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

Investigation of Test Case Prioritization for Model-Based Testing

João Felipe Silva Ouriques

Tese 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 Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dra. Patrícia Duarte Lima Machado
(Orientadora)
Dra. Emanuela Gadelha Cartaxo
(Co-orientador)

Campina Grande - Paraíba - Brasil
©João Felipe Silva Ouriques, Dezembro de 2017

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

O93i Ouriques, João Felipe Silva.
Investigation of test case prioritization for model-based testing / João Felipe Silva Ouriques. – Campina Grande, 2017.
138 f. : il.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2017.
"Orientação: Profa. Dra. Patrícia Duarte Lima Machado, Profa. Dra. Emanuela Gadelha Cartaxo”.
Referências.

1. Priorização de Casos de Teste. 2. Teste Baseado em Modelo. 3. Detecção de Faltas. I. Machado, Patrícia Duarte Lima. II. Cartaxo, Emanuela Gadelha. III. Título.

CDU 004.43(043)

"INVESTIGATION OF TEST CASE PRIORITIZATION FOR MODEL-BASED TESTING"

JOÃO FELIPE SILVA OURIQUES

TESE APROVADA EM 12/12/2017

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

EMANUELA GADELHA CARTAXO, D.Sc, UFCG
Orientador(a)

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

WILKERSON DE LUCENA ANDRADE, Dr., UFCG
Examinador(a)

JULIANO MANABU IYODA, PhD, UFPE
Examinador(a)

ANAMARIA MARTINS MOREIRA, PhD, UFRJ
Examinador(a)

CAMPINA GRANDE - PB

Acknowledgements

Here I express my gratitude to everyone that contributed, either directly or even indirectly, to the endeavors needed to traverse and conclude this doctorate research. Besides the lessons learned as a researcher, all the people mentioned here have taught me other important life lessons.

First and foremost, I would like to thank **Raquel Andrade** for all her support and love. Several hard moments were alleviated by her, either acting to do something about or just by being close. For following me and allowing me to follow her on hard decisions.

My family for all their love, understanding, and encouragement. The first supporters of my interest on the knowledge were my parents, **Valdenice Margarida** and **Flaviano Ouriques**, which made huge efforts to provide me the best education possible. Besides, I would like to thank my brother **Paulo Victor** for helping me to cultivate curiosity about how things work. Furthermore, I would like to thank **Joceli Andrade** and **Edvaldo Barros** by their advices and encouragement.

I would like to thank **Emanuela Cartaxo** and **Patrícia Machado** by their supervision, motivation, and for giving me autonomy to follow my instincts regarding our research. For providing feedback on our research steps and being good research partners. Besides, for connecting me with **Ingenico do Brasil**, which was a great partner during this research, providing us with case studies, crucial for our progress.

I also thank my friends **Francisco Gomes** and **José Bener** by their constant contact and for cheering me up when I was down and complaining, either in person or on our chat group. Since we share some major life changes, I owe them my admiration and gratitude for the support in the process.

Furthermore, I thank the staff of **Programa de Pós-Graduação em Ciência da Computação (PPGCC-UFCG)** by their support regarding administrative tasks. I also would like to remark the financial support provided by the **Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq)** and by the **Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES)**.

Contents

1 Introduction **3**

1.1 Problem Statement and Proposed Solutions 6

1.2 Research Questions and Methodology 11

1.2.1 Research Methodology 13

1.3 Contributions 15

1.4 Chapter Final Remarks 15

2 Background **17**

2.1 Software Testing 17

2.2 Model-Based Testing 20

2.2.1 Modeling a System's Behavior 21

2.2.2 Generating Test Cases 24

2.2.3 Test Purpose 28

2.3 Test Case Prioritization 29

2.3.1 Studied Techniques 31

2.3.2 Evaluation Metrics 43

2.4 Empirical Research in Software Engineering 45

2.4.1 Data Analysis 47

2.5 Chapter Final Remarks 50

3 Investigation on Relevant Factors **51**

3.1 Preliminary Results 52

3.2 Investigation with Industrial Artifacts 55

3.2.1 Investigated Techniques 55

3.2.2 Systems, Test Suites, and Faults 57
3.2.3 Planning and Design 60
3.2.4 Results and Analysis 62
3.2.5 Threats to Validity 70
3.3 Chapter Final Remarks 71

4 Hint-Based Adaptive Random Prioritization - HARP 73
4.1 What is a Hint? 74
4.2 Proposed Algorithm 75
4.2.1 Asymptotic Analysis 78
4.2.2 Running Example 79
4.3 Empirical Evaluation 80
4.3.1 Experiment on Hint's Quality 81
4.3.2 Exploratory Case Study 90
4.4 Chapter Final Remarks 98

5 Clustering System Test Cases for Test Case Prioritization - CRISPy 99
5.1 Proposed Algorithm 100
5.1.1 Cluster Creation and Thresholds 100
5.1.2 Clusters and Test Case Prioritization 102
5.1.3 Asymptotic Analysis 103
5.1.4 Running Example 104
5.2 Empirical Evaluation 105
5.2.1 Comparative Case Study 106
5.2.2 Experiment on Tuning Potential 110
5.2.3 Experiment Comparing CRISPy to HARP 114
5.3 Chapter Final Remarks 116

6 Literature Review 118
6.1 Black-Box TCP Techniques 119
6.1.1 Structural Aspects 119
6.1.2 Dissimilarity-Based 120

6.1.3 Soft-Computing TCP Techniques 121
6.2 Chapter Final Remarks 123

7 Conclusions and Final Remarks 124
7.1 Research Questions and Contributions 124
7.2 Discussion on Practical Aspects 126
7.3 Future Work 127

Acronyms

AHC - *Agglomerative Hierarchic Clustering*

APFD - *Average Percentage of Fault Detection*

CRISPy - ***C**luste**RI**ng for **S**ystem test case **P**rioritization*

HARP - ***H**int-based **A**daptive **R**andom **P**rioritization*

ICP - *Interleaved Cluster Prioritization*

LTS - *Labeled Transition Systems*

MBT - *Model-Based Testing*

SUT - *System Under Test*

TCP - *Test Case Prioritization*

V&V - *Verification and Validation*

List of Figures

1.1 Execution of an MBT TCP technique. 6
1.2 Overall flow of activities, artifacts and research lines. 13

2.1 Activity flow of the Model-Based Testing process. 21
2.2 Elements of an LTS: The circles represent the states, the arrows represent the transitions and the state with an arrow without source, is the initial one. . . 23
2.3 The LTS model representing the behavior of a login/password system. . . . 24
2.4 Motivation for the application of the test case prioritization. 30
2.5 Visual interpretation of two scenarios of APFD. The area on the second chart (right) is bigger, indicating that the specific order of test cases in that test suite allows testers to reveal faults sooner. 44
2.6 Example of useful plots for descriptive statistics. 48

3.1 Overview of the performed case study. 62
3.2 Overall data for all techniques. 64
3.3 Performance of the techniques by system 65
3.4 Boxplots of S2 and S4 samples. 67
3.5 Boxplots of **ShortTC** and **LongTC** samples. 68

4.1 Experiment Design Overview 86
4.2 Boxplots of the raw data collected in the experiment execution. 87
4.3 Summary about the questionnaire's participants. 91
4.4 Summary about the collected data. 96

5.1 Graphical demonstration of the Jaccard distances between test cases. 104
5.2 Graphical demonstration of the cluster merging step. 105

5.3 Box plot with the samples from each technique, showing their medians and spreading. 108
5.4 Box plots with the samples configured with the investigated distance functions and thresholds. 112
5.5 Box plots showing samples of the difference between the test suite size and the number of generated clusters, for each distance threshold. Notice that the variables are directly related. 112
5.6 Box plots showing samples of the investigated techniques. 115

List of Tables

1.1 Test suite that tests a simple login and password verification system. 7

1.1 Test suite that tests a simple login and password verification system. 8

1.2 Test cases order produced by the greedy and reverse-greedy techniques for the test suite from Table 1.1. 9

1.1 Test suite that tests a simple login and password verification system. 9

2.1 Test suite generated from the LTS depicted in Figure 2.3. 26

2.1 Test suite generated from the LTS depicted in Figure 2.3. 27

2.2 Characteristics of the kinds of empirical research [87]. 46

2.3 Summary of some important hypothesis tests. 49

3.1 Overview about the investigated systems and related test suites. 59

3.2 Side by side comparison between the original and replication studies. . . . 63

3.3 Effect sizes of pairwise comparisons of the investigated techniques. Each cell contains the result of the comparison between the technique from the line i and the one in the column j. The diagonal (lighter grey) contains the comparison $\hat{A}_{12}(i, i)$, which always lead to effect size 0.5 and it is not relevant for our purposes. The darker grey area omits values that are complementary to the presented values. Bold faced values represent medium and large effect sizes. 66

3.4 Effect sizes of the comparisons for each technique between **ShortTC** and **LongTC** 69

4.1 1st Iteration: Similarities among candidates and test cases already prioritized 80

4.2 2nd Iteration: Similarities among candidates and test cases already prioritized 80

4.3 Illustration of the good hint generation 83
4.4 Metrics about the artifacts used as experiment objects 86
4.5 Effect sizes of pairwise comparisons 88
4.6 The indications collected from the questionnaires, as well as their reasons. In bold we highlight regions that a majority is achieved. 93
4.7 Metrics about the test suites investigated in the case study. We measure the shortest and longest test cases in amount of steps, expected results and conditions that the test case describes. 95

5.1 Nonparametric pairwise comparisons: dark grey (upper right) area are the effect sizes calculated by $\hat{A}_{12}(i, j)$, where i is the technique from the row and j is the one from the column. Analogously, the light grey (lower left) area contains the p-values of Mann-Whitney tests comparing j and i. We omitted the values with lower orders of magnitude to improve visualisation. 109

Abstract

Software Testing is an expensive and laborious task among the Verification and Validation ones. Nevertheless testing is still the main way of assessing the quality of systems under development. In Model-Based Testing (MBT), test cases are generated automatically from system models, which provides time and cost reduction. On the other hand, frequently the size of the generated test suites leads to an unfeasible complete execution. Therefore, some approaches are used to deal with the costs involved in the test case execution, for example: Test Case Selection, Test Suite Reduction and the Test Case Prioritization (TCP). In order to improve resource consumption during system level testing, whereas not eliminating test cases, we explore TCP in our work. In this doctorate research we address the general problem of proposing a new order for the execution of system level test cases, specifically the ones generated through MBT approaches and considering that historical information is not available to guide this process. As a first step, we evaluate current techniques in this context, aiming at assessing their strengths and weaknesses. Based on these results, we propose two TCP techniques: HARP and CRISPy. The former uses indications (hints) provided by development team members regarding error prone portions of the system under test, as guidance for prioritization; the latter clusters test cases in order to add variability to the prioritization. HARP is based on the adaptive random strategy and explore test cases that cover steps related to the hints. To validate it, we perform two studies and from them we conclude that, with good hints, HARP can suggest effective prioritized test suites, provided that one avoid situations that the hints are not consensual in the development team. On the other hand, we propose CRISPy to complement HARP, i.e. to apply it when HARP is not indicated. To validate it we perform three studies and, even being a simple clustering technique, it already proposes effective prioritized test suites in comparison to current TCP techniques. Besides, depending on the kind of the test suites, choosing an adequate distance function could be a tuning step. We detail the implementation of all investigated and proposed techniques, and publish the artifacts related to the performed empirical studies in a companion site, enabling researchers to verify and reproduce our findings.

Resumo

Teste de Software é uma tarefa cara dentre as de Verificação de Validação. No entanto, ainda é a mais utilizada para medir a qualidade de sistemas em desenvolvimento. No Teste Baseado em Modelo (TBM), casos de teste são gerados automaticamente de modelos de sistema, o que reduz tempo e custos. Por outro lado, frequentemente o tamanho das suites de teste geradas leva a uma execução completa inviável. Assim, algumas abordagens são usadas para lidar com os custos da execução dos casos de teste: Seleção de Casos de Teste, Minimização de suítes de teste e Priorização de Casos de Teste (PCT). Com a finalidade de melhorar o consumo de recursos durante o teste no nível de sistema, ao mesmo tempo não eliminando casos de teste no processo, nós exploramos a PCT neste trabalho. Nesta pesquisa de doutorado abordamos o problema de propor uma nova ordem de execução para casos de teste de sistema, especificamente os gerados através de abordagens de TBM, considerando que informação histórica não está disponível para guiar este processo. Como primeiro passo, avaliamos as técnicas atuais no nosso contexto de pesquisa, com a finalidade de avaliar suas caracteristicas. Baseado nisto, propomos duas técnicas de PCT: HARP e CRISPy. A primeira usa indicações de membros do time de desenvolvimento sobre porções suscetíveis a erros (as ditas dicas), como guia para a priorização; a segunda cria *clusters* de casos de teste com a finalidade de adicionar variabilidade a priorização. HARP se baseia na estratégia aleatório-adaptativa e explora casos de teste que cobrem passos relacionados às dicas. Para validá-la, executamos dois estudos empíricos e deles, concluimos que, com dicas boas, HARP é capaz de sugerir sequências de casos de teste efetivas, mas é necessário evitar situações em que as dicas não são consensuais entre os membros do time de desenvolvimento. Por outro lado, propomos CRISPy para complementar HARP. Para validá-la, realizamos três estudos empíricos e, mesmo sendo uma técnica simples de *clustering*, já consegue propor sequências de casos de teste efetivas em comparação as outras técnicas investigadas. Além disso, a depender dos tipos de casos de teste considerados, a escolha de uma função de distância adequada pode ser um passo de adequação da técnica. Detalhamos a implementação de todas as técnicas propostas e publicamos todos os artefatos relacionados aos estudos empíricos realizados em uma página web, o que permite a reprodução dos nossos resultados.

Chapter 1

Introduction

During software development, the involved personnel produce and maintain a set of resources, including requirement documents, progress reports, source code, and test cases. Among the activities performed to develop a system, Verification and Validation (V&V) plays an important role during software development, since it assesses the quality of the developed software and indicates aspects that could be improved. V&V comprises a set of activities that project members perform in order to deal with quality and, among them, Software Testing is often used to decrease the chances of delivering a buggy product and to increase the quality of software artifacts. Therefore, a great deal of a project's budget and resources is often devoted to V&V tasks [78].

During testing, testers execute the System Under Testing (SUT) providing a set of controlled inputs and check the respective outputs, looking for errors, anomalies, and/or non-functional information [78]. Although widely used and extensively studied, software testing is known to be costly. Beizer [6], Ammann and Offutt [2] point that testing alone can consume nearly 50% of a project's budget. Therefore, research has been conducted aiming at reducing the costs related to software testing.

A system can be tested in different levels of abstraction (or granularity) [78]: *unit testing* focuses on verifying individual and small units, e.g. functions/procedures or even entire classes; *component testing* deals with the interaction of several units, when they provide and consume services to and from each other; ***system testing*** considers all (or some) components in a system as a whole, checking whether they meet their original objectives. On top of that, Myers [56] argues that the high level of abstraction that the SUT is faced makes system

testing the most difficult level of testing. Therefore, by its characteristics, this testing level is essentially behavioral or **black-box**, which means that inner structures of the SUT code, such as iterative and conditional commands, are not explicitly taken into consideration.

In this scenario, Model-Based Testing (MBT) [83] has been researched as an alternative to the traditional testing. It uses models describing the system behavior to automatically perform test case generation. Among the main advantages of using MBT, we can list [83]: i) *time and cost reduction* - automation is one of the pillars of MBT, which helps the software development team to save time and resources when building testing artifacts; and ii) *artifacts soundness* - as test artifacts are derived from system models, MBT decreases the chances of creating erroneous artifacts.

Due to the characteristics of most test case generation algorithms used in MBT, which are based on graph search algorithms, such as depth-first and breadth-first search, generated test suites tend to be massive, which may jeopardize the test case management task [83]. To cope with the size and feasibility of the test suites, which also affects other research fields such as regression testing, a number of techniques have been presented in the literature. Most of them can be classified into three approaches [88]: Test Case Selection [11,29,31], Test Suite Reduction or Minimization [16,74], and Test Case Prioritization (TCP) [22,38,43].

Test case selection aims at guiding model-based test case generation in order to control more strictly the model portions that are going to be tested, reducing the number of generated test cases [83]. However, the same term is also applied in the code-based testing context; here a technique selects a subset of the test cases to be scheduled for execution, according to a resource limitation, such as available execution time or even a certain share of the available test suite [25, 31]. Note that, even though both applications are essentially different, they have the same objective, which is to reduce the number of test cases to be executed. In turn, **test suite reduction** focus on defining a subset of test cases from the original suite, but keeping a set of test requirements that is also satisfied by the original test suite. However, one of the downsides of using them is that there is a chance of discarding test cases that unveil faults uniquely [36].

Therefore, in some scenarios, both test case selection or reduction may fall short on proposing an effective smaller test suite that still reveals all faults that the original test suite would unveil. On the other hand, the TCP problem suggests determining a permutation

that minimizes an evaluation function, defined according to a prioritization objective [22]. As a direct consequence, a TCP technique does not eliminate test cases from the original test suite; instead, it helps to wisely consume resources with the most relevant ones [70]. By not discarding test cases, TCP is flexible, since it allows the tester to decide how many test cases are going to be run, even without knowing how much resource – such as time, testers and hardware – are available in advance. This decision is often delayed until the test cases are finally executed and taken according to projects' aspects, such as budget and/or delivery time, and so on. Besides, TCP is complementary to test case selection and test suite reduction, allowing a test manager to apply TCP uniquely or on a suite already reduced. Thus, TCP has been widely studied, and it is also faced as a good alternative to address the "high cost of testing" problem.

Both code-based and model-based test suites can be prioritized using TCP techniques [46]. Rothermel et al. [71] proposed a classification for TCP techniques, with respect to the kind of artifacts they use as input:

- **TCP applied to regression testing** relies on the assumption that the system already evolved and there is access to the modification history of artifacts, such as model/code fault reports [76, 88]. Regression TCP techniques rely mainly on information from previous testing execution, such as fault reports of previous executions regarding the available test cases, and code/model modifications. In this work, when we mention ***historical information*** from this point on, we refer to the aforementioned kind of information;

- **General TCP**, on the other hand, is the term used to refer to techniques that do not make this same assumption [12], i.e. they just deal with artifacts related to the current version of the software, or even the first one.

The research regarding regression TCP is mature and it already has several verified and replicated findings [54]. On the other hand, in the same research Lu et al. argue that current TCP techniques present significant variations on early fault detection capability in the presence of new test cases. Even though the knowledge available in the regression testing is useful to leverage TCP, they may not be available, for example, during the initial iteration

of system testing, since test cases have never been executed before, or whether the development team does not maintain properly these artifacts, due to the costs associated or lack of organization. Therefore, further investigation is needed to better understand this context.

Therefore, we **focus on the general TCP problem in the system testing context, considering test suites generated through MBT approaches, and assuming that historical information from previous testing executions is not available**. As direct consequence, regression and code-based testing is out of our scope. Besides, Another important detail is that, due to the available formal background and its flexibility, we use Labeled Transition Systems (LTS) as our main way to model systems' behavior (we discuss it further in Section 2.2.1).

1.1 Problem Statement and Proposed Solutions

The execution of a TCP technique usually involves some constructs, as described in Figure 1.1; it takes as input a set of test cases and suggests a new order for them, potentially using complementary information to leverage this process. The main constructs are the ones directly linked to the TCP technique, which are the test suite and other complementary information. Since the construct *test suite* is the main object of study of the whole software testing activity and it is directly linked to the **TCP technique**, we should understand how is the relationship between them before proposing any new technique.

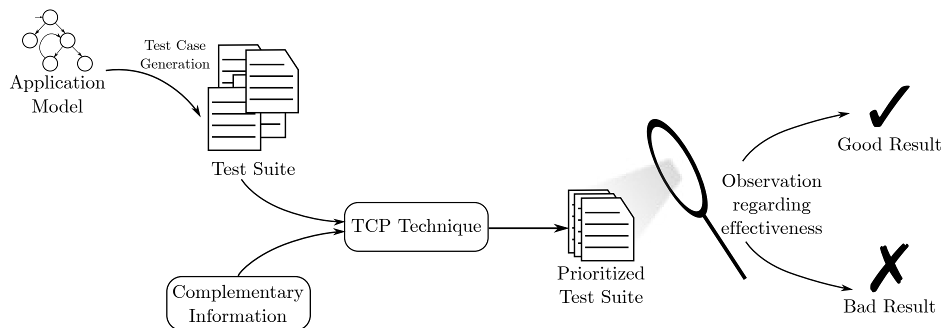


Figure 1.1: Execution of an MBT TCP technique.

Another aspect of our context that affects the test suites and, in turn the TCP technique, is the test case generation in MBT. Given that we just have one kind of model in this thesis, we use a single test case generation algorithm to keep this aspect as neutral as possible in our research. On the other hand, the system model needs to be investigated, since test suites are

generated from them and they vary their characteristics according to the behavior of different systems.

To model the system behavior, one can represent different changes of control flow by branches, joins and loops: branches represent changes in control flow triggered by conditions; joins represent the junction of different control flows into a single one; and loops represent the repetition of steps. Since these three elements define the layout of an LTS and affect characteristics of generated test cases, such as number, length and redundancy, we study them in order to understand their effect on TCP techniques.

Furthermore, other aspect that interact with a TCP technique is more evident in experimental environments; when a TCP technique produces a prioritized test suite, its success is usually measured by "how soon" it detects the faults able to be detected by the test suite. To do so, we also should understand how TCP techniques interact with the test cases that fail, provided that if a technique supposes that a particular kind of test case must have priority over others, it could lead to better or poorer results.

To illustrate how TCP techniques may be affected by different characteristics of test cases that fail, consider a fictitious system that implements login and password verification. We discuss modeling and test case generation processes afterwards (Sections 2.2.1 and 2.2.2), but for now, just consider the test suite from Table 1.1; observe that the test cases contain steps (second column), whose labels' prefixes identify their meaning: "C -" represents a **condition** to be met by the system; "S -" denotes a **user input**; and "R -" represents an **expected result** that the system must provide. Moreover, assume that after a tester executes the referred test cases, only the scenario that represents the successful login fails, in other words, **TC1** fails.

Table 1.1: Test suite that tests a simple login and password verification system.

Label	Steps	Size
TC1	**C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - match → R - Show the main screen of the system**	**8**

It continues on the next page

Table 1.1: Test suite that tests a simple login and password verification system.

Label	Steps	Size
TC2	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: "Login and password do not match" → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - match → R - Show the main screen of the system	15
TC3	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: "Login and password do not match" → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: "Login and password do not match"	15
TC4	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: "Login and password do not match" → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: "Login not found"	12
TC5	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: "Login not found" → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - match → R - Show the main screen of the system	12
	It continues on the next page	

Table 1.2: Test cases order produced by the greedy and reverse-greedy techniques for the test suite from Table 1.1.

Technique	Test cases sequence
greedy	TC2, TC4, TC6, TC3, TC5, **TC1**, TC7
reverse-greedy	**TC1**, TC7, TC4, TC7, TC3, TC6, TC2

Table 1.1: Test suite that tests a simple login and password verification system.

Label	Steps	Size
TC6	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: “Login not found” → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: “Login and password do not match”	12
TC7	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: “Login not found” → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: “Login not found”	9

Then, suppose the application of two basic prioritization techniques, considering test case size (the third column, labeled “Size”, in Table 1.1) as the test requirement and fault detection as prioritization goal: i) **greedy**, which chooses iteratively test cases with the highest number of steps and ii) **reverse-greedy**, which chooses iteratively test cases with the lowest number of steps. By applying these two techniques, we can obtain the orders presented in Table 1.2. Note that **greedy** places the failure represented by TC1 near to the end of the sequence – a poor result for a TCP technique. On the other hand, **reverse-greedy** places the failure in the first position – conversely, a desirable behavior for a prioritization technique. TC1 has the fewest number of steps along with TC7.

In this example, it is fairly easy to identify the factor that has influenced the results. However, for more sophisticated techniques and more complex test suites, it is not always obvious to identify all factors involved and how each particular technique can be affected by

them. To make TCP techniques applicable in practice, it is important to understand why one is more successful than other. Is the amount of steps in a test case the only factor to uncover fails first? Or are there more factors that influence the results?

Thus, our research problem is *to propose an execution order for system level model-based test suites, taking into consideration factors that could affect this result, aiming to detect faults as soon as possible, leading to a better use of resources during testing*.

In this work we investigate factors that affect TCP techniques in our research context [65]. We have evaluated whether the *amount of test cases that fail*, the *model layout* (represented by the amount of branches, joins, and loops) and *characteristics of test cases that fail* (if they traverse many or few branches, joins, or loops) affect the ability of revealing faults of a set of TCP techniques. We have set up three different empirical studies to evaluate these factors and we have discovered that the model layout does not affect significantly the techniques. However, the characteristics of the test cases that fail, specifically the amount of branches/steps they exercise, affect the investigated techniques. Therefore, it is important to explore different sources of information to leverage TCP and exploring other strategies than the ones investigated in the referred study. We discuss these findings in more details and a replication using industrial artifacts in Chapter 3.

Disregarding whatever previous information about faults and failures of the system may be available, most of the prioritization techniques are based on information about system models, test suites, or even the SUT code. Other source of information is the expertise of team members [82, 89] and other metrics from the development process, for example customer priority scores, and the occurrence of risk factors (such as loss of power, unauthorized user access or slow throughput) [79]. Note that considering the expertise of team members could be costly, since it would require much interaction [89]. Whereas this expertise may be valuable to indicate the occurrence of faults, one should be aware of the circumstances that it could be misleading, i.e. if there is no consensus among the team members, it could be more valuable not using this expertise.

Therefore, we proposed a simple process of collecting expertise of team members, such as developers and managers, about portions of the system they work on, and encoding it as what we named **hints**. They are indications that may point to the faults locations, such as portions of the system that some developer considered hard to implement, that uses some

external and/or unreliable library, or that suffered a schedule shortening. Besides, in order to use these hints, we proposed a technique called **H**int-based **A**daptive **R**andom **P**rioritization (HARP). It is based on the adaptive random strategy and takes into account the hints to suggest a new order for the input test cases. In our evaluation, HARP showed potential to suggest more effective test suites, in comparison with the original adaptive random prioritization technique [38]. On the other hand, HARP may be negatively affected by hints that do not point to potential problems. Trying to reduce the occurrence of these scenarios, we suggest using hints just when there is consensus among the team members.

In order to provide a complementary solution, indicated to the circumstances that we suggest not using hints, we explore another kind of strategy to solve TCP. As an example of a more refined strategy, clustering has been recently explored in the TCP literature [24,30,89]. Since it is essentially based on the notion of distance, also being explored in recent research [25], we can apply it without relying on historical information, which is the main aspect of our research context. Thereby, we proposed a second technique called **C**luste**RI**ng for **S**ystem test case **P**rioritization (CRISPy). Our evaluation suggests that it is a good complement to HARP, since it is effective in comparison to others techniques that also do not take into consideration specialists' expertise.

In the next section we discuss our research questions and the methods applied to answer them.

1.2 Research Questions and Methodology

We designed our research to explore the aforementioned aspects, through the following questions:

> *(RQ1) Does the model layout, represented by the number of branches, joins, and loops, affect the fault detection capabilities of the prioritization techniques in the investigated context?*

Many elements interact during the execution of an MBT TCP technique, such as: the system model, the technique itself and human intervention (if required). These elements may cause influence on the operation and on the result of a prioritization technique. Since

test cases are generated from system models in an MBT approach, the layout of this model affects directly the set of test cases and also may affect the performance of the techniques.

> *(RQ2) Do characteristics of test cases that fail affect the fault detection capabilities of prioritization techniques in the investigated context?*

As introduced by the previous research question, the conjunction of different factors may influence the performance of prioritization techniques. Characteristics of test cases that fail may privilege some techniques and jeopardize the performance of others, in other words, particularities of each technique could make them sensitive to different characteristics of test cases that fail. To evaluate that, we categorized faults and the related failures according to different characteristics and performed experiments with TCP techniques measuring their fault detection capability across different categories.

> *(RQ3) How can we systematize and collect expertise from the developers or/and managers, and use it to leverage Test Case Prioritization in the defined context, aiming to reveal faults sooner in the testing process?*

Intending to incorporate the expertise of the development team in the prioritization process, we defined a process based on use case documents to collect indications from developers and managers (the called ***hints***), related to functionalities more prone to fail. In turn, HARP uses these hints as a layer of guidance during its operation.

> *(RQ4) How can we prioritize test suites, even when the expertise available is not enough to guide TCP, i.e. what to do when this expertise would hinder instead of being useful?*

Based on the results obtained on recent researches, clustering is effective on representing the similarity between test cases. Therefore, as a complementary strategy, we proposed CRISPy aiming at gathering similar test cases and sampling them in order to reveal faults sooner during test execution.

1.2.1 Research Methodology

In order to answer these research questions, we conducted our research using a variety of investigation strategies, as depicted on the roadmap in Figure 1.2. The first main step was a literature review about the current techniques able to be used in our context. Besides some techniques designed directly for MBT, other general techniques that make weak assumptions about artifacts were adapted to our context. This task gave us an overview about what kinds of techniques were available.

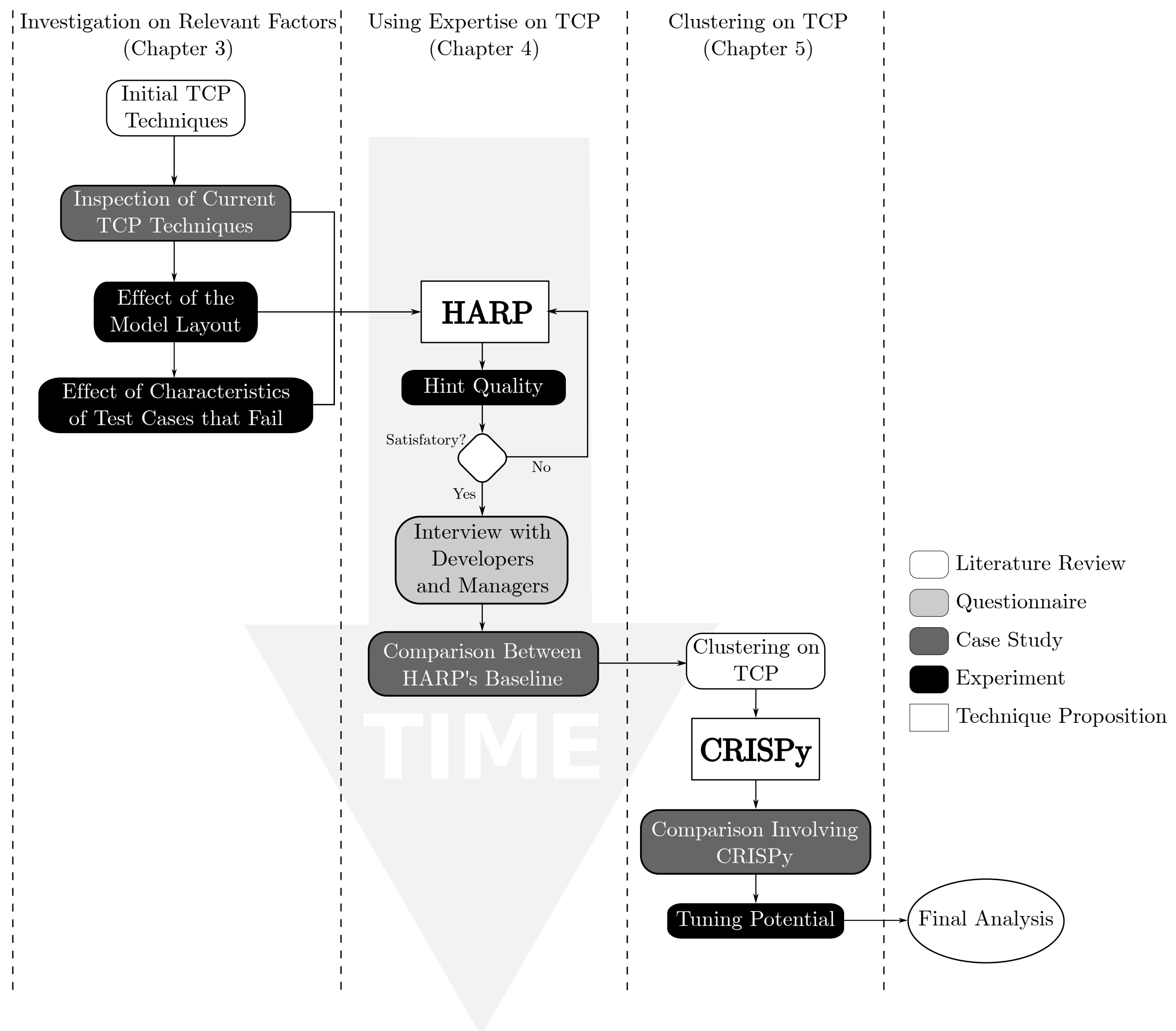


Figure 1.2: Overall flow of activities, artifacts and research lines.

In order to build our common ground regarding our operational environment, we implemented the techniques able to perform on the same set of premises as our scope definition, evaluation metrics, and the entire structure to collect and parse test suites and fault reports.

Then we carried out an exploratory case study comparing the aforementioned techniques, in order to have the first impressions on how they perform with industrial and academic artifacts. From that point, we realized the need for a deeper investigation about important factors that could affect the investigated techniques.

We performed that investigation on important factors resorting to artificial and industrial artifacts, always isolating as much as possible the involved factors on experiments, but every study still presented limitations, which were mitigated in a subsequent one. Even though we already have evidence supporting our claims, other factors remain to be studied and this research line is still open to contributions. Our findings encouraged us to use the expertise of team members as leverage for TCP in our context as hints, trying to avoid the downsides of current techniques; therefore we designed and implemented HARP. Its evaluation consisted of an experiment on the effect of hint's quality, followed by an iteration of refinement, and a case study using actual hints from development teams, collected through a questionnaire, comparing HARP to its baseline. Among the findings in this last study, we found out that there are circumstances that we should avoid using HARP, such as when hints do not point to faults, which is hard to avoid but we try to address it by collecting hints from people that actually worked/developed the use case in question, or when the hints could misguide HARP by the lack of consensus among the team members.

Investigating different strategies recently applied to solve TCP problem in our context, such as neural networks, clustering and genetic algorithms, we identified clustering as a good candidate to complement HARP, in other words, to be used in situations where HARP is not indicated. It is a good candidate because it is simple to implement and understand, at the same time, it represents clearly the notion of resemblance among elements in general, which is important to introduce variability. Then we designed and implemented CRISPy as a technique that clusters test cases to improve variability in TCP; its validation comprised a comparative case study involving a diverse selection of TCP techniques, including another clustering technique, which follows a different strategy; an experiment aiming at revealing CRISPy's tuning capabilities by varying clustering configuration parameters; and a comparison between HARP and CRISPy, illustrating their potential for complementation.

There is a clear relationship between the three vertical lanes depicted in Figure 1.2 and our main contributions, which we detail in the next section.

1.3 Contributions

We present several contributions throughout this document, but at this point, we highlight:

1. Our **empirical investigation** regarding the **important factors that affects TCP techniques**, including planning, data analysis scripts, and results;

2. A **systematic procedure to collect indications** from developers and/or managers, about error prone regions of use cases; and **HARP**, as a technique that use these indications;

3. **CRISPy**, as a TCP technique based on Agglomerative Hierarchic Clustering (AHC), able to operate when it is not indicated to use HARP;

4. Our **empirical investigation** regarding both **HARP** and **CRISPy**, also including planning, data analysis scripts, and results.

Besides, we developed a companion website to store artifacts related to our studies. It contains enough details to leverage the repetition of all of our studies by others researchers; it is available at `https://goo.gl/Tz7Jpx`.

1.4 Chapter Final Remarks

This introduction presented an overview of our entire research. Further, we discuss details about every research step in a dedicated chapter, according to the following organization.

Chapter 2 covers the main subjects involved in this research, such as software testing, model-based testing and the system modeling activity, and test case prioritization. Chapter 3 reports a sequence of studies investigating the effect of some factors on the performance of current TCP techniques. Chapter 4 details how to collect hints from the development team and to use them to provide another layer of guidance to the prioritization process, the HARP algorithm itself, and its evaluation. Chapter 5 describes design considerations about CRISPy, its algorithm, and the empirical investigation about its efficacy. Chapter 6 reviews the literature around the TCP problem, discussing the current findings and trends, and relating the proposed techniques with the context investigated in this thesis. Finally,

Chapter 7 discusses our conclusions about the whole research and points paths for future research.

Chapter 2

Background

This chapter provides the foundations about the topics covered in this research. The main topic approached here is Software Testing. This activity in the software development process might focus on the source code of the system under development or even in models that represent system's behavior [83]. From the model-based point of view, there is the particularity of modeling the behavior of a system and the automation of test case generation, which provides the test suites that exercise the system under development.

Some approaches are executed during software testing aiming to improve resource consumption; in our research, we focus on TCP, which contributes by seeking to satisfy sooner some quality criterion. Techniques apply different strategies to solve the TCP problem and we detail several of them used during our investigation. Furthermore, in order to investigate and evaluate these techniques we must use a systematic and empirical approach. Empirical Software Engineering focuses on answering research questions based on observing the objects of study, testing hypotheses, and analyzing them using statistical methods.

2.1 Software Testing

In software development processes, there are some tasks that aim to assess the quality of the product being developed, whose objectives are: to verify if the software satisfies the client's needs, a concept named **validation**, and to evaluate the outputs provided by the software against the ones defined in its specification document, which is the concept of **verification** [68, 78]. In current development processes, the majority of efforts to assess

the quality of the developed systems are applied on verification tasks, more specifically on Software Testing [78].

There are some definitions of software testing in the literature, but we can synthesize them in the following ones:

- According to Myers [56], software testing is the process of executing a program aiming to find errors;
- Craig and Jaskiel [17] state that software testing is a concurrent process in the life cycle of the Software Engineering, and focus on measuring and improving the quality of the system being tested.

The former one is simpleminded, in the sense that it affirms that the unique objective of testing is to find problems in the developed program. It is possible to obtain much more information with the test case execution, hence the later definition is more accepted for being more complete and in line with the current software development processes. Thus, the later one is broadly accepted in the literature by its completeness and alignment with the current software development processes.

The basic unit of the whole software testing activity is the **Test Case** (TC) [68]. Shortly, software testing techniques aim to select which test cases will be executed, exercising the SUT. Thereby, a test case comprises [41]:

- **Inputs**
  - preconditions: conditions that hold prior to the test execution;
  - steps: the inputs that exercise the system;
- **Outputs**
  - postconditions: conditions that hold after the execution of the test case;
  - expected results: the responses (numeric, textual, files) that the SUT must produce.

A test case that executes and provokes a wrong or unexpected output is part of the process of correcting the SUT. According to Jorgensen [41], an **error**, is when a person makes a

mistake while specifying or coding the system. The **fault** is the representation of the error in the artifact in question, and the **failure** is when a fault executes and an unexpected behavior of the system is observed. As an example of the application of the aforementioned concepts, consider that a developer makes an error and puts a wrong number in the source code. Then, a tester develops a test case that executes the fault, and the system produces a result different from the expected, thus the failure arises.

Regarding the awareness of internal details of the SUT, authors classify testing methods as [17, 41, 56, 78]:

- Functional or Black Box: the test that is based on the view where a system is considered as a function, which receives a set of values and produces a result. The black box notion comes from the "lack of visibility" with respect to the details of internal structures of the SUT, thus the tester just takes into account the inputs and the expected outputs;

- Structural or White Box: the test that is designed with full knowledge and awareness of SUT's internal structures. Jorgensen [41] suggests the metaphor of clear box, instead of white box, because the ability of "seeing inside the box".

None of these methods are better than the other, because they have different application contexts; instead, they are complementary. If the testing will be executed in a lower abstraction level element, as a class or a method, considering the conditional and iterative commands, the structural test is often applied. On the other hand, when considering elements with greater abstraction level, *e.g.* a component or even the whole system, taking into account the proposed results and their functionalities, the functional test is more indicated [7].

In Functional testing methods, test cases might be produced even before the software be developed, because they are based on the system's requirements [7]. These documents specify the requirements that the system must meet and, depending on the kind of development process, they are available since when the developers begin to code the particular requirement. If these documents are expressed using a formal, or even semi-formal language, they might provide some automation for the ST process, reducing costs and efforts. This premise is the basis for the subject of the next section, the model-based testing approach.

2.2 Model-Based Testing

MBT, is a functional testing approach that uses system models to conduct the activities of the testing process, for instance test case generation or evaluation of the acquired results from the execution [20]. According to Utting and Legeard [83], MBT is the automation of the design of functional tests.

Then, what is a model? According to Utting and Legeard [83], a model is a smaller representation of the system that will be developed, but it must be detailed enough to contain accurate descriptions about system's characteristics. Pezzè and Young [66] further state that a model must provide enough details for test case generation, as we will discuss later.

The general MBT process, depicted in Figure 2.1, begins by **modeling** (Step 1) the system based on its requirements. There are several notations to represent requirements, such as UML diagrams, State Machine Diagrams, Petri Nets or LTSs. We discuss the modeling activity in Section 2.2.1. Besides the aforementioned benefits, MBT also provides a way to support communication with clients.

In Step 2, the **generation** of test cases takes place. To do so, algorithms mainly based on graph search, usually depth-first search, traverse automatically the previously created model. The product of the generation process is a set of test cases, in other words, the test suite. At this point, this test suite is ready for manual execution, relying on the interpretation of the tester. However, in order to execute a generated test suite automatically, it should be augmented with more information, such as input data, extra code to adapt it to the SUT, and information about verdicts, through the **concretization** (Step 3).

Both kinds of test execution produce reports containing the reported faults and failures, i.e. test cases that reveal these faults, as result and regardless the kind of execution, testers still need to **analyze** (Step 4) these results, e.g. which portions of the SUT need debugging or refactoring for correcting faults.

In this thesis, we focus on manual test case execution, therefore, all studies we performed consider manually executed test suites, which means that our results are just valid to this context; besides, we do not discuss in detail test case concretization, since it is out of our scope.

Although all the benefits already discussed, MBT has limitations. It requires more exper-

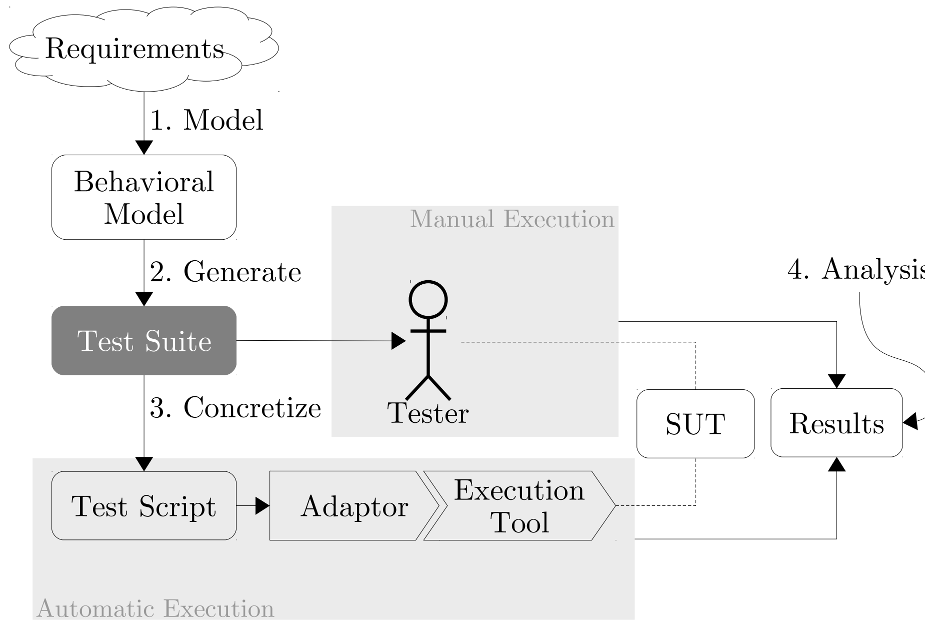


Figure 2.1: Activity flow of the Model-Based Testing process.

tise of the tester comparing with the manual testing process, since in order to generate good models, the tester must know in details every system requirement to be modeled, as well as the model notation. Thus, the quality of test suites is directly proportional to the quality of the models [83].

2.2.1 Modeling a System's Behavior

To model the behavior of a system is a way of expressing its execution flows. The first decision is to define the *abstraction level* of the model, in other words, which details of the SUT will be exposed/omitted [83]. In our work, since we focus on system testing, we deal with a higher abstraction level, detailing just the functionalities of the system, omitting inner control structures (iterative and conditional commands), as a functional testing strategy.

The next step is to select the kind of notation. There are some kinds of notation/languages to express models [83]:

- State-Based: the system is represented as a set of variables, whose values define its state. The operations over variables are stated through *pre* and *post conditions*. As example, we cite UML Object Constraint Language (OCL) and Java Modeling Language (JML);

- Transition-Based: when the system changes its behavior, a transition is used to represent this change. Typically, it uses a graphical notation of nodes and arcs, e.g. UML state machine and activity diagram, input/output automata and LTS;

- History-Based: the system is represented as a set of valid traces of its behavior over time. Some notions of time might be considered, as continuous/discrete or points/intervals, implying different temporal logics. As examples, we can list the System Description Language (SDL) and UML Sequence Diagrams;

- Functional: the system is represented as a set of mathematical functions, which provides support for theorem proofing. Normally it is more abstract and harder to write than other notations and it is not widely used in the MBT context;

- Operational: it describes the system as a set of executable and potentially parallel processes. It tends to be used to model protocols and distributed systems. As example, we can cite Communicating Sequential Processes (CSP) with an algebraic notation, and Petri Nets with a graphical representation;

- Statistical: it describes a system by a probabilistic model of its inputs and events. It is used more often to model environment instead of systems itself. As example, we can refer to Markov chains;

- Data-Flow: it concentrates on the data flow through the SUT, rather than the control flow. As example, there is the Matlab Simulink with their block notations.

The Transition-Based kind of notation is widely used because of its visual representation, availability of supporting tools, and it is best for control-flow representation [83]. In our research, we use LTS as our model representation. An LTS is defined in terms of states and labeled transitions linking them. Formally, an LTS is a 4-tuple $< S, L, T, s_0 >$, where [19]:

- S is a non-empty and finite set of *states*;

- L is a finite set of *labels*;

- $T \subseteq SxLxS$ is a set of triples, the *transition relation*;

- s_0 is the *initial state*.

The LTS, besides this algebraic representation, might be presented graphically and the elements are depicted in Figure 2.2. In the figure, we can see the composition of the elements of the formal definition: $S = \{1, 2, 3, 4, 5, 6\}$, $L = \{a, b, c, d, e\}$, $T = \{< 1, a, 2 >, < 2, b, 3 >, < 1, c, 4 >, < 4, d, 5 >, < 4, e, 6 >\}$ and $s_0 = 1$.

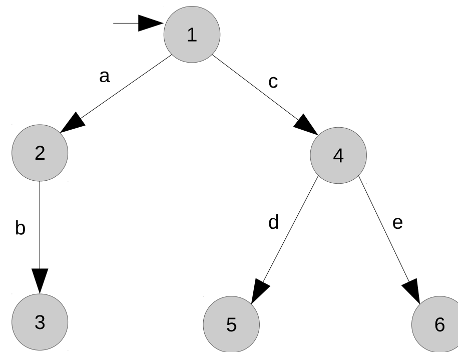


Figure 2.2: Elements of an LTS: The circles represent the states, the arrows represent the transitions and the state with an arrow without source, is the initial one.

To model a system using this notation, one may represent **user steps**, **system responses**, and **system conditions** as transition's labels. To represent a condition or decision, it is necessary to branch the state and add a transition representing each alternative. For the sake of notation, the transition label has reserved prefixes: i) *“S - ”* for user steps; ii) *“R - ”* for system responses; and iii) *“C - ”* for conditions. It is also possible to represent the junction of more than one flow and loops. In the context of our work, we use LTS to model use cases. A use case may present three kinds of flows: i) *base*, describing the most common use scenario that is completed successfully; ii) *alternative*, defining a user's alternative behavior or a different way that the user has to do something; and iii) *exception*, specifying the occurrence of an error returned by the system.

Figure 2.3 contains an example of a use case that verifies login and password, already discussed previously, modeled as an LTS, comprising one base and two exception flows. The first transition represents the initial action of the system, which is when the system shows its main screen. It can be interpreted as an initial condition for the operation of the system. In the next action, the user must fill in the login field and the following action, the system verifies if the filled login is valid. Depending on the validity of the provided login, the execution follows two different paths, expressed by the branch state labeled as “4”. If the

login is invalid, a loop step that shows an error message takes the execution flow back to the action of filling the login field; otherwise, the user must fill in the password field. Then, the system verifies if the provided login and password match; if they match, the system shows its main screen, otherwise it shows another error message and also takes the execution flow back to the state labeled as “2”.

Note that, if one traverses this LTS, it would follow the possible scenarios for this use case, involving the interaction with a user and possible system responses, which is what happens on test case generation.

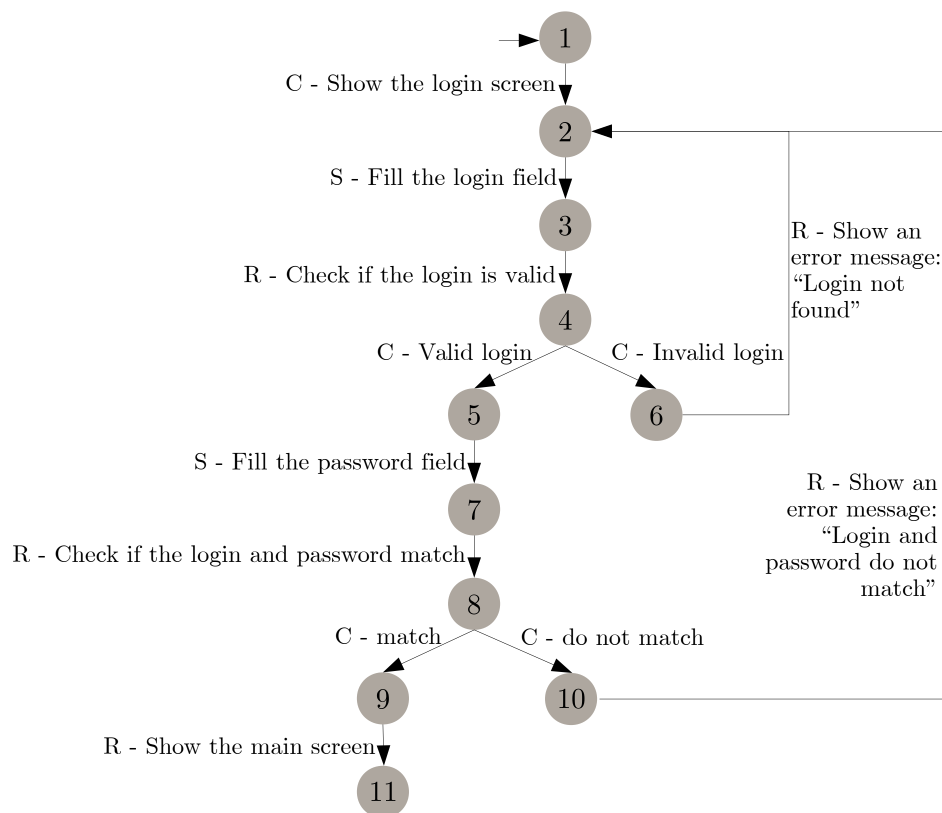


Figure 2.3: The LTS model representing the behavior of a login/password system.

2.2.2 Generating Test Cases

The test case generation step takes as input the specification model. The algorithm that effectively generates the test cases varies depending on the type of the notation that the system was modeled. Considering the usage of LTS in our work, generating test cases is equivalent to traverse the model using an algorithm of search in graphs, i.e. Depth-Search

First - DFS or Breadth-First Search - BFS. DFS is widely used instead of BFS due to its simplicity of implementation, using a recursive structure [14].

Formally, specifying the concept of test case presented in Section 2.1, a test case $tc = (l_1, l_2, \cdots, l_n)$ is a sequence of n labels, which represent model elements. Each l_i is a label representing *user steps* to be performed, *expected results* that the system must present for a given input, or *conditions* to be met before (precondition), during, and after (postcondition) the test case's execution. Note that, since the label comes from the model as result of its traversing, it already contains the prefix that indicates the kind of element it represents, as discussed in the previous section. Therefore, a test suite is a set of test cases, which means that all elements are unique and their order is not defined. Using the example model in Figure 2.2, a simple generation process would generate three test cases, $TS = \{(a, b), (c, d), (c, e)\}$.

A test case is a path calculated by the effect of traversing the LTS from its initial state. By using this notion, just a single loop transition would lead to the problem of the infinite test suite [83] and because of that, the algorithm must be equipped with a coverage criterion. Coverage criteria are ways to delimit the aspects of the model that the generation algorithm should consider, representing the intent of the testing and avoiding the infinite generation of test cases, acting as a stop criteria [2]. We can enumerate a list of coverage criteria types:

- Structural Coverage: the criteria are associated with the coverage of structural elements of the model, e.g. to cover every state/transition of the LTS or traverse each loop just once;

- Requirement Coverage: it is used to specify that the test suite must cover every test requirement, e.g. the generated test suite must cover every mention to a specific set of functionalities;

- Test Purpose: it defines which portions of the model must be explored and it is also known as explicit test case definition [35].

In order to increase the number of covered scenarios, we consider a test case as a path from the initial state of the LTS, either to another one with no child or a to another one that satisfies the stop criteria. In our research, we used a simple test case generation algorithm.

We used a DFS implementation to traverse the LTS recording the paths from its initial state, equipped with a structural coverage criterion called **all-n-loop-paths** [15, 83], which states that the algorithm should keep exploring the LTS until a loop be covered at most n times. As a way of keeping a balance between expressing some degree of repetition due to loop transitions on generated test cases, and still generating a feasible number of test cases, we considered $n = 2$ for our studies.

As an example, we generated the test suite discussed in Section 1.1 using this algorithm. For the sake of visualization, we bring the test suite here again, in Table 2.1. Note that, for instance, TC3 traverses twice the loop where the password for the provided login does not match. When the algorithm detects that it is traversing this loop for the second time, the exploration does not proceed, it stores the path as a test case, and it backtracks to the next path.

Table 2.1: Test suite generated from the LTS depicted in Figure 2.3.

Label	**Steps**
TC1	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - match → R - Show the main screen of the system
TC2	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: "Login and password do not match" → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - match → R - Show the main screen of the system
	It continues on the next page

Table 2.1: Test suite generated from the LTS depicted in Figure 2.3.

Label	**Steps**
TC3	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: “Login and password do not match” → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: “Login and password do not match”
TC4	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: “Login and password do not match” → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: “Login not found”
TC5	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: “Login not found” → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - match → R - Show the main screen of the system
TC6	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: “Login not found” → S - Fill the login field → R - Check if the login is valid → C - Valid login → S - Fill the password field → R - Check if the login and password match → C - do not match → R - Show error message: “Login and password do not match”
TC7	C - Show the login/password screen → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: “Login not found” → S - Fill the login field → R - Check if the login is valid → C - Invalid login → R - Show error message: “Login not found”

The result of the test generation process is the test suite, which is ready for manual test execution or concretization in an automatic execution context. However, due to limitations

already discussed, some processing might be performed between test case generation and execution, and the test case prioritization is one of them.

2.2.3 Test Purpose

Test purpose is a construct that provides the ability of indicating explicitly what are being tested. According to Vain *et al.* [84], a test purpose is a specific objective or property of the SUT that the development team wants to test. In practice, applying a test purpose to a test case generation process represents a guiding process, resulting in a smaller test suite in comparison to generating test cases for the whole system model.

There are different ways for representing test purposes. Jard and Jeron [35] present a formal approach to represent test purposes in conformance testing. This representation uses an extended version of an LTS, which includes input and output actions (IOLTS). The same format is also used to model the related SUT. Therefore, with the system and the test purpose respecting the same rules, the latter is able to guide the testing process through the former.

Cartaxo et al. [10] propose a test purpose also based on LTS models, but considering the "accept" or "reject" modifier. This notation is more flexible, since it does not need to define specific inputs and outputs, just the model steps represented by the edges of the LTS. Moreover, this representation can be also applied in the MBT context by considering system level test cases. It is expressed using the following guidelines:

- The purpose is a sequence of strings separated by commas;
- Each string is a label of an edge from the model or an "*", which works as a wildcard, indicating that any label satisfies the purpose;
- The last string is "accept", if the test case agrees with the test purpose definition, or "reject" otherwise.

Inspired by the idea of guiding the test case generation process, we introduce the **Prioritization Purpose** as a representation of a hint. Instead of acting in the test case generation, the prioritization purpose acts as a filter for natural language test cases, possibly generated from system models through MBT approaches. Its syntax is a variation of Cartaxo's test purpose [10]. We adapted it by omitting the last string, which is *accept* or *reject*, adopting

just the acceptance as default behavior. This modification was needed because we focus only on hints regarding error-prone regions of the system, regardless the safer (which would be represented by the rejection). Therefore, our prioritization purpose is defined as follows:

- A sequence of strings separated by a vertical bar or pipe ("|");
- Each string is the label of an element of the system model, e.g. a user step of a use case, or a "*", which is a wildcard, representing any label.

Our proposed TCP technique called HARP takes as input a set of prioritization purposes, representing hints, and during its operation, a purpose filter defines two sets: one with test cases that satisfy the prioritization purpose (or related to the hints) and other with test cases that do not satisfy the prioritization purpose (or not related to the hints).

As an example of this filtering process, for the test suite introduced in the previous chapter and also in Table 2.1, consider the prioritization purposes **"* | C - Valid login | *"** and **"C - match | R - Show the main screen | *"**. For the first one, we can see that there are six test cases that satisfy it: TC1, TC2, TC3, TC4, TC5, and TC6. In turn, the second one will not accept any test case, because, even though it has only valid labels, no test case begins with **"C - match"**.

2.3 Test Case Prioritization

The order that the test cases were generated (automatic approach) or listed (manual approach) may not be interesting to the execution purposes. To illustrate that:

1. it could be necessary to spend much time preparing the preconditions for the test cases along the execution [53];
2. in the considered initial order, faults could take too long to be revealed;
3. the test requirement required to stop the execution process could take too much time to be met.

TCP techniques reorder test cases from a test suite in order to achieve a certain objective as soon as possible in the testing process. In this research we focus on fault detection, in other

words, we intend to reorder test cases to find faults sooner in the testing process. Figure 2.4 expresses a motivation for applying TCP techniques.

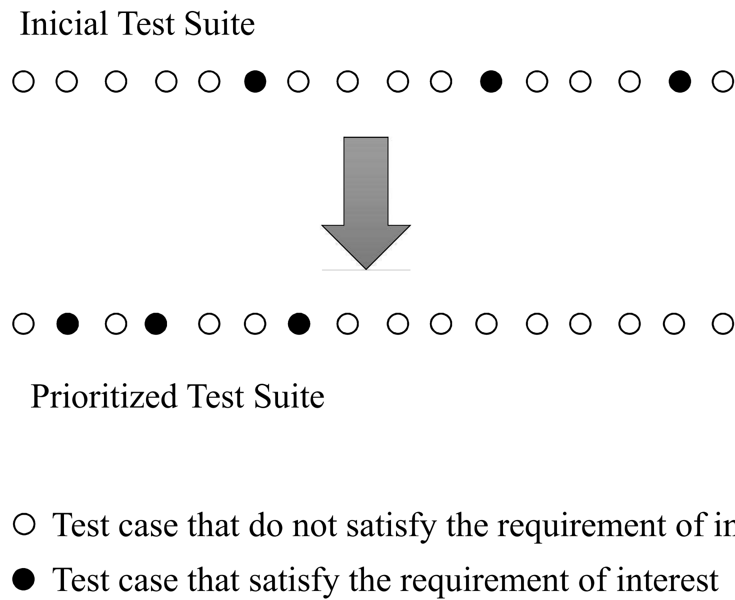


Figure 2.4: Motivation for the application of the test case prioritization.

Formally, we can define the test case prioritization problem as [21, 36, 39]:

Given: A test suite TS, the set PTS of all permutations of TS, and a function $f : PTS \to \mathbb{R}$.

Problem: To find a $TS' \in PTS \mid \forall\, TS'' \in PTS\ (TS'' \neq TS') \cdot f(TS') \geq f(TS'')$.

In other words, for the set of all permutations of the available test cases, the problem is to select the one that maximizes the evaluation function, i.e. a function that measures how good a prioritized test suite performs regarding the referred objective.

TCP has two clear limitations: i) to analyze the whole set of all permutations of TS; and ii) to define the evaluation function f. Since the number of permutations of a set A is equals to $|A|!$, if TS is big, analyzing the elements from PTS might demand an unfeasible amount of resources, as discussed by Lima et al. [53]. Moreover, the evaluation function f is related to the prioritization goal, for example, to reduce test case setup time [53] and, more commonly, to accelerate fault detection. Depending on the prioritization goal, the required information to define f is not available beforehand, making its accurate/precise definition impossible. For instance, considering the prioritization objective explored by Lima et al. – to reduce the configuration time of the whole test suite – the required information to calculate it

is already available in the test suite itself, which are the inputs and outputs of each test case. Nevertheless, considering early fault detection, the information regarding where the faults are or which test cases detect them is not available in advance, then a technique is not able to maximize it. Therefore, techniques that focus on fault detection, estimate fault information through surrogates [89].

TCP can be applied in code-based and model-based contexts, but it has been more applied in the code-based context and it is often related to regression testing, as suggested by Catal and Mishra [12]. This way, Rothermel *et al.* [72] proposed the following classification:

- General test case prioritization - TCP is applied any time in the software development process, even in the initial testing activities;
- Regression testing prioritization - TCP is performed after a set of changes was made. Therefore, it can take into consideration information gathered in previous runs of existing test cases to help prioritizing the test cases for subsequent runs.

The work presented in this thesis focus on general test case prioritization; in the next chapters we will use the term Test Case Prioritization in the general context. For our research, we studied and implemented several techniques and evaluation metrics to integrate our operational environment to perform our empirical studies and we detail them in the next section.

2.3.1 Studied Techniques

As the first step of the investigation, we performed a systematic mapping in the related literature aiming to know the techniques and evaluation strategies applied to the context we are investigating. Details about the protocol, execution and results are in a technical report [62], also available at our companion site (refer to Section 1.3)[1].

In this section we provide an overview of every technique explored in this thesis. Some of them was initially proposed and evaluated for code-based testing but, because of loose requirements regarding the SUT source code, they underwent adaptations to the MBT context, making some assumptions.

[1]Direct link: `https://goo.gl/wAFV3J`

STOOP

Kundu *et al.* [48] proposed a technique called **Stoop**, based on UML Sequence Diagrams. It has its operation illustrated by Algorithm 1.

Algorithm 1 Execution flow for Stoop

```
1:  function STOOP_PRIORITIZE(SDS)
2:      SG ← ∅
3:      for sd ∈ SDS do
4:          SG ← SG ∪ convertSDtoSG(sd)
5:      end for
6:      fullSG ← mergeSG(SG)
7:      testCases ← generateTestCases(fullSG)
8:      testCases ← removeSpuriousPaths(testCases)
9:      prioritizedTS ← sortDescendingOrder(testCases, AWPL)
10:     return prioritizedTS
11: end function
```

In Lines 2, 3 and 4, the sequence diagrams passed as input are converted into sequence graphs (SG). SG is a graph that considers just the messages between boundary elements of the system (*e.g.* user, screen, speakers, mouse, and sensors), excluding the internal actions of the modeled system. Then, the SGs are merged into a single one in Line 5, corresponding to the total boundary actions of the system. This merging process may create paths in the final SG that are not specified in the SGs separately, which are the spurious paths; they must be excluded after comparing with the individual SGs.

The authors merged the test case generation into the algorithm, which is performed in Line 7. It is based on the DFS algorithm, instrumented with the **one loop path**, which implies that a loop must be traversed at most once. After that, the algorithm eliminates spurious paths in Line 8.

After that, in Line 9, the test cases are prioritized based on structural metrics. The chosen metric is the Averaged Weight Path Length - $AWPL$, which emphasizes the importance of the most common steps in the test case execution. The authors assume that common steps need to be well tested because they will be frequently exercised [48]. Therefore, the

algorithm sorts the test cases in descending order by their AWPL values. The algorithm calculates this metric using the expression $AWPL(tc) = \frac{\sum_{i=1}^{m} eWeight(e_i)}{m}$, where e_i is the $i-$th edge of tc, m is the size of tc, and $eWeight$ is the weight of en edge, which is the number of test cases that traverse e_i.

StringDistance

Proposed by Ledru *et al.* [49], and better detailed and evaluated later by the same research group [50], **StringDistance** introduces a general strategy based on distance concept and uses some distance functions, evaluating the string representation of the test cases to be prioritized. We depict its operation in Algorithm 2.

Algorithm 2 Execution flow for StringDistance

```
1:  function STRINGDISTANCE_PRIORITIZE(TS)
2:      prioritizedTS ← ∅
3:      calculateDistances(TS)
4:      nextTC ← maxMinDistance(TS, TS)
5:      TS ← testsuite − nextTC
6:      prioritizedTS ← prioritizedTS ∪ nextTC
7:      while TS ≠ ∅ do
8:          nextTC ← maxMinDistance(TS, prioritizedTS)
9:          TS ← testsuite − nextTC
10:         prioritizedTS ← prioritizedTS ∪ nextTC
11:     end while
12:     return prioritizedTS
13: end function
```

The technique uses string distances as guide, in order to select the next test case to be placed in order. The algorithm calculates pairwise distances among the test cases in Line 3. In Line 4, the main routine calls a function $maxMinDistance(TS, testsuite)$ to select the next test case. It selects the test case among the initial test suite with the minimum distance to the other ones; then in Lines 4 and 5, it moves this test case to the prioritized test suite.

Lines 7 to 11 comprise the main loop of the technique: while there is some test case

not yet placed in order, the next test case $t \in testsuite$ to be place in order is the one that maximizes the minimum distances between the test cases from $prioritizedTS$.

The authors propose using four functions to calculate the string distance:

- Hamming distance: it compares two strings of equal length and its value is the number of tokens that differ;
- Edit distance: it calculates the smallest number of editions needed to transform one arbitrary string in another one. It uses a dynamic programming approach to calculate every transformation possibility and select the smallest one;
- Cartesian distance: it compares two strings of equal length n according to the ASCII values of their characters. Its value is given by the expression $Cart(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$, where $X = x_1, x_2 \ldots, x_n$ and $Y = y_1, y_2 \ldots, y_n$ are the compared strings;
- Manhattan distance: it is similar to the Cartesian distance, and it is calculated by $Man(X,Y) = \sum_{i=1}^{n}|x_i - y_i|$, where $X = x_1, x_2 \ldots, x_n$ and $Y = y_1, y_2 \ldots, y_n$ are the compared strings.

The authors proposed and evaluated this technique for code-based test suites, but for our context, we assumed a string representation of system level model-based test cases, formed by concatenating the labels of their steps, as a substitute for the code-based test cases and then added it to our experimental setup.

FixedWeights

Proposed by Sapna and Mohanty [75], it prioritizes test cases generated from UML Activity Diagrams. The reason of the technique's name is the assignment of nodes and edges weights. The authors assume that the test cases that represent the main scenarios of the system must be tested first. Thus, the technique treats the test cases with lower number of steps and fewer branches and loops as the main scenarios. The weights of the nodes are fixed and guided by the original activity diagram element. We summarize it in Algorithm 3.

The first step (Line 2) is to convert the activity diagram into a tree intending to eliminate loops. For that, it applies a modified DFS, which **traverses a loop at most twice**, which is

Algorithm 3 Execution flow for FixedWeights

```
1:  function FIXEDWEIGHTS_PRIORITIZE(AD)
2:      tree ← convertActivityDiagram(AD)
3:      assignNodeWeights(tree)
4:      assignEdgeWeights(tree)
5:      TS ← traverse(tree)
6:      for all tc ∈ TC do
7:          calculateWeight(tc)
8:      end for
9:      prioritizedTS ← sortAscendingOrder(TS, weight)
10:     return prioritizedTS
11: end function
```

similar to the generation process we introduce in Section 2.2.2. Then, the algorithm assigns node weights in Line 3 and, considering the node v, its weight is:

$$W(v)\begin{cases}3 & \text{fork/join nodes}\\ 2 & \text{branch/merge nodes}\\ 1 & \text{activity nodes}\end{cases}$$

The next step is to calculate the weights of the edges (Line 4). The weight of the edge $e = \{v_1, v_2\}$ is calculated by the expression $W(e) = (v_1)_{in} \cdot (v_2)_{out}$, with $(v_1)_{in}$ representing the number of incoming edges of the node v_1 and $(v_2)_{out}$ the number of outgoing edges of v_2. After that, the algorithm uses a simple DFS to traverse the tree and derive the test cases in Line 5.

The node and edge weights are used to calculate the weight of a test case (Line 6). Considering the test case tc with x nodes and y edges, $W(tc) = \sum_{i=1}^{x} W(v_i) + \sum_{j=1}^{y} W(e_j)$, i.e. the test case weight is the sum of the weights of its nodes and edges. Then, in Line 9, the algorithm sorts the test cases in ascending order by their weights.

Algorithm 4 Execution flow for PathComplexity

```
1: function PATHCOMPLEXITY_PRIORITIZE(AD)
2:     cfg ← convertActivityDiagram(AD)
3:     calculateNodeWeights(cfg)
4:     TS ← generateTestCases(cfg)
5:     for all tc ∈ testsuite do
6:         travNodes ← numberOftraversedNodes(tc)
7:         weight ← calculateWeight(tc)
8:         predNodes ← numberOfPredicateNodes(tc)
9:         logConditions ← numberOflogicalConditions(tc)
10:        tcComplexity = travNodes + weight + predNodes + logConditions
11:        complexities.put(tc, tcComplexity)
12:    end for
13:    prioritizedTS ← sortDescendingOrder(TS, complexities)
14:    return prioritizedTS
15: end function
```

PathComplexity

Proposed by Kaur *et al.* [43], this technique prioritizes test cases generated from UML Activity Diagrams. The first step is to convert the input diagram into a control flow graph. After the conversion, the prioritization process is based on the nodes of the generated graph and on the **FANIN(v)** and **FANOUT(v)** functions. The former is the number of incoming flows to the vertex v and the latter is the number of outgoing flows from the vertex v. We describe the technique in Algorithm 4.

The first step, in Line 2, is to convert the activity diagram into a control-flow graph (CFG), keeping the kind of the equivalent element from the input activity diagram. In Line 3, the algorithm calculates the weight of the CFG's vertexes, through to the expression $weight(v) = FANIN(v) \cdot FANOUT(v)$.

In Line 4, the algorithm generates the test cases from the CFG, traversing a loop at most once. In Lines 5 to 12, the algorithm calculates the complexity of each generated test case. The Lines 5 to 12 contain the calculation of four properties from a test case, that are

the number of traversed nodes, the sum of the weights of the nodes, the number of predicate nodes and logical conditions, respectively. A predicate node is a node with multiple outgoing edges and a logical condition is an edge that represents a logical condition satisfied. Thus, the algorithm sorts the test cases in descending order by their complexities in Line 13.

Total and Additional Coverage of Steps

Elbaum et al. [22] suggested using a greedy strategy to prioritize test cases. The total approach (**ST**) sorts the test cases by the total amount of code statements that they cover. Moreover, the additional one (**SA**) also sorts them, but adjusts iteratively the statements already covered by the currently test case sequence, i.e. the next test case in order is the one that covers more statements not yet covered so far. Since our test suites are based on high abstraction steps instead of code statements, we adapted the techniques for our context by considering the coverage of steps.

Adaptive Random Prioritization

This strategy distributes the selected test case as spaced out as possible based on a distance function [13]. During the execution of a technique that follow this strategy, it keeps two collections of test cases: the prioritized sequence and the candidate set (the set of test cases that are randomly selected without replacement). Besides, it is also based on varying the test cases by selecting the **farthest away** between the candidates and the already prioritized ones. We represent the general adaptive random strategy in Algorithm 5.

In lines 2 to 5 the algorithm selects randomly the first test case to be place in order. Then, from Line 6 to 9, it performs the main loop, which composes randomly a candidate, limited by 10 test cases, and selects the next test case to be placed in order. The next test case comes from the candidate set; the algorithm assembles a matrix with the minimum distances between the candidates and the already prioritized and selects the candidate with the maximum distance.

The general strategy is based on a distance function and in this thesis, we explored three of them:

- **Jaccard distance**: The use of this function in the TCP context was proposed by Jiang

Algorithm 5 The general algorithm for the adaptive random prioritization.

1: **function** ART_PRIORITIZE(U_TS)
2: $prioritizedSequence \leftarrow \emptyset$
3: $firstChoice \leftarrow randomSelectOne(U_TS)$
4: $prioritizedSequence.append(firstChoice)$
5: $U_TS.remove(firstChoice)$
6: **while** $U_TS \neq \emptyset$ **do**
7: $candidateSet \leftarrow$ ***generateCandidateSet***(U_TS)
8: $nextTestCase \leftarrow$ ***selectNextTestCase***$(prioritizedSequence, candidateSet)$
9: **end while**
10: **return** $prioritizedSequence$
11: **end function**

et al. [38]. Let t_1 and t_2 be two test cases, and $steps(t)$ as the set of steps covered by the test case t, the jaccard distance function is:

$$Jac(t_1, t_2) = \frac{|steps(t_1) \cap steps(t_2)|}{|steps(t_1) \cup steps(t_2)|}$$

- **Manhattan distance**: Zhou [91] applies this strategy using Manhattan function to compare the distance between test cases. Consider two test cases t_1 and t_2, represented by $t_1 = (S_{11}, S_{12}, \cdots, S_{1i})$ and $t_2 = (S_{21}, S_{22}, \cdots, S_{2i})$, which are the steps covered by the two test cases respectively. Both sequences have the same length i, which is the total number of steps in the system model. Moreover, each $S_{xy} \in \{0, 1\}$, where 1 indicates that the test case x traverses the step y, 0 indicates otherwise. Thus, the function is defined as follows:

$$Man(t_1, t_2) = \sum_{x=1}^{i} |S_{1x} - S_{2x}|$$

- **MBT Similarity**: Coutinho et al. [15] proposed a function to measure the similarity between model-based test cases and applied it on test suite reduction. Since it is already suitable to evaluate test cases in the context we investigate, we contribute with the literature on TCP by applying this function on ARP. This function is defined as follows:

$$Sim(t_1, t_2) = \frac{nip(t_1, t_2) + |sit(t_1, t_2)|}{\left(\frac{|t_1| + |t_2| + |sdt(t_1)| + |sdt(t_2)|}{2}\right)}$$

Where:

- t_1 and t_2 are two test cases;
- $nip(t_1, t_2)$ is the number of identical transition pairs between two test cases;
- $sdt(t)$ is the set of distinct transitions in the test case i;
- $sit(t_1, t_2)$ is the set of identical transitions between two test cases, i.e. $sdt(t_1) \cap sdt(t_2)$.

Clustering

Clustering is the idea of grouping similar elements and, as consequence, creating categories or clusters [86]. It also includes the notion of distance or similarity between elements as the way of creating these categories. These clusters could be exclusive, so that an element belongs just to a single cluster, or probabilistic, when an element belongs to more than a cluster, but with certain probability. These elements are usually numeric arrays representing metrics (or indexes) of some real-world entity of interest, and the evaluation functions, e.g. Euclidean distance, establish the relationship between them. However, it is also possible to use the entities itself, along with a way of comparing them.

In TCP, there are some techniques that use clustering as a strategy to reorder test cases. We detail a technique that clusters elements of a transition-based model and reorders test cases based on these groups, and other that clusters test cases directly.

Clustering Elements of the Model A research group has shown a long-term interest in the clustering strategy. They proposed three techniques using it: Fuzzy C-Means (FCM) [8], Gustafson Kessel Clustering (GK) [9] and Neural Network Based Clustering [28]. These three papers consider the same context: the system is modeled through an Event Sequence Graph (ESG), which is a graph with a definition similar to the LTS. It is composed by events, which are the nodes of the ESG, and arcs, corresponding to the edges. Therefore, test cases

are generated from an ESG in a similar fashion than from an LTS. These three techniques follow a common reasoning, represented by the Algorithm 6:

Algorithm 6 Execution flow for the techniques that cluster elements of the model.

```
1:  function CLUSTERING_PRIORITIZE(ESG)
2:      TS ← generateTestCases(ESG)
3:      events ← getEvents(ESG)
4:      calculateIndexes(events)
5:      numCLusters ← calculateNumberOfClusters(events)
6:      clusters ← generateClusters(events)
7:      calculateImportanceLevel(clusters)
8:      preferenceDegrees ← calculatePreferenceDegree(TS)
9:      prioritizedTS ← sortDescendingOrder(TS, preferenceDegrees)
10:     return prioritizedTS
11: end function
```

The algorithm also includes the test case generation in Line 2. After that, it calculates the indexes associated with these events in Line 3. These indexes are a set of functions that evaluate the structure of the model, e.g. the number of incoming and outgoing edges associated with the event and the number of occurrences within the generated test cases. In the paper, the authors suggest a set of eleven functions, but they mention that other functions might be used. The algorithm defines the number of clusters $c = 4$ for their evaluations. On the other hand, they argue that one can use other strategies to calculate that, for instance the *index-based cluster validity algorithm* [44].

The difference between these three techniques is just the clustering algorithm, which is called in Line 6. The first one uses **Fuzzy C-Means**, which is a variant of the classic clustering algorithm K-Means. Here, clusters are probabilistic and the event will be in the cluster with higher probability, which is the membership degree. Once the distance of each element to the center of the cluster is a straight line, given by the Euclidean distance, every cluster assumes a spherical format.

The **Gustafson Kessel Clustering** is an evolution of the Fuzzy C-Means by employing another way to represent the center of a cluster and the distance measure. Thus, this

algorithm might identify clusters with different shapes, as an ellipse. The third mentioned technique is **Neural Network-Based Clustering**. It uses an unsupervised learning algorithm that minimizes the euclidean distance from the elements to the centers of the clusters. Due to the simplicity of implementation, we just implemented the **Fuzzy C-Means** variant on our structure.

Clustering Test Cases The other technique that uses the clustering strategy is the Interleaved Clustering Prioritization (ICP) [89]. Instead of model elements, this technique cluster test cases directly, which weakens the requirements about the formalism and the general structure of test cases. It is called interleaved because it performs clustering and sorting process in two levels: inside of each cluster (intra-cluster) and between the clusters (inter-cluster), and then the algorithm interleaves test cases from the clusters, defining the new order.

The technique uses the expertise from the testers and developers to evaluate the clusters' degree of importance. The clustering method is a simple agglomerative hierarchical clustering algorithm, detailed in Algorithm 7

Algorithm 7 Agglomerative hierarquical algorithm.

1: **function** AH_PRIORITIZE(TS)
2: $dendogram \leftarrow \emptyset$
3: $clustersList \leftarrow createClusters(TS)$
4: $dendogram.addLeaves(clustersList)$
5: **while** $clustersList \neq \emptyset$ **do**
6: $minDistanceClusters \leftarrow getMinimumDistanceClusters(clustersList)$
7: $newCluster \leftarrow minDistanceClusters[0] \cup minDistanceClusters[1]$
8: $clustersList \leftarrow clustersList - minDistanceClusters$
9: $clustersList \leftarrow clustersList \cup newCluster$
10: $dendogram.addParentNode(newCluster, minDistanceClusters)$
11: **end while**
12: **return** $dendogram$
13: **end function**

The algorithm keeps a tree of clusters (dendogram) and, in Line 3 and 4, it adds each

unitary cluster as a leaf of the dendogram, i.e. at this point, the dendogram contains one cluster for each test case. The loop in Lines 5 to 11 finds two clusters with the minimum distance between them, let c_1 and c_2, based on hamming distance function between the code-based test cases. These two clusters take part of a new cluster, which is added into the tree as a parent node for c_1 and c_2.

Once with the clusters assembled, a set of pairwise comparisons involving test cases from different clusters are used to sort them. Experts are expected to perform these comparisons in order to define an importance order among the clusters. The authors show an expression calculating the number of comparisons needed to sort the clusters, which is $\frac{k(k-1)}{2}$, with k being the number of clusters. They consider the maximum number of $k = 14$ in order to achieve the number of 91 comparisons, since they aim for a value lesser than 100. The ordered sequence of clusters is the input for the interleaving process.

Considering OC as an ordered cluster of test cases, OOC as the ordered sequence of ordered clusters and $OOC_i[j]$ as the j-th test case from the i-th ordered cluster, the interleaving process happens according the Algorithm 8.

Algorithm 8 Interleaving test cases.

1: **function** INTERLEAVE_CLUSTERS(OOC)
2: $OTS \leftarrow \emptyset$ (OTS is the ordered test suite)
3: $i \leftarrow 1$
4: **while** $OOC \neq \emptyset$ **do**
5: $OTS \leftarrow OTS \cup OOC_i[1]$
6: $OOC_i \leftarrow OOC_i - OOC_i[1]$
7: **if** $OOC_i = \emptyset$ **then**
8: $OOC \leftarrow OOC - OOC_i$
9: **end if**
10: $i \leftarrow (i + 1)$ mod $size(OOC)$
11: **end while**
12: **return** OTS
13: **end function**

All techniques already mentioned in this section have intersection with our research con-

text and compose the validation framework for our new proposed techniques. Along with them, we implemented a set of mechanisms to assess their effectiveness, related to the prioritization objective. In the next section we will detail the evaluation metrics to assess the fault detection capability proposed in the literature.

2.3.2 Evaluation Metrics

Besides TCP techniques, our validation framework also contains two metrics originally proposed to evaluate fault detection of TCP techniques, regardless of whether they are code or model-based, general or regression TCP. In the literature, two metrics are usually used to assess effectiveness regarding fault detection: F-Measure, which is, given the execution order, the number of test cases executed until the first failure happens; and the Average Percentage of Fault Detection (APFD), which expresses how rapidly a prioritized test suite detect faults.

F-Measure is straightforward because it just counts the number of test cases executed before the first failure is revealed. For a test suite $ts = \{tc_1, tc_2, \cdots, tc_n\}$ with n test cases, assuming that the fifth test case reveals the first fault, $F{-}Measure(ts) = 4$. Therefore, the metric varies between 0 when the first test case already fails, and $n - 1$ when just the last test case fails. As consequences, the metric only considers the first failure, which could oversimplify the effectiveness analysis, but still usable to evaluate failures instead of faults, i.e. the mapping between test cases and the faults they reveal is not available.

In addition, APFD is more powerful since it takes into account all fault able to be discovered, as well as the test cases that reveal them. Rothermel et al. [72] introduced this metric and since then, several studies has been using APFD to assess the fault detection of TCP techniques, usually comparing techniques directly [30, 33, 38, 49, 50, 54, 82, 89, 90].

Let ts be the same test suite introduced in this section, containing n test cases, TF_i the index of the first test case that detects the i-th fault, and m the number of faults able to be detected by ts, the expression is:

$$APFD(ts) = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{n \cdot m} + \frac{1}{2n}$$

Besides the numerical result (an average), APFD can be visualized as the area under the curve representing the relation between the test cases on the prioritized test suite (x-axis) and

the percentage of the faults revealed so far (y-axis). APFD values range from 0 to 1, such that higher values indicate that the evaluated test suite reveals faults *sooner*, on the other hand, lower values mean the faults are revealed later as we execute the entire test suite. Figure 2.5 summarizes the visual interpretation of APFD. The left image depicts a standard value for APFD, which is 0.5, for a visual reference. If 100% of the faults are revealed earlier, the associated APFD increases, as depicted on the right image. Note that in the left figure the test suite does not reveal any fault until 10% of its test cases are executed, whereas the right figure shows a more desirable result for a prioritized test suite where faults are found sooner, such that by executing 70% of the test cases all faults are revealed.

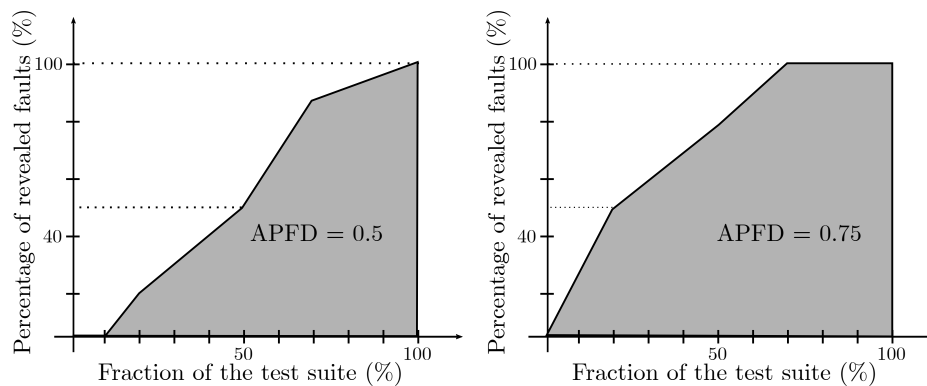


Figure 2.5: Visual interpretation of two scenarios of APFD. The area on the second chart (right) is bigger, indicating that the specific order of test cases in that test suite allows testers to reveal faults sooner.

Ideally, it is possible to compare two APFDs visually, because it enables to spot differences between the general shape of the curves. However, visual analysis is not feasible on the analysis performed in this thesis, since we compare several techniques, test suites and trials.

Thus, on the empirical studies we performed, we relied on central tendency measures and statistical tests to answer our research questions. In the next section we discuss relevant aspects on empirical research in software engineering and data analysis.

2.4 Empirical Research in Software Engineering

The software development process is not an easy activity, because it is a set of human effort, well-defined methods and even art, either for small projects or for the bigger ones. In the late sixties, aiming to mitigate problems, such as the lack of quality and satisfaction of the client's requirements and bad estimation of budget, a set of good practices were established and used in different companies. The term *Software Engineering* was coined with the objective of gathering these practices and make them patterns, supporting the software development [78].

The development and maturation of the software engineering leaded to a need for deeper understanding of their inner relationships and components [5]. Therefore, practitioners needed to conduct scientific researches, measuring, comparing and observing behaviors and artifacts. The research based on the observation of phenomena is the **empirical research**. Depending on the purpose and the level of control, it is possible to classify the empirical research into three types: the survey, the case study and the experiment [87].

The *survey* is frequently applied in the social sciences context [4] and, in the software engineering context, it is applied in research that involves methods, tools or techniques that were used sometime in the past. This kind of research might be performed to describe characteristics from a population, explain some past event or act as an introduction for a deeper and more focused study [87]. The most common way to gather data in the survey is an interview form [4, 87], that might be answered by the researcher or through an interview, conducted by a member of the research team.

In turn, a *case study* is a research method that aims to evaluate a single entity or phenomenon in its real-life context [45]. Case studies are suitable to compare techniques in an everyday situation, using typical configurations and artifacts; since they are based on real scenarios, they have more power to present evidence about the particular investigate scenario then the surveys, on the other hand they do not have much power of generalization [87].

In the *experiment*, the researcher has the control of the studied variables and of the allocation of the elements to the subjects, which often leads the experiments to be performed in a laboratory. Through the experiment, we seek to verify cause-effect relations in the world, observing the variables that reflect them. It is suitable to confirm theories, explore relationships among variables, validate models and measures, and even confirm elements from the

popular culture [87].

In Table 2.2 there is a comparison between the empirical research methods, regarding control of the execution and measurement, cost of the investigation and replication capability. **Execution control** relates to how much control the researcher has over the study execution. For instance if during a case study the development team decides to change drastically the objective of the current delivery due to some client's new demand, the researcher could need to restart or even stop the study; on the other hand, in a controlled experiment, the researcher has total control of its execution. Regarding **measurement control**, it means how free the researcher is to define the measures to be collected. For example, in a survey, a researcher can define some measures, but in the end, he or she still depends on the people's opinions and responses.

Table 2.2: Characteristics of the kinds of empirical research [87].

	Survey	**Case Study**	**Experiment**
Execution control	No	No	Yes
Measurement control	No	Yes	Yes
Investigation costs	Low	Medium	High
Ease of replication	High	Low	High

The two other aspects are very related, which are the **investigation cost** and the **ease of replication**. Both relate to how much resource the investigation requires. Usually, case studies could cost less than experiments, since the investigator follows its execution as an observer and keeps interaction as subtle as possible. On the other hand, replication in case studies could be a challenge because, besides the resources needed for a replication, could be hard to evaluate a similar context than the original study.

The major pillar of the empirical research is the hypotheses testing, which aims to reject or obtain evidences about the validity of hypotheses postulated based on observing the world [26]. We test them through statistic methods and by applying the concept of statistical significance. The only action we should take regarding a hypothesis is to reject it or not; it can sound simple but, when one is not able to reject a hypothesis, it does not imply that it should be accepted. We provide some guidelines on data analysis, hypothesis testing, and effect size analysis in the next section.

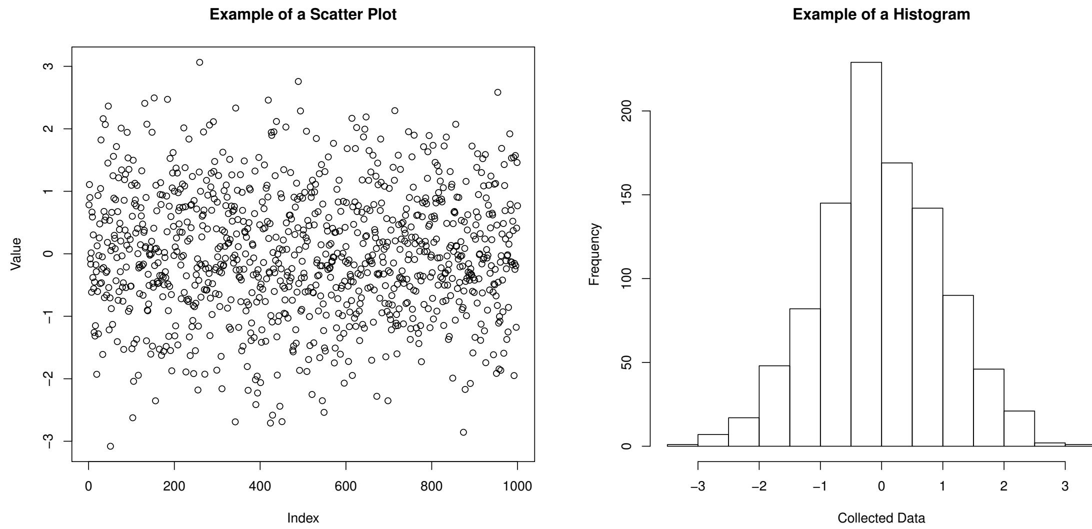
2.4.1 Data Analysis

As a first approach upon the collected data, the researcher usually applies descriptive statistic analysis to provide a simplified, but yet meaningful overview about the data. Among the most important tools, some of them deserve to highlight:

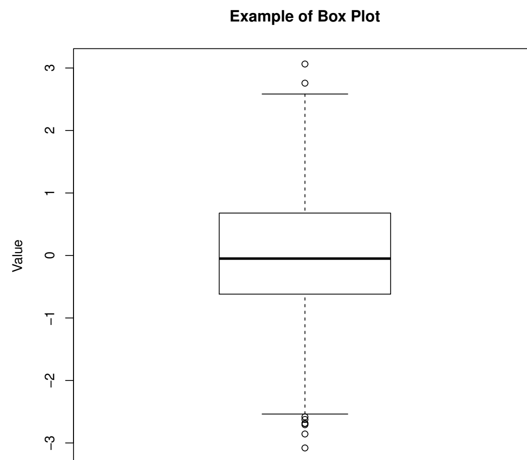
- Central tendency measures (mean, median, mode): they express the tendency of a group of data by a single value;
- Dispersion measures (standard deviation, variance): they express in a single value how disperse/concentrated the data is;
- Scatter plot: two-dimension plot that displays each data point as a (x, y) point, as depicted in Figure 2.6a;
- Histogram: plot that shows data grouped into categories or ranges, as exemplified in Figure 2.6b;
- Box (and whisker) plot: graphical representation of some important measures, as quartiles, mean, median, and outliers, as shown in Figure 2.6c;

After this step, in order to deepen the data analysis and begin to make inferences by testing hypotheses, the researcher may check whether the data satisfy the assumptions stated for each test. For example, **parametric tests** usually assume that data follow normal distribution and, particularly the analysis of variance – ANOVA – (refer to Table 2.3) also assume that the variances of the compared samples are similar or at least comparable. On the other hand, the **non-parametric tests** do not make these kinds of assumption about the data. Table 2.3 summarizes several hypothesis tests and their applications [87].

In order to test a null hypothesis, the researcher usually compares the $p{-}value$ of the equivalent test against a statistical significance or α. It means the probability of rejecting the null hypothesis when it is in fact true, i.e. **Type I error**. If $p{-}value = \alpha$, the test's null hypothesis must be rejected, otherwise there is no evidence to do so. As a good practice, the researcher should provide the p-values of the performed tests in order to increase transparency regarding their conclusions [3]. However, if a test compares several samples and the null hypothesis of equality among them is rejected, one is not able to determine which



(a) Scatter Plot (b) Histogram



(c) Box Plot

Figure 2.6: Example of useful plots for descriptive statistics.

Table 2.3: Summary of some important hypothesis tests.

Name	Objective
Anderson-Darling Cramér-von Mises Kolmogorov-Smirnov	They test whether a sample follows normal distribution.
Fligner-Killeen	It verifies whether samples with the same size present equal variances.
Bartlett	Same as Fligner-Killeen, but the samples could have different sizes.
T-Test	It compares the means of two samples.
Mann-Whitney	Non-parametric alternative to the t-test.
ANOVA	It compares two or more alternatives.
Kruskal-Wallis	Non-parametric alternative to the ANOVA.
Tukey	Post-hoc test that compares pairwise samples. Frequently used as a further analysis after ANOVA.
Bonferroni	Correction on the significance applied on multiple tests, usually along with an array of pairwise Mann-Whitney tests, as an alternative to the Tukey test.

samples are different and how significant this difference is. To do so, there is a tool that is able to provide this interpretation, which is the effect size analysis.

In this thesis, we applied non-parametric effect size analysis by using the $\hat{A}_{12}$ statistic [85]. Let s_1 and s_2 be two numerical samples, $\hat{A}_{12}(s_1, s_2) = p$ calculates the probability p that the result of a comparison between s_1 and s_2 is true. Since p is a probability, $0 \leq p \leq 1$, we can evaluate its result by two different points of view [67]: which is the greatest sample (Equation 2.1) and how big is this difference, i.e. the effect size itself (Equation 2.2).

$$\hat{A}_{12}(s_1, s_2) = \begin{cases} p > 0.5, & \mathbf{s_1} \text{ is the greatest;} \\ p < 0.5, & \mathbf{s_2} \text{ is the greatest;} \\ p = 0.5, & \text{the samples are } \textbf{indistinguishable}. \end{cases} \tag{2.1}$$

$$\hat{A}_{12}(s_1, s_2) = \begin{cases} \text{if } p > 0.71 \text{ or } p < 0.29, & \textbf{large} \text{ effect size;} \\ \text{if } p > 0.64 \text{ or } p < 0.36, & \textbf{medium} \text{ effect size;} \\ \text{otherwise,} & \textbf{small} \text{ or negligible effect size.} \end{cases} \tag{2.2}$$

For instance, suppose two TCP techniques, A and B, and we compare them using APFD

by running both techniques with a set of test suites. The $\hat{A}_{12}$ statistic measures the probability that running A yields a higher APFD than running B. Therefore, if we get the result $\hat{A}_{12}(A, B) = 0.67$, A performs better than B (from Equation 2.1), with a medium effect size (from Equation 2.2).

2.5 Chapter Final Remarks

We discussed in this chapter the fundamental aspects about the subjects involved in this research. Most of the discussion focused on TCP applied in the context of MBT, including an in-depth description of the techniques implemented and investigated in this thesis, and effectiveness evaluation metrics regarding fault detection. Furthermore, we also covered empirical investigation in software engineering, along with remarks on data analysis tools and guidelines. In the next chapters, we discuss and detail the steps performed for our investigation.

Chapter 3

Investigation on Relevant Factors

In this chapter we intended to investigate to what extent TCP techniques are affected by factors related to characteristics of system models and test cases itself, addressing our first two research questions, which are:

> *(RQ1) Does the model layout, represented by the number of branches, joins, and loops, affect the fault detection capabilities of the prioritization techniques in the investigated context?*
>
> *(RQ2) Do characteristics of test cases that fail affect the fault detection capabilities of prioritization techniques in the investigated context?*

The main ideas were: to gather evidence about the effect of different factors; to learn about strengths and weaknesses of each technique; to use the obtained knowledge on the following steps of the research, to propose new techniques which try to explore the strengths and avoid the weaknesses revealed.

To do so, we designed and performed a sequence of experiments evaluating separately several factors that interact directly and indirectly with MBT TCP techniques. This investigation includes the following independent variables:

- **General TCP techniques**: the techniques itself are our object of study;
- **The model layout**: aspects that define the general appearance of the model that represents the SUT's behavior. We represent it through: depth or the number of steps

from the first state (root) to the states without child (leaves); branches, or number of states that have more than one child; joins, or the number of states that have more than one parent; and loops, which is the number of steps that lead to a previous state in the control flow;

- **Characteristic of test cases that fail**: the size of test cases that fail (short or long test cases fail), how many branches, joins, and loops the test cases that fail traverse.

We report our results in this chapter, including a general discussion on findings already published, and a study repeating previous conditions, but dealing with existent threats to validity, i.e. using industrial artifacts and a more representative selection of TCP techniques.

3.1 Preliminary Results

The results presented in this section were also produced during our research and compose our body of investigation. The first work with our results [64], as well as an extended and more detailed version [65], comprised a set of empirical studies analyzing the aforementioned factors:

1. An exploratory study evaluating the effect of the number of test cases that fail in the test suite on the investigated TCP techniques;

2. An experiment evaluating the influence of the model's layout on the investigated TCP techniques. In the study, we used synthetic LTS generated according to parameters related to the model layout, in order to control carefully this factor;

3. An experiment evaluating the influence of characteristics of test cases that fail on the investigated TCP techniques. In this study we also used synthetic LTS and fault models to control the characteristics of the test cases that fail.

From the first study we found out that the investigated TCP techniques presented significant statistical differences when the number of test cases that fail in the investigated test suites varied. However, one should not be confident to use only this information when choosing a technique. On the one hand, techniques based on random choices had a good performance when the amount of failures is low. However, performance's growth rates did not follow a

pattern, in other words, some techniques improved their performances when the amount of failures increased, whereas some others decreased. Thus, these results were consonant with our research hypothesis, which stated that prioritization techniques present different abilities of revealing failures given the number of test cases that fail in the test suite, motivating the investigation of other factors.

The next step is to observe the relationship between the TCP techniques and test suites. Since these test cases are generated from LTS models, as a first step, we varied the layout of the LTS models aiming to observe the effect on the TCP techniques. To do so, we used an automatic LTS generator [18] to generate synthetic artifacts, varying their layouts as already discussed in the beginning of this chapter. In order to control the injection of faults, we used fault models able to inject randomly faults in models' transitions, representing the scenario that faults may occur everywhere.

Considering those synthetic LTS with different layouts by varying the amount of branches, joins, and loops, *we expected test suites with varied lengths*. For instance, cascade branches (or branches distributed at different levels of the model) may lead to a variety of short to long test cases. Moreover, *we expected that test cases could be more or less redundant with respect to covering common transitions*, particularly the more branches, joins and loops the model had, more redundancy the generated test cases would have, since they might cover common prefixes of the branches and joins, as well as repeated sequences of loops. After generating all synthetic LTS and test suites, both expectations were confirmed. As results, even with all the diversity explored by the factors and treatments, the configurations of layouts we considered for each treatment of the independent variables did not show statistical difference on the behavior of the studied techniques.

Since the general layout of the system models does not seem to affect significantly TCP techniques, we decided to investigate other characteristics, now related to the test cases that fail and how TCP techniques deal with them. For this study, we kept the model layout as close to constant as possible (several slightly different LTS, generated using the same parameters, as in the previous study). This time, instead of allowing faults everywhere in the model, fault models exercised the following characteristics:

- Longest test cases, i.e. the ones that comprise more transitions (**LongTC**);

- Shortest test cases, i.e. the ones that comprise fewer transitions (**ShortTC**);
- Test cases that traverse more branches (**ManyBR**);
- Test cases that traverse fewer branches (**FewBR**);
- Test cases that traverse more joins (**ManyJOIN**);
- Test cases that traverse fewer joins (**FewJOIN**);
- Essential test cases, i.e. the ones that uniquely cover a specific transition in the model (**Essential**);

The artifacts related to our preliminary investigation can be accessed at `https://goo.gl/EftpNA`. Summarizing the results from these artifacts, we remark:

- There was no best performer among the techniques;
- Techniques that present some random aspect were less affected by the characteristics;
- The ''Essential test case" characteristic did not affect significantly the techniques;
- There were no significant differences between the pairs of characteristics (**LongTC**, **ManyBR**) and (**ShortTC**, **FewBR**);
- Characteristics of the test cases that fail affected the investigated techniques because techniques performed well in one scenario and worse in others.

Since this study revealed a result indicating a factor that affect significantly TCP techniques, we decided to reproduce it, dealing with some threats to validity, such as the use of synthetic artifacts, in order to review this result. Therefore, in the next sections we detail the study we performed, also evaluating characteristics of test cases that fail, but now with industrial artifacts.

3.2 Investigation with Industrial Artifacts

This study reproduces the circumstances evaluated in the third study, mentioned in the last paragraph, which considered synthetic system and fault models to represent exactly the specific investigated conditions. However, it represented an important threat to validity for the investigation, which is the use of synthetic artifacts. The objective of this replication is twofold: i) to reduce the threats to validity presented in the original study by using industrial artifacts and ii) to provide evidence that the results presented in both studies are not merely artifactual; to do so, we modified some aspects related to the operationalization and population, refined the research question, but kept the same variables, operational environment and experimenter. Therefore, according to the classification proposed by Gómez et al. [27], this is a changed-operationalizations/populations replication.

In this study, **we compare a set of general and system level TCP techniques, investigating the influence of different characteristics of test cases that fail on them**. Besides using industrial test suites and fault reports as artifacts, we also have a more diverse and representative set of TCP techniques when compared to the previous studies.

In this investigation, we intend to answer the following research questions:

- **FACT_Q1: Is there a best performer among the investigated techniques, with respect to the ability of revealing faults?** We aim at comparing the investigated techniques and either point out the one that presents the highest APFD value, or report if they are statistically similar, alternatively;

- **FACT_Q2: Is the ability of revealing faults of the investigated techniques affected by the size of the test cases that fail?** We aim at measuring the effect sizes of the APFD measured from the investigated techniques when operating with test suites with test cases that fail with different sizes;

3.2.1 Investigated Techniques

Both research questions refer to a set of investigated techniques. In Section 2.3.1 we introduce and detail all the techniques used in this thesis; they are a representative and inclusive set of TCP techniques able to work on our investigated context, even though some slight

modification must be done. They are:

- Controls: **Opt** provides the optimal performance, just reached taking into account the fault records, and represents the best result a technique could present. Analogously, **Ran**, which is the random prioritization, represents a total lack of strategy, being frequently used as a lower bound;

- Adaptive Random: it is a prioritization technique proposed by Jiang et al. [38], which focuses on spreading as much as possible the ordering of test cases through the input test suite, using a notion of distance. Here we investigate four variants of this technique, labeled as **ARPJac**, **ARPMan**, **ARPSim1**, and **ARPSim2**, respectively;

- Sorting based on functions: other set of techniques suggest the order of test cases based on the value calculated by a function. **FW** is a technique [75] that assigns fixed weights to different elements of UML activity diagrams. Kundu et al. [48] introduced a technique called **Stoop**, which calculates the Averaged Weight Path Length - AWPL for each test case and sorts them based on that. Besides, Kaur et. al [43] proposed **PC** taking into consideration a metric called information flow to evaluate each step of the test cases. It prioritizes them by summing the values for each of their steps;

- String distance: Ledru et al. [50] proposed prioritizing test cases based on the resemblance between string representations of the involved test cases. The authors propose using four different functions: Hamming, Euclidean, Manhattan, and Levenshtein distances. Among these well-known functions, we consider all but Levenshtein, because it is very time consuming and leads to a result comparable to Manhattan distance [50]. Therefore, we refer to these variations in our study as **SDh**, **SDe**, and **SDm**, respectively;

- Greedy coverage of steps: Elbaum et al. [22] suggested using a greedy reasoning to prioritize test cases. The total approach sorts the test cases by the total amount of code statements that they cover, whereas the additional one also sorts them, but adjusts iteratively the statements already covered by the currently test case sequence, i.e. the next test case in order is the one that cover more statements not yet covered so far. Since our test suites are based on high abstraction steps instead of code statements,

we adapted the techniques for our context by considering the coverage of steps. We identify these two techniques as **ST** and **SA** in our study.

3.2.2 Systems, Test Suites, and Faults

In MBT processes, the system's models are often the input artifacts. Test cases are generated from these models and later provided to testing techniques. As experiment objects, we use test suites from six industrial systems[1], but we are not able to reveal further details about them due to a Non-Disclosure Agreement. We provide a brief description about the systems, as follows:

- S1 - a system that provides communication between mobile devices and payment terminals;
- S2 - embedded system that collects biometric data for controlling student attendance to a school class;
- S3 - desktop system that interacts with payment terminals, sending test commands and collecting its results;
- S4 - system for management of general scholarly activities, including equipment's management, which interacts with S2 to have access to student attendance data;
- S5 - a system that analyze failure log files on payment terminals;
- S6 - system to manage lending of equipment/software, maintenance logs, and control of bills.

The development team described the behavior of each system using a controlled natural language to write use cases, as part of the team's requirements specification practice. Besides, they also decided to generate test cases for each use case independently, composing test suites by the union of test cases generated from each test suite. From each use case description, the team automatically generated an LTS such as the one described in Section 2.2.1. The testing team then generated test suites from these LTS models using the same

[1] Developed as part of a cooperation between Ingenico do Brasil and our research lab.

test case generation algorithm we used in our preliminary study (refer to Section 3.1), which traverses loops at most twice, and uploaded them to a test case execution management tool - TestLink[2]. This process was also applied on the other industrial systems investigated in every study reported in this thesis.

Then we collected *the executed test suites* and *produced fault reports* from TestLink, after the testing team validated and executed them. These two artifacts were the inputs for our experiment. The description of a fault on those reports consists of a high level description, including the unexpected behavior and a possible high level cause. From the reports, we collected test cases that revealed each fault, which is an important artifact in our study.

Due to a non-disclosure agreement, we are not able to reveal further details about these systems and test suites, but we summarize some important data about the test suites related to the investigated systems in Table 3.1, where for each row, we have the following data about the referred test suite:

1. TS Size: amount of test cases in the test suite;

2. Mean TC Size: the arithmetic mean of the test case sizes in the referred test suite;

3. Shortest TC: the size of the shortest test case in the referred test suite;

4. Longest TC: the size of the longest test case in the referred test suite;

5. #Faults: the number of reported faults for the test suite;

6. #Failures: the number of test cases that fail because of the reported faults.

Note that, even though the sizes of the test suites are relatively small, considering the samples grouped by system gives a more precise idea about the size of the investigated industrial systems. Besides, the complete execution of the test suites of each system can be prohibitive, demanding the use of TCP techniques. Firstly, the test suites contains system level test cases, which means that they represent a high level usage of the SUT. Secondly, test cases are executed manually and frequently they make use of different devices that are prepared and operated by more than one tester for a single test case execution. Finally,

[2]http://testlink.org/

Table 3.1: Overview about the investigated systems and related test suites.

		TS Size	Mean TC Size	Shortest TC	Longest TC	#Faults	#Failures
S1	TS1	10	9.7	5	13	2	2
	TS2	9	9.5	5	11	2	2
	TS3	4	6.7	6	7	2	2
S2	TS1	10	10.7	9	12	1	3
	TS2	14	15.5	7	22	1	4
	TS3	24	15	6	21	2	20
S3	TS1	4	17	15	19	1	2
	TS2	5	22.6	13	41	2	3
S4	TS1	4	8.5	5	12	1	2
	TS2	6	10.8	5	17	2	2
S5	TS1	3	27	5	39	1	1
	TS2	3	18.3	17	19	2	2
	TS3	5	15	15	15	1	1
	TS4	4	10	7	17	1	1
S6	TS1	6	10.3	7	13	2	2
	TS2	9	5	5	5	3	4
	TS3	6	10.3	7	13	2	2

the setup conditions for some test cases can be rather costly, for instance specific network conditions/failures and time requirements.

Under these circumstances, the execution costs of a single test case can be quite high so that executing the complete test suite (that encompasses a single use case) may not be feasible at times, and all suites most of the times, particularly if we consider that manual test case execution:

- Requires the tester to be detail-oriented and inquisitive as usually execution is more than following the test case scripts. The tester needs to closely observe the behavior of the system regarding expected results in order to define the right verdict. Identifying the expected results may not be trivial;
- Proper documentation of failures and possible defects is required and this should be made right after each failure occurs. It might be the case that execution may be repeated a number of times so that the tester gets the right clues to describe the failure;

- Failures may be intermittent and not easy to reproduce, demanding possible rework and redesign of the test case execution procedure.

Furthermore, regarding Table 3.1, it is important to remark that faults and failures are not the same concept. For each fault, a number of test cases fail. There are situations where for each fault, we had exactly one failure; on the other hand, there are situations where we had more than one failure for a fault. The APFD metric considers the first test case that fail for each fault. This is why we need to consider both information.

3.2.3 Planning and Design

In order to represent characteristics of test cases that fail, the preliminary study took several ones into consideration. However, for this replication we focus only on their size. The reason is that we found out in the preliminary study that the responses of the investigated techniques to different number of branches that the test cases traverse and their sizes presented similar results. Moreover, since the system models are not experimental objects of this replication, i.e., they are not automatically generated as part of the experiment, it is difficult to deal with model branches and joins. Furthermore, we do not consider the **Essential** characteristic for the replication, since no significant differences among the techniques were observed in the preliminary study for this characteristic. Therefore, we just consider **LongTC** and **ShortTC** as characteristics for this replication.

However, in this replication, we deal with industrial test cases and fault reports. Finding experimental objects that strictly meet the original **LongTC** and **ShortTC** descriptions could limit our scope of investigation since we would need to search for test suites and history of executions where just the largest and the shortest test case would fail, respectively. It is reasonable to assume that approximations of the longest and shortest test cases are also good representatives. Therefore, we extend the definition of **LongTC** and **ShortTC** characteristics.

We establish a relation between the sizes of test cases that fail and the remainder of the test suite: i) Long test cases are those that exercise more steps of the system than the average number, possibly traversing loops - including the longest ones as in the preliminary study, and ii) short test cases exercise fewer execution paths than the average, usually straight and

not traversing loops. Since we know the faults in advance, considering the set of available test suites and the test cases that fail for each one, we classify the test suites that meet each of following characteristics:

- *LongTC*: test suites that every test case that fail is longer than the average size (From Table 3.1: S1-TS3, S4-TS1, S4-TS2, S5-TS1, S5-TS2, and S5-TS4);
- *ShortTC*: test suites that every test case that fail is shorter than the average size (From Table 3.1: S3-TS2 and S6-TS3);
- *ConstantSizeTC*: test suites where all test cases have the same size (From Table 3.1: S5-TS3 and S6-TS2);
- *MixedTC*: The test suites that do not fit in any of the previous profiles (From Table 3.1: S1-TS1, S1-TS2, S2-TS1, S2-TS2, S2-TS3, S3-TS1, and S6-TS1).

In order to properly address FACT_Q2, we discarded all test suites in the *ConstantSizeTC* and *MixedTC* categories since they neither resemble any characteristic investigated in the original study, nor provide any significant variation of the factor. Rather, we only consider **LongTC** and **ShortTC** categories. To measure the effect of these two treatments of the factor, we analyze the effectiveness of the investigated techniques considering test suites that are in **LongTC** and **ShortTC** categories separately, i.e. we compare the techniques when they prioritize test suites that satisfy **LongTC** conditions with their performance prioritizing test suites that satisfy **ShortTC** conditions. Notice that, although we present a different way of defining both **ShortTC** and **LongTC**, we still analyze the same variable, that may include an extended set of approximate values.

Considering the artifacts already referred, we executed every technique 1000 times with each test suite, according the suggestion of Arcuri and Briand [3], since the techniques perform random choices; from every single trial we measured the APFD ($technique \mid testsuite \rightarrow APFD$). Figure 3.1 presents an overview of our study. We performed every trial for this study on an Ubuntu 16.04 Linux machine, with a Core i5 processor and 6 GB of RAM memory.

To summarize the differences and similarities between both studies involving characteristics of test cases that fail, Table 3.2 presents, side by side, the characteristics of each study.

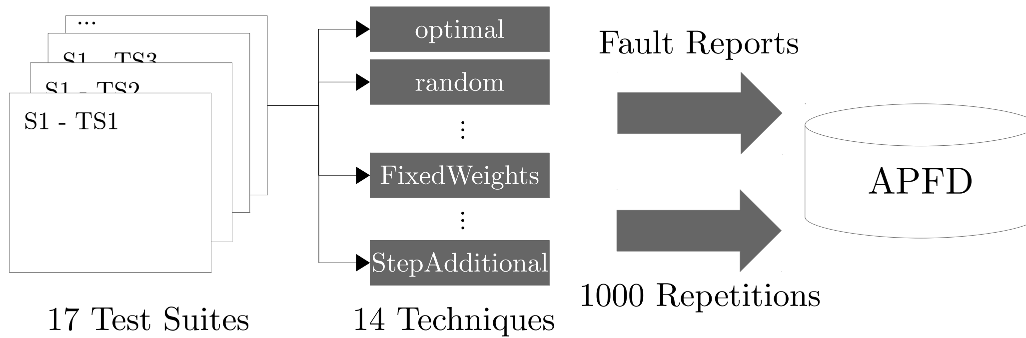


Figure 3.1: Overview of the performed case study.

Underlined text indicates common aspects of both studies.

3.2.4 Results and Analysis

As a first approach to analyze the collected data[3], Figure 3.2 summarizes the overall performances of the investigated techniques. Note that the samples compared here congregates data from several test suites, then the joined sample of **Opt** contains different values and the related boxplot represents variation. In this study we have two experiment controls, **Opt** and **Rand**, but for the statistical analysis we just remove **Opt** since it is not a viable technique, which would bias severely the comparisons, mainly Kruskal-Wallis tests; **Rand**, on the other hand, is still a viable TCP technique, then we keep it for the data analysis.

Analyzing visually the Figure 3.2, one can notice that there are differences among the investigated techniques. Moreover, apart from **Opt**, **PC** presents the highest median. Nevertheless, this is not yet an evidence that PC is the best performer; we must investigate further to address our research questions.

When we test the hypothesis that *all samples are statistically equal* through a Kruskal-Wallis test, we obtain *p-value* $= 2.2 \cdot 10^{-16}$. Therefore, even with overlaps among box plots, techniques present different performances considering the whole set of test suites.

Since some behaviors can be hidden by gathering data from all systems, Figure 3.3 depicts the performances of the techniques grouped by system. Notice that the techniques perform differently across systems, but again **PC** appears in 3 out of 6 systems with the highest median (also excluding **Opt**). Analogously, we also perform Kruskal-Wallis tests

[3]The direct link to the subpage in our companion web site is `https://goo.gl/I4cPgt`.

Table 3.2: Side by side comparison between the original and replication studies.

Aspect	Original	Replication
Research Question	How general TCP techniques behave when test suites with certain properties fail?	Is there a best performer among the investigated techniques, with respect to the ability of revealing faults? Is the ability of revealing faults of the investigated techniques affected by the size of the test cases that fail?
Independent Variables	**TCP techniques**: ARPJacMaxMin, ARPManMaxMin, FixedWeights, Stoop **Characteristics of test cases that fail**: LongTC, ShortTC, ManyBR, FewBR, ManyJoin, FewJoin, Essential Number of faults defined by a fault model: fixed value equals to 1	**TCP techniques**: Optimal, Random, ARPJacMaxMin (renamed ARPJac), ARPManMaxMin (renamed ARPMan), ARTSimMaxMin, ARTSimMaxMax, FixedWeights (renamed FW), Stoop, PathComplexity, StringDistanceHamming, StringDistanceEuc, StringDistanceMan, StepTotal, StepAdditional **Characteristics of test cases that fail**: LongTC, ShortTC Actual fault reports
Dependent Variables	APFD	APFD
Experimental Objects	31 synthetic LTS, similar to industrial ones, and test cases generated from them	17 test suites from six different industrial systems
Data Analysis Techniques	Descriptive statistics, visual analysis, and hypothesis testing	Descriptive statistics, visual analysis, hypothesis testing, and effect size analysis

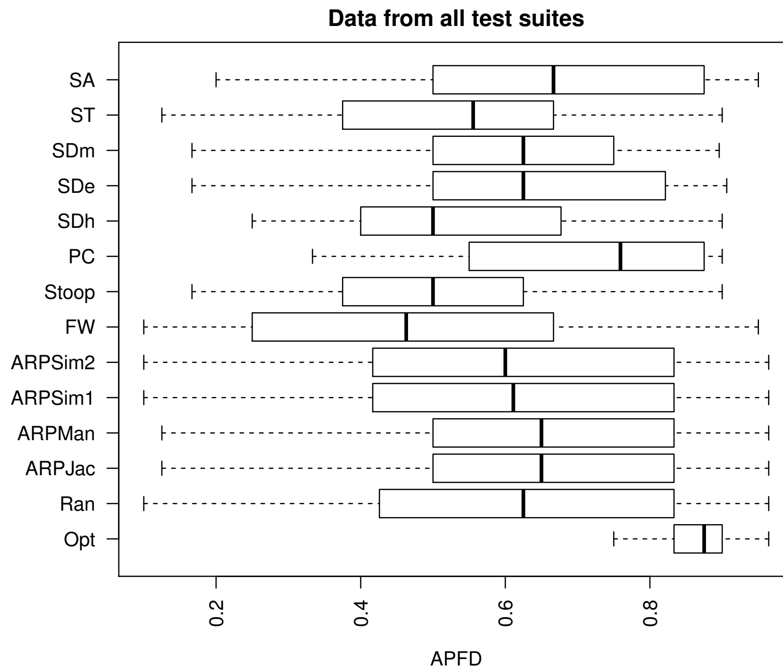


Figure 3.2: Overall data for all techniques.

on each of the six data sets and all of them result in the same p-value of the previous test, which is enough to reject all null hypotheses of equality among the samples. Thus, in order to address FACT_Q1 and FACT_Q2 we need to investigate in more details.

FACT_Q1 To address this question, we adopt a non-parametric approach for data analysis. Since we have a high number of repetitions and there is no practical difference between both approaches, non-parametric tests are able to test with confidence [3].

A possible approach to consistently identify the best performer would be to apply pairwise hypothesis tests between techniques, with a correction regarding statistical significance, e.g. an array of Wilcoxon tests with Bonferroni correction. However, these tests would just suggest the best performer, without clarifying how big are the differences between them. Thus, we need to apply effect size analysis.

Considering data from Table 3.3, the major portion of the comparisons between every pair of technique results in small effect sizes. Just observing the bold-faced values, which are the medium and large effect sizes, we remark two findings:

- **FW** presents consistently a bad performance, which means that just applying fixed weights based on the kind of model element the test suite traverses does not lead to a

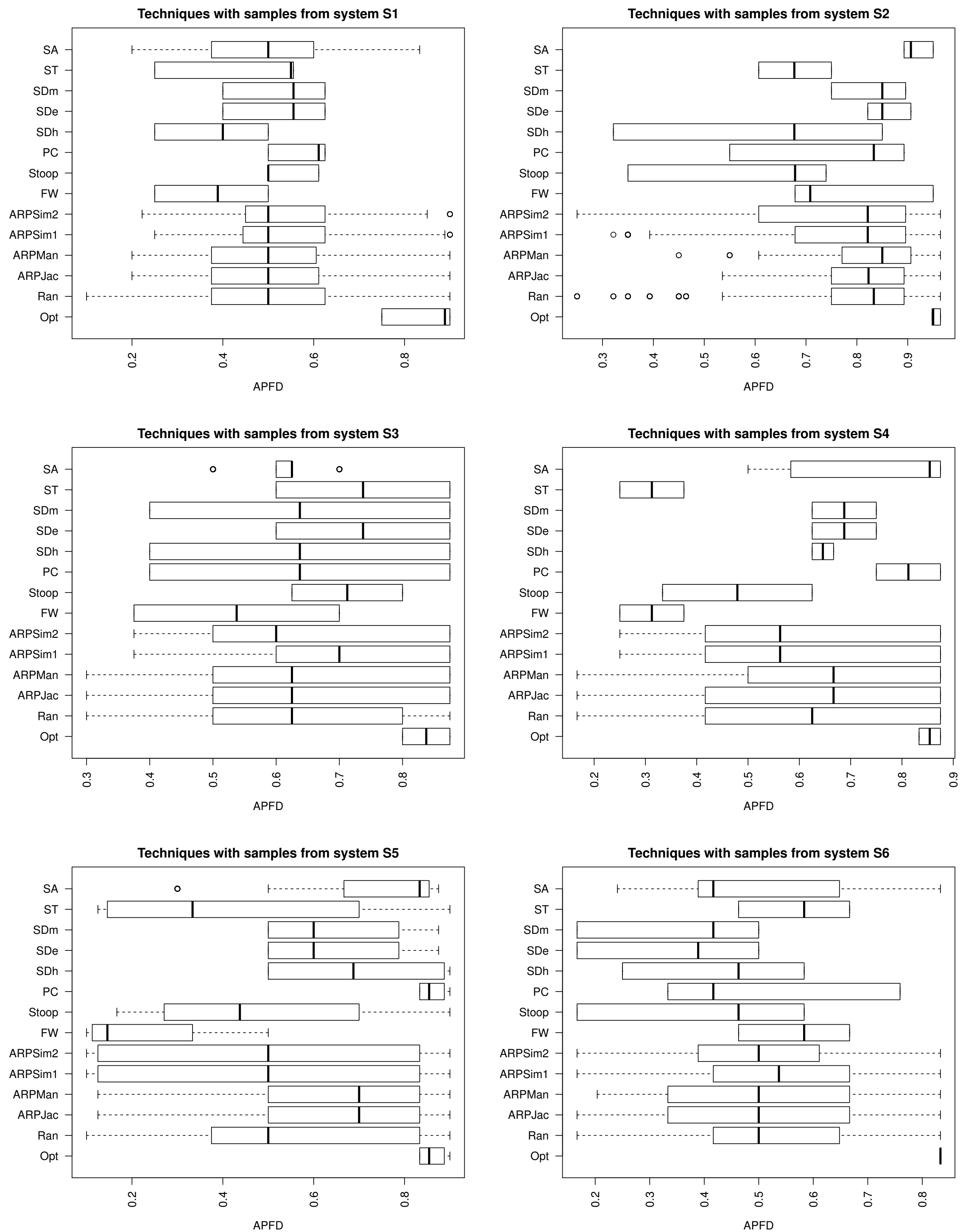


Figure 3.3: Performance of the techniques by system

Table 3.3: Effect sizes of pairwise comparisons of the investigated techniques. Each cell contains the result of the comparison between the technique from the line i and the one in the column j. The diagonal (lighter grey) contains the comparison $\hat{A}_{12}(i, i)$, which always lead to effect size 0.5 and it is not relevant for our purposes. The darker grey area omits values that are complementary to the presented values. Bold faced values represent medium and large effect sizes.

	Ran	ARPJac	ARPMan	ARPSim1	ARPSim2	FW	Stoop	PC	SDh	SDe	SDm	ST	SA
Ran		0.47	0.46	0.51	0.52	**0.67**	0.60	0.37	0.54	0.46	0.48	0.59	0.40
ARPjac			0.49	0.54	0.55	**0.70**	0.64	0.40	0.57	0.49	0.50	0.62	0.43
ARPMan				0.54	0.56	**0.70**	**0.64**	0.40	0.57	0.50	0.5	0.63	0.43
ARPSim1					0.51	**0.65**	0.58	0.37	0.52	0.45	0.47	0.57	0.39
ARPSim2						**0.64**	0.57	0.36	0.51	0.44	0.45	0.55	0.38
FW							0.41	**0.21**	0.37	**0.28**	**0.29**	0.41	**0.26**
Stoop								**0.27**	0.45	**0.35**	0.37	0.50	**0.31**
PC									**0.65**	0.59	0.61	**0.71**	0.51
SDh										0.42	0.43	0.53	0.36
SDe											0.51	0.62	0.43
SDm												0.60	0.42
ST													**0.32**
SA													

satisfactory ability of revealing faults;

- **PC** is the technique that presents more positive results, followed by **SA**. Therefore, the strategies that these techniques apply, such as information flow metric, and additional coverage of steps, could be efficient in the context studied. Since $\hat{A}_{12}(PC, SA) = 0.51$, both techniques are almost statistically indistinguishable.

Therefore, by the visual analysis of the boxplots and the small effect size between **PC** and **SA**, we do not have evidence to suggest that any technique is consistently better than the other ones, in other words:

Considering data from all test suites, there is no best performer among the investigated techniques with respect to the fault detection capability.

PC is among the best techniques but its performance is statistically indistinguishable from other techniques. The technique with performance more similar to **PC** is **SA**. However, for some systems a particular technique performs well and for others, badly, corroborating the findings from our previous studies.

As an example of the aforementioned variation, consider two systems for which techniques presented different performances, say S2 and S4, from our study (see Figure 3.4 for a side by side comparison). Notice that **FW**, **SDe**, and **ST** present a good performance prioritizing test suites from S2, whereas in S4 their performances decrease to poor levels. On the other hand, **PC** and **SDh** are more accurate in S4 than in S2, as suggested by less spread box plots in S4. The source of these variations are particularities of each test suite and system and how the investigated techniques interact with these particularities, for instance variations in the sizes of test cases that fail.

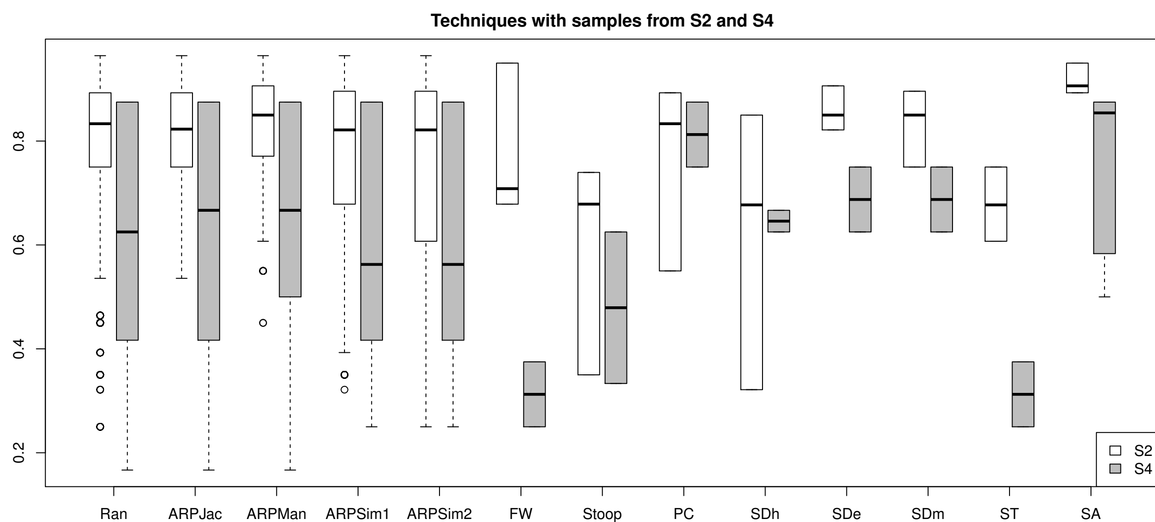


Figure 3.4: Boxplots of S2 and S4 samples.

FACT_Q2 As strategy to reveal the effect of varying the size of the test cases that fail, we compare the techniques with themselves in both characteristics of the test cases that fail and measure the effect size of these comparisons. Figure 3.5 shows side by side the box

plots from the two considered characteristics. Note that there are significant differences on the performance of the techniques for the different characteristics, particularly for **FW**, **PC**, **SDh**, and **ST**. Thus, this clear change of behavior when comparing visually both box plots suggest that the investigated techniques are sensitive to the size of the test cases that fail.

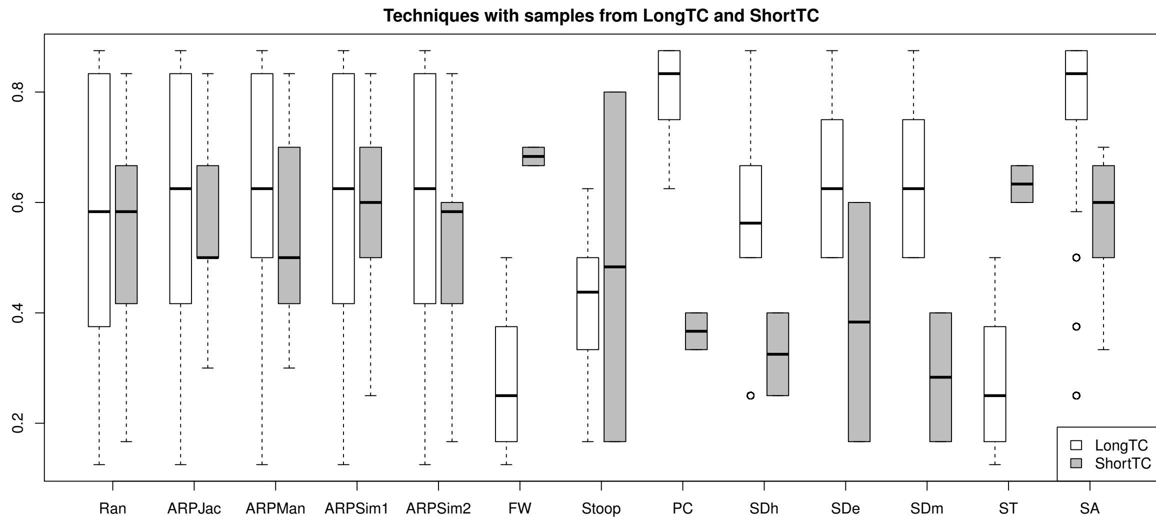


Figure 3.5: Boxplots of **ShortTC** and **LongTC** samples.

Whereas the visual analysis just provides an initial clue about the differences, the effect sizes in Table 3.4 quantify these influences. Based on the results of $\hat{A}_{12}$ statistic:

- All random-based techniques – (**Ran**, **ARPJac**, **ARPMan**, **ARPSim1**, and **ARPSim2**) – present small effect size. Among the adaptive random techniques, the different functions do not appear to affect significantly the results.
- Even though **Stoop** does not rely strongly on random choices, it also presents small effect size.
- The other techniques present a large effect size, in other words, they are strongly affected by the variation of the studied characteristics.

Thus, based on the results obtained, we can conclude that:

> The investigated techniques are affected by the size of the test cases that fail, but not in the same way.

Table 3.4: Effect sizes of the comparisons for each technique between **ShortTC** and **LongTC**

Technique	$\hat{A}_{12}$	Effect Size	Technique	$\hat{A}_{12}$	Effect Size
Ran	0.4443	Small	Stoop	0.5833	Small
ARPJac	0.3945	Small	SDh	0.0833	Large
ARPMan	0.374	Small	SDe	0.1666	Large
ARPSim1	0.4725	Small	SDm	0	Large
ARPSim2	0.3892	Small	ST	1	Large
FW	1	Large	SA	0.1608	Large
PC	0	Large			

Techniques that present random choices in its operation are less sensitive than the other ones because techniques that use random choices to guide their operation make fewer assumptions about the relationship between structural aspects of the test cases and their ability of revealing faults. For instance, ARP techniques have a mechanism to create candidate sets randomly, where the next test case to be placed in order is randomly selected among the ones not yet prioritized, before the distance function is applied. Whereas we already expected this result, we were surprised by the low importance of having different distance functions on the results; maybe the selected ones, even using different properties of test cases, capture the same notion of distance between test cases. On the other hand, this result may be an indication that some level of randomness for a technique in the investigated context could help to provide a more general result.

Other technique that present an interesting behavior is **Stoop**. It seems less affected by the variation of the investigated characteristics of the test cases that fail, but still performs badly, as discussed in the FACT_Q1 analysis. This low effect may be due to the assumption that their authors introduce; they consider important test cases that exercise common steps, i.e., steps that many other test cases also exercise. Therefore, Stoop puts them in the first positions to assure a good coverage of more important functionalities of the SUT. Note that this assumption does not take into account the size of test cases, but it may not lead to a good ability of revealing faults either.

The other techniques also appear to be affected by the assumptions made by their authors. **FW** gives preference to test cases that cover main scenarios of the use cases, which are

usually shorter than the ones that cover alternative and exception scenarios; **ST** and **SA** sorts test cases by the number of steps, the former does not consider the feedback of the steps already covered, and the latter does; and **SDh**, **SDe**, and **SDm** put the most different test cases in the first positions and, since the distance functions are not normalized by the test case size, the techniques tend to put long test cases in the beginning.

3.2.5 Threats to Validity

We evaluate the validity of this study by discussing its threats and how we dealt with them. Regarding **conclusion validity**, when comparing performances between **ShortTC** and **LongTC**, we compared samples with different sizes. To mitigate that threat, we repeated each pair technique/test suite a high number of times in order to perform non-parametric analysis with confidence.

Concerning **construct validity**, since we considered industrial test suites and fault reports, we increased this validity in contrast to the other studies performed previously [64,65]; this change provided us a more direct relationship between the theory and the observation. Another aspect of the construct validity is the definition of **ShortTC** and **LongTC**, which are the characteristics investigated in this study. Although there might be other ways of defining these characteristics, in order to keep variability - more than one test suite in each treatment - we used the average test case size of the test suite as a threshold to define whether the test cases that fail are short or long.

With respect to **internal validity**, we investigated only the effect of the size of test cases that fail on the techniques, but there may be other aspects that we did not control in this study that may affect them, for example the number of test cases in each test suite in the sample or the proportion of test cases that fail. We report the data about the investigated test suites to provide transparency to the replicated study setup.

About **external validity**, even though we used real and industrial systems, we were not able to generalize for any other kind of system. Besides, since there is a direct relationship between the system model and the generated test suites, we were not able to generalize the results for either manually created test suites or generated from some other kind of model. Therefore, we believe that similar conclusions hold for similar systems. Furthermore, some of these are small test suites, possibly making them not good candidates for prioritization.

This argument is even stronger when test suites are automatically executed. Nevertheless, the seventeen test suites involved in this study were executed manually which is usually a highly demanding and costly process, specially regarding the level of the testing and the technology involved in the execution. Therefore, even small manual test suites could benefit from prioritization, since the execution/analysis savings can compensate for the prioritization costs.

3.3 Chapter Final Remarks

This chapter reported the results of an investigation about factors that could affect the performance of TCP techniques. It has begun investigating the model layout, represented by the number of branches, joins and loops of the model, and a set of characteristics of the test cases that fail. The effect of the model layout and characteristics of the test cases that fail were investigated and reported in two studies [64, 65]. The results showed that, even varying the structures responsible to represent the model layout, the techniques presented behaviors statistically similar, which is an evidence of low or negligible effect. Even with the low influence, we could still argue that the number of loops that a model has could be still important, since it also affects the redundancy on generated test suites. On the other hand, characteristics of the test cases that fail definitely affect the techniques, since we vary the characteristics and we observed the techniques varying effectiveness. However, this study just considered synthetic artifacts, i.e. system and fault models, which is a significant threat to external validity. Therefore, considering the same research questions, variables and objectives, we repeated the study that evaluated characteristics of test cases that fail, but now using industrial artifacts, and reported its details in this chapter.

For this study, we compared a set of 5 families of TCP techniques, with a total of 14 techniques, using 17 test suites for system testing of 6 industrial systems, developed in cooperation with Ingenico do Brasil. By comparing directly the techniques, we did not have empirical evidence to suggest a clear best performer. In this sense, in similar contexts, a tester may opt for lower cost techniques such as the **SA** or **PC** because they are easy to implement and use a simple heuristic to propose a prioritized sequence of test cases. Since the investigated techniques were sensitive to the variation of the sizes of test cases that fail, their

performances may vary depending on these characteristic.

Therefore, it is important to investigate ways of reducing this variation by trying to incorporate other sources of information, yet independent of previous test executions. To illustrate, we could use experiences that developers have during the early stages of the system development, even before any testing. Furthermore, we could apply different underlying theories to solve the TCP problem, but still subject to our main assumption, which is that previous information about other testing executions is not available e.g. soft computing methods or machine learning approaches. Another important aspect of the study is the lower influence of the random based techniques, which may be an indication that keeping a random aspect in a TCP technique may help to reduce the dependence of external characteristics.

Chapter 4

Hint-Based Adaptive Random Prioritization - HARP

Results of our previous investigation suggested that: keeping some degree of randomness could be a good idea to propose more general results; incorporating other sources of information than structural aspects of the test suite and system model can be valuable to guide prioritization; and adaptive random techniques showed promising results among the ones that make random choices. Based on these findings we introduce Hint-Based Adaptive Random Prioritization (HARP), which is a technique based on the adaptive random strategy, that uses hints from developers and managers to prioritize test cases from a test suite. The investigation reported in this chapter focuses on our third research question, which is:

> *(RQ3) How can we systematize and collect expertise from the developers or/and managers, and use it to leverage Test Case Prioritization in the defined context, aiming to reveal faults sooner in the testing process?*

HARP is targeted for system test suites, here approached in the MBT context, with test cases expressed as sequences of steps. Although we use LTS as model to represent system behavior and generate test cases from it, HARP does not make any assumption about a particular type of model or test case generation algorithm. Nonetheless, the functions used assume that test case steps defining the same user action/condition/result contain the same text, which matches with the model-based test case generation process traversing transitions; if we rather consider test suites written manually, there would be no guarantee that the labels

are the same and the functions would evaluate the similarity incorrectly.

The technique combines the advantages of evenly exploring the test suite with the guidance provided by hints in order to improve the performance of the ARP technique, since it reduces random choices. By observing the adaptive random prioritization algorithm [38, 91] (refer to Algorithm 5 in Section 2.3.1), we identified three possible points of improvement: i) how the first test case is chosen, which is originally a fully random choice; ii) the function that calculates the distance between two test cases; and iii) the function that represents the distance between two sets of test cases, which is used to select the next test case to be placed in order. Here, we modify these three aspects in order to reduce the effects of random choices, to represent more adequately the resemblance relationship regarding the kind of test cases we have in this context, and to take into consideration the indication from the development team about portions of the system under test that may present problems (hints).

4.1 What is a Hint?

Hints are indications that the development team give - using use case documents about occurrences regarding the system under development/testing - that may contribute to the insertion of faults, such as portions of the system that some developer considered hard to implement, that uses some external and/or unreliable library, or that suffered a schedule shortening. Our hypothesis is that the aforementioned occurrences are estimators for faults and taking them into consideration should lead to better fault detection capabilities. Observe that the hints relate to the current version of the SUT, which does not violate our initial assumption regarding historical information.

We designed HARP to process hints encoded as prioritization purposes using the notation presented in Section 2.2.3. Encoding hints as prioritization purposes is a manual, but simple task. A single hint may be a single step in a use case or a sequence of steps (or even a whole flow). In the first case, suppose that the development team indicate a step with text “single user input" as hint; so the equivalent prioritization purpose is pp_1 =“* | single user input | *", indicating that any test case that traverse “single user input" is filtered by pp_1. On the other hand, in the second case, let the hint comprises a pair of user input and expected result with texts “single user input" and “single system response" respectively; the equivalent

prioritization purpose is pp_2 =“* | single user input | single system response | *", indicating that any test case that describes the referred action and expects that particular response is filtered by pp_2. The same idea applies for a whole flow. Therefore, the output of this step is a set of hints $PP = \{pp_1, pp_2, \cdots, pp_n\}$ translated into prioritization purposes.

As illustration, considering the login and password verification system of the example introduced in Section 2.2.1, suppose that the testers suspect that the error messages are not correct or are being acquired using an error-prone strategy. Therefore, they could codify hints as the set $PP_{ex} = \{$“ $*$ | C - Invalid Login | $*$ ”, “ $*$ | C - do not match | $*$ ”$\}$. The set contains two hints, each one about a scenario where the SUT prints an error message.

HARP uses prioritization purposes as filters for test cases, creating two categories of test cases, one with the test cases related to hints and other with the not related ones; it uses these categories to create a secondary layer of guidance to prioritize test suites. We detail HARP's algorithm in the next section.

4.2 Proposed Algorithm

HARP is based on the idea of exploring test cases similar (or “close”) to the ones related to the hints provided by the development team, but still giving chance of other test cases appear in the sequence. The next test case to be placed in order always comes from a set of candidates, so this test case can be related to the hint or not. Since the creation of the candidate set is iterative, i.e. a new set of candidates is created for every iteration of the algorithm, we do not always select the most similar to the already prioritized among the unprioritized test cases; instead, the randomness in the candidate set generation inserts variability.

We depict HARP's operation in Algorithm 9. It starts with an empty sequence of test cases (Line 2). In Line 3, it filters U_TS using every prioritization purpose from PP and the remainder of the algorithm uses the result set as hint-related test cases. In Line 4, the algorithm defines the first test case to be placed in order (`firstChoice`). For that, it selects randomly one from the filtered set. This selected test case is then included in the last position of the prioritized sequence (Line 5), and removed from the original set (Line 6) and from the filtered set (Line 7).

Algorithm 9 The main procedure.

```
1:  function PRIORITIZE_HARP(U_TS, PP)
2:      PTS ← []
3:      filtTCs ← filter(U_TS, PP)
4:      first ← firstChoice(filtTCs)
5:      PTS.append(first)
6:      U_TS.remove(first)
7:      filteredTCs.remove(first)
8:      while U_TS ≠ ∅ do
9:          c ← genCandSet(U_TS, filtTCs)
10:         nextTC ← selectNext(PTS, c)
11:         PTS.append(nextTC)
12:         U_TS.remove(nextTC)
13:         filtTCs.remove(nextTC)
14:     end while
15: return PTS
16: end function
```

The loop in lines 8-14 assembles the rest of the prioritized suite. To do so, in its first step, it generates a set of candidates (Line 9). The candidate set generation process selects randomly one test case at a time, while the set keeps increasing branch coverage and size less than or equals to 10, as discussed by Jiang et al. [38] and empirically evaluated by Chen *et al.* [13]. We modified the candidate set generation by increasing the chances of hint-related test cases not yet placed in order be selected. In order to decide whether the next test case to compose the candidate set comes from the hint-related ones or not, we define a random variable following uniform distribution $V \sim U(0, 1)$. Therefore, iteratively, if the algorithm samples a value $v < 0.5$, it selects randomly a test case among the ones filtered by the prioritization purposes, otherwise selects randomly a test case among the ones not yet prioritized but not related to the hints.

The next step is to select the next test case to be placed in order among the candidates by calling `selectNext` (Line 10). Algorithm 10 gives details on how it works. First, it builds

Algorithm 10 The select next test case procedure

1: **function** SELECTNEXT(p_TS, c_S) ▷ p_TS is the already prioritized test sequence ▷ c_S is the candidate set
2: $d \leftarrow array[p_TS.size][c_S.size]$
3: **for** i = 0 to p_TS.size - 1 **do**
4: **for** j = 0 to c_S.size - 1 **do**
5: $d \leftarrow$ **sim(p_TS[i], c_S[j])**
6: **end for**
7: **end for**
8: $index \leftarrow$ **maxValue(d)**
9: $nextTestCase \leftarrow c_S.get(index)$
10: **return** $nextTestCase$
11: **end function**

a matrix (Line 5) that represents the resemblance between candidates and the test cases that were already prioritized. Since we use the notion of similarity to represent the resemblance between test cases, the matrix contains similarity values. Then, the algorithm selects the candidate with the highest similarity (Line 8), i.e. the next test case will be the most similar to the previous prioritized ones. In order to measure the similarity between test cases we use the similarity function proposed by Coutinho et al. [15], which was proposed specifically to the MBT context, being fully compatible with test cases generated with different algorithms, since it takes into account, besides the common transitions, their frequency in the test cases. The function is defined by: $similarity(i, j) = \dfrac{nip(i, j) + |sit(i, j)|}{\left(\dfrac{|i| + |j| + |sdt(i)| + |sdt(j)|}{2}\right)}$, where:

- i and j are two test cases;
- $nip(i, j)$ is the number of identical transition pairs between two test cases;
- $sdt(i)$ is the set of distinct transitions in the test case i;
- $sit(i, j)$ is the set of identical transitions between two test cases, i.e. $sdt(i) \cap sdt(j)$.

4.2.1 Asymptotic Analysis

To investigate how costly it would be to run **HARP** in practice, we analyze its asymptotic bounds. In order to perform this analysis, we consider its worst execution scenario, which is when the maximum number of candidates (10), are selected at every iteration. Moreover, consider a test suite with n elements and, for simplification, let us assume that the test cases from the original test suite have the same size, t. The execution time for every step is as follows:

1. `similarity`: The similarity calculation depends on the involved test cases sizes, however, as we are simplifying sizes to t, we can say its time is $2 \cdot O(t)$ or $O(t)$;
2. `maxValue`: The definition of the maximum similarity traverses a matrix with <number of candidates> columns and <number of test cases already prioritized> lines. The number of candidates will be in the interval of 1 and 10, and in the worst case, it will be 10 all iterations. The number of test cases already prioritized depends on the initial suite. Thus, the time is $O(10 \cdot n)$ or $O(n)$;
3. `selectNext`: The selection of the next test case to be placed in order, once more, depends on the number of candidates and on the number of test cases already prioritized. The similarity is calculated for every pair of test cases and, following the same reasoning, the time is $10 \cdot n \cdot O(t)$ plus $O(n)$ of the `maxMaxValue` execution. Thus, the execution time is $O(t \cdot n) + O(n)$ or $O(t \cdot n)$;
4. `genCandSet`: The generation of a candidate set depends on its size, that must be at most ten, and the increasing coverage of requirements. Considering the worst case, the time is constant, thus $O(1)$;
5. `firstChoice`: The choice of the first test case depends on the amount of test cases from initial test suite and the time necessary to evaluate if a single test case matches the prioritization purpose. The function just iterates over the initial test suite and verify if each test case is accepted by the prioritization purpose. This verification just depends on the size of the test case and, since we assume that the size of the test cases are the same (equals to n), the verification is $O(t)$. Thus, the whole method executes in $n \cdot O(t)$ or $O(t \cdot n)$ time;

6. `prioritize_HARP`: This is the main function, which calls the above mentioned ones, and its execution time is the total execution time of HARP. Thus, HARP execution time is $O(t \cdot n) + n \cdot (O(1) + O(t \cdot n)) = O(t \cdot n) + O(n) + O(t \cdot n^2)$, which is equals to $O(n^3)$, if $n \geq t$ or $O(t \cdot n^2)$, otherwise.

4.2.2 Running Example

In this section we provide a visual and practical understanding on how HARP works with an example. To do so, consider the login and password verification, and Table 1.1 (on Page 9) contains the generated test suite.

Besides, suppose that the development team responsible for building this system identifies the verification of the login in the database as an error-prone step of the system, particularly the invalid login verification. This step may be risky because it verifies a non-encrypted data from the database and, if the developer did not think carefully when coding it, he/she might have left an open window for security attacks (e.g., SQL injection). Therefore, the tester formulates the hint as a set with a single prioritization purpose $logError$ ={“* | [C - Invalid login] | *"}.

Considering the original test suite, to propose a new execution order, HARP chooses the test case to be placed in the first position by filtering the input test suite according to the provided hint. The algorithm filters the test suite by visiting every test case, traversing it and verifying whether its sequence of the steps matches the prioritization purpose, a process similar to the evaluation of a regular expression. The filtered subset is $filtTCs = \{TC4, TC5, TC6, TC7\}$. Then, suppose that the first test case randomly selected from the filtered set is **TC6**. Therefore, the algorithm removes it from the untreated test suite and makes $PTS = [TC6]$.

Then, HARP creates a candidate set by randomly and iteratively selecting test cases from the untreated test suite or from the hint-related set. Suppose that the set is $c_1 = \{TC1, TC3, TC5\}$, where just $TC5$ comes from the hint-related set. The next step is to calculate the similarities between the elements from current prioritized sequence and the ones from the candidate set. The calculated values are in Table 4.1.

Considering these values, **HARP** selects the test case with the highest similarity, **TC5**, then the algorithm adds it to the prioritized sequence, making $PTS = [TC6, TC5]$.

Table 4.1: 1st Iteration: Similarities among candidates and test cases already prioritized

	TC6
TC1	63.15%
TC3	71.11%
TC5	72.72%

Now, in the second iteration of the main loop of the algorithm, it proceeds by generating a new candidate set, and suppose the second candidate set is $c_2 = \{TC2, TC4\}$, where $TC4$ comes from the hint-related set. In the next step, it calculates the similarities between the test cases from c_2 and PTS, as can be seen in Table 4.2.

Table 4.2: 2nd Iteration: Similarities among candidates and test cases already prioritized

	TC6	**TC5**
TC2	59.57%	68.08%
TC4	90.90%	72.72%

By comparing the maximum similarities for the two candidates, which are 68.08% and 90.90%, the next test case to be placed in order is **TC4**, making $PTS = [TC6, TC5, TC4]$. The algorithm repeats this process until the last test case is placed in order. After the whole execution, adding one test case at a time, a proposed sequence could be $PTS = [TC6, TC5, TC4, TC2, TC3, TC7, TC1]$.

After this discussion about HARP's details and a brief running example, we detail its empirical evaluation in the next section, covering a general rationale, variables and planning, results, and validity evaluation regarding the performed studies.

4.3 Empirical Evaluation

In order to provide evidence about its applicability and efficacy with respect to fault detection capability, we intend to investigate the following questions:

- **HARP_Q1: Does the quality of a hint affects HARP's effectiveness?** In this question we want to evaluate if a good hint leads to a significantly better fault detection capability, in comparison with when HARP receives a bad hint. We evaluate this to

observe whether the random choices in HARP's algorithm affect the guidance provided by the hints;

- **HARP_Q2: Are developers and managers able to provide good hints?** We are not interested in evaluating the participants themselves. Instead, we try to find out whether the hints provided by them approximate portions of the system that really contain faults;

- **HARP_Q3: Is the hint collection process costly?** We try to find out how easy it can be to implement a process aiming to collect required information to derive the prioritization purpose, i.e, the hints. This information is important since we need to justify costs inserted by our technique;

- **HARP_Q4: Is HARP able to outperform the original Adaptive Random Prioritization technique, considering actual hints?** To address this question, we compare HARP with the baseline version of the Adaptive Random Prioritization proposed by Jiang et al. [38], observing their fault detection capability.

The evaluation encompasses two studies: The first one addresses HARP_Q1 through an experiment whose objective was to evaluate the effect of hints with different abilities to point to actual failures; The second one addresses HARP_Q2, HARP_Q3 and HARP_Q4 through a case study, in which we collected hints from development teams using a questionnaire, relate them to reported faults, and apply the collected hints, as well as the related systems and test suites, to HARP and its baseline technique. We prepared a sub page in our companion site, available at `https://goo.gl/FH3b5m`, containing the collected data and R scripts for data analysis.

4.3.1 Experiment on Hint's Quality

We define this experiment using the framework based on key concepts suggested by Wohlin *et al.* [87]:

- The **objects of study** are **HARP**, **hints** and the **test suites** generated from models that represent Systems Under Testing (SUT);

- The **purpose** of the study is to evaluate whether HARP is really affected by the hint's quality;
- The **quality focus** is fault detection capability, measured through APFD;
- The **perspective** is from the tester point of view, i.e., the person that executes a TCP technique in a testing team;
- And the **context** is Model-Based Testing.

In order to guide this investigation and address HARP_Q1 properly, we provide a manual and systematic procedure to derive the artificial hints used in this study, since we need to represent both extremes of the situation: when the team provides either an accurate hint (one that points to test cases that fail) which we call a **good hint**, or a poor hint (one that does not point to any failure), which we call a **bad hint**. We define them with previous knowledge about faults and failures collected from fault reports provided during the development process.

Hint Definition

To define a bad hint is straightforward; we create a prioritization purpose with steps not related to any fault and make sure that any of the filtered test cases reveal a fault. On the other hand, a good hint must represent a scenario where the filtered test cases reveal a fault, whereas not biasing the investigation. The definition of a good hint for this study followed a systematic and manual approach:

- For a specific fault of a given model, composed by its cause and the test cases that fail because of it, we define a prioritization purpose with the label of the edge more related to the cause of the fault;
- If the whole set of test cases filtered by this prioritization purpose fails, using it can bias the investigation, since a test case that fail would be chosen as the first one in sequence proposed by HARP. Therefore, we add more edges related to the cause in order to avoid this scenario;

- If it is not possible to create a single prioritization purpose able to filter test cases that fail and that not fail, we define other prioritization purpose related to the same hint and we restart the process aiming at filtering a set of test cases that does not bias the study.

As an example of this approach, consider once more the login and password verification system and the generated test suite in Table 2.1. Besides, assume that there is a fault on the error message regarding the login verification, but it just manifests when this message appear for more than once. The hint definition process begins by creating a prioritization purpose with the label of a single transition related to the fault, which is "C - Invalid Login". The first row of Table 4.3 shows the filtered test cases and the proportion of them that fail. Since we know that the fault just occurs when the user provide a nonexistent login at least twice in a row, we can be even more precise by adding the same label twice. The second row in the same table shows a prioritization purpose that specifies this condition case. However, note that this prioritization purpose filters only the test case that reveal the fault, which bias the study, because HARP would always place TC7 the in the first position in the prioritized test suite. Thus, according to the aforementioned approach, " * | C - Invalid Login | * " is the final prioritization purpose representing a good hint.

Table 4.3: Illustration of the good hint generation

Test Purpose	**Filtered Test Cases**	**% TC Reveal Fault**
" * | C - Invalid Login | *"	TC4, TC5, TC6, and TC7	25%
" * | C - Invalid Login | * | C - Invalid Login | * "	TC7	100%

We use the proportion of test cases that fail among the filtered ones to judge how good a hint is. Intuitively, for a bad hint 0% of the filtered set fail, on the other hand, following the aforementioned approach to derive good hints leaded to a proportion between 20% and 50%. These values represent the best prioritization purposes we are able to generate, considering the selected use cases and test reports, and without being extreme, i.e. not filtering 100% of the test cases and being equal to the baseline ARP technique, neither filtering one test case that unveil the fault. We use this approach to define the possible values for a variable investigated in this study.

Investigated Variables

Since we want to evaluate the effects of the hint's quality on HARP, observing variations by adopting different functions that evaluate resemblance between test cases, we define the following variables:

- **Independent Variables**
  - Hint quality: good and bad hints;
  - Resemblance function: Similarity function [15], and Jaccard distance function [38];
- **Dependent Variable**
  - Average Percentage of Fault Detection - APFD

In addition to the variables and their possible values that defined HARP's behavior during the experiment, we also discuss the experiment objects, which are the set of systems, test suites, and faults analyzed in this experiment.

Systems, Test Suites, and Faults

As experiment objects, we resort to two industrial systems: the **Biometric Collector** and the **Document Control**[1]. Both systems were modeled through a set of LTS, representing their use cases. The testing team generated the test cases for each use case using the same test case generation algorithm, which traverses the LTS covering loops at most twice, and uploaded them to a test case execution management tool - Testlink[2]. Then, we collected the test suites and use them in our experiment. To the purpose of our investigation, we consider only the ones that have at least one reported fault. They were reported by testers also using Testlink. Thus, our systems, test suites and faults are:

- Biometric Collector

[1]Developed as part of a cooperation between Ingenico do Brasil and our research lab
[2]http://testlink.org/

- CB01: which contains two faults, one revealed by two test cases, and the other by a single test case;
- CB03: which contains one fault revealed by a single test case;
- CB04: which contains four faults, two revealed by two test cases each, and the remaining two revealed by a single test case each;
- CB05: which contains three faults, one revealed by two test cases, and the remaining two revealed by a single test case each;
- CB06: which contains one fault revealed by a single test case.

- Document Control
  - CD01: which contains two faults, one revealed by three test cases, and the other by a single test case;
  - CD02: which contains one fault revealed by a single test case;
  - CD03: which contains one fault revealed by a singe test case;
  - CD04: which contains two faults, revealed by a single test case each;
  - CD05: which contains two faults, one revealed by three test cases, and the other one by just a single test case.

Since our objective is to provide a hint and detect a related fault earlier in the testing process, each experiment object is a pair $< ts, f >$, where ts is a test suite related to a use case and f is a fault recorded by executing this test suite. For instance, for CB01, we considered as objects both <CB01,Fault1> and <CB01,Fault2>. Thus, according to the list of models and faults discussed previously, we worked with 19 experiment objects. Although we are not able to provide the artifacts itself, Table 4.4 summarizes some metrics about them.

Planning and Design

Our experimental units are the pairs <HARP algorithm, resemblance function>, and the treatments are both good and bad hints (defined using the process in Section 4.3.1) for each pair $< ts, f >$. In order to increase precision and power of statistical analysis, we repeated independently the application of each experimental unit and treatment to every experiment

Table 4.4: Metrics about the artifacts used as experiment objects

	Biometric Collector					Document Control				
	CB01	**CB03**	**CB04**	**CB05**	**CB06**	**CD01**	**CD02**	**CD03**	**CD04**	**CD05**
Branches	9	9	11	17	4	7	7	5	2	9
Joins	3	3	4	11	0	4	4	5	2	9
Loops	2	2	1	6	0	2	2	0	1	0
Max Depth	12	20	36	32	14	22	22	12	14	12
Min Depth	8	4	14	12	6	10	10	10	14	10
Gen. TC	16	14	16	67	3	10	10	5	3	9

object 1000 times, according guidelines proposed by Arcuri and Briand [3]. Figure 4.1 presents the experiment overview.

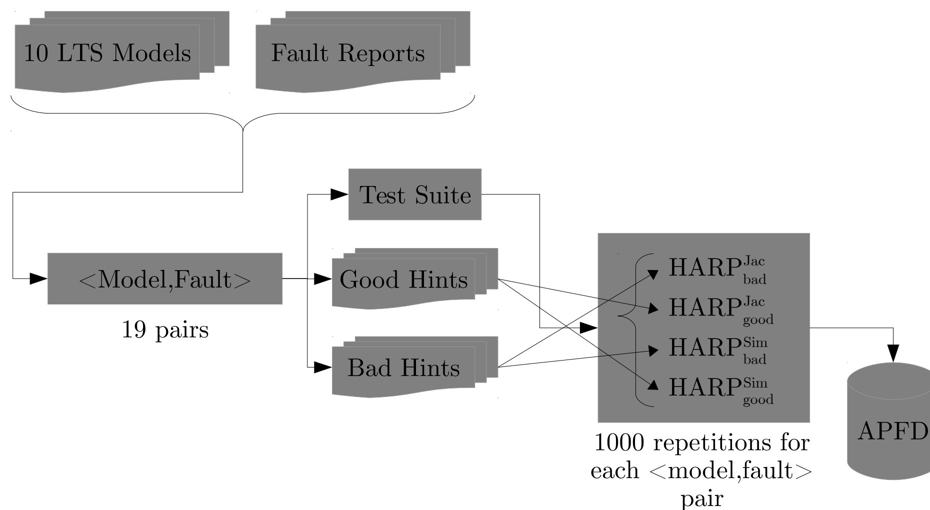


Figure 4.1: Experiment Design Overview

Results and Analysis

In this section we present and discuss the results of the study. As a first insight about the collected data, we perform a visual analysis (see Figure 4.2). From this figure we remark: i) HARP with good hints is more accurate because of a more compact box plot, however we notice some outliers; ii) it is already noticeable the difference between good and bad hints, regardless the applied resemblance function; iii) both functions appear to have a similar behavior across the levels of the variable hint quality.

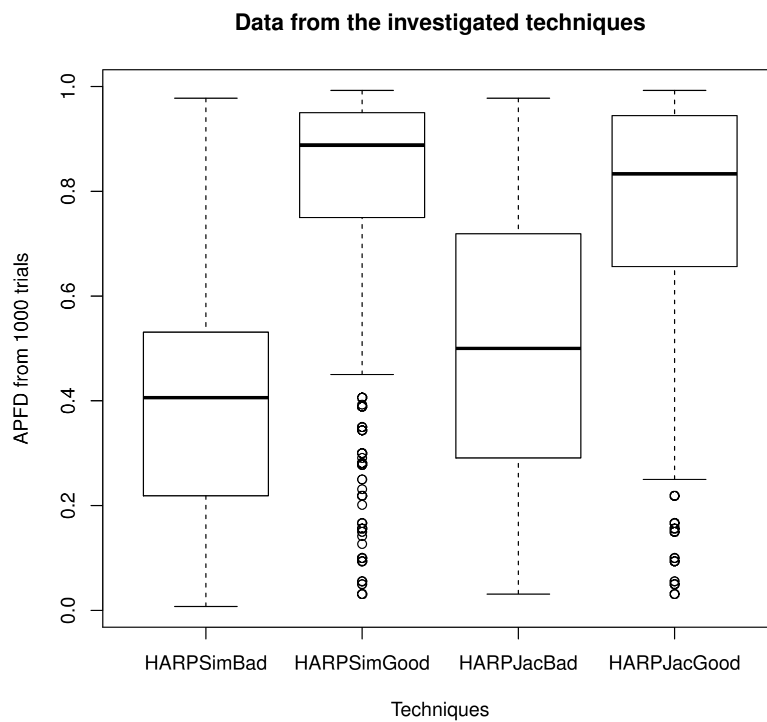


Figure 4.2: Boxplots of the raw data collected in the experiment execution.

Considering both alternatives that receive good hints (*HARPSimGood* and *HARPJacGood*), one is able to notice some outliers, marked as small hollow dots in the respective box plots. It happens mainly because the HARP's candidate set generation may take many iterations to select a hint related test case. It varies based on the sampled values from the uniform random variable that guides the process; although very uncommon, these outliers are not mistakes. Therefore, they must compose the sample to investigate the differences through the effect sizes.

Effect Size Analysis To measure the effect sizes and hence properly compare the treatments, we perform a set of parwise comparisons with the $\hat{A}_{12}$ statistic and Table 4.5 the calculated values.

Table 4.5: Effect sizes of pairwise comparisons

Function - Hint	**Technique B**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**Effect Size**
Similarity - Bad	Similarity - Good	0.0713	Large
Similarity - Bad	Jaccard - Bad	0.3628	Small
Similarity - Bad	Jaccard - Good	0.1271	Large
Similarity - Good	Jaccard - Bad	0.8484	Large
Similarity - Good	Jaccard - Good	0.5801	Small
Jaccard - Bad	Jaccard - Good	0.2237	Large

Addressing **HARP_Q1**, we evaluate the effect sizes separately between the pairs HARPSimBad | HARPSimGood and HARPJacBad | HARPJacGood, isolating the effects of each resemblance function. The $\hat{A}_{12}$ statistic value for these comparisons are 0.0713 and 0.2237 respectively, which suggests that on both comparisons the effect sizes are large favoring the alternative with good hints; in other words:

> The quality of a hint affects significantly HARP's effectiveness.

As a further analysis, we observe whether the investigated resemblance functions act differently during HARP operation. To do so, we compare the effect sizes of the pairs HARPSimBad | HARPJacBad and HARPSimGood | HARPJacGood, i.e. changing the resemblance

function but keeping the level of the variable hint quality. The values for A statistic are 0.3628 and 0.58 respectively. Both values indicate a small effect size, although the first one is close to the medium effect size threshold. Therefore, we do not have enough evidence to affirm that any function is better or more indicated to be used in HARP, in other words both functions express the same notion of resemblance. On the other hand, the similarity function proposed by Coutinho et al. [15] already takes into consideration the particular context we work, which is system level testing, with test suites based on labels representing user actions, conditions and expected results, potentially generated through MBT approaches. Because of that, we suggest using the similarity function proposed by Coutinho et al. instead of Jaccard function.

Threats to Validity

We evaluate the validity of our experiment by discussing its threats and how we deal with them. Concerning **internal validity**, the variation of APFD might be also affected by random choices during the techniques execution. In order to deal with this threat, we repeat the executions according to Arcuri and Briand's work [3]. Besides, the execution order for each technique in every repetition is defined randomly to ensure independence among them.

About **construct validity**, since we generate the hints based on prior knowledge about faults and test cases that fail, the process of defining good and bad hints may be biased. Aiming to mitigate this risk, we provide a systematic procedure to do that and we submit both good and bad hints to all experimental units, in order to distribute evenly the effects on them.

Concerning the **external validity**, even though we use real and industrial systems, a sample with only two systems are not enough to provide proper generalization. Although we were not able to generalize the results due to sample size, we try to be as most realistic as possible by using models that represent the investigated systems, so we believe that similar conclusions would hold for similar systems.

4.3.2 Exploratory Case Study

After evaluating the hint's quality in a controlled environment, in order to address our other research questions, we compare the performance of HARP with our baseline, which is the adaptive random prioritization technique proposed by Jiang et al. [38]. For the comparison, we collected a set of hints from development teams using a questionnaire, and applied these hints in an empirical comparison between both techniques, labeled as **HARP** and **ARTJac** respectively.

Applied Questionnaire

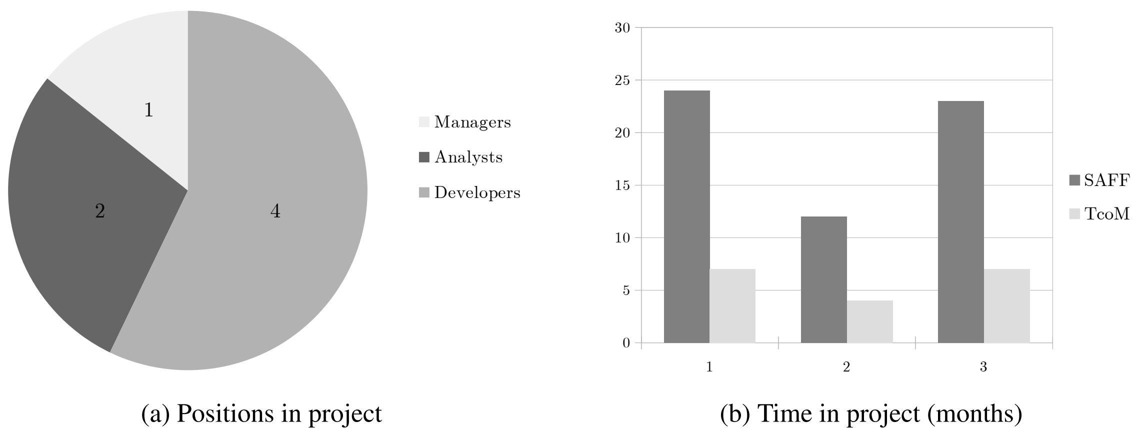
Introducing the idea of getting information from the development team, we perform a questionnaire [1] to collect indications about portions of the system that suffered some occurrence during its development, which are the hints, and investigate whether these hints are related to faults. This is an observational method that helps to assess opinions and attitudes of a sample of interest. In the following sections, we discuss its methodology and results.

Participants and Investigated Systems For this investigation, our population of interest are small development teams that have an interaction with both industry and academy. Therefore, we consider teams from two industrial projects of the Ingenico do Brasil[3]. SAFF is an information system that manages status reports of embedded devices; and TMA is a system that controls the execution of tests on a series of hardware parts and manages their results. More details regarding these systems, as well as their behaviors, cannot be unveiled due to a non-disclosure agreement (NDA).

Since it is hard to find available industrial development teams for participating on our investigation, and as we do not intend to generalize the results for a larger population, we establish a **non-probability sampling strategy** [42]. With that sample, we believe that the teams performing in research laboratories in partnership with companies are well represented. Therefore, four members from SAFF and three from TMA, which are assigned in the project during different intervals of time, compose our sample, as reported in Figure 4.3.

To select the target use cases, we checked the use case documents of both systems, which are described using a natural language-based use case notation and previous system testing

[3]www.ingenico.com.br



(a) Positions in project

(b) Time in project (months)

Figure 4.3: Summary about the questionnaire's participants.

execution results. Among them, we select use cases that have at least one reported fault. These reports consist of the test cases that fail and a cause of the failure in high level abstraction, instead of a bug in related code, since the context is system level testing. Among the considered use cases, and aiming to keep a balanced design, we selected two of them for each system. Note that there are seven participants and different numbers from both teams participated; whereas for TMA the three participants worked in both use cases, for SAFF one developer worked in one of the investigated use cases and other developer in another, which leads to four participants.

The participants present different maturity levels, varying from undergraduate students with development experience to professionals. SAFF is an ongoing project and it is been under development for the last two years, whereas TMA for the last seven months. However, one participant of each team started working on the project after it had started, which balances the participants experience in the projects. Figure 4.3b summarizes the time of activity of the participants in their projects. Note that the participants acted on the development of the use cases they responded the questionnaire about, using the model-based process described by Jorge et al. [40], in other words the participants designed and/or implemented the use cases considered in this study, therefore they actually faced the reported problems.

Questions and Related Information In order to address **HARP_Q2** and **HARP_Q3**, we devise a questionnaire intending to reveal three kinds of information:

- **Experience**: we ask the participant's position in the team, and how long the partici-

pant had been working in the project. The intention is to evaluate whether the team experience affects the ability to give hints;

- **Use case critical regions**: we attach to the questionnaire the correspondent use case document and we ask the participant to underline a region in the use case (a single step or an entire flow) that he/she believes that was jeopardized during the development activities before system level testing. For that, we instruct them to solely use his/her experience acquired when implementing the use cases. Moreover, we ask them to provide possible reasons for the choice. We provided an initial list of reasons: *Inherent complexity* – when the use case presented aspects that leaded to a complex implementation; *Neglected due to schedule* – when some other demand had to be satisfied in advance and the schedule became shorter; and *Already present any sort of problem before* – when the indicated step or flow presented a non-expected behavior in early steps of development and unit testing. Besides, the participants are free to write down any other reason they find adequate. Comparing the answers of this question with the faults and failures reported, we are able to know if it is really possible to gather good hints;

- **How time-consuming and hard was to perform the task**: we also ask to measure the time in minutes they spend to read the attached artifacts and to answer the questionnaire, and whether the participant considers it difficult to read the provided use case document and to answer the related questions. We want to evaluate whether the whole process to obtain hints is feasible.

Collected Data and Results In order to address **HARP_Q2**, we resort to the portion of the questionnaire in which the participants underline their hints and explain their reasons. We guide our analysis by observing whether the major part of the participants indicate the same portions with the same reason. A summary of this analysis is available in Table 4.6, and the bold face cells mark the situations that the participants assign the same hint, i.e. they indicated the same portion of the use case and the same cause.

Considering the Use Case 1 from SAFF (UC1), the participants report an inherent complexity problem hint, but two of them point the same step. Analyzing the fault report related

Table 4.6: The indications collected from the questionnaires, as well as their reasons. In bold we highlight regions that a majority is achieved.

		Participant 1	Participant 2	Participant 3
SAFF	UC1	**Base Flow/Step 4** **Inherent Complexity**	Base Flow/Step 3 Inherent Complexity	**Base Flow/Step 4** **Inherent Complexity**
	UC2	Exception Flow 1/Step 7 Previous Problems	**Exception Flow 1** **Inherent Complexity**	**Exception Flow 1** **Inherent Complexity**
TMA	UC1	Alternative Flow 4/Step 1 Inherent Complexity	**Base Flow/Step 4** **Inherent Complexity**	**Base Flow/Step 4** **Inherent Complexity**
	UC2	– No Difficulty	**Precondition** **No Difficulty**	**Precondition** **No Difficulty**

to this use case, we verify that the step pointed by the majority of the participants is directly related to a fault. In turn, for the Use Case 2 (UC2), there are divergences about the reasons of the indications. Two participants report an inherent complexity whereas the other assigned it to problems that already appeared during the development and that is not unveiled on early tests (unit testing). However, when relating it to the actual fault, all participants successfully report the same flow, and it is also related to a fault. Therefore, for this use case, the participants also provide a useful hint.

Now, considering the first use case (UC1) of TMA project, the participants report that an inherent complexity regarding the underlined region could be the source of some problem. Two of them point the same step in the base flow and this step is related to a fault, according to the fault reports. On the other hand, concerning Use Case 2, a different but possible scenario emerges. Every participant wrote freely that they had no difficulty developing this use case, but two of them report the use case's precondition as the most affected region, and the other one do not provide any hint. The fault report contained a fault for this use case, but it is not related to the provided answers. In a conversation out of investigation they discussed that the related precondition is hard to be satisfied, since some network requirements should be met. This is a situation that requires further investigation, since it is hard to measure how a precondition can be related to fault detection.

Considering the use cases where a majority of participants indicate the same portion with the same cause, allied to the fact that every participant worked in the respective use case, in three out of four, they provide a good hint, i.e., a region that actually contained a fault. Therefore, based on our results, we are able to infer that:

> Members of development teams, with similar characteristics than the investigated here, may provide good hints about functionalities that they have worked with, but it is necessary to observe whether the members agree on their conclusions.

To address **HARP_Q3**, we analyze the measurements in minutes collected from the questionnaires and the answer of a question about how hard they think that is to perform the questionnaire. All participants found the task of finding error-prone regions in a use case easy to perform. Moreover, the whole questionnaire was applied on an average of 4.17 minutes when considering the SAFF use cases, and 3.67 for the TMA ones.

Considering these results, we suggest that:

> The impact of using this approach to collect hints in other projects with similar characteristics tend to be low.

The most direct way of collecting those hints is how we did in this questionnaire, which was asking directly developers and managers to point the steps from the use case. Similar questionnaires could be also used in other industrial projects without much effort. We believe these questionnaires should be handed right between a development and a system testing stage, which is the moment the team still has fresh knowledge regarding what was recently developed. Alternatively, these hints could be collected iteratively during the development phase and then used right before the test case execution to prioritize the system level test cases. Moreover, we believe that would be possible to collect the same information in different ways, for instance including them in commit messages, which enables automatic collection.

With the collected responses, we generated hints manually, encoding them in prioritization purposes according the description in Section 4.1, in order to feed the empirical comparison between HARP and ARPJac.

Table 4.7: Metrics about the test suites investigated in the case study. We measure the shortest and longest test cases in amount of steps, expected results and conditions that the test case describes.

System	**#TC**	**Shortest TC**	**Longest TC**	**#Faults**	**#TC that fail**	**#TC filtered by hint**
SAFF	60	3	89	9	13	6
TMA	32	7	41	5	8	8

Planning and Design

We followed the system testing activities on SAFF and TMA systems, which comprised **executing manually the available test cases** on the systems, which ran on small terminals with a proprietary operational system, and **recording their results**, associating the test cases that failed and a cause described in a high level of abstraction. Therefore, we collected these artifacts and use them in our case study. To illustrate the dimensions of the executed test suites, Table 4.7 summarizes relevant testing related metrics.

Note that the number of hints for each system is different because, as a result of the questionnaire, there was a use case in TMA system where the participants did not report any problem, therefore we consider just the hint for use case 1.

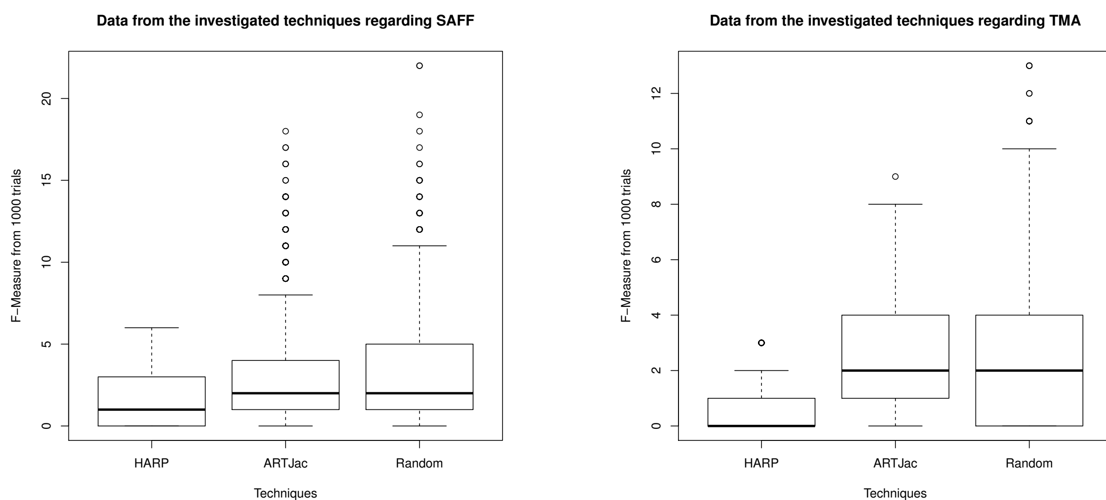
In this case study, just HARP uses hints to guide its operation, and both techniques take the two aforementioned test suites as input. We need to repeat the executions of these techniques because they both make random choices. Therefore, we run each technique 1000 times independently with the same input, as suggested by Arcuri and Briand [3].

As measure of fault detection capability, instead of measuring the APFD, which takes into consideration the detection rate of all faults that the test suite is able to unveil, we use F-Measure [91], which considers the ability of detecting the first fault. A test case reveals a fault when it fails, in other words, when the system produces outputs different from the expected. We are using F-Measure instead of APFD because the participants provided just a single hint for each use case in the questionnaire and when applying this hints in the whole test suite, would be unfair that the other use cases not investigated affect the measurements.

Results and Analysis

A visual representation of the collected data is depicted in Figure 4.4. Besides HARP and ARTJac, we added the Random technique as a lower boundary for techniques' performance. A good performance presents a low F-Measure value, which ranges between 0, when the first test case already reveals a fault, to $n - 1$, when the last test case reveals a fault. Since both test suites have different sizes, we analyze both of them separately.

Regarding SAFF, based on how spread are the boxplots in Figure 4.4a, HARP presents a more accurate performance than the other two techniques, presenting fewer outliers; however visually it is not possible to conclude which of the techniques present the lowest F-Measure. By calculating the effect size of the comparison between HARP and ARTJac we obtain 0.4106 favoring HARP, but with a small effect size. It suggests small gains through the application of HARP.



(a) Boxplot with the results from SAFF system

(b) Boxplot with the results from TMA system

Figure 4.4: Summary about the collected data.

On the other hand, analyzing visually the boxplots from Figure 4.4b, the improvements either in accuracy or in earlier fault detection are clearer. The effect size measurement of the comparison between them is 0.2442, which means a large effect favoring HARP. Thus, considering these two systems and addressing **HARP_Q4**:

Considering the available sample, HARP is able to outperform ARTJac when using actual hints.

Even though the small differences between the techniques regarding the median of F-Measure, which are the lines in the center of the boxplots in Figure 4.4, this difference may still provide significant savings in the MBT context and in a context with manual test case execution. For instance, during the development of the systems considered in our evaluation, reports of the test case execution show they took between 8-60 minutes, including setup, evaluation, and recording the results using tools. In general, a single test case execution took 16-27 minutes and any reduction in the amount of test cases executed before revealing faults is representative, in a process point of view.

Threats to Validity

In this section, we discuss aspects that may represent some limitations of case study. To do so, we discuss separately the questionnaire and the comparison itself.

Regarding the questionnaire, when analyzing our results, we must be aware that the participants answered the questionnaire after the use cases were implemented. Therefore, we rely on their memory regarding the system. To reduce the learning effect and the memory bias, we asked about critical regions and not explicitly about steps and/or flows that could contain faults. The relationship between the responses from the questionnaire and the revealed faults was established afterwards by comparing both artifacts. Moreover, as we intended to investigate developers from projects mixing undergraduates and professionals in cooperation with industry, we were limited to a small and selective sample.

Due to our limited sampling strategy, we are not able to generalize our results to different contexts. However, since all elements involved in this study were generated on real projects (e.g., real developers, systems, and use cases), we believe our results are still valid. Nonetheless, our results suggest that similar questionnaires can be used in different contexts.

The sampling strategy and size also affected the comparison itself. We are not able to draw more general conclusions due to the low number of systems in our case study, however both systems are industrial. Moreover, the hints were provided by the actual development teams. Although we use two hints from SAFF and one from TMA, we are sure that the hints

were given by people that really worked in these use cases. Therefore, to reduce the effect of the other use cases' faults in the ability of reveal faults, we use F-Measure instead of APFD in the case study.

4.4 Chapter Final Remarks

In this chapter we report details about HARP, including: what are the hints and how they are encoded; how HARP uses them, as well as details of HARP's operation; a thorough asymptotic analysis of its algorithm; and a running example considering a test suite already explored on previous moments in the document in order to illustrate its operation. The differences between HARP and its bottom line, which is the original Adaptive Random Prioritization strategy, are now clear, including the aspects we intervened and the reduced effect of random choices.

Furthermore, we report two studies. The first one an experiment evaluating the effect of hints' quality on the effectiveness of HARP, and the second one is a case study comparing HARP with the original adaptive random prioritization, using actual hints collected from two development teams. Whereas the experiment suggests that HARP is affected by hints' quality, the case study shows that development teams may provide hints able to improve HARP's effectiveness in comparison with its baseline.

Regarding scalability, note that the two investigated test suites, SAFF and TMA, contain 60 and 32 test cases respectively (see Table 4.7). It is already a substantial number of test cases, mainly considering manual execution, but we believe that for much larger test suites, the cost of applying HARP could increase due to the hint collection process. If it is turned to an automatic process, for instance based on analyzing commit messages, this cost would be alleviated. Therefore, we argue that the limitations for applying HARP are more related to the hint collection process than the size of test suites to prioritize.

Thus, based on the results of both studies, we suggest using HARP just with hints pointed by a majority of the team members, thereby we believe that situations that a bad hint misguide HARP's operation are minimized. For the situations we suggest not using the collected hints, i.e. when we believe that a hint could misguide HARP's execution instead of being helpful, we propose another technique whose details we discuss in the next chapter.

Chapter 5

Clustering System Test Cases for Test Case Prioritization - CRISPy

Motivated by the results presented by other researches on clustering and software testing, exploring another underlying strategy to approach TCP, and aiming at complementing HARP's usage, we propose CRISPy. The investigation discussed in this chapter addresses our fourth research question, which is:

> *(RQ4) How can we prioritize test suites, even when the expertise available is not enough to guide TCP, i.e. what to do when this expertise would hinder instead of being useful?*

CRISPy is targeted for black-box system test suites and operates without expert intervention. Whereas HARP focuses on MBT test suites, CRISPy does not make this assumption, trying to generalize for a more general manual black-box system testing context. Besides, CRISPy also tries to diversify the steps covered by the available test cases using a notion of distance.

As we discussed in Chapter 3, techniques with random choices are less affected by characteristics of test cases that fail, specifically their size, since they do not apply judgment based on these characteristics; on the other hand, they present a certain lack of accuracy. In this way, we resort to a cluster technique that is not affected by random choices, and therefore it might present more accurate results. Besides, it uses functions that do not judge the distance of test cases based on their sizes, for instance Jaccard or Levenshtein distance.

In the next sections we present: the proposed algorithm (Section 5.1), as well as a running example to illustrate its operation; the empirical evaluation (Section 5.2), where we presented planning, results, and validity of our studies; and finally the final remarks (Section 5.3).

5.1 Proposed Algorithm

CRISPy uses a different way of inserting variability to the TCP process, which is grouping similar test cases together using a clustering strategy, and sampling them observing their sizes. We understand that it could also be a way of being biased by the size of test cases that fail, but we believe that the choice of distance functions, and the clustering process itself, may keep this bias reduced.

CRISPy comprises five major steps: calculating pairwise distances between the input test cases; calculating the limit distance, which is how close two clusters should be to be merged together, using the distances and a threshold; cluster definition following the Agglomerative Hierarchic Clustering (AHC); and the Interleaved Clustering Prioritization (ICP), which put clusters and test cases in order.

5.1.1 Cluster Creation and Thresholds

Following Algorithm 11, in order to calculate the distance between the pairs of test cases (Line 3), the algorithm uses a distance function and stores the values is a matrix. It is a distance function because low values mean that the compared test cases are similar and high values mean that the test cases are different. Besides, the function does not need to return normalized values, since next step takes care about that. We experiment three different distance functions on our empirical studies, which will be detailed afterwards.

At this point, the algorithm has already calculated the minimum and maximum distances and, in Line 4, we define a range of values representing the most similar test cases. The distance **threshold** is a percentage that defines the size of this range by normalizing the distances. If we define a small distance threshold, say 0.1 or 10%, the range is very narrow, which means that few clusters will be merged in the next step, ending up with several small clusters; otherwise defining a high threshold, say 0.7 or 70%, the algorithm tends to merge the majority of the clusters together.

Algorithm 11 The main procedure.

```
1:  function PRIORITIZE_CRISPY(U_TS)            ▷ U_TS is the not yet prioritized test suite
2:      prioritizedSequence ← []
3:      dMatrix ← calculatePairwiseDistances(U_TS)
4:      limDist = minDist + [(maxDist − minDist) · threshold]
5:      clusters ← AHC(U_TS, limDist)
6:      for all c ∈ clusters do                  ▷ intra-cluster prioritization
7:          c.sort()
8:      end for
9:      clusters.sort()                          ▷ inter-cluster prioritization
10:     i ← 1
11:     while clusters.size() > 0 do
12:         next ← clusters[i, 1]                ▷ Selects the first test case from i-th cluster
13:         prioritizedSequence ← next
14:         clusters[i].remove(next)
15:         if clusters[i].size() == 0 then
16:             clusters.remove(i)
17:         end if
18:         i ← (i + 1) % clusters.size()        ▷ Interleaves clusters that still contain test cases
19:     end while
20: return prioritizedSequence
21: end function
```

The next step is to calculate the clusters using AHC, illustrated in Algorithm 12. The main idea is to begin with unitary clusters, one for each test case (Line 3-5), try to find a pair of clusters close enough to be merged, i.e. if the distance between these two clusters falls in the range $[minDist, limDist]$, and then merge them (Line 7-9). In order to be conservative, we consider the distance between two clusters as the minimum among the test cases from both clusters.

Algorithm 12 Agglomerative Hierarchical Clustering.

1: **function** AHC($U_TS, limDist$) ▷ $limDist$ how close clusters should be to be merged
2: $clusters \leftarrow \emptyset$
3: **for all** $tc \in U_TS$ **do**
4: $clusters.add(cluster(tc))$ ▷ One cluster for each test case
5: **end for**
6: $closeClusters \leftarrow true$
7: **while** $closeClusters$ **do**
8: $closeClusters \leftarrow fusePairClusters(clusters, limDist)$
9: **end while**
10: **return** $clusters$
11: **end function**

5.1.2 Clusters and Test Case Prioritization

Following the steps after the AHC in Algorithm 11, it performs the ICP with the clusters calculated in the previous step. It prioritizes test cases from each cluster (intra-cluster prioritization, Lines 6-8), then prioritizes the clusters (inter-cluster prioritization Line 9), and finally interleaves the first test cases from the clusters (Lines 11-19), always respecting both inter and intra cluster orders. Regarding intra-cluster prioritization, the algorithm sorts decreasingly the test cases by their size, i.e. test cases containing more steps have preference over the shorter ones within each cluster. Similarly, regarding inter-cluster prioritization, the algorithm sorts the clusters decreasingly by the average size of their test cases.

Note that we took measures to implement a stable algorithm aiming to reduce the effect of random choices, e.g. both intra and inter cluster sorting are stable. At the same time, we

are trying to increase the chances of detecting faults by giving preference to longer test cases from each cluster. Note that after the clusters are assembled, interleaving longer test cases from each cluster leads to diversification. At the same time, we understand that it could also be a way of being biased by the size of test cases that fail, but we believe that the choice of distance functions keeps it controlled.

5.1.3 Asymptotic Analysis

In order to provide a more general evaluation of CRISPy's algorithm, we analyze its asymptotic bounds. Consider a test suite with n elements and, for simplification, assume that the test cases from the original test suite have the same size, t. The execution time for every step is as follows:

1. `calculatePairwiseDistances`: In order to calculate the pairwise distances of the test cases to be prioritized, this function assembles a $n \times n$ matrix and fills one of the diagonals with the distances between every pair of test cases, since most distance functions are commutative. Therefore the cost of filling the matrix is $\frac{O(n^2)}{2} \cdot d$, where d is the evaluation cost of the distance function. Assuming this cost as a function of the sizes of the compared test cases, the total cost is $t \cdot \frac{O(n^2)}{2}$ or $t \cdot O(n^2)$;
2. `AHC`: The Agglomerative Hierarquical Clustering is a classic algorithm, which operates in $O(n^2 \log(n))$;
3. `Cluster sorting`: For this task, there are two levels of sorting, the intra-cluster sorting depending on the size of each cluster, and the inter-cluster sorting depending on the number of clusters formed. The worst case cost is when just a single cluster is formed, which leads to cost $n \log n$ from a classic sorting algorithm;
4. `ICP`: To interleave the sorted test cases from the sorted clusters is always a matter of performing n operations, since every test case must be in the prioritized sequence;
5. `prioritize_CRISPy`: This is the main function, which calls the other ones to prioritize the input test suite, therefore its cost is CRISPy's total execution cost. The total cost is: $t \cdot O(n^2) + O(n^2 \log(n)) + n \log n + n$, which is equals to $O(n^2 \log(n))$.

5.1.4 Running Example

To illustrate its operation, we follow a running example on the same test suite we used for the previous technique (refer to Table 1.1). Notice that the test cases always begin from the step "Show the login and password screen" and follow their steps, which means that they present redundancy. In our example, we demonstrate the execution using Jaccard function with a threshold of 0.4 or (40%).

Each point in Figure 5.1 represents the distance of a pair of test cases from our example. The minimum distance is $d(TC_4, TC_6) = 0$, the maximum is $d(TC_2, TC_7) = 0.75$, and since the considered threshold is 0.4, the distance range is $[0, 0.3]$.

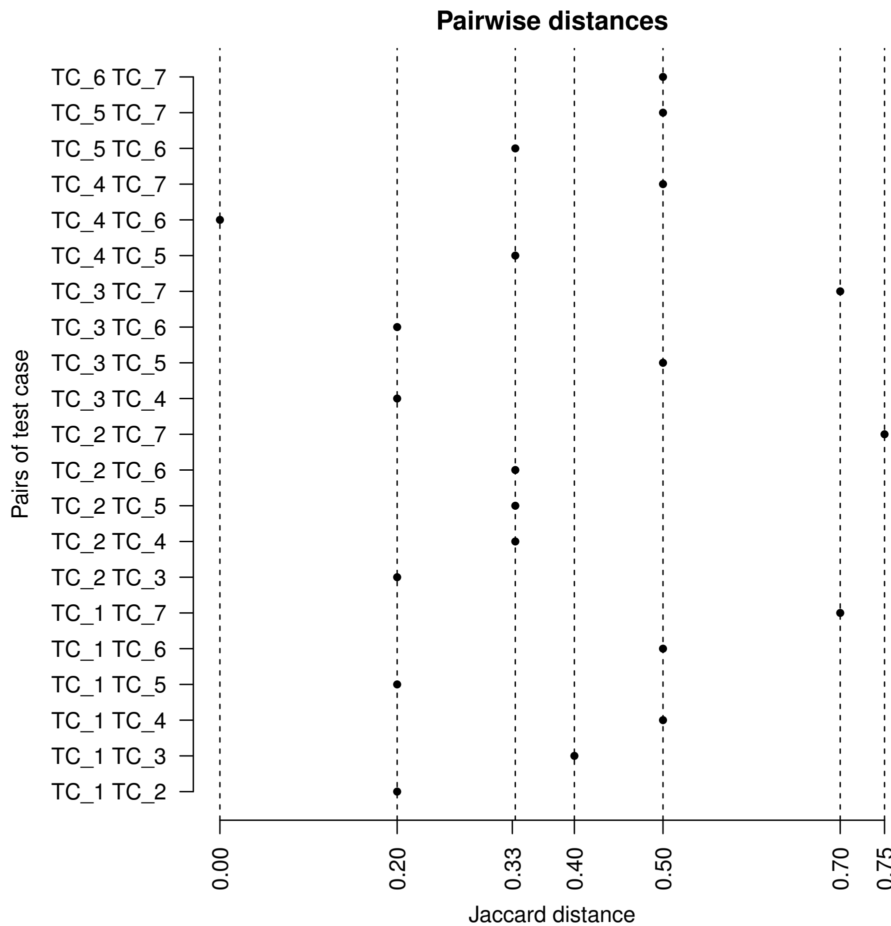


Figure 5.1: Graphical demonstration of the Jaccard distances between test cases.

Figure 5.2 summarizes each step of the merging process. One can notice, for instance, when the algorithm tries to merge clusters $\{TC1, TC2, TC5\}$ and $\{TC3, TC4, TC6\}$, the minimum distance is $d(TC2, TC3) = 0.2$, then in the next step, they appear in the same

cluster. On the other hand, the minimum distance between any other test case and TC7 is 0.5, which is greater than the limit distance, therefore it will be placed alone in a cluster, as in the last step in the aforementioned figure.

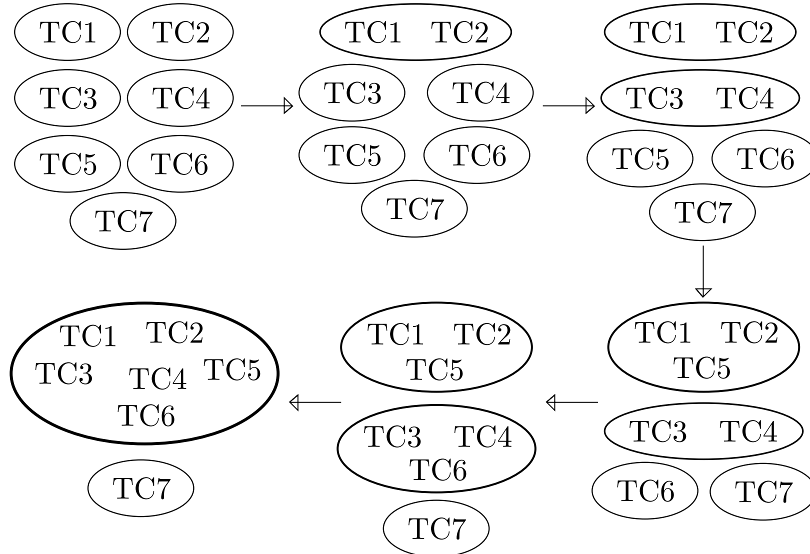


Figure 5.2: Graphical demonstration of the cluster merging step.

On our example, CRISPy suggests the sequence $[TC2, TC7, TC3, TC5, TC6, TC4, TC1]$ as final result.

5.2 Empirical Evaluation

We intend to provide two degrees of assessment, one regarding the effectiveness of the proposed technique in comparison to other techniques from the literature, and other with respect to which aspects one could modify the proposed technique aiming at tuning it. Therefore, we investigate the following research questions:

- **CRISPY_Q1 - How does CRISPy perform in comparison to other TCP techniques in the same context?** We perform this comparison with a basic configuration of CRISPy in order to show its capabilities;

- **CRISPY_Q2 - Regarding tuning, to what extent different distance functions and thresholds affect the performance of the proposed technique?** On clustering tech-

niques, modifying attributes in order to represent better the notion of distance is often a way of tuning.

- **CRISPY_Q3 - How CRISPy perform in comparison with HARP, considering hints with different qualities?** We investigate that in order to demonstrate the ability of CRISPy to complement HARP in situations that collected hints are not indicated to be used.

To address these research questions, we perform three separate studies. For CRISPY_Q1, we set up an empirical comparison involving a relevant set of techniques, including CRISPy, evaluating their ability of revealing faults (see Section 5.2.1). To address CRISPY_Q2 we perform an experiment, evaluating the effect of different distance functions and distance thresholds on the performance of our technique (Section 5.2.2). In order to answer CRISPY_Q3, we repeat the experiment reported in Section 4.3.1, but adding CRISPy, aiming at verifying how it compares to HARP performing with good and bad hints (refer to Section 5.2.3).

As a supplementary source of details and a place to gather images, data, and data analysis scripts, improving the reproducibility of our study, we prepared a specific sub page in our companion site, available at `https://goo.gl/mQGmJL`.

5.2.1 Comparative Case Study

To address our first research question regarding CRISPy, we carry out an empirical comparison using a set of 17 test suites, related to six different systems. We compared CRISPy with several other techniques, which adopt different strategies to solve the TCP problem.

Investigated Variables

In order to answer CRISPY_Q1, we configure CRISPy using Jaccard distance as distance function and using a distance threshold of 40%. We have done so because Jaccard function already appear on this study on other technique and we do not want other source of variation. Besides, this threshold seems reasonable to make sure that the similar test cases be together in the same clusters, but being conservative at the same time.

For this comparison we control and observe the following variables:

- **Independent Variables**
  - TCP techniques: opt, rand, RA, SD, PC, ARPJac, Fuzzy, CRISPy;
  - Test Suites: the same set of 17 test suites involved in the study reported in Chapter 3, whose details are in Table 3.1.
- **Dependent Variable**
  - Average Percentage of Fault Detection - APFD

All these techniques, detailed in Chapter 2, receive the same artifact as input, which is a test suite, and provide a prioritized test suite as result.

Planning and Design

Since the objective is to compare the techniques, we execute each one of them with every test suite as input, submitting the techniques to the same workload. Although our technique is deterministic, several others rely on random choices. Therefore, we execute all of them 1000 independently scheduled times to observe better the trends among the possible results [3], which suggests a balanced design. For each repetition, the execution order for the techniques is defined randomly to reduce any bias related to the execution environment, and we measure their effectiveness through APFD.

Results and Analysis

Figure 5.3 provides an overview of every technique. Test suites are an important source of variation, since even **Opt** varies and the other techniques present a stretched box plot. As we already would expect, techniques based on some random choice (**ARPJac**, **Fuzzy**, **Rand**) present greater variation. On the other hand, excluding **Opt**, **Crispy** and **PC** present smaller variations, which means less variation across test suites. We argue that this variation is a sign that we still need to understand and control more aspects of the test suites/systems to provide a more precise assessment.

Verifying assumptions for using parametric tests, by visual analysis of q-q plots, normality tests (Anderson-Darling, Cramér-von Mises, and Kolmogorov-Smirnov) and variance

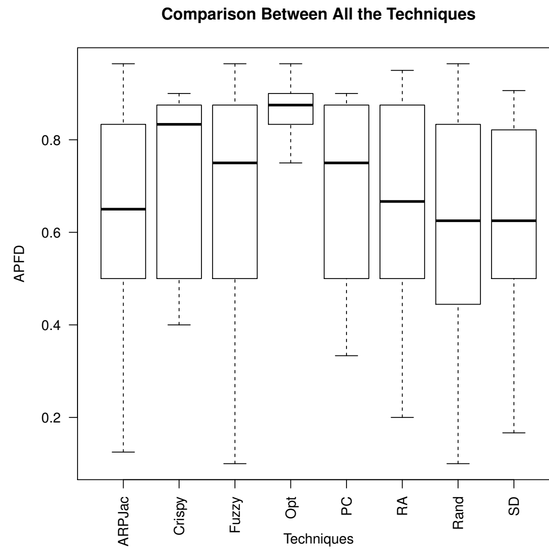


Figure 5.3: Box plot with the samples from each technique, showing their medians and spreading.

test (Fligner-Killeen), we do not have evidence to support neither the normality of residuals nor equivalence of variances (homoscedasticity). Therefore, since we have a representative sample size, we proceed with non-parametric analysis.

To compare the performances of the techniques, we resort to pairwise Mann-Whitney tests and effect size analysis through the $\hat{A}_{12}$ statistic [67, 85], which assesses the samples, suggest the one that present the higher performance and suggests how relevant this difference is. Regarding the results of Mann-Whitney tests from Table 5.1, only three pairs of techniques are comparable and one of them (**PC** and **RA**) present a high $p - value$. On the other hand, regarding effect size analysis and complementing the visual analysis from Figure 5.3, **Crispy** is the only technique that has advantage over all the others (excluding **Opt**); in comparison with **Rand** and **SD**, the effect sizes are very close to the medium effect size (the limit value is 0.64).

Even though all effect sizes measured are considered small, considering the high costs of manual setup and execution, and the lack of complementary information, any improvement on fault detection would be a benefit to testing teams. Therefore:

Table 5.1: Nonparametric pairwise comparisons: dark grey (upper right) area are the effect sizes calculated by $\hat{A}_{12}(i, j)$, where i is the technique from the row and j is the one from the column. Analogously, the light grey (lower left) area contains the p-values of Mann-Whitney tests comparing j and i. We omitted the values with lower orders of magnitude to improve visualisation.

	ARPJac	Crispy	Fuzzy	PC	RA	Rand	SD
ARPJac		0.388	0.423	0.413	0.431	0.528	0.494
Crispy			0.539	0.527	0.527	0.631	0.602
Fuzzy				0.488	0.494	0.597	0.568
PC					0.498	0.604	0.576
RA			0.074	0.654		0.592	0.566
Rand							0.466
SD	0.079						

Even a standard configuration of CRISPy performs slightly better than the other TCP techniques considered in this comparison.

Besides, we still need to investigate other variations on CRISPy, in order to improve its ability of proposing better general results. The first investigation is presented in Section 5.2.2.

Threats to Validity

We evaluate the validity of this study by pointing aspects that could hinder it and discussing our practices to alleviate them. Regarding **construct validity**, we sampled the TCP techniques to be compared by relevance on previous researches and by their ability of representing our context of interest; we did not compare with HARP because we did not collect hints for every test suite available.

In terms of **internal validity**, we implemented and tested carefully our implementation, the other techniques, as well as the rest of our experiment environment. This whole structure was developed in Java and was instrumented to export the data for human readable text files, ready to be analyzed by our R script.

With respect to **conclusion validity**, we provided analysis of statistical and practical significance. To do so, we tested the samples for parametric analysis, reported results considering the whole distributions and in isolation by test suite, and provided effect size analysis.

Regarding **external validity**, since we evaluated 17 test suites from six different systems, we are still not able to generalize our findings for other systems from different companies and diverse domains. We understand that having a more diversified selection of test suites suitable for system testing and manual execution, from different domains and companies, is needed before we can suggest more robust and general results.

5.2.2 Experiment on Tuning Potential

In the first moment of our evaluation, we configured CRISPy with a simple and conservative set of parameters, in order to provide a first insight about its effectiveness. However, we understand that depending on the elements compared, a clustering strategy could benefit from tuning. In this study, we control two factors that guide CRISPy's execution, which are the functions that calculate the distances between pairs of test cases, and the distance thresholds, which define how close two test cases need to be from each other to be placed in the same cluster.

Investigated Variables

Since we exercise the same algorithm and we fix this variable, in order to tune it, we change their configuration parameters, which are:

- **Independent Variables and factors**
  - Distance function: Jaccard, Levenshtein (edit) distance, and the Normalized Compression Distance (NCD);
  - Distance threshold: 0.1, 0.2, 0.3, 0.4, 0.5.
- **Dependent Variable**
  - Average Percentage of Fault Detection - APFD

As we already discussed in Section 5.1, the only test artifact it takes as input is the test suite. Therefore, we use the same 17 test suites, related to six different systems, already used in Chapter 3 and in our previous study in this chapter.

Planning and Design

Since the objective is tuning CRISPy, considering the same implementation, we configure it with each pairwise combination of both factors and execute them using the same 17 test suites, as in the previous setup. We intend to investigate the effect of each factor on CRISPy, proposing a combination of them able to improve its ability of revealing faults.

Results and Analysis

We follow a similar strategy to analyze data regarding CRISPY_Q2. Verifying the assumptions for parametric tests, normality of residuals and equivalence of variances, we have no evidence to support that the residuals follow normal distribution. Since every sample is a configuration of the same base algorithm and each of them is submitted to the same workload, as expected, they present similar variances (p-value of 0.998 of a Fligner-Killeen test). Therefore, we also followed a non-parametric strategy, because the samples just have 17 data points, one for each test suite.

By checking the box plots with **Crispy** samples configured with different distance functions and thresholds in Figure 5.4, one can notice that Levenshtein function presents a greater skew to higher values than the other functions, even though the whole distributions seem similar. Since we are evaluating the configurations with the same workload, we already expect that something similar to it could happen. Levenshtein function is more sensitive to differences on strings than Jaccard and NCD, because it calculates the operations to transform one string on other. Therefore, considering that the test cases on our context are mainly sequences of human readable strings, we argue that Levenshtein function is slightly better on representing test case distances.

Regarding the Thresholds, refer to Figure 5.4, **Crispy** configured with 0.1 and 0.4 present similar skewness in contrast to the others. Since the hierarchic clustering begins with a test case in each cluster and tries to merge them according to the threshold of distances, the algorithm with 0.1 threshold merges few clusters, as depicted in Figure 5.5, and the

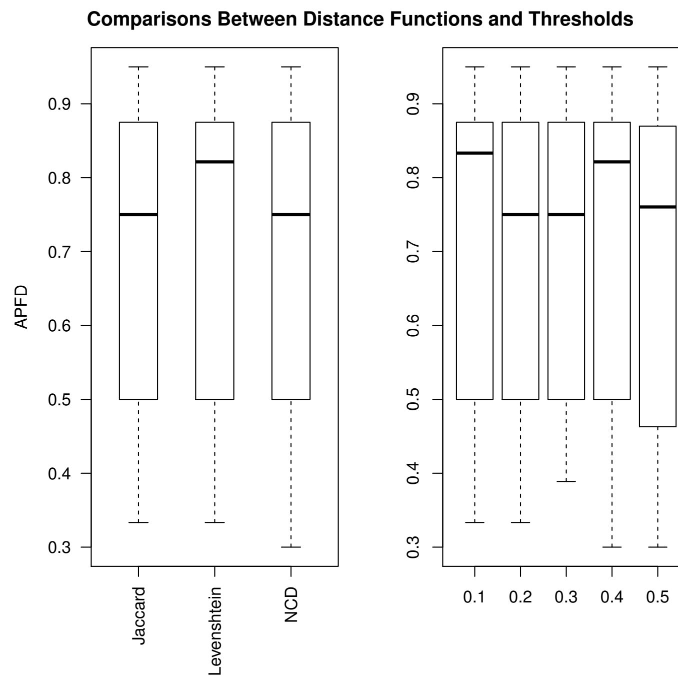


Figure 5.4: Box plots with the samples configured with the investigated distance functions and thresholds.

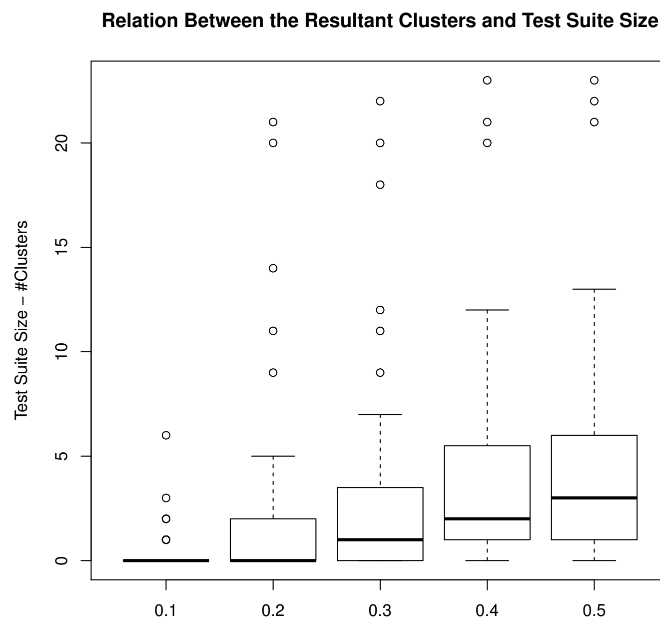


Figure 5.5: Box plots showing samples of the difference between the test suite size and the number of generated clusters, for each distance threshold. Notice that the variables are directly related.

hierarchic clustering is almost as not clustering at all. Therefore, we argue that particular characteristics from test suites affected that result. On the other hand, the 0.4 we notice the formation of different numbers of clusters across the test suites, which also evidences that fixing the amount of clusters could be a problem for traditional clustering on TCP. Thus, we argue that being radical, i.e. several small clusters versus few big clusters, is not a good strategy in our context. Summarizing:

> The distance function should be defined taking into consideration the kind of test suite to be compared. Besides, distance thresholds around 0.4 seem to be reasonable to avoid extreme situations regarding test suite size versus number of clusters.

Threats to Validity

As the other studies reported in this thesis, we evaluate its validity by discussing limitations and how we address them. Regarding **construct validity**, observe that the variable distance function presents several other possibilities than the investigated on this study. To limit the treatments, but keeping it still representative, we also sampled them based on results from previous research that point them as good candidates for the comparison between test cases. Furthermore, even though the factor distance threshold ranges from 0 to 1, we just considered values below 50% as treatments because from that point on, we would put most test cases in a single cluster, which jeopardizes the prioritization strategy.

In terms of **internal validity**, we implemented and tested carefully our implementation, as well as the data analysis script. Although we controlled the only two variable parameters of the hierarchic clustering, we know that other aspects could also affect CRISPy's result, for instance the intra and inter cluster prioritization. Here we keep them fixed to reduce this effect and we intend to investigate it in future work.

With respect to **conclusion validity**, we provided analysis of statistical and practical significance for both studies. To do so, we tested the samples for parametric analysis, reported results considering the whole distributions and in isolation by test suite, and provided further analysis on the number of generated clusters trying to explain effects from the investigated test suites.

Regarding **external validity**, since we evaluated 17 test suites from six different systems, we are still not able to generalize our findings for other systems from different companies and domains. We only can propose more robust and general results upon having a more diversified selection of test suites suitable for system testing and manual execution, from different domains and companies.

5.2.3 Experiment Comparing CRISPy to HARP

Our third study regarding CRISPy's evaluation focus on addressing CRISPY_Q3. Here we repeat the study reported in Section 4.3.1, in which we defined bad and good hints using a manual and systematic process, adding CRISPy to this structure. We do that in order to observe whether we can suggest using CRISPy when HARP is not indicated, in other words whether CRISPy performs better than HARP with bad hints.

The aforementioned study already suggests that HARP really performs better with good hints, which is showed by a large effect size for all comparisons, regardless the distance function used. Our initial hypothesis is that CRISPy performs better than HARP with bad hints, which would be an indication that they are complementary.

Investigated Variables

Already following the indications from previous studies, we just consider the similarity function for HARP and Levenshtein function for CRISPy. Besides, we use the same process to derive good and bad hints from the same test suites investigated in the original study. Therefore, our variables are:

- **Independent Variables**
  - Hint quality: good and bad hints;
  - Proposed TCP techniques: HARP and CRISPy;
- **Dependent Variable**
  - Average Percentage of Fault Detection - APFD

Planning and Design

Observe that we are not able to apply the derived hints to CRISPy, because it assumes that they are not useful; therefore, we just apply them to HARP, creating **HARPGood**, **HARP-Bad** and **CRISPy** as compared techniques. Then we submit them to each one of the ten available test suites.

Since CRISPy is deterministic, we just need to execute it once with each test suite; on the other hand, we repeat HARPGood and HARPBad 1000 times for each test suite, in order to observe any performance trend [3].

Results and Analysis

In order to provide a more general view of the data collected from the three combinations evaluated here, Figure 5.6 contains side by side box plots. It is already clear the superiority of HARPGood, as we expected; on the other hand, just comparing their medians, CRISPy performs better than HARPBad. Since they are close to each other, we should investigate how significant is the difference.

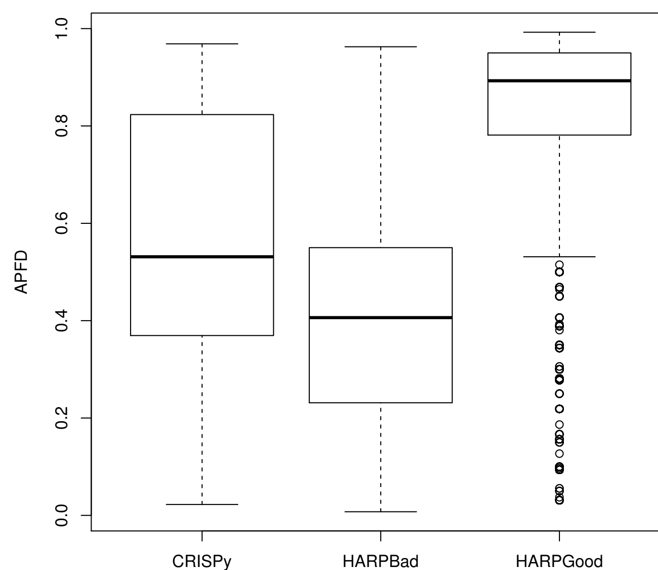


Figure 5.6: Box plots showing samples of the investigated techniques.

Analyzing the effect size of these differences through the $\hat{A}_{12}$ statistic, for the compar-

isons HARPBad | HARPGood, CRISPy | HARPBad, and CRISPy | HARPGood we have 0.0694, 0.6828, and 0.242 respectively. These results confirm the large effect size of the comparisons favoring HARPGood over the other two techniques; besides it also suggests that the effect size is medium favoring CRISPy over HARPBad.

Since CRISPy, which does not use any kind of secondary information perform significantly better than HARP misguided by bad hints, we can suggest using it when hints are not useful, in other words:

> When hints are not available or when developers and managers do not agree about them, we discourage applying them on HARP and using CRISPy instead.

Threats to Validity

As the other studies, we assess the validity of this one by discussing their limitations and how we deal with them. With respect to **construct validity**, we have the notion of bad and good hints, derived with previous awareness about fault and failures. As we already discussed in the original study, we provided a systematic procedure to derive the hints, trying to be less biased as possible.

Regarding conclusion validity, we repeated HARPGood and HARPBad 1000 times to observe the tendencies, whereas CRISPy just was run once. It would be a major problem if we have performed paired tests; since we just perform visual comparisons and non parametric effect size analysis, it does not violate any assumption.

Concerning **external validity**, we used test suites and faults from two industrial systems, which is not enough to provide the desired degree of generalization. Therefore, we expect that our findings should hold for systems and test suites with similar characteristics than the ones investigated.

5.3 Chapter Final Remarks

In this chapter, we report details about CRISPy including, its algorithm, a running example, and the empirical investigation on its effectiveness. It is a clustering-based technique, which

focuses on grouping similar test cases, according to some distance function, and interleaving test cases from each cluster.

We performed three studies to evaluate its performance: a case study comparing a basic version of it against a selection of techniques, including other clustering technique; an experiment evaluating the effect of different distance functions and thresholds; and a repetition of a study already performed to evaluate our other technique, but now involving CRISPy.

The results of our studies suggest that, even in a standard setup, CRISPy is able to outperform other techniques, but we still need to control other sources of variation in order to draw more general conclusions. Regarding tuning, we found out that depending on the structure and representation of the test suites available, a distance function able to represent better the resemblance among them benefits CRISPy. Regarding scalability, we believe CRISPy would perform well even with larger test suites, since it just takes as input a test suite to be prioritized and during the empirical studies each execution took few milliseconds.

Furthermore, in comparison to HARP, CRISPy performs better when just bad hints are available, which means that it complements HARP when the development team does not agree about the collected hints.

Chapter 6

Literature Review

We began discussing the related literature in Section 2.3.1, by detailing several TCP techniques that integrated our operation environment for empirical studies. Taking into account the initial set of papers investigated, identified as result of a systematic mapping [62], we analyze the recent literature that cites them using Google Scholar. Even though we do not perform a fully systematic review, we believe we cover the relevant literature regarding our research context.

The Test Case Prioritization problem has been investigated in several levels of testing, with different kinds of test case design; however, most of TCP techniques already proposed focus on code-based test suites and the regression testing context [12, 23, 47]. Authors explore different strategies to solve the TCP problem, including greedy reasoning [21, 55], risk analysis [79, 80], adaptive random [37, 38, 91], and soft computing methods, as genetic algorithms [34, 57, 69], clustering [9, 89], search algorithms [52].

There is a common assumption in the TCP context focusing on fault detection, which is: techniques assume that covering structural elements as soon as possible increases the chances of revealing faults earlier in the testing process [12, 77]. As example of these structural elements, we have statements, if-else branches, and functions from code-based research, and steps, branches, actions of the UML activity diagram from model-based research. In this research we also work with this assumption; besides, as already discussed earlier in this thesis, we also consider that historical information regarding previous test executions is not available, and that test cases are targeted for manual execution. Therefore, we need to explore other sources of information in order to solve the TCP problem. We survey the research on

black-box TCP techniques, including the model-based ones, and on cluster-based testing techniques. Complementing the details already provided in Section 2.3.1, we discuss the main sources of information used and the relationship with our research for each topic.

6.1 Black-Box TCP Techniques

As an initial provocation on black-box testing, one could argue: "how is it possible for a technique being able to propose a good execution order regarding fault detection, even not being aware of internal details of the SUT?" Maybe there is some relationship between elements in a high level of abstraction and the occurrence of faults. Henard et al. [33] performed a study comparing white-box and black-box TCP techniques. They claim that diversity-based techniques perform best among black-box ones, and, even though white-box techniques have more information about the SUT available, they do not present a significantly higher performance than black-box ones. This result suggests the need for further investigation about the behavior of black-box MBT techniques.

We have already introduced and detailed several techniques in this context in Section 2.3.1 and, even though some of them were originally evaluated using code-based test suites, the techniques themselves do not make strong assumptions about code coverage, for instance Elbaum et al. [22] proposed greedy techniques that prioritizes test cases based on how much code elements, such as statements and if-else branches, they exercise. Note that we can, analogously, face the coverage of model elements, such as steps or branches, or even of general steps performed by testers, as done by Hemmati et al. [32]. With that in mind, we adapted some of them, initially proposed and evaluated with code-based test suites, to work in our context as common ground of the current state of research.

In the next sections we discuss important aspects that characterize the operation of current black-box techniques, and we classify them as: structural aspects based, dissimilarity-based, and soft-computing.

6.1.1 Structural Aspects

Techniques that take into account directly structural aspects consider the model or source code elements, such as edges, nodes, branches, and procedures, and sort test cases based on

some function. Regarding MBT, Kundu et al. [48] proposed a technique here labeled **Stoop**, which is based on UML Sequence Diagrams and sorts test cases based on how common are the steps they traverse. Sapna and Mohanty [75] proposed another similar technique (labeled **FixedWeights**), which attributes fixed values for each element from the input activity diagram and sorts test cases based on the sum of these values. Kaur et al. [43] also approached TCP based on test cases generated from activity diagrams. Their technique, which we label **PathComplexity**, takes into account the number of incoming and outgoing transitions from each action from the model, and sorts the generated test cases giving preference to the ones that traverse actions with more information flow, in other words, actions that have more incoming and outgoing transitions. These techniques participated on several studies reported in this thesis (refer to Chapter 2 for more information), and among them, just PathComplexity presented a good overall performance.

6.1.2 Dissimilarity-Based

Observing the dissimilarity based techniques, which are the ones that aim at establishing a resemblance relationship between test cases and vary them in order to increase the likelihood of revealing faults sooner. Ledru *et al.* [49] proposed, and evaluated in more detail [50], a technique called **StringDistance**. They consider the textual representation of test cases to vary the test cases in order. Jiang et al. [37, 38] and Zhou [91] propose adaptive random prioritization (**ARP**) techniques, which has a random component, and also have the distance notion to spread the test cases across the coverage space. Thomas et al. [81] also apply the idea of varying the coverage of black-box test cases. They use a text analysis technique called *topic modeling* to compare test cases. In their work, topics are keywords, such as variables and function names, that may carry meaning, which they assume that test cases that cover similar topics, are more similar. Later the idea was applied on textual representation of black-box test cases [32]. This approach takes into consideration test cases for manual execution, which is a compatible context of application, considering our research. Note that all these techniques pursue variation as a way of achieving sooner fault detection. We evaluated StringDistance and ARP in our research and both of them gave us insights about different aspects of our work.

Given that HARP is based on the adaptive random strategy [37, 38, 91], it also has a

dissimilarity facet; on the other hand, instead of keeping the exploration "as far as possible", it explores the vicinity of the provided hints. Besides, it takes as input expert indications, which is a more common practice among the techniques based on soft-computing strategies.

6.1.3 Soft-Computing TCP Techniques

Instead of using a simple strategy to prioritize test cases, e.g. sorting the test cases based on the value calculated by an arbitrary function, other authors investigate the use of soft-computing methods, such as genetic algorithms and clustering, to solve the TCP problem.

Fevzi Belli, Mubariz Eminov, and Nida Gokce present a long term research in this context [8,9,28]. They propose techniques that prioritize test cases by clustering model elements (events from Event Sequence Graphs) and giving preference degrees for the test cases based on the importance of the events they cover: the first one using an unsupervised neural network; the second one, a clusterer using a fuzzy version of the c-means algorithm; and the last one, the Gustafson-Kessel clustering algorithm. In their empirical investigation, they propose that the use of soft-computing might present a high computational cost, while none of them perform significantly better than the others. We studied their technique based on the fuzzy version of the c-means algorithm in our research, which motivates us to explore the application of more soft-computing method in future work.

Genetic algorithm (GA) is a strategy proposed in the Artificial Intelligence field, which mimics mechanisms of the biological evolution in chromosome level. All techniques that implement this strategy are based on coding a candidate solution, also called chromosome, in a simple but structured way, so that it can be manipulated, e.g. a test case can be represented as an array of booleans, where each position refers to a branch of a model, and a true value means that the test traverses that branch and false otherwise. Based on that representations, the algorithm performs random operations combining two chromosomes (crossover) and mutating some aspect of a chromosome; these new chromosomes originate the next generation. However, they must be filtered by a fitness function, which evaluates the adaptability of a chromosome.

Sabharwal et al. [73] proposed a GA TCP technique able to prioritize test cases generated from UML activity diagrams. They map a test case to a chromosome, and it is represented by a sequence of the bits indicating if the test case contains the decision nodes of the initial

model. The fitness function is the same function used by the technique PathComplexity [43] to order test cases. Even though this GA technique is not evaluated empirically in the paper, the good performance presented by PathComplexity encourages us to investigate it in future work.

CRISPy is a clustering technique based on Yoo et al. [89] and Fang et al. [24] work; therefore, it is classified as a soft-computing technique according to our classification. In comparison to these two works, the former uses Hamming distance to calculate distances, and incorporates expert knowledge to order clusters, sampling test cases respecting these order to create the prioritized test suite. On the other hand, the latter use Levenshtein distance to evaluate the ordered sequences of program entities that test cases exercise, and prioritize test cases just by interleaving test cases from each cluster, considering all of them having the same importance. Both techniques were evaluated with code-based test suites and, using a similar reasoning, we adapted them to be used as inspiration for our work.

There is a subset of TCP techniques that also rely on information provided by specialists. This category has intersections with the others aforementioned. The most popular way of incorporating inputs produced by specialists is by performing a set of pairwise comparisons, and using their results to assist the prioritization task. Yoo et al. [89] use the comparisons to attribute importance degrees to clusters in their technique; on the other hand, Tonella et al. [82] apply case base reasoning, which is another artificial intelligence technique, to prioritize test cases. This set of pairwise comparisons depends on the amount of test cases available and it tends to be an expensive task.

Another way of incorporating is augmenting system models with risk and probability of fault occurrence scores, as done by Stallbaum et al. [80]. Even though the values could be incorporated without any previous knowledge of historical artifacts, mainly regarding risk, it is hard to provide a punctual and still valid information about risk without this knowledge, which we assume absent. HARP is inspired on the idea explored by Stallbaum et al., since we also use a use case model to get information from experts. However, hints are indication about problems or impediments that occurred in early stages of the SUT development, instead of risk scores. Besides, we provided evidence regarding the low cost and impact of the hints collection process.

6.2 Chapter Final Remarks

In this chapter we discussed details not yet addressed about the research related to ours. As a complement of the description of the techniques investigated in this thesis (refer to Section 2.3.1), here we address which strategies have been used to solve the TCP problem and which are the sources of information available.

The prioritization techniques consider different strategies to solving the TCP problem and we classified them into: structural aspects, dissimilarity-based, and soft-computing. Some techniques should be in more than one category, but we discussed them where it seems to be the most relevant. The majority of the techniques commented in this chapter were implemented and evaluated during our studies, and among them, some underwent adaptations to fit out context, which is also our contribution.

Chapter 7

Conclusions and Final Remarks

Test Case Prioritization (TCP) is an activity dedicated to optimize test case execution by rescheduling test cases aiming at achieving some testing goal. The research on black-box system level, including Model-Based Testing, TCP is still ongoing and still faces several issues, for example, proposing an effective prioritized test suite in the presence of new test cases, in other words, when there is no historical information about previous execution of available test cases [54]. Here we contribute by analyzing weaknesses and strengths of current techniques and investigating different ways of approaching the TCP problem.

In this research, we propose two TCP techniques, named **HARP** and **CRISPy**, for system level test suites, in a context that historical information regarding failures/faults may not be available. In order to guide prioritization, HARP uses information derived from the expertise of development teams, named "hints". We represent these hints as prioritization purposes that work as filters, accepting test cases related to the represented information. HARP is based on the Adaptive Random Prioritization, which focus on exploring the input test cases using a distance notion. On the other hand, CRISPy still uses static information about the test suites, but uses clustering as guiding strategy in order to find faults sooner.

7.1 Research Questions and Contributions

In this section we discuss our conclusions, how they address our initial research questions, and what are the related artifacts available, e.g. papers, websites with supplementary material, and so on.

Regarding our first two research questions, we performed an array of studies evaluating different characteristics around the execution of TCP techniques in the investigated context. Whereas we noticed that some of them are more successful on specific scenarios and not in others, in general we argue that the most important lesson on this investigation is to try to estimate the characteristics of the test cases that fail and then use a technique adapted to that circumstances. Therefore, we take up our research questions and comment our findings and contributions regarding them.

(RQ1) Does the model layout, represented by the number of branches, joins, and loops, affect the fault detection capabilities of the prioritization techniques in the investigated context?

In [64] we began this investigation using synthetic artifacts, such as system models and fault models. In [65], we broadened the studies by adding a larger set of TCP techniques. By analyzing the variation of aspects that define the model layout, we did not observe any variation regarding techniques' effectiveness, which means that these variables, and the model layout itself by consequence, do not affect the fault detection capability of the investigated techniques. We publish the artifacts related to this particular discussion in [60].

(RQ2) Do characteristics of test cases that fail affect the fault detection capabilities of prioritization techniques in the investigated context?

On the other hand, considering the investigation regarding characteristics of test cases that fail, we improved an experiment with this idea across the two aforementioned papers, and as described in Chapter 3, which details a replicated study using industrial test suites. After these studies, we argue that TCP techniques are sensitive to test cases that fail with different characteristics, particularly their lengths. Therefore, in order to improve the performance of TCP techniques, even without access to historical artifacts, it is necessary adding other sources of information or investigate different types of strategies to achieve a better fault detection. We submitted this replication [63] and currently it is under minor review. Artifacts related to this particular investigation are in [61].

(RQ3) How can we systematize and collect expertise from the developers or/and managers, and use it to leverage Test Case Prioritization in the defined context, aiming to reveal faults sooner in the testing process?

Motivated by the first investigation, we proposed encoding the expertise of developers

and managers about steps of use cases more prone to contain faults as prioritization purposes, which are the *hints*, and using them as a layer of guidance for a TCP technique. Therefore, we designed HARP, which uses the hints to indicate which test cases should have preference. Since our findings suggest that HARP's ability of revealing faults depends on the quality of the hints, we also suggested a guideline for using the hints on HARP just when there is a consensus among the team members that worked on the SUT. Besides, HARP performed better than its baseline on our validation, when using actual hints for two industrial systems. We also provided another companion website [59] containing details, enabling the repetition of our analysis.

(RQ4) How can we prioritize test suites, even when the expertise available is not enough to guide TCP, i.e. what to do when this expertise would hinder instead of being useful?

As a complementary investigation, i.e. when HARP is not indicated, we also considered another strategy with a different underlying theory to propose a complementary technique. We applied clustering to design CRISPy, which is able to cluster similar test cases and interleave clusters in order to produce a more diverse prioritized test suite, aiming to detect faults earlier on the system testing process. Our studies suggest that the distance notion applied to cluster test cases is important, whereas the distance functions must be aligned with the representation of the test suites, i.e. since in our context the test suites comprise essentially text, Levenshtein distance is a good way of representing distances. Besides, even a basic configuration of CRISPy was already able to indicate promising results and with more tuning, it may be even more successful. We also produced a companion website [58] with data and script for data analysis.

7.2 Discussion on Practical Aspects

From the practical point of view, here we discuss orientations for the practitioners or test managers who work on a similar context and desire to apply our results on their testing processes. We present them in topics in order to simplify their understanding. They are:

- **Pay close attention on the research context**: be aware that our findings are specific for black-box system testing level, HARP explicitly assumes that test suites should be generated from behavioral models (e.g. transition based models, specially labeled

transition systems) and CRISPy deals with textual test suites comprising steps, conditions and expected results, both for manual execution. We did not evaluate them on other contexts, therefore they could perform differently;

- **Applying HARP requires consensus about hints**: since the quality of the hints affects directly its performance, just use them if the personnel, or at least a simple majority, that worked with the system under test point to a common set of portions that may contain a set of faults; otherwise, HARP would be misguided;

- **HARP and CRISPy complement each other**: if the involved personnel did not agree regarding the provided hints, use CRISPy instead;

- **CRISPy still need tuning**: depending on characteristics of test suites to be prioritized by this technique, other distance functions or strategy to decide on how close test cases are to be in the same cluster could be more appropriate. Since our test cases are mainly plain semi-structured text, a function that compared text is appropriate, as discussed in Chapter 5.

7.3 Future Work

This research has clearly three distinct lines: i) investigation about related factors that affect the performance of TCP techniques; ii) investigation on the relationship between personal reports from developers and managers about previous problems and the occurrence of faults; and iii) investigation on the use of different strategies for solving the TCP problem. During the whole doctorate course, we produced evidence supporting our claims, but other hypotheses arose from them and still require investigation.

Regarding i and ii, we did not evaluate factors involving personal traits or even the relation between the occurrence of technical debts [51] and faults. This is a long term investigation that requires cooperation with companies aiming at collecting data about technical debts and fault records, and relating the kinds of debts and the occurrence of faults. Longitudinal studies suit well for this investigation. These result may point to good estimators for faults, which would be valuable for TCP.

With respect to iii, we began to investigate the use of clustering as a different strategy for solving the TCP problem on our research context. We plan to apply other soft computing methods, such as neural networks and genetic algorithms, as well as search-based strategies, e.g. tabu search, to solve this problem. As examples of these applications we can refer to Sabharwal et al. [73] and Nejad et al. [57]. Besides, the natural language processing, e.g. topic analysis, also seems a good topic to be explored, as shown by the good results reported by Hemmati et al. [32].

Bibliography

[1] Anne Adams and Anna L. Cox. *Questionnaires, in-depth interviews and focus groups*, pages 17–34. Cambridge University Press.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1–10, May 2011.

[4] E. Babbie. *Survey Research Methods*. Wadsworth, 2 edition, 1990.

[5] V. R. Basili, R. Selby, and D. Hutchens. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, 12(7):733–743, July 1986.

[6] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2 edition, June 1990.

[7] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Verlag John Wiley & Sons, Inc, 1 edition, 1995.

[8] Fevzi Belli, Mubariz Eminov, and Nida Gökçe. Coverage-oriented, prioritized testing – a fuzzy clustering approach and case study. In Andrea Bondavalli, Francisco Brasileiro, and Sergio Rajsbaum, editors, *Dependable Computing*, volume 4746 of *Lecture Notes in Computer Science*, pages 95–110. Springer Berlin Heidelberg, 2007.

[9] Fevzi Belli and Nida Gökçe. Test prioritization at different modeling levels. In *Advances in Software Engineering*, volume 117 of *Communications in Computer and Information Science*, pages 130–140. Springer Berlin Heidelberg, 2010.

[10] Emanuela G. Cartaxo, Wilkerson L. Andrade, Francisco G. Oliveira Neto, and Patrícia D. L. Machado. LTS-BT: a tool to generate and select functional test cases for embedded systems. In *SAC '08: Proc. of the 2008 ACM Symposium on Applied Computing*, volume 2, pages 1540–1544, New York, NY, USA, 2008. ACM.

[11] Emanuela G. Cartaxo, Francisco G. Oliveira Neto, and Patrícia D. L. Machado. Test case selection using similarity function. In *MOTES 2007*, pages 381 – 386, 2007.

[12] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, pages 1–34, 2012.

[13] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer Science - ASIAN 2004*, volume 3321/2005 of *Lecture Notes in Computer Science*, pages 320–329. Springer Berlin / Heidelberg, 2004.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction of Algorithms*. MIT Press, 3 edition, 2009.

[15] Ana Emília Victor Barbosa Coutinho, Emanuela Gadelha Cartaxo, and Patrícia de Duarte Lima Machado. Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal*, pages 1–39, 2014.

[16] Ana Emília Victor Barbosa Coutinho, Emanuela Gadelha Cartaxo, and Patrícia Duarte de Lima Machado. Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal*, 24(2):407–445, Jun 2016.

[17] Rick D. Craig and Stefan P. Jaskiel. *Systematic Software Testing*. Artech House, 2002.

[18] Francisco Gomes De Oliveira Neto, Robert Feldt, Richard Torkar, and Patricia D.L. Machado. Searching for models to evaluate software technology. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering, CMSBSE 2013 - Proceedings*, pages 12–15, 2013.

[19] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using spin. *STTT*, 2(4):382–393, 2000.

[20] Ibrahim K. El-Far and James A. Whittaker. *Model-Based Software Testing*. John Wiley & Sons, Inc., 2002.

[21] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[22] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions in Software Engineering*, February 2002.

[23] Sebastian G. Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[24] Chunrong Fang, Zhenyu Chen, Kun Wu, and Zhihong Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2):335–361, 2014.

[25] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 223–233, 2016.

[26] H.G. Gauch. *Scientific method in practice*. Cambridge University Press, 2003.

[27] Omar S. Gómez, Natalia Juristo, and Sira Vegas. Understanding replication of experiments in software engineering: A classification. *Information and Software Technology*, 56(8):1033 – 1048, 2014.

[28] Nida Gökçe, Mubariz Eminov, and Fevzi Belli. Coverage-based, prioritized testing using neural network clustering. In Albert Levi, Erkay Savaş, Hüsnü Yenigün, Selim Balcısoy, and Yücel Saygın, editors, *Computer and Information Sciences – ISCIS 2006*, volume 4263 of *Lecture Notes in Computer Science*, pages 1060–1071. Springer Berlin Heidelberg, 2006.

[29] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[30] H. Hemmati, Z. Fang, and M. V. Mantyla. Prioritizing manual test cases in traditional and rapid release environments. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015.

[31] Hadi Hemmati, Andrea Arcuri, and Lionel C. Briand. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.*, 22(1):6, 2013.

[32] Hadi Hemmati, Zhihan Fang, Mika V. Mäntylä, and Bram Adams. Prioritizing manual test cases in rapid release environments. *Software Testing Verification and Reliability*, 27(6):1–25, 2017.

[33] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 523–534, New York, NY, USA, 2016. ACM.

[34] Yu Chi Huang, Chin Yu Huang, Jun Ru Chang, and Tsan Yuan Chen. Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history. In *Proceedings - International Computer Software and Applications Conference*, COMPSAC '10, pages 413–418, Washington, DC, USA, jul 2010. IEEE Computer Society.

[35] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, August 2005.

[36] Dennis Jeffrey and Neelam Gupta. Experiments with test case prioritization using relevant slices. *Journal of Systems and Software*, 81(2):196 – 221, 2008.

[37] Bo Jiang and W. K. Chan. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In *Proceedings - International Computer Software and Applications Conference*, number 123512, pages 190–199, 2013.

[38] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and T. H. Tse. Adaptive random test case prioritization. In *ASE*, pages 233–244. IEEE Computer Society, 2009.

[39] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Software Eng.*, 29(3):195–209, 2003.

[40] Dalton Jorge, Patrícia D. L. Machado, Francisco G. Oliveira Neto, and Ana Emília V. B. Coutinho. Integrando teste baseado em modelos no desenvolvimento de uma aplicação industrial: Benefícios e desafios. In Wilkerson de Lucena Andrade and Valdivino Alexandre de Santiago Júnior, editors, *8th Brazilian Workshop on Systematic and Automated Software Testing*, volume 2, pages 101 – 106, 2014.

[41] P. C. Jorgensen. *Software Testing - a Craftsman Approach*. CRC Press, 2 edition, 2002.

[42] Mark Kasunic. Designing an effective survey. Technical report, Carnegie Mellon University, September 2005.

[43] Preeti Kaur, Priti Bansal, and Ritu Sibal. Prioritization of test scenarios derived from uml activity diagram using path complexity. In Vidyasagar Potdar and Debajyoti Mukhopadhyay, editors, *CUBE*, pages 355–359. ACM, 2012.

[44] Do-Jong Kim, Yong-Woon Park, and Dong-Jo Park. A novel validity index for determination of the optimal number of clusters. *IEICE Transactions on Informations and Systems*, E84-D(2):281–285, February 2001.

[45] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, Jul 1995.

[46] B. Korel, G. Koutsogiannakis, and L.H. Tahat. Application of system models in regression test suite prioritization. In *IEEE International Conference on Software Maintenance*, pages 247–256, 2008.

[47] B. Korel, L.H. Tahat, and M. Harman. Test prioritization using system models. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 559–568, Budapest, 2005. IEEE.

[48] Debasish Kundu, Monalisa Sarma, Debasis Samanta, and Rajib Mall. System testing for object-oriented systems with test case prioritization. *Softw. Test., Verif. Reliab.*, 19(4):297–333, 2007.

[49] Yves Ledru, Alexandre Petrenko, and Sergiy Boroday. Using string distances for test case prioritisation. In *ASE*, pages 510–514. IEEE Computer Society, 2009.

[50] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Autom. Softw. Eng.*, 19(1):65–95, 2012.

[51] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[52] Zheng Li, Mark Harman, and Robert M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, apr 2007.

[53] Lucas Lima, Juliano Iyoda, Augusto Sampaio, and Eduardo Aranha. Test case prioritization based on data reuse an experimental study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 279–290, Washington, DC, USA, 2009. IEEE Computer Society.

[54] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16. ACM, 2016.

[55] Camila Loiola Brito Maia, Rafael Augusto Ferreira do Carmo, Fabrício Gomes de Freitas, Gustavo Augusto Lima de Campos, and Jerffeson Teixeira de Souza. Automated Test Case Prioritization with Reactive GRASP. *Advances in Software Engineering*, 2010:1–18, 2010.

[56] Glenford J. Myers. *The Art of Software Testing, Second Edition*. Wiley, 2 edition, 2004.

[57] F. M. Nejad, R. Akbari, and M. M. Dejam. Using memetic algorithms for test case prioritization in model based software testing. In *2016 1st Conference on Swarm Intelligence and Evolutionary Computation (CSIEC)*, pages 142–147, March 2016.

[58] João Felipe Silva Ouriques. Clustering system test cases for test case prioritization. `https://sites.`

google.com/site/joaofso/research/experiments/clustering-system-test-cases-for-test-case-prioritisation.

[59] João Felipe Silva Ouriques. Hint-based adaptive random prioritization. https://sites.google.com/site/joaofso/research/experiments/hint-based-adaptive-random-prioritization.

[60] João Felipe Silva Ouriques. Influencing factors on tcp techniques. https://sites.google.com/site/joaofso/research/experiments/influencing-factors-on-tcp-techniques.

[61] João Felipe Silva Ouriques. Replication of failure characteristics experiment. https://sites.google.com/site/joaofso/research/experiments/replication-of-failure-characteristics-experiment.

[62] João Felipe Silva Ouriques. A sistematic mapping about the general test case prioritization, in the context of the model-based testing. Technical report, Federal University of Campina Grande, November 2013.

[63] João Felipe Silva Ouriques, Emanuela Gadelha Cartaxo, and Patrícia Duarte Lima Machado. Test case prioritization techniques for model-based testing: A replicated study. Under minor corrections, available at: https://arxiv.org/abs/1708.03240.

[64] João Felipe Silva Ouriques, Emanuela Gadelha Cartaxo, and Patrícia Duarte Lima Machado. On the influence of model structure and test case profile in the prioritization of test cases in the context of model-based testing. In *XXVII Simpósio Brasileiro de Engenharia de Software*, October 2013.

[65] João Felipe Silva Ouriques, Emanuela Gadelha Cartaxo, and Patrícia Duarte Lima Machado. Revealing influence of model structure and test case profile on the prioritization of test cases in the context of model-based testing. *Journal of Software Engineering Research and Development*, 2015.

[66] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques.* Wiley, 2007.

[67] S. Poulding and John A. Clark. Efficient software verification: Statistical testing using automated search. *Software Engineering, IEEE Transactions on*, 36(6):763–777, Nov 2010.

[68] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 5th edition, 2001.

[69] S Raju and G V Uma. Factors Oriented Test Case Prioritization Technique in Regression Testing using Genetic Algorithm. *European Journal of Scientific Research*, 74(3):1450–216, 2012.

[70] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: an empirical study. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 179–188, 1999.

[71] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27:929–948, 2001.

[72] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Software Eng.*, 27(10):929–948, 2001.

[73] S. Sabharwal, R. Sibal, and C. Sharma. Prioritization of test case scenarios derived from activity diagram using genetic algorithm. In *Computer and Communication Technology (ICCCT), 2010 International Conference on*, pages 481–485, Sept 2010.

[74] Sreedevi Sampath and Renée C. Bryce. Improving the effectiveness of test suite reduction for user-session-based testing of web applications. *Information and Software Technology*, 54(7):724–738, 2012.

[75] P.G. Sapna and H. Mohanty. Prioritization of scenarios based on uml activity diagrams. In *Computational Intelligence, Communication Systems and Networks, 2009. CICSYN '09. First International Conference on*, pages 271 –276, july 2009.

[76] Amanda Schwartz and Hyunsook Do. Cost-effective regression testing through adaptive test prioritization strategies. *Journal of Systems and Software*, 2016.

[77] Yogesh Singh, Arvinder Kaur, Bharti Suri, and Shweta Singhal. Systematic literature review on regression test prioritization techniques. *Informatica (Slovenia)*, 36(4):379–408, 2012.

[78] Ian Sommerville. *Software Engineering*. Addison Wesley, New York, 8th edition, 2006.

[79] Praveen Ranjan Srivastva, Krishan Kumar, and G Raghurama. Test case prioritization based on requirements and risk factors. *SIGSOFT Softw. Eng. Notes*, 33(4):7:1–7:5, July 2008.

[80] Heiko Stallbaum, Andreas Metzger, and Klaus Pohl. An automated technique for risk-based test case generation and prioritization. In ... *on Automation of software test*, page 67, New York, NY, USA, 2008. ACM.

[81] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, Feb 2014.

[82] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 123–133, Sept 2006.

[83] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kauffman, first edition, 2007.

[84] Jüri Vain, Kullo Raiend, Andres Kull, and Juhan P. Ernits. Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer 0002, editors, *ASE*, pages 363–372. ACM, 2007.

[85] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[86] I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann series in data management systems. Morgan Kaufman, 2005.

[87] Claes Wohlin, Per Runeson, Martin Host, Magnus C. Ohlsson, Bjoorn Regnell, and Anders Wesslen. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[88] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[89] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 201–212, New York, NY, USA, 2009. ACM.

[90] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201, Piscataway, NJ, USA, 2013. IEEE Press.

[91] Zhi Quan Zhou. Using coverage information to guide test case selection in adaptive random testing. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, pages 208 –213, july 2010.