test environment and collection instrumentation

In the ePol development environment, the developer had two options when she wanted to test her changes. She could run the tests through the Maven test command [5] or run a script created by previously developers. This alternative test script is designed to give the developer the option to run test profiles (unit, integration, and system), and try to shorten the test execution time a little. In ePol, the Maven test phase was set up to always inspect the code, after running the tests, to look for code smells. This setting increased the total duration of the Maven test command because the alternate script ran Maven through a particular configuration (Surefire[6]) that eliminated this analysis phase, thereby decreasing the running time by a few minutes.

We created a script to collect data automatically. Whenever someone ran a test, our script captured information about that execution, to allow us to reproduce that run later. For our script to run through the Maven testing phase, we have changed the POM[7]. That way, whenever it finished the testing phase, it would call our script. However, this did not work when the developer ran the alternate test script, so we also had to make a change to it to call our script after the tests ran.

In our study, we choose to use previous executions to perform the techniques. This

[5]The mvn test command is a phase of the Maven build lifecycle (https://maven.apache.org/ref/3.5.3/maven-core/lifecycles.html) that performs all system tests.

[6]https://maven.apache.org/surefire/maven-surefire-plugin/index.html

[7]Project Object Model - https://maven.apache.org/pom.html

means that it was only after capturing some test executions that we begin our experiment. No interaction was made with the developer during her activity.

3.3.2.2 Data collection

The principal information captured by our script was all code modified from the last synchronized index (last local or repository commit) to that moment, the list of the tests executed, the total duration, and the result of the execution. The script also captured which code branch the developer was working on. We did this to help group information from the same change.

As shown in Figure 3.2, our script was run at two different points in the development process. The first point occurred within the development environment when the developer who was coding some modifications (1) decided to run some tests to verify her changes (2). Immediately after the tests were run, our script was called regardless of the result (3), and the data were collected (4). Only when this collection ended that also ended the test run. The second point occurred within the continuous integration environment when the developer finished her changes and submitted the code to review (5). At this point, she or the reviewer needed to run the entire system regression test suite to validate the changes before submitting it to the main repository (6). In this case, our script also was called immediately after the tests ran, regardless of the result (7), and the data were collected (8).

In Figure 3.3, we can see how the data flow worked after the script was called. Each time a test execution was completed and called our script (1), it performed the following steps: created a file with the collected data (2), compressed this file (3) and sent it to a remote application (4). Our role was to regularly download new data from this application (5), extract a summary of the information, and add it to a spreadsheet (6) so that at the end of the collection phase, we can analyze this data quickly.

3.3.2.3 Filtering data collection

A summary of the filtering collected data be seen in Table 3.1. As showed, after nearly a year of collecting the data (02/22/18 to 12/18/18), we began the analysis and selection phase of the experimental units. During this time, we collected approximately 6800 test executions.

Of these executions, we disregarded 436, because they were collected during the collection script creation and adjustment period (02/22/18 and 03/16/18). We also removed all

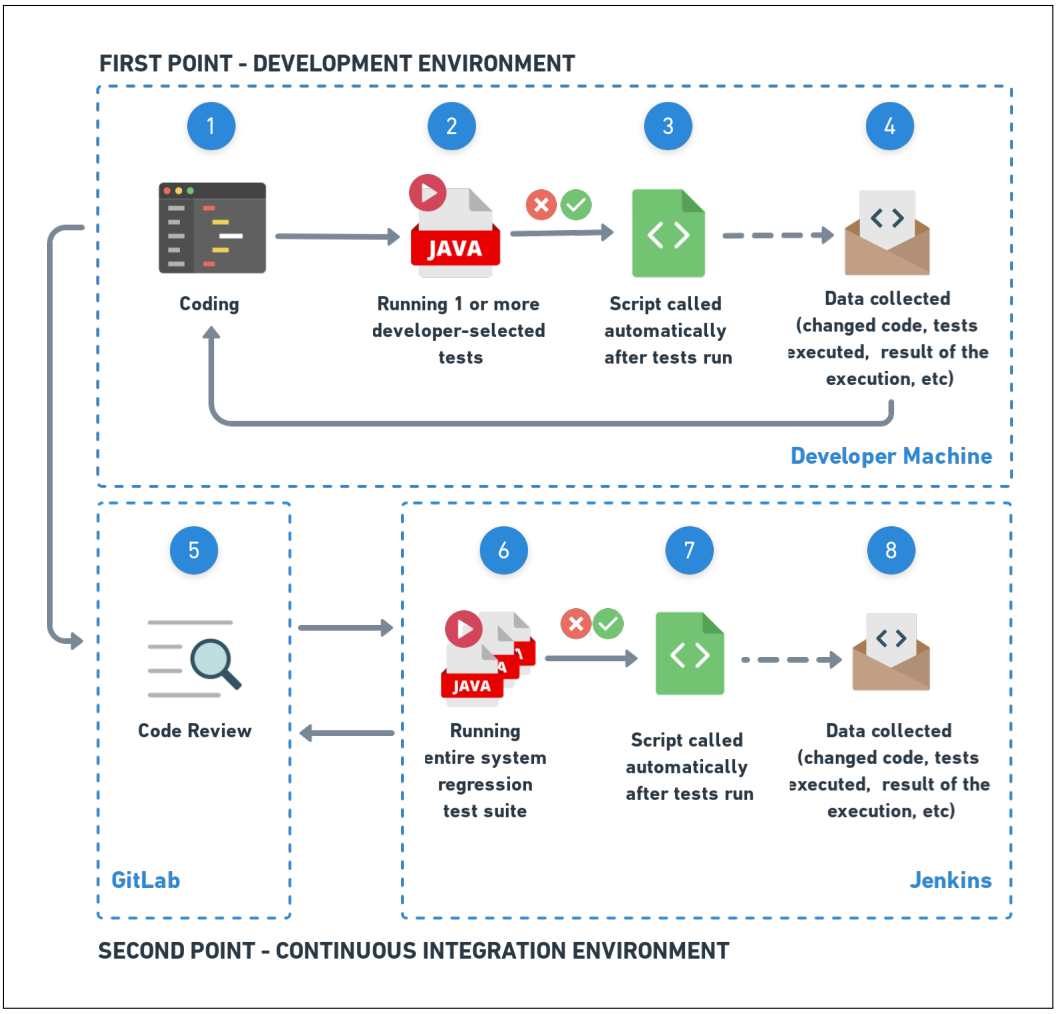

Figure 3.2: Scenarios where the data collection script occurred.

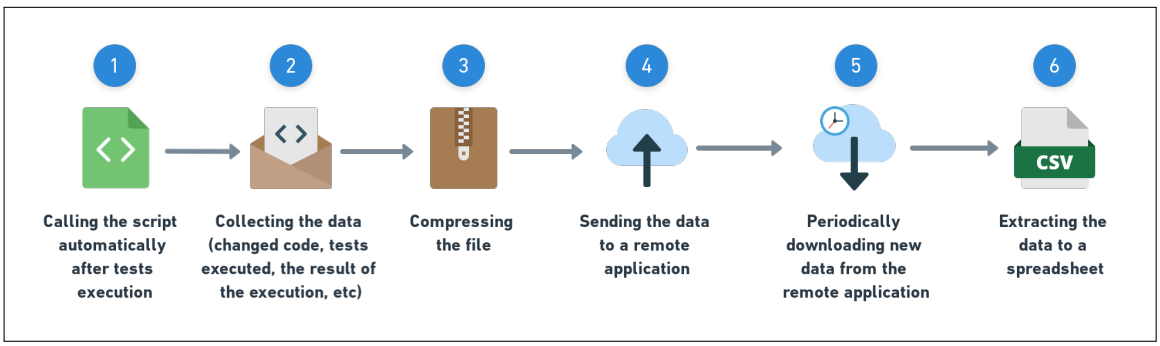

Figure 3.3: Data flow after script call.

executions collected after 10/30/18 (519 executions) because it was when the ePol started migrating its continuous integration system from Jenkins to GitLab.

Then, we grouped the executions by branches, totaling 316 branches. We decided to exclude branches that did not refer to system changes, such as branches created for the training of new developers, merge branches, version branches, etc. The list with the names of the 65 branches removed is available in our website[8].

Finally, we removed 43 executions because they came with empty files. For some reason, our script was called under circumstances that it could not capture any data, neither who was the user, the current branch, the tests executed. Consequently, the file was empty, so we decided to remove them.

At the end, we had 5263 executions (251 branches) as possible experimental units. As mentioned earlier, our experimental unit is a test suite execution that detects a system fault. So, we created a script that looked for branches that had at least one failed test execution and, the cause of this failed execution could not be a configuration problem (any service unavailable or .war with deployment issue). The script returned 164 branches out of 251, so we ignored the 87 non-returned branches.

3.3.3 Applying prioritization techniques

With the remaining 164 branches, we move on to the TCP techniques execution phase. Here, the first step was to extract the changed code and coverage information from each experimental unit.

3.3.3.1 Extracting the information necessary to run the techniques

We start by analyzing the execution history of each branch. Each one had an average of 30 runs. Our goal was to identify failed executions and gather information about them.

We analyze each branch's executions one by one, looking for what the developer did differently from one execution to another. Whenever we encountered a failed run, we identified the type of change that was being made (bug fix, functionality added or refactoring), its ID (each run collected had an ID that identified it), the code that corrected the fault, the Merge

[8]https://sites.google.com/view/isabellycavalcante-research/

Amount	Description	How was the removal done?
6845 executions	Total data collected between 02/22/18 and 12/18/18.	-
436 executions	Data collected during the collection script creation and adjustment period.	The removal was done manually filtering the executions that occurred between 02/22/18 and 03/16/18.
519 executions	Data collected during the continuous integration system migration period.	The removal was done manually filtering the executions that occurred after 10/30/18.
65 branches	Branches that did not refer to system changes.	The removal was done manually analyzing the branches' name and then executing a script to remove all executions that belonged to any of these branches that would be removed.
45 executions	Executions that came with empty files.	The removal was done manually filtering the executions that came with empty files.
87 branches	Branches that have not failed executions or branches that have executions that failed because of configuration problems.	The removal was done by running a script that first grouped the executions of each branch. And then, this script would try to find at least one failed execution in each branch. If it did not find, it would remove all executions linked to this branch.
4943 executions (164 branches)	The final total of possible experimental units.	-

Table 3.1: Summary of the discarded tests runs.

Request (MR) of the change and the code changes that impacted in the execution.

We extracted this last item from the code that came in the file initially collected and from the commits made before execution. As mentioned before, the collected code was a result of the changes made between the last commit until the moment that ran the tests. Therefore, it was necessary to check for commits before execution. If there are any commits, its changes should be included.

We analyze the execution history of 48 of the 164 branches. Even with a script, this process takes a long time because even though it compares and shows executions with the help of a tool, we still need to look at executions one by one to understand the developer's thinking.

Almost all the 48 branches were maintenance about bugs in the system. And this was because the 164 branches were ordered alphabetically by their names, and we selected the first 48 branches. After the history analyses, we excluded 16 of the 48 branches because we saw that their failed executions occurred in particular scenarios. For example, they were branches with discontinued changes, that is, that the developers did not complete them, or were branches with executions that failed due to intermittent faults. Our website[9] has the list of the 16 branches excluded by specific scenarios and their reasons.

With the remaining 32 branches, we were able to generate 43 experimental units because some of these branches had more than one execution that detected system faults. The average of failed tests from the 43 test suite executions used in the experiment was approximately 14 tests. This number was because two units changed a large entity of the system in their maintenance, and as a result, many tests failed after inserting a fault. Without them, the number of failed tests decreases to approximately four tests. A list of all experimental units also can be seen on our website.

Each one of these experimental units went through 2 steps to extract its data coverage, as shown in Figure 3.4.

Step 1 aimed to run the entire regression test suite of the experimental unit to generate the coverage data. To do this, we tried to replicate the environment that it had previously run. We created a script that put the repository on the same version as the day of execution, added the changes made by the developer, and configured some of the external services to

[9]https://sites.google.com/view/isabellycavalcante-research/

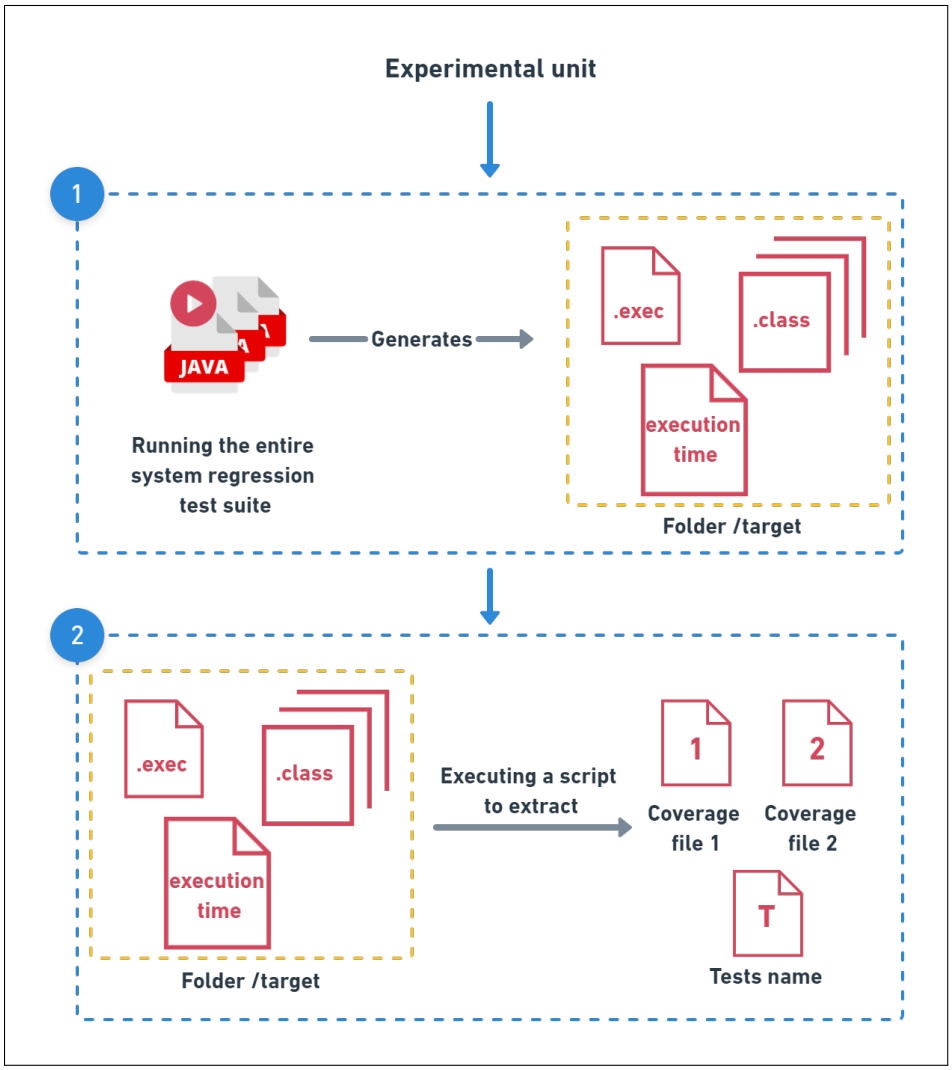

Figure 3.4: Extraction steps of the data coverage information.

use the same version. Then, it ran the entire regression test suite. In the end, we manually checked whether the failed tests were as expected.

Some experimental units have had more tests failing than expected. For example, in the developer execution, three tests failed, and in our execution, five failed. This inconsistency occurs because, as it was the developer who chose which tests to run, she may end up not choosing all the tests that would identify the fault. Also, at this stage, we noticed that executions collected between March and May had a group of tests that always failed in our re-executions. Since we are unable to understand why they fail, we decided to remove them from the test suite before proceeding with the next steps.

Step 2 used the files of the folder */target* generated in the previous step to extract the coverage information. To do this, we had to create a second script that, based on the .exec and .class files, created two files with the coverage information extracted in different formats. The Total-stmt, Add-stmt, ARTMaxMin, and Genetic techniques would use the first, and the Echelon and EchelonTime techniques would use the second. The difference between them was that the second explicitly stated which lines of a class were exercised by each test, while the first did not refer to which lines were, but only whether or not it covered.

With these two coverage files created we passed to the phase of performing the prioritization techniques.

3.3.3.2 Performing the TCP techniques

To automate the execution of all techniques, we created a new script. As showed in Figure 3.5, the script worked by receiving the two coverage files that are used by each technique, the run time file per test, and the code change file. Then, the script executed the algorithm of all techniques and returned a file with the prioritized list of each technique. After that, we could move onto the metric application phase.

All the steps created to perform this experiment, from the re-execution of the test suites to obtaining the prioritized lists, took almost 9 hours to run in just one experimental unit. Because of that, we could not increase the total of the experimental units, staying with a total of 43 test suite executions applied in all the six TCP techniques.

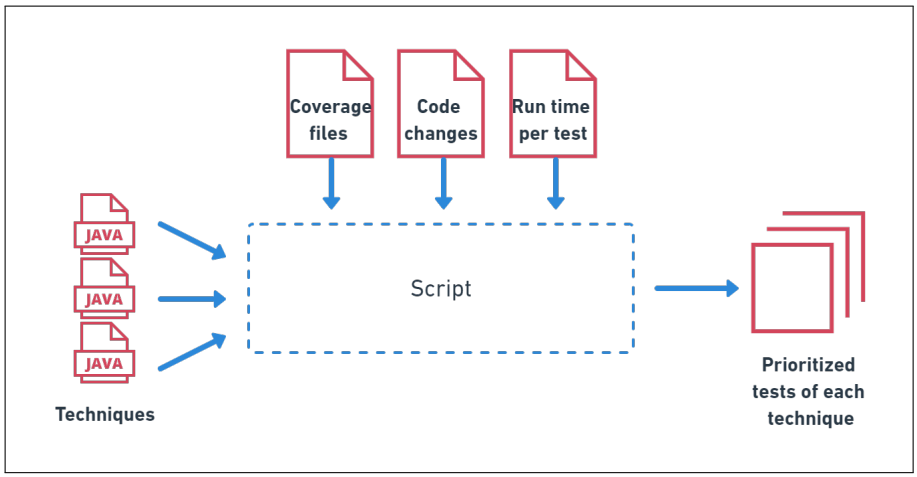

Figure 3.5: Script executing the TCP techniques.

Chapter 4

Results

In this chapter, we present a summary of our experimental findings.

4.1 Answers to research questions 1 to 4

As described in Section 3.3.3, we applied six TCP techniques to 43 experimental units, resulting in a total of 258 prioritized test suites. For each suite, we calculated the APFD and F-Spreading values, which were analyzed and tested to help us discuss answers to research questions 1 to 4, related to the performance of these techniques in the CI process.

In a companion website[1], we provide the calculated values for both metrics, and the scripts used in the RStudio tool[2].

4.1.1 RQ1: Do the chosen TCP techniques improve the system test suite fault detection rate?

Figure 4.1 presents boxplots with the distribution of APFD values per technique. All distribution boxes overlap, which may imply similar APFD distributions for all techniques. Other noteworthy information is that the Echelon and Echelon Time techniques had the highest medians, with values close or equal to 100%, suggesting that they performed better than the

[1] https://sites.google.com/view/isabellycavalcante-research/

[2] A tool that uses a programming language to generate graphs and statistical calculations (https://rstudio.com/)

rest of the techniques. We can also see that the Original order has the worst median compared to the chosen TCP techniques, which may indicate that any one of these techniques, if applied by ePol developers, might have improvements on the system's fault detection rate.

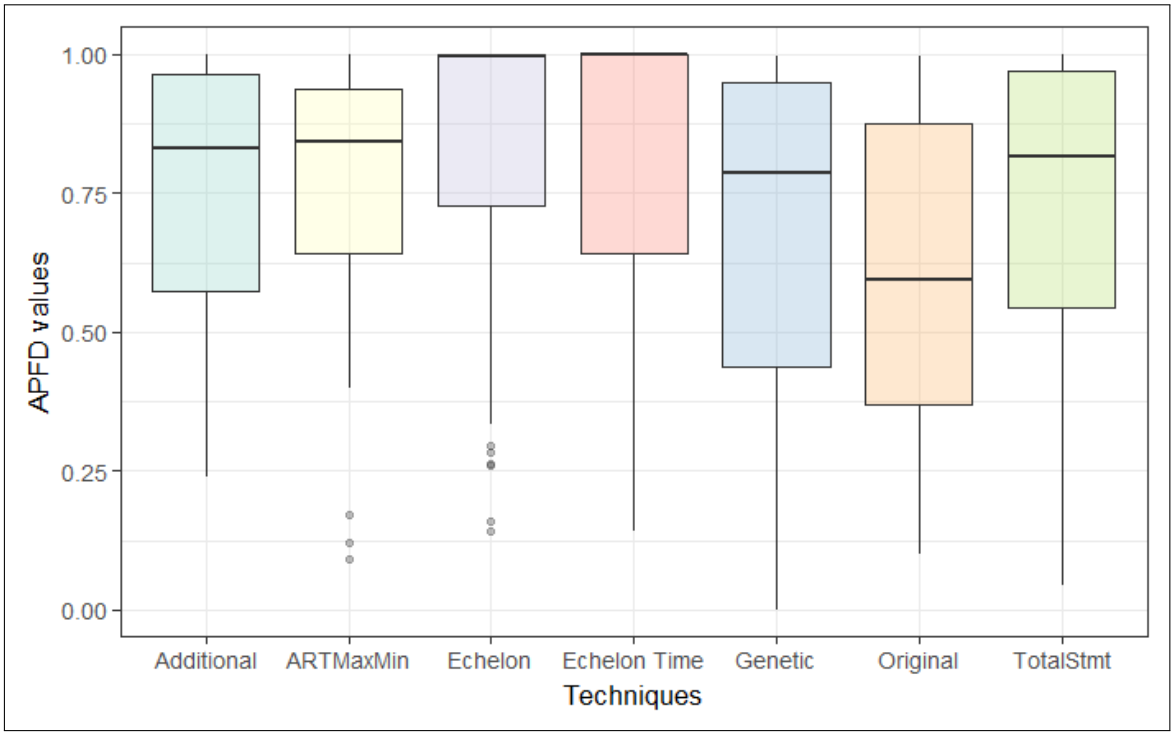

Figure 4.1: Boxplots with the APFD distribution per technique.

We also performed a statistical analysis to compare these distributions. Since our collected data is only a sample of the real world, we used the Bootstrap [20] method to make an inference about the population. Figure 4.2 shows the Confidence Intervals (CI), with 95% confidence, of the median APFD values with 10,000 replications for each strategy. As we can see, Echelon and its extension, Echelon Time, have the higher ranges, which indicate that they may be the techniques with the highest fault detection rate of the experiment. The CIs also reinforce the idea that all techniques might improve the suite's fault detection rate since some of their range is above the Original range.

To have a statistical understanding of the techniques behavior, we first performed the *Shapiro-Wilk* normality test. It verifies if the population of each technique is normally distributed [19]. The significance level used in our analysis is 0.05, which is a value commonly used for these tests. Table 4.1 shows the calculated p-value for each technique. According

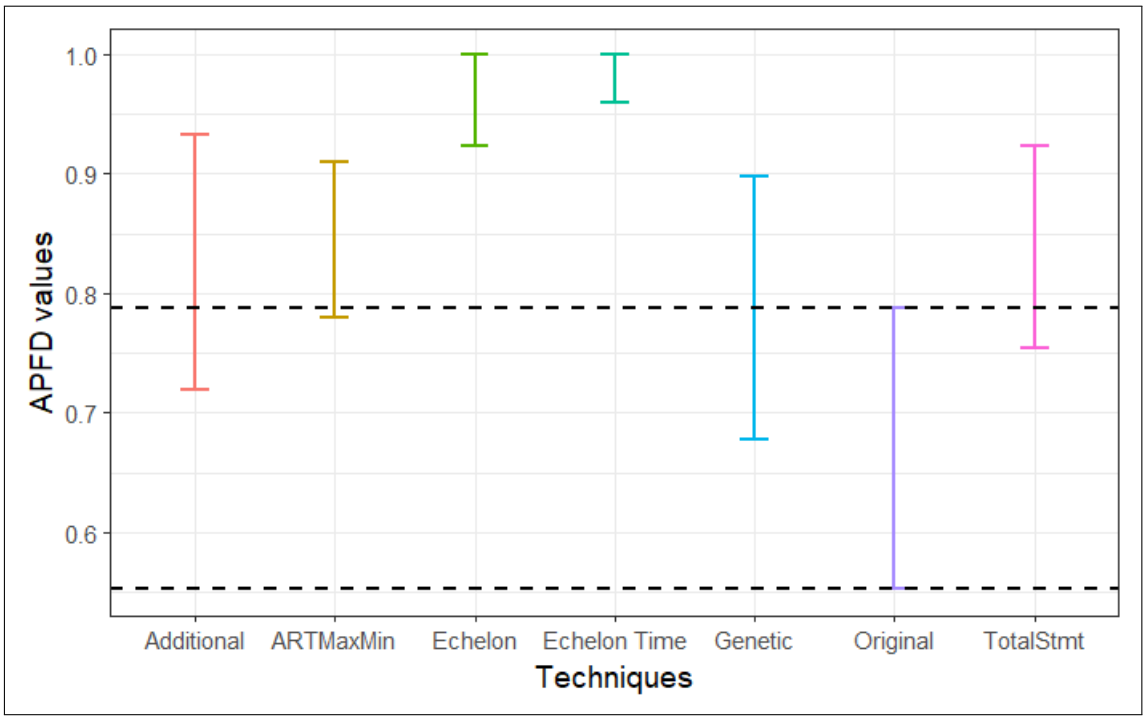

Figure 4.2: Confidence interval for the APFD median values of each technique.

to the test, all techniques follow a non-normal distribution. Thus, for testing distribution differences, we need a non-parametric test.

Technique	**p-value**
Total-stmt	1.828e-05
Add-stmt	0.0002250
Echelon	8.023e-09
EchelonTime	6.248e-09
ARTMaxMin	1.272e-05
Genetic	0.0001572
Original	0.0118455

Table 4.1: Shapiro-Wilk Normality Test for APFD metric.

To analyze whether techniques have similar distribution functions, we performed the *Kruskal-Wallis* test. It verifies if the medians of all groups are equal. We obtained as p-value **1.137e-07**, which means that at least one technique has a population with different behavior

from at least one other technique.

To try to identify the pairs that have a different distribution, we performed the *Conover* test with Bonferroni adjustment, a pairwise test for multiple comparisons that tests if two groups have the same distribution [4]. As we can see in Table 4.2, the results show that only the pairs involving the Echelon and Echelon Time, detached in gray cells, presented a population difference. For the rest of the techniques, this test cannot tell if they are different.

	Add	**ART**	**Ech**	**EchTime**	**Genetic**	**Original**	**Total**
Add	-	-	-	-	-	-	-
ART	1.00000	-	-	-	-	-	-
Ech	0.04151	0.02002	-	-	-	-	-
EchTime	0.03400	0.01622	1.00000	-	-	-	-
Genetic	1.00000	1.00000	0.00066	0.00051	-	-	-
Original	0.76219	1.00000	6.9e-06	5.1e-06	1.00000	-	-
Total	1.00000	1.00000	0.01821	0.01474	1.00000	1.00000	-

Table 4.2: Conover test (p-value) for APFD metric. In gray cells, the pairs with population difference.

To complement the results of the Conover test, we calculated the effect size of each technique over the others. It gives the magnitude of the difference between two groups of observations [12]. As previously tested, our data follows a non-parametric distribution, so we choose to use the *Cliff's Delta* measure. The Table 4.3 shows the calculated values for each pair, and the interpretations [12] for theses values follow the Equation 4.1.

$$\delta_{ij} = \begin{cases} +1 \rightarrow Group1_i > Group2_j, & \forall_i, \forall_j \\ -1 \rightarrow Group1_i < Group2_j, & \forall_i, \forall_j \\ 0 \rightarrow Group1_i = Group2_j, & \forall_i, \forall_j \end{cases} \tag{4.1}$$

The results show that **all the six techniques can improve the suite's fault detection rate significantly when compared to the Original system order**. The Echelon and Echelon Time techniques had the most significant effect with 0.526 and 0.514, respectively, followed by the Additional, ARTMaxMin, and Total techniques, with a minor improvement. The

		1st Group						
		Add	**ART**	**Ech**	**EchTime**	**Genetic**	**Original**	**Total**
2nd	**Add**	-	-	-	-	-	-	-
G	**ART**	0.014	-	-	-	-	-	-
r	**Ech**	-0.394	-0.438	-	-	-	-	-
o	**EchTime**	-0.392	-0.440	-0.028	-	-	-	-
u	**Genetic**	0.138	0.112	0.490	0.490	-	-	-
p	**Original**	0.300	0.290	0.526	0.514	0.138	-	-
	Total	0.026	0.006	0.422	0.424	-0.118	-0.254	-

Table 4.3: Cliff Delta measurement result for APFD metric. In gray cells, the pairs with the largest effects.

Genetic technique had the smallest improvement but still managed to be better than the Original. These results confirm the findings of visual analysis.

4.1.2 RQ2: Are the chosen TCP techniques equally efficient according to the APFD metric?

To answer this question, we return to the results of the Cliff Delta measure in Table 4.3 to classify the techniques according to their effect size. Starting with the Additional and ARTMaxMin techniques, we can see that Additional outperforms ARTMaxMin, but with a value very close to zero, meaning that there was almost no difference between them. The same occurs with Total vs. ARTMaxMin and Total vs. Additional. Therefore, we consider that all these three techniques are equally efficient.

The Echelon and Echelon Time outperform all other techniques with a significant effect, which means they are the most efficient of the group. When we compare them, we see an effect size close to zero, showing that both are equally efficient. The Genetic technique loses to all the other techniques, wining just of the Original, meaning that it had the worst efficient of the group.

In short, **the chosen TCP techniques are not equally efficient** and we can rank them by their APFD result as:

Echelon = Echelon Time > Additional = Total = ARTMaxMin > Genetic > Original

4.1.3 RQ3: Do the chosen TCP techniques decrease the spread of failed test cases in the system test suite?

Unlike APFD, in the F-Spreading metric, it is desirable that a technique returns a low value. A low value of F-Spreading means that the technique was able to group test cases that revealed the fault. With that in mind, Figure 4.3 shows the boxplots with the distributions of F-Spreading values per technique. Even the Echelon and Echelon Time having a large variation in their results, 50% are close to or below 0.25, indicating a small spread of their failed tests. We can also see the behavior of Total approaching the Original order. However, as its median was lower, it may have a slight improvement in the spreading of tests. On the other techniques, all have medians higher than Original, and their data variations are small, suggesting that all three should have a higher spread than Original.

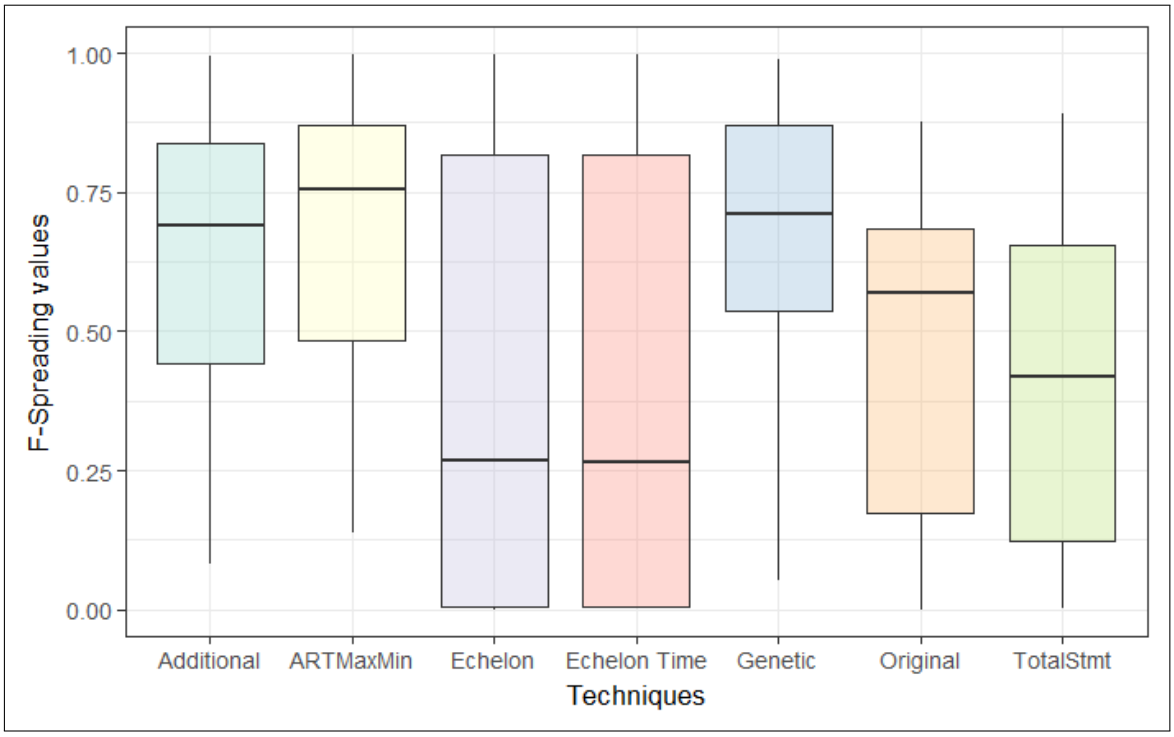

Figure 4.3: Boxplots with the F-Spreading distribution per technique.

We also used the *Bootstrap* method here to make CIs with the F-Spreading values. Figure 4.4 shows the CIs, with 95% confidence, of the median F-Spreading values with 10,000

replications for each strategy. It confirms the observations that we made before in the boxplot chart: high variability of Echelon and Echelon Time values, Total with a behavior close to the Original, and the high values of the other three techniques.

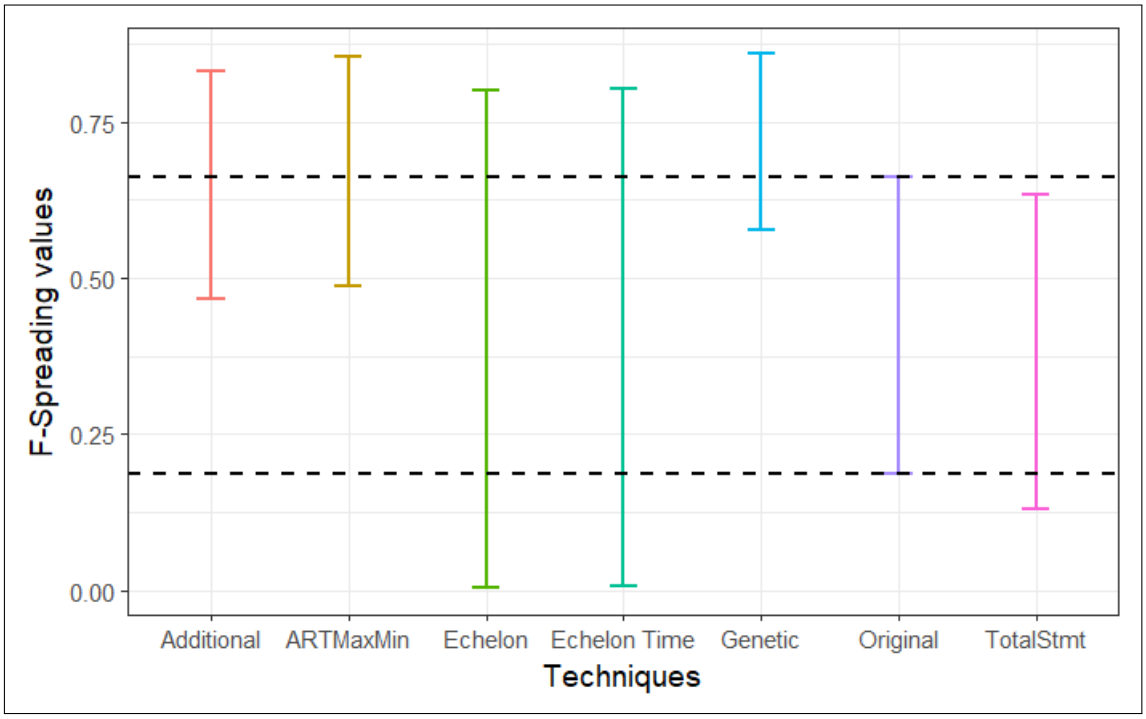

Figure 4.4: Confidence interval for the F-Spreading median values of each technique.

For a statistical understanding of the techniques behavior on this metric, we performed the *Shapiro-Wilk* normality test. Table 4.4 shows that only Echelon, Echelon Time, and Original follow a non-normally distribution. The other four strategies (Additional, Total, ARTMaxMin, and Genetic) follow a normal distribution.

As one part of our techniques showed a normal distribution and the other part a non-normal distribution, we decided to continue the analysis by applying non-parametric tests. Parametric tests were also applied but only to confirm the results found in non-parametric ones. The values returned by the parametric tests can be seen on our website.

The p-value obtained in the *Krukal-Wallis* test was **0.02021**, meaning that at least one technique has a population with different behavior from at least one other technique. Therefore, we pass to the pairwise analysis to try to identify which ones are different. Table 4.5 shows the values calculated for the *Conover* test, and what we can see is all pairs had a p-value higher than 0.05, which means that with this test, we have no significance to affirm

Technique	**p-value**
Total-stmt	0.1575312491
Add-stmt	0.1684358376
Echelon	0.0005090073
EchelonTime	0.0005225778
ARTMaxMin	0.0837448360
Genetic	0.1160940417
Original	0.0350797699

Table 4.4: Shapiro-Wilk Normality test for F-Spreading metric. In gray cells, the techniques that follow a non-normally distribution.

which techniques behave differently.

	Add	**ART**	**Echelon**	**EchelonTime**	**Genetic**	**Original**	**Total**
Add	-	-	-	-	-	-	-
ART	1.00	-	-	-	-	-	-
Echalon	0.90	0.37	-	-	-	-	-
EchalonTime	1.00	0.42	1.00	-	-	-	-
Genetic	1.00	1.00	0.33	0.38	-	-	-
Original	1.00	1.00	1.00	1.00	1.00	-	-
Total	1.00	0.45	1.00	1.00	0.40	1.00	-

Table 4.5: Conover test (p-value) for F-Spreading metric.

Table 4.6 shows the values calculated using the *Cliff's Delta* measure. Again, this measure shows the magnitude of the difference between two groups of observations. This means that if one group is superior to another, their F-Spreading values are higher than the second group. In this scenario, the second group is better than the first, according to F-Spreading, because their values are lower.

Looking at the results of *Cliff's Delta* for F-Spreading, we can see that the differences between the strategies are very small. The Original order was superior to the **Echelon, Echelon Time, and Total** by very little. This means that all these three techniques **can**

have a smaller spread of the failed test cases than the Original order. The other three remaining techniques (**Additional, Genetic, and ARTMaxMin**) were superior to the values of the Original order, meaning that they **cannot have a smaller spread of the failed tests than the Original**.

		1st Group						
		Add	**ART**	**Echelon**	**EchelonTime**	**Genetic**	**Original**	**Total**
2nd	**Add**	-	-	-	-	-	-	-
G	**ART**	-0.076	-	-	-	-	-	-
r	**Echalon**	0.332	0.364	-	-	-	-	-
o	**EchalonTime**	0.336	0.364	-0.030	-	-	-	-
u	**Genetic**	-0.084	-0.006	-0.378	-0.378	-	-	-
p	**Original**	0.284	0.344	-0.138	-0.136	0.406	-	-
	Total	0.390	0.484	-0.100	-0.096	0.474	0.100	-

Table 4.6: Cliff's Delta measure results for F-Spreading metric.

4.1.4 RQ4: Are the chosen TCP techniques equally efficient according to the F-Spreading metric?

To answer this question, we return to the results of the Cliff Delta measure in Table 4.6, in order to classify the techniques according to their effect size. We started by listing the techniques that have the same behavior: Echelon = Echelon Time and Additional = ARTMaxMin = Genetic. This second group shows superiority over the values of the first group in all pairs involving elements of both groups. Thus, the first group obtained a better F-Spreading result. The remaining technique, Total, shows superiority over the values of the Additional, ARTMaxMin, and Genetic, but not over the values of the two Echelons. Therefore, the efficiency ranking for the F-Spreading metrics is: Echelon = Echelon Time > Total > Additional = ARTMaxMin = Genetic, **this show that the chosen TCP techniques are not equally efficient.** The final rank, after adding the results found in the *RQ3*, is:

Echelon = Echelon Time > Total > Original > Additional = ARTMaxMin = Genetic

4.2 Answers to research questions 5 and 6

Until now, we were using the system default order to compare with the TCP techniques order. However, in research questions 5 and 6, we are interested in analyzing the test selection made manually by developers. They use this selection to validate their changes in the development environment, where the time constraint is greater than in the CI environment and where there is no need to run the entire test suite.

To answer these research questions, we used only 29 of the 43 experimental units because they were the ones that contained test executions made in the development environment.

4.2.1 RQ5: Was the developer able to select all tests that detected the fault?

The tests executed on the 29 experimental units were selected by the developers using their knowledge of the changes made. On average, he selected one test class to run. This class contained 50 test cases in maximum, which means that, on average, the developer executed 50 test cases.

We want to know with this question if their selection can include all the tests that detected faults in the code. That is if the developer selection can be effective. For this question, we analyzed the relationship between the tests executed by the developers and tests that detected faults in these 29 units.

Developers selected the tests correctly in 20 of the 29 units, which means that in 69% of the units, they were able to select all tests that detected the fault. In all these cases, there was only one test to be selected. In eight units (28%), they selected only part of the tests, and in only one (3%), they did not select any of the tests that detected the fault.

We analyzed the changed classes of these 20 units that were correctly selected and saw that in 16 of them, the developer had changed or added the selected test. This may indicate that ePol developers tend to choose tests that have recently been changed or added. Of the 43 initial units, 21 had at least one failed test that had been changed or added, confirming the developer's instinct to select these tests.

In summary, **ePol developers can select tests in the scenario where only one test fails**, but in situations in which more than one test fails, they may not perform well.

4.2.2 RQ6: Are time-constrained prioritization techniques more efficient than manual selection in the development environment?

Now that we have seen that the developer selection can correctly detect at least one failed test, we want to know if a selection made from a prioritized list[3] can be more effective at detecting failed tests than the manual developer selection.

To help us answer this question, we first remove from the 29 experimental units, the one the developer failed to select. Then, for each of the remaining 28 units, we list the most effective technique[4] at fault detection and collect the total execution time of the tests selected by the developer. Finally, we created some visual analyses to describe our data.

Figure 4.5 compares, per experimental unit, the number of failed tests detected by the developer selection, the number of tests detected by the selection made on the most effective technique, and the total number of tests that had to be detected. For example, in unit number eight, the developer took 37 seconds to execute her selection, with this time, she can detect the only failed test, while the TCP technique did not. In approximately 43% of the units, the developer selection detected more failed tests than the TCP techniques, in 18%, the techniques detected more, and in 39%, they detected the same amount.

In the second visual analysis, we want to show a new view of how TCP techniques would behave if used in the development environment. Figure 4.6 shows the comparison between the execution time of the tests selected by the developer, with the time that the most effective technique took to perform the same amount of failed tests. We use a circle to represent the execution time of the selected tests and a triangle to represent the execution time of the most effective technique. For example, in the unit number eight, the execution time of the tests selected by the developer was 37 seconds, while that the prioritized list took approximately 3 min to run the same amount of failed tests. The experimental unit number 28 was an outline because it is very high the difference between the failed tests selected by the developer and the tests that failed. This occurred because the failure was related to the bigger entity of the

[3] As described before in Section 3.1.1, in our study, a prioritized list selection works by taking the tests from the top positions while the sum of their time execution does not exceed the total time execution of the developer selection.

[4] The most effective technique in the 28 experimental units was Echelon Time. The relationship of each experimental unit with its most effective technique was listed on our website.

Figure 4.5: Comparison, per experimental unit, of the number of failed tests that the most efficient technique detected utilizing the time it took the developer to perform her selection, with the number of tests the developer detected, and with the total number of tests that had to be detected.

system. And the developer was not able not to trail all the tests that could detect the problem.

In 12 of 28 units, TCP techniques took longer to perform failed tests than the developer selection. And in the units that techniques were faster, the execution time of the developer-selected tests was very close to theirs. This is indicative that TCP techniques may not be worthwhile in the development environment.

In none of these analyses, we include in the execution time of the developer selection the time used to select which tests would be run to validate the changes. The execution time here consists only of the time used to run the selected test cases.

With these two analyses, we can conclude that **prioritization techniques cannot be more effective than the developer manual selection**. Because they usually detect the same number of failed tests or less than the developer selection using the same execution time. And even when they can detect the same amount of failed tests faster, the gain is tiny compared to when they take longer.

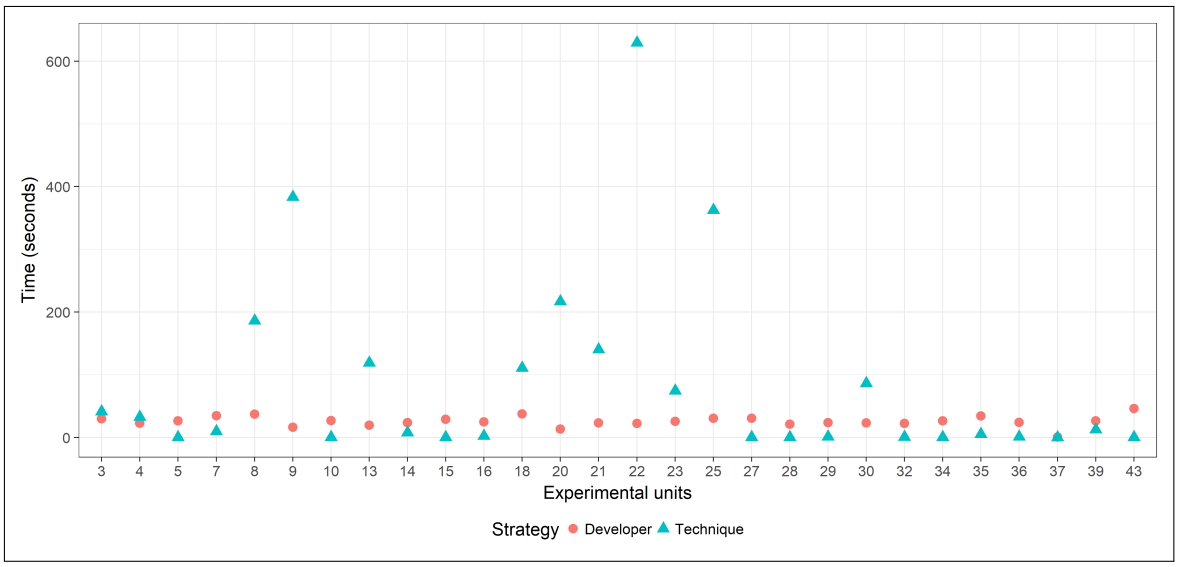

Figure 4.6: Comparison between the execution time of the developer-selected tests, and the time that the most effective technique of each respective unit, took to perform the same amount of failed tests.

Chapter 5

Discussion

We present in this chapter a discussion about our findings and the threats to validity of our study.

5.1 Prioritization tradeoffs

Our study shows that the Echelon and Echelon Time are the most effective techniques in fault detection among the evaluated TCP techniques in the scenario where all tests need to be performed. Most studies cite Echelon in their related work to use it as an example of a prioritization approach [14; 8; 9; 7]. To our knowledge, only Alves et al. [3] work uses it for comparison with other techniques. In their work, they proposed a new approach based on refactoring and compared it with Echelon, Total, and Additional techniques to evaluate its effectiveness. Echelon achieved significant results in APFD, beating Total and Additional, confirming our findings. Prioritizing tests that cover newly changed lines seems most effective because it is where probably occurs the system fault, since before such changes, the regression tests were passing.

We also show that Echelon and Echelon Time are the most effective techniques in the reduction of the spread rate of failed case tests, followed by the Total. This is probably because all tests that cover any changed line are grouped at the beginning of the prioritized list, and as we have seen from the APFD result, these tests are the ones that probably fail, so the tests that fail end up grouped. No study analyzes the behavior of the Echelon using this metric. However, in the work of Torres et al. [25], they investigated and compared the other

four techniques used in our work (Total, Additional, Genetic, and ART) using a variation of the F-Spreading metric. The difference between the F-Spreading and its variation (M-Spreading) is that M-Spreading works with several faults being detected by the test suite, whereas F-Spreading works with only one fault, which is why we chose it. They showed that among the four techniques, the Total had the best result, confirming our findings.

Selecting a prioritization technique depends not only on its efficiency in the achievement of a specific objective but also on its cost to execute [24]. Thus, we decided to analyze the cost of using each technique.

First, we look at the effort to gather the information needed to perform the techniques. Since they all depend on the same data coverage information[1], the overhead is the same. However, the Echelon technique and its extension, Echelon Time, need one more information for its execution (the changed classes), giving them an extra cost to execute.

In the work of Srivastava et al. [23], they propose the Echelon using a binary code-based approach, which has an advantage over using source code because its extraction of code difference avoids complex static analysis making it faster [13]. In our study, we choose to implement this technique using the source code approach, once the authors provided only a pseudo algorithm in their work. Therefore, as we had to implement it, we decided to do it in the same language as the other techniques to facilitate the collection of coverage and the automation of the experiment. To extract the changes, we had to use the Unix `diff`, a simple and quick tool [23]. We did this extraction manually because we were dealing with past data. There was no way to know the original commit used to clone the repository to get the difference automatically. We believe the cost to automatically extract this information is low.

Second, we consider the time necessary to run the techniques. Table 5.1 shows the average processing time of each technique during the case study. As we can see, ART and Genetic techniques are the slowest to generate the prioritized list, making it almost impossible to incorporate them in the CI process. They take this time because they need to do more operations than the others to prioritize. For Echelon and Echelon Time, the higher the number of changed lines, the longer it takes to process the tests that exercise those lines.

In the end, when we analyze the cost-benefit of a technique, we weigh the results it

[1]Coverage information collection aspects are discussed in more detail in Section 5.2

Technique	Time (s)
Total-stmt	0:00:01.694
Add-stmt	0:07:15.113
Echelon	0:03:07.165
EchelonTime	0:03:13.759
ART	3:56:42.036
Genetic	3:12:33.696

Table 5.1: Average processing time of each technique. In gray, the technique with the shortest time of processing.

achieves versus the cost it takes to implement. Thus, we can conclude that ART and Genetic presents a prohibitive time cost execution, so they must be discarded. **If the developers' time constraints are an issue, then the Total approach would be better**. However, the **Echelon technique can give better results in discovering relevant faults and grouping them**.

5.2 Coverage Processing

According to Elbaum et al. [7], keeping the coverage data information updated per test is not feasible in continuous integration environments. Data collection requires the execution of the entire suite, making the process even more expensive. Because of that, there is only one study, to the best of our knowledge, that shows the results of using a coverage-based technique in CI environments.

We performed a study to know if TCP techniques that use coverage information could be effective in these environments. We had sufficient time to execute the entire suite test before performing the techniques because it was outside the real CI environments. To implement these techniques in practice require a solution to the coverage processing problem.

One alternative for scenarios such as ePol, in which developers need to run the entire test suite before sending code to the main repository, is to create a **coverage memory**. During the mandatory execution, the system could collect the coverage information per test for this

version, and when the code was sent, the coverage information would be sent as well. In such scenario, the next time the test suite needs to be run, the system will look for the latest version that has coverage collected for that branch and can already begin the prioritization process with that information.

We illustrate this idea, in Figure 5.1. When the developer A or her reviewer decides to validate the code, the entire regression suite is executed (1), and the coverage information is collected (2). If all tests pass and the reviewer gives permission, the developer can submit the code (3). Looking at the final step in more depth, we can see it will consist of two actions: sending the code to the main repository (4) and sending the collected coverage of that code to an online repository (5).

Then, when a developer B decides to perform the regression set, the system will retrieve the coverage information and begin the prioritization stage (6), only to then perform the regression tests and collect the coverage data.

In the work of Lu et al. [11], they show the importance of using up-to-date coverage information for test prioritization. They also show that added tests influence a good prioritization. The solution that we propose does not include coverage of recent changes made by the developer. However, we believe that by making use of available coverage information and giving high priority to added or changed tests, we will already have a good prioritization. Unfortunately, this solution is yet to be tested and remains for future work.

5.3 Project-related issues

During the experiment execution, we notice some scenarios related to the project that caused the low performance of some techniques.

The first one occurred due to the way developers designed the ePol REST layer test long before our experiment. They used an interface to simulate calls to the endpoints, which means that, the REST classes of this layer are not called directly by the tests, but by this simulator. For example, suppose the developer wants to create a test to verify the addition of a person to the system. Instead of the test call the *PersonREST* addition method, it should call the *PersonRest* simulator that, which in turn, will make a call to the endpoint of the *PersonREST* addition method. As this call occurs through the framework it is not possible

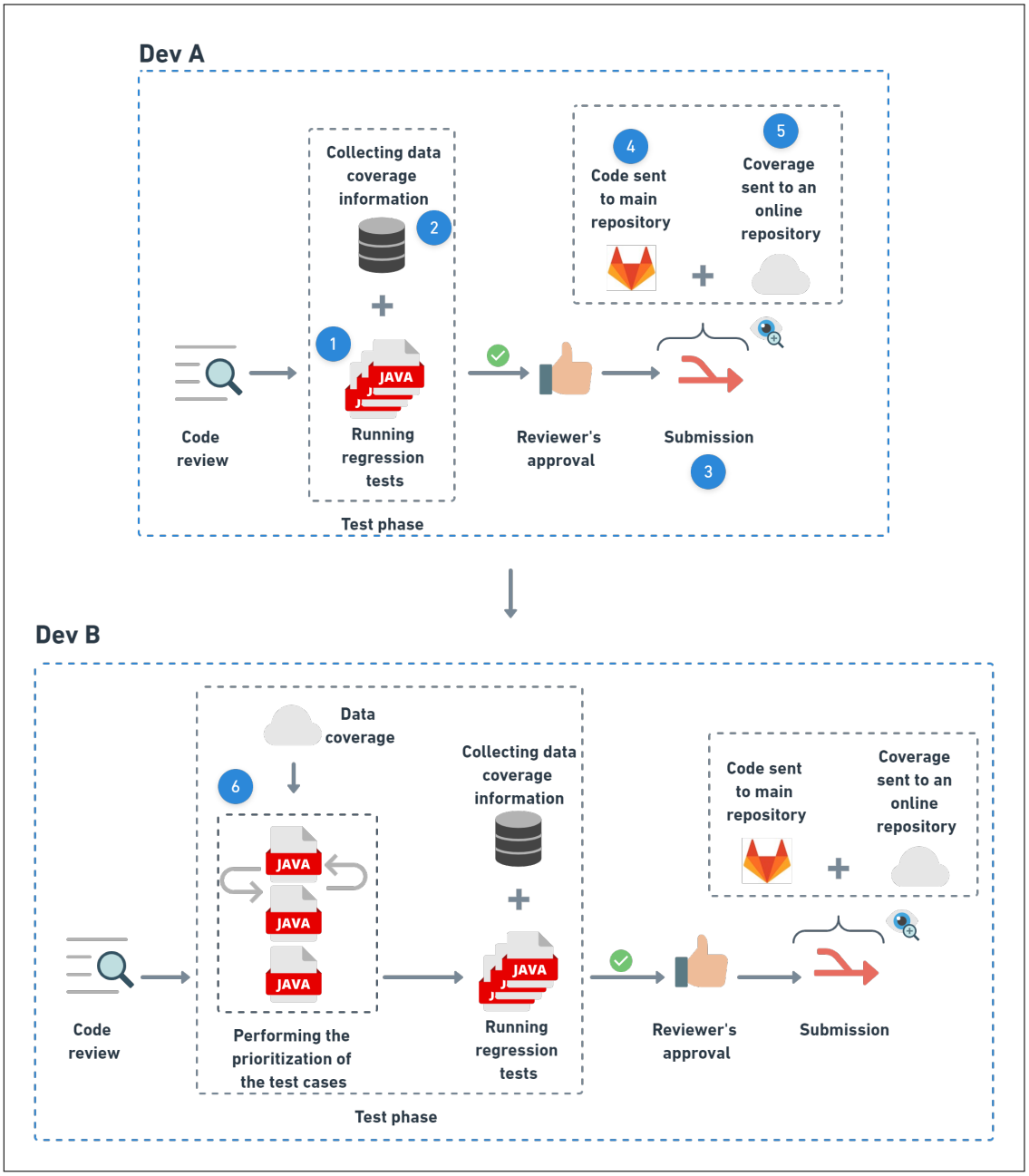

Figure 5.1: Scenario using the coverage memory solution.

for the *Jacoco* to track it. Because of that, when we collect coverage information from tests that exercise this layer, we cannot see the REST classes in the coverage list. In summary, what happens in this scenario is that when **the developer changes some lines in one REST class, the tests that exercise these lines do not contain coverage information about them, and in the end, techniques as Echelon cannot prioritize well**.

Of the 12 units in which Echelon under-performed, three were for this reason. Even with coverage information missing from the REST classes, the other techniques are not affected because they only use the total coverage information. In the three units, the techniques with the best results are Genetic, ART, and Original.

The second scenario involves Original strategy. In two of the 43 experimental units, the original order out-performed all the techniques in the APFD metric. The justification is that the failed tests of these units had names that benefited by the alphabetic order and that they occur in scenarios that the coverage did not work well, as REST tests.

The third scenario occurred on experimental units that failed due to added or changed tests. Of the 12 units where Echelon under-performed, four were because these tests failed on lines preceding the call of changed methods, and thus there was no coverage of the modified parts. This is further indicative that **added or changed tests should be considered when proposing a new prioritization technique**. This finding confirms the previous work of Lu et al. [11].

5.4 Practice of Testing

In *RQ5* and *RQ6*, we analyzed the tests selected by developers aiming to validate their changes during the development process. The *RQ5* results showed that the developer selection always includes at least one failed test. So, it performs better in the scenario where there is only one failed test to be identified. And in *RQ6*, we showed that the techniques, when time-constrained, are less effective than manual developer selection. Using the same time used by the developers in practice, the techniques executed the same amount or fewer failed tests than developer selection.

These are interesting results because some works showed that TCP techniques improve the order used by the developers [7; 5]. However, this is only true when the entire suite needs

to be executed. When comparing the effectiveness of a manual selection made by developers with that of TCP techniques under a time constraint, we see that using the techniques does not seem worthy.

This result may be related to the number of failed tests in the suites used in the experiment. A comparison involving information on the total coverage of the regression suite could complement our results, however we did not include this data to be collected. Processing the *Jacoco's* .exec file could give us this information, but that will be a point to be added in the future.

Analyzing the nine units where the techniques lost with a big difference in Figure 4.6, we saw that their APFD values are relatively high (median at 0.85). This indicates that a high APFD does not mean that the failed tests will be in the first positions. An example of this is experimental unit 13, its APFD is 0.970 with the test that detects the fault at position 121º, which is great if the developer is going to perform all 4142 tests. But within the development environment, he wants to run the minimum amount of tests to validate his code. Performing 121 tests is not the best option here if only one matter.

In summary, if the developer does not mind waiting a few minutes longer, prioritization techniques are the best option as they do the work of order the tests and running them up to a certain maximum time. However, if the developer wants to minimize the test execution time, his selection can be the best option.

5.5 Threats to validity

The threats to the internal validity of our study are: 1) the correctness of our tools and scripts for running experiments, 2) the correctness of the implementation of the TCP techniques and 3) the random aspect of the techniques execution. For the first one, to minimize the errors, we tested some of them to assure the correctness, and to the others, we validated using some small examples of test suites and programs. For the second, we reused almost all the implementations of the techniques of other works [11] or tools [15]. The only implementations that we made were the Echelon and its extension, and to reduce the threat, we did some tests to assure the correctness. And for the third one, as only two techniques had random calls in their algorithm and as they were the most time consuming, we decide to execute only time.

However, this affects only two of the six techniques. To reduce this threat, we would need more resources to run more times these algorithms.

The threats to external validity include 1) the number and type of experimental units used (almost all the experimental units were of bugs correction) and 2) the number of faults in the experimental units. For the first one, we try to include the maximum number of experimental units in the experiment. However, the time to process and execute the TCP techniques was too long. So, for this threat, only an extension of this study would include more experimental units in the analysis. And for the second, maybe adding seeded faults to complement the real ones. However, according to the purpose of this study, these faults can be regarded as appropriate.

The threat to construct validity mainly appoint to the metrics used to evaluate the effectiveness of TCP techniques. We used APFD and F-Spreading, aiming to reduce this threat. The first is the most used metric in different prioritization works [14], and the second complements the first one. However, to reduce even more this threat, more studies using other metrics are required.

Chapter 6

Related Work

Many prioritization techniques have been proposed and evaluated in the literature. In this chapter, we present the most relevant works in the area and related them to our study.

6.1 Prioritization techniques and their evaluation

In 1999, Rothermel et al. [16] introduced several prioritization techniques, including Total-stmt and Add-stmt, and empirically examined their ability to improve the fault detection rate of a test suite. It was the first work to use the APFD metric in its evaluation, metric which was only formally defined years later [6]. They used seven C programs as subjects with inserted faults. The experimental results showed that all prioritization techniques can improve the rate of fault detection of test suites. Looking just at the results involving the two techniques used in our study, we can see that Total outperformed the Additional in all the seven programs, which is similar to our findings using a Java program with real faults.

Following the works in the coverage area, Alves et al. [3] proposed a refactoring-based selection/prioritization approach to help developers to detect faults more effectively. Their motivation is that even though widely used coverage techniques produce good results, they are general-purpose techniques, while their approach is specifically to detect refactoring faults. To evaluated their approach, they performed a case study in a real Java system using six versions with created faults and compared to the other six prioritization techniques, including Total-stmt, Add-stmt, and Echelon using the APFD and F-Measure metrics. The case study showed that their approach outperformed the other techniques in almost every

case. The Echelon also showed good results, especially when compared with the other techniques. However, as it does not differentiate the type of code changes applied, it ended losing for the proposed technique. Comparing it with our study, a case study involving more experimental units and using real faults, we get similar results involving the three techniques (Total-stmt, Add-stmt, and Echelon).

Years later, the authors of this work published a new one [2] extending the solution with a complete definition of refactoring fault models and a broader evaluation. The approach called Refactoring-Based Approach (RBA) was evaluated in this new work using three real open-source projects with seeded refactoring faults and compared with the same six prioritization techniques. However, in this work, in addition to the APFD and F-Measure, they used a new metric called F-Spreading, which is a rate that measures how the failing test cases are spread in a prioritized test suite. The results involving the three techniques common to our study showed that the Echelon almost have the same effectiveness as the traditional techniques for APFD values. And for F-spreading, we saw that Echelon loses to traditional techniques, remaining roughly equal to the random order, which diverges from our results. We believe this occurs because of the specificity of the context that involves only a few types of refactoring, while ours involves other types of changes.

Srivastava and Thiagarajan [23] proposed a binary code-based approach for test prioritization based on program change (Echelon). Its main idea is to identify the changed blocks between the two versions (working at the binary level) and order the tests according to the number of modified blocks covered by each test. In their work, they measured the Echelon performance on a large Microsoft product binaries without comparing it with other techniques or approaches. According to the authors, the results showed that Echelon is a fast technique and scales well to large programs, making it suitable for use in development environments. In our study, we implement the Echelon in Java language and replace the binary way to identify the changed blocks to a source-code one, that is, we used only the technique algorithm. And, even not been faster as the binary approach, our results showed that the Echelon effectiveness is still better than the other evaluated techniques.

Jiang et al. [8] proposed in their work a set of ART prioritization techniques guided by white-box coverage information. For the evaluation, they used 11 programs with real and seeded faults and compared the ART family with random ordering and six existing coverage-

based prioritization techniques using the APFD metric. The results showed that the ART techniques were superior to random order, and one of them is statistically comparable to the best-studied coverage-based prioritization techniques in terms of the fault detection rate. This result is similar to ours because when we analyzed the traditional prioritization techniques we saw that ARTMaxMin had the same effectiveness than Total-stmt and Add-stmt techniques.

6.2 Empirical studies

To initiate the section of empirical studies, we present the Lu et al.'s work [11]. They conducted an empirical study to investigate the effectiveness of existing test prioritization on eight open-source Java projects with seeded faults. Their main goal was to evaluate the TCP techniques by considering the influence of the added tests and real source code changes and considering the influence of time budgets in prioritization. This work is very similar to ours because they used almost all techniques that we used (Total, Additional, Genetic, and ART-MaxMin) and evaluated them on a real-world Java program using the APFD metric. The differences were that we used real faults, and we include two other techniques. Their results showed the importance of including the added test when proposing new test prioritization techniques. They also showed that among the test prioritization techniques, the Additional and Genetic were the most effective. The first find is similar to what we found in our study. However, the second one is a little different because we saw that Genetic had the worst result between the techniques.

Torres et al. [25] made a similar study to the last work. They compared four TCP strategies using the same design, subjects, and techniques implementations from Lu et al. [11]. However, besides the APFD metric, they evaluated using the F-Spreading, which is the same metric that we used in our study. The APFD result showed similar results found in Lu et al.'s work, meaning that they also disagree with part of our finds. However, concerning the F-Spreading, they showed similar results to ours. The Total presented the lower spreading of all techniques and followed by Additional and Genetic with the same results. Only the ARTMaxMin that, in Torres et al.'s work, under-performed all the techniques, and in ours, it presented the same result that Additional and Genetic.

Now, from the studies involving prioritization on the CI environment, we started with Elbaum et al. [7] work. In it, they evaluated a new approach for applying regression testing in continuous integration development environments more cost-effectively. Their approach uses not only test case prioritization but also the regression test selection (RTS)[1]. According to the authors, traditional RTS and TCP techniques are difficult to apply in the CI environment because they tend to rely on code instrumentation and apply only to discrete, complete sets of test cases. So, their approach innovates by using only failure history on its logic. It works as follows: the RTS technique uses time windows to track how recently test suites have been executed and revealed failures, to select test suites to be applied during pre-submit testing[2]. Then, the TCP technique based on such windows prioritizes test suites that must be performed during subsequent post-submit testing.

Their empirical study was made using an extensive data set of test suite execution made available by Google and involved, besides the new approach, a baseline approach that utilized all test suites in the change list, and a random RTS approach. They used the APFD metric to evaluate the TCP technique and measured the percentages of failures detected by the RTS technique. The results showed that the TCP technique can reduce delays in fault detection during post-submit testing. However, to the RTS technique they showed that even when it reduces the testing execution cost, it can delay the detection of faults.

Liang et al. [10] also proposed and evaluated a new TCP technique for using in a continuous integration environment. As the work of Elbaum et al. [7], they also believe that traditional TCP techniques are difficult to apply in the CI process. So, their approach is also based on execution history and test suite failure. The main difference from the previously presented work to this is the moment that this technique prioritizes the tests and the moment that it collects the window information. In their study, they used the data set from Google and other project managed under Travis, an open-source CI server, and compared their approach with a baseline in which test cases are executed in the original order and an optimal order. The metric used to evaluate the study was the APFDc, a variation of APFD metric that takes the test case cost into consideration. The results showed that the technique proposed can

[1]RTS techniques attempt to select test cases that are important to execute [7].

[2]Pre-submit testing and post-submit testing are terms used in this work to refer to the phase of testing made by the developer before submitting the code, and the phase after submitting, respectively.

be more effective than the original order, but not as effective as the optimal order. According to the authors, the new technique is lightweight and quick, allowing it to be sufficiently responsive in CI environments.

The relation between our study and the two works previously presented is that both focus on TCP techniques applied in CI environments. The difference is that we used traditional TCP techniques in our study. And although these works point that techniques that use coverage information are difficult to apply, we showed that they still can present great results.

Finally, we present the Nardo et al. work [5], the study more related to ours. They performed an industrial case study of coverage-based prioritization techniques on a real-world system with real regression faults. Four techniques were used in the study: Total, Additional, Total Coverage of Modified Code and Additional Coverage of Modified Code. The four techniques were applied using five different structural coverage criteria (function, calls, statements, and branches). Besides the four techniques, the study evaluated a random, optimal and original order. It involved four versions of a system written in C++, and the metric used in the evaluation was the APFD. The results showed that the Additional technique using finer grained coverage criteria outperformed all other techniques used in the study, including random ordering. This result differed from ours because, for us, the modification-based technique had the best result between the other techniques. As the authors did not specify from which work they used the algorithm we can not compare it as the same technique. Another differences with relation to our work are that we include techniques with different approaches (ARTMaxMin and Genetic) and we include one more metric.

Chapter 7

Conclusion

Although test prioritization has been intensively investigated, few industrial case studies have been done involving real faults and assessing TCP techniques that use coverage information in a system that uses the CI environment. In this work, we conducted a case study on a large evolving web system to evaluate and compare the effectiveness of six prioritization techniques that use coverage information with what occurs in practice, focusing on their possible use in the development process. In our study, we used real faults inserted by the developers during maintenance.

To answer the *RQs*, we applied the TCP techniques to 43 experimental units, resulting in a total of 258 prioritized test suites. For each suite, we calculated the APFD and F-Spreading values, which were analyzed and tested. The case study shows that in the scenario where it is needed to run all tests, **Echelon and Echelon Time are the most effective techniques in fault detection and reduction of the spread rate of failed tests, among the evaluated TCP techniques**. When they are compared, considering the cost-benefit of implementation, the result is still the same. Also, all the TCP techniques presented the best result than the original order used in the CI environment.

Another significant result is that, although TCP techniques had better results than the original order, **they were not more effective at fault detection than developer selection when time constraints are higher**. They usually detect the same number of failed tests or less using the same execution time. Developer selection is maybe more effective because he almost always selects changed/added tests, which are the tests that usually fails, whereas the techniques do not prioritize these tests. **Added or altered tests should always be consid-**

ered when proposing a new prioritization technique.

This comparison may be unfair to the techniques because they did not include this type of test in their heuristic. However, this was the scenario for almost half of the experimental unit, which means that it is a common scenario. In the future, we want to extend the present work by increasing the number of experimental units focusing on the situations in which this type of test is no present to investigate whether the effectiveness of the techniques increases.

Also, as future work, we want to implement in practice the Echelon technique in the CI environment to test if the developers get used to prioritization techniques in their processes. We want to implement the coverage memory solution to analyze whether, using the available coverage information, we still get a better result than the original order. And finally, implement an extended version of the Echelon technique that will include in its prioritization process the added/changed tests, so we can evaluate if it can be more effective than developer selection.

Bibliography

[1] https://martinfowler.com/articles/continuousIntegration.html. Acesso em: 26 de dezembro de 2019.

[2] E. L. G. Alves, P. D. L. Machado, T. Massoni, and M. Kim. Prioritizing test cases for early detection of refactoring faults. *Softw. Test. Verif. Reliab.*, 26(5):402–426, August 2016.

[3] E. L. G. Alves, P. D. L. Machado, T. Massoni, and S. T. C. Santos. A refactoring-based approach for test case selection and prioritization. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 93–99, May 2013.

[4] W. J. Conover and R. L. Iman. On multiple-comparisons procedures. Los Alamos Scientific Laboratory, February 1979.

[5] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche. Coverage-based test case prioritisation: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 302–311, March 2013.

[6] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb 2002.

[7] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[8] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, Nov 2009.

[9] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

[10] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: Continuous prioritization for continuous integration. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 688–698, May 2018.

[11] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 535–546, May 2016.

[12] G. Macbeth, E. Razumiejczyk, and R. Ledesma. Cliff's delta calculator: A nonparametric effect size program for two groups of observations. *Universitas Psychologica*, 10:545–555, 05 2011.

[13] A. P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008.

[14] R. Mukherjee and K. S. Patnaik. A survey on different approaches for software test case prioritization. *Journal of King Saud University - Computer and Information Sciences*, 2018.

[15] J. H. Rocha, P. D. L. Machado, and E. Alves. Priorj - priorizacao automatica de casos de teste junit. In *Proceedings of Third Brazilian Conference on Software: Theory and Practice (CBSoft) - Tools Section*, volume 4, pages 43–50, Natal, 2012.

[16] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 179–188, Aug 1999.

[17] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct 2001.

[18] 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.

[19] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, December 1965.

[20] K. Singh and M. Xie. Bootstrap: a statistical method. *Unpublished manuscript, Rutgers University, USA. Retrieved from http://www.stat.rutgers.edu/home/mxie/RCPapers/bootstrap.pdf*, 2008.

[21] Y. Singh, A. Kaur, B. Suri, and S. Singhal. Systematic literature review on regression test prioritization techniques. *Informatica*, 36(4):379–408, December 2012.

[22] I. Sommerville. *Engenharia de software*. PEARSON BRASIL, São Paulo, 9 edition, 2011.

[23] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes*, 27(4):97–106, 2002.

[24] D. Suleiman, M. Alian, and A. Hudaib. A survey on prioritization regression testing test case. In *2017 8th International Conference on Information Technology (ICIT)*, pages 854–862, May 2017.

[25] W. N. M. Torres, E. L. G. Alves, and P. D. L. Machado. An empirical study on the spreading of fault revealing test cases in prioritized suites. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 129–138, Jul 2019.

[26] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings The Eighth International Symposium on Software Reliability Engineering*, pages 264–274, November 1997.