GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

Doctoral Thesis

Orchestration Strategies for Regression Testing of Evolving Software Systems

PhD Program in Computer Science: 34th cycle

Supervisors:

Prof. Antonia Bertolino
antonia.bertolino@isti.cnr.it
Prof. Breno Miranda bafm@cin.ufpe.br

Author:

Renan Domingos Merlin Greca
renan.greca@gssi.it

Internal advisor:

Dr. Ludovico Iovino
ludovico.iovino@gssi.it

May 2023

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

"There are many potentially valuable academic insights that just wither and die on the vine because they aren't pushed far enough to entice corporations or policy makers to adopt them. This intermediary step is time consuming, it's tedious, it's maybe expensive, and it's really not rewarded enough in academics."

Steven Levitt

Abstract

Context: Software is an important part of modern life, and in most cases, it provides tremendous benefits to society. Unfortunately, software is highly susceptible to faults. Faults are often harmless, but even small errors can cause massive damage depending on the context. Thus, it is crucial for software developers to adopt testing techniques that can help locate faults and guarantee the functionality of both individual components and the system as a whole. Today, there is a trend towards *continuously evolving software*, in which it is desired that changes such as new features and corrections are delivered to end users as quickly as possible. To ensure correct behavior upon release, development teams rely on *regression testing* suites, which serve to validate previously-correct features and, when well-designed, avoid the propagation of faults to end users. However, the desire for velocity that comes with continuously evolving software places an additional burden on regression testing practices, as running complete test suites can be a costly process in large-scale software. This challenge has generated a need for novel regression testing techniques, a topic which now enjoys a robust literature within software engineering research. However, there is limited evidence of this research finding its way into practical usage by the software development community; in other words, there is a disconnect between academia and industry on the subject of software testing techniques.

Objective: To improve applicability of regression testing research, we must identify what are the main causes of this apparent gap between software engineering academics and practitioners. This is a multifaceted goal, involving an investigation of the literature and of the state of practice. A related goal is to provide examples of *test suite orchestration strategies* that draw from academic advancements and could provide benefits if implemented on real software.

Method: This thesis tackles the aforementioned challenge from multiple directions. It includes a comprehensive systematic literature review covering research published between 2016 and 2022, focusing on papers that bring techniques and discussions that are relevant to the applicability of regression testing research. Along with data extracted from the papers themselves, this discussion of the existing literature includes information received directly from authors through a questionnaire, as well as a survey performed with practitioners, seeking to validate some of the reported findings.

Test suite orchestration strategies can be a step towards bridging the so-called *industry-academia knowledge gap*. To that end we propose a combined approach for regression testing, including techniques extracted from the literature that have promising qualities. This approach is an initial experiment with full test suite orchestration and extended approaches are also discussed.

To get a closer understanding of the state of regression testing in a practical sense, a series of interviews were conducted in collaboration with a large technology company. During a seven-week process, we were able to interact with the team and learn the test practices performed on a daily basis and have some insight on the long-term test strategies for the company. The responses of the interviews are reported, edited for readability and confidentiality reasons, and these results are discussed within the larger context of the study.

The results from the above components of the studies are then aggregated into two notable outputs. First, a live repository of literature is made available online, containing the current results of the literature review and with the opportunity of expansion as more research is performed in this topic. Then, we provide a list of the most notable challenges for the implementation of regression testing techniques in practice, that were identified during the development of this entire study.

Results: This thesis provides the following contributions: a comprehensive literature review of applicable regression testing research; additional context on the literature provided by the authors of cited papers; a preliminary test suite orchestration strategy combining robust techniques from the literature; interviews with practitioners at a major technology company that highlight the challenges faced daily by developers and testers; a live repository of papers to aggregate relevant literature in one online location; a list of challenges that can serve as guidelines for researchers or even as research directions in their own right.

Conclusion: There is still much work to be done by the software engineering research and development communities in order to completely close the gap that exists between them. To a great extent, the motivations of researchers and practitioners are not aligned — while in academia, proposing theoretically sound novel approaches is encouraged to obtain publications, in industry there is a need for techniques that are proven to reduce effort and/or costs. This can only be solved by close collaboration between the two sides, yet a question of who is willing to fund these experiments remain. The data and discussions provided in this thesis show that, although difficult, this is not an impossible problem to solve and there are certain clear steps that can be taken by researchers and practitioners alike to begin addressing it.

*In memory of Prof. Dr. Luiz Felipe Paula Soares
and Prof. Dr. Francisco de Paula Soares Filho*

Acknowledgements

I am incredibly grateful for the guidance and orientation provided by my advisors, professors Antonia Bertolino, Breno Miranda and Ludovico Iovino. I would also like to thank GSSI professors Luca Aceto and Michele Flammini, who have been supportive in times of need since my early days at the institute.

Thanks to Milos Gligoric, from University of Texas at Austin, who provided significant contributions to the discussions and results presented in Chapter 4. I also thank Sigrid Eldh, from Mälardens Universitet, who arranged the collaboration that allows the discussion present in Chapter 5. I am also grateful to the people at the industrial partner, who received me with grace and gave me the opportunity of interacting with their team, but who shall remain anonymous due to confidentiality concerns. If you are reading this, you know who you are.

Thanks to my parents, Lizmari and Edison Greca, and my grandparents, Glacial and Aquiles Merlin, who gave me the foundational values and education that have guided me through life until this point. Additionally, I would like to remember my grandparents Berenice and Eros Greca, who I wish could be here to share their part in this achievement, and my uncles Felipe and Chico, who encouraged me to explore the world.

Thanks to my colleagues Debashmita Poddar, Konstantin Prokopchik and Alex Coto for the companionship throughout the PhD and the many board game nights that kept us sane during the lockdowns.

I would also like to thank professors Luiz Albini and Eduardo Todt from UFPR, for the education I received before coming to Italy and for the warm welcome I received when visiting my *alma mater*.

Thanks to Marco Rotilio, whose music lessons were an outlet of creativity.

Finally, I would like to acknowledge and thank my friends: André Ramos, Arthur Alves, Arthur Pieri, Cainã Trevisan, Dácio Augusto, Darren Kerwin, Douglas Novelli, Eric Bunese, Felipe de Lara, Felipe Gugelmin, Gustavo Henrique, Janderson Oliveira, Lucas Knopki, Luiz Roveran, and Pedro Vicente. Even thousands of kilometers away, you are all special to me.

Contents

List of Figures

List of Tables

Abbreviations

APFD: **A**verage **P**ercentage of **F**aults **D**etected

CI/CD: **C**ontinuous **I**ntegration/**C**ontinuous **D**elivery (or **D**eployment)

FAD: **F**unctional **A**rea **D**omain

FOSS: **F**ree and **O**pen-**S**ource **S**oftware

IR&A: **I**ndustrial **R**elevance and **A**pplicability

LIRT: **L**ong **I**nterval **R**egression **T**est(ing)

MCT: **M**ulti-**C**omponent **T**est(ing)

RT: **R**egression **T**esting

SIRT: **S**hort **I**nterval **R**egression **T**est(ing)

SLR: **S**ystematic **L**iterature **R**eview

SUT: **S**ystem **U**nder **T**est

TCP: **T**est **C**ase **P**rioritization

TCS: **T**est **C**ase **S**election

TSA: **T**est **S**uite **A**mplification or **A**ugmentation

TSR: **T**est **S**uite **R**eduction

TR: **T**rouble **R**eport

XFT: **X** (Cross) **F**unctional **T**eam

Chapter 1

Introduction

Software has become an ubiquitous part of every-day life, be it in computers, smartphones, vehicles, or other devices. Well-functioning software can be a major asset for most people, helping to reduce operational costs, reduce time spent on tasks, and even save lives. However, it is already in the common sense that software is not always perfect, and, from time to time, it may behave incorrectly or unexpectedly.

A recent and devastating example is the Boeing 737 Max, an aircraft that was involved in two fatal crashes in 2018 and 2019, and had to be grounded for months, due to what was likely a software fault [81]. Needless to say, this software fault caused the tragic loss of many lives, as well as billions of dollars of expenses to Boeing, airlines, airports and passengers.

Despite this example, encountering bugs is an almost universal experience to people who interact with software. Most bugs are not harmful or fatal, appearing frequently as inconveniences that may bother users, or perhaps incurring additional costs to a company that relies on the software. From an industry perspective, these bugs are costly — fixing software defects is always more expensive after release and, in critical infrastructure systems, costly steps such as hardware redundancy must be employed to ensure uninterrupted service if the software is not deemed to be 100% reliable.

As such, it is important for companies and communities developing software to utilize methods to mitigate the possibility that faulty software will reach production. Today, most commercial and open-source software products are accompanied by a *test suite*, a series of automated tests that are used to provide a level of certainty that parts of a software, both in isolation and in conjunction, correctly perform the tasks to which they are assigned. One widely-adopted software testing technique is called *regression testing* (RT); its primary role is to execute the test suite with a certain frequency, in order to guarantee that recently introduced changes to the software do not affect previously-correct behavior. However, in large-scale software development (that is, with multiple developers and a large codebase), it is usually unfeasible to execute every test after every change, either because changes are too frequent, or because there are too many tests, or both. This is aggravated by the fact that most software is now developed in a *continuous* manner, meaning software that is developed in an iterative and cyclical process, resulting in a short turnaround time between the design of a requirement, the development of a feature, and the delivery of an update to customers.

Although RT is an active research topic, the research community's efforts over the years to mitigate RT cost and complexity do not seem to have produced the desired impact. A 2010 study [38] aiming at understanding RT practice already highlighted several divergences between software testing research and practice, notwithstanding in 2017 Garousi and Felderer [46] still called them as "*worlds apart*". Indeed, in a recent systematic review of the RT literature aiming at identifying approaches with *industrial relevance and applicability* (henceforth referred to as IR&A), Ali et al. [4] could only select 38 primary studies out of an initial pool of 1068 collected works. In other words, their study would imply that less than 4% of the published works on RT could be of interest to industry.

This difficulty of bringing theoretically-sound approaches into real-world use by developers is a major challenge in software testing research. There are many reasons why this happens; for example, many academic works on the topic aim for highly-precise techniques that, when applied in practice, are too slow to be useful or unrealistically require resources that are not easily available, e.g. an extensive log of test execution history. On the other hand, many practitioners already apply coarse-grained techniques that provide some reduction in costs, but that could do much better with further research and experiments, although companies are reluctant to spend time and money on improving a testing workflow instead of delivering new features. In other words, there is a potential mismatch in motivations of researchers and practitioners who work with software testing, causing the so-called *industry-academia knowledge/technology transfer gap.*

This thesis brings forth a discussion on this apparent gap, extracting information from a comprehensive literature review in combination with in person interviews at a major technology company. With this, we aim to expose ongoing challenges that prevent most software testing research from seeing real-world use and provide directions for future researchers to act upon. In addition, as a proof-of-concept approach, we introduce *orchestration strategies* for regression testing, with the objective of managing multiple RT techniques, such as test case prioritization (TCP), test case selection (TCS), test suite reduction (TSR) and test suite amplification(TSA).

Many approaches for RT techniques have already been proposed in the literature [132, 76, 62, 87]. Our research goal here is not that of inventing yet another approach, but rather to understand if and how TCS, TCP, TSR and TSA should be used in combination, i.e.: when a new software version is released, is it more convenient to apply a TCS approach or instead a TCP one? Intuitively, a combination of all techniques would provide the most benefit, but this could result in additional challenges and drawbacks. Notwithstanding the vast literature on regression testing, such type of questions remain largely unanswered.

1.1 Document structure

The following paragraphs provide a summary of the remainder of this document.

As a background, Chapter 2 provides a detailed description of the challenges involved with regression testing and continuously evolving software systems. It also introduces the concept of test suite orchestration and the techniques that can be components of an overarching test strategy. Finally, it describes the four groups of techniques that are delineated as the scope for this thesis: test case prioritization (TCP), test case selection (TCS), test suite reduction (TSR) and test suite amplification (TSA).

Then, Chapter 3 is a comprehensive systematic literature review covering advances in regression testing research between the years 2016 and 2022. The focus is to identify papers that propose techniques that are either validated in practical experiments, or are promising candidates for real-world use. In this process, we have identified 79 papers covering the four aforementioned groups of regression testing techniques. To obtain an updated understanding of the research beyond what is included directly in the papers, we contacted the authors directly, asking them about the long-term impact of their research after publication. Furthermore, we also contacted a number of industry practitioners to better understand their relationship with the ongoing research in the field.

A proof-of-concept test suite orchestration strategy is introduced in Chapter 4, combining two previously existing techniques from literature and drawing conclusions regarding the effectiveness and efficiency of the combined approach versus the individual techniques. This work is left open to extension with probable paths highlighted in the chapter.

During a seven-week period, an investigation was conducted in partnership with a large technology company; the results of which can be found in Chapter 5. The primary objective of this period was to understand how testing is done at the company and identify the testing issues that most commonly affect the practitioners. A series of interviews were conducted with team members involved with the testing process, from which it is possible to determine the positives of their current process, the points that could be improved with new techniques, and potential avenues for closer collaboration with academic work. Certain parts of the extracted information are under a non-disclosure agreement, thus here we present as much as possible without infringing this contract.

Combining data and findings from the preceding chapters, Chapter 6 provides a list of ongoing challenges for the real-world relevance of software testing research. Some of these were brought up by authors; others emerged during the interviews at the industrial partner; others still are based on our own observations of the information we collected and analyzed during the development of this work. This chapter is designed to be a set of suggestions for researchers to keep in mind while developing their next work, as well as as potential research directions in their own right.

To provide a long-term usefulness to this work, we introduce an online live repository of research in Chapter 7. This is based primarily on the findings of Chapter 3, developed into an interactive website that will be systematically updated over the years, as long as the related research questions remain relevant. The goal is to provide a destination to researchers in the field of regression testing to easily gather a bibliography of research that has proven applicability or potential for it. Practitioners interested in adopting new techniques on their projects can also look for studies that are related to the challenges they face, opening up an avenue for contact and collaboration. Authors of the cited papers and engaged readers are encouraged to contribute to the repository, either by adding additional details about the included papers or by suggesting additional papers that fit the topic and criteria.

Finally, Chapter 8 provides a summary and final thoughts on the results of this research. It also acknowledges the threats to validity of the preceding chapters and, finally, lists the papers that have been developed during the PhD period.

In summary, the contributions of this research are as follows:

- A comprehensive literature review highlighting research with a high degree of applicability between the years 2016-2022 [52].
- Follow-up information from the authors of the cited papers, detailing their long-term impact and the challenges that prevent implementation of a technique.
- A proof-of-concept for test suite orchestration, combining existing robust research and drawing paths for future expansion [53].
- A series of interviews with practitioners at a major technology company, presenting an overview of how testing is performed there and what are the issues faced by the team that could be alleviated by software testing research.
- A list of challenges gathered from the preceding work, which can help future researchers address barriers to applicability and should help research turn into practice.
- A live repository of papers extracted from the literature review, which will serve as a starting point for researchers and practitioners seeking to develop new and practical regression testing techniques.

Chapter 2

Background

Some core concepts form the basis of the discussion found in this thesis. In particular, the focus is on regression testing (RT) techniques for continuously evolving software. Collectively, we view RT techniques as parts of an overarching test suite orchestration strategy. This chapter provides a brief introduction to these concepts, including additional details on the specific RT challenges that are tackled in this research.

First of all, it is important to clarify the definitions of two sets of terms that are related, but offer distinct nuance to the discussion of software testing techniques.

Failure, fault and error: these terms are widely used throughout this thesis. While they may seem interchangeable, there are key distinctions that must be highlighted, as defined by the IEEE standard 1044-2009 [66]. A *failure* occurs when a system produces an incorrect or unexpected outcome, leading to a failing test case — without further investigation, the cause of the failure might be unknown. A *fault* is a specific part of the software that is incorrect; e.g. a logical or semantical issue in the source code that, when executed, will produce a failure. Finally, an *error* is the human action that leads to a fault, such as having written incorrect code, or misinterpreted the system requirements. Terms like *anomaly*, *defect* or *bug* can be used when referring generally to unexpected software behavior.

Efficacy, effectiveness and efficiency: these "three Es" are commonly used words when describing the results of a technique or experiment. Here, we follow dictionary definitions closely [31]. Given a certain task and one or more techniques designed to accomplish it, *efficacy* is a binary assessment of whether a technique accomplishes the desired task at all; *effectiveness* provides a more nuanced description of how well the task is accomplished, and can be used to compare multiple alternatives; and *efficiency* is a quality related to the time, cost and/or effort spent to accomplish the task.

2.1 Regression Testing of Evolving Software Systems

In the early stages of commercial software development, computer programs were designed, produced and distributed mostly like physical retail goods. That is, there was an initial planning and design phase, followed by an extensive development period and, on a certain deadline, the software would be shipped embedded with hardware, or pressed onto disquettes or CDs that could be mailed to customers or made available in store shelves. The advent of the Internet made it possible to completely alter this paradigm. Now, these three phases still exist in commercial software, but happen much faster and can be repeated iteratively as needed. In other words, software companies can initially design and develop the “minimum viable product” to be delivered to customers online and, with the software already in use, updates can be develop to add new features, improve existing ones, or correct bugs that can be detected[1].

We denote software developed and released as ever-evolving products as *continuously evolving software*. The concept of evolving software was introduced in the 1970s by Lehman [77], although it was in the 1990s that the term and paradigm gained widespread use, due to the accelerated delivery methods becoming available [97]. It can also happen that programs that were originally designed according to the traditional release cycle are, at some point, adapted and converted to be continuously evolving (e.g. Microsoft Windows shifted from yearly “service packs” to weekly online updates).

The shift to evolving software, which correlates to the pivot to agile development practices in the mid-2000s, also caused a significant change to how software testing is viewed and addressed. Previously, it was common to see testing as its own stage of development; certain teams had members solely responsible for testing the source code, which was

[1]This is not the same for all types of software; for example, embedded systems cannot always rely on the ability of online updates; meanwhile video games generally deliver complete products on a given deadline to account for distribution and marketing schedules, often followed by an extensive post-release update cycle which resembles the evolving software paradigm.

usually a manual process. Nowadays, it is common practice for developers to write and test their own code, and have an active role in the maintenance of the regression testing suite, a practice encouraged by the agile method [89]. This has the advantage of speeding up the testing process, although as a drawback it can cause testing to be seen as a "second-class citizen" by developers, who would rather create new features than test existing ones.

Regression testing is the part of software testing concerned with testing previously existing components of a system to guarantee that recent changes in the codebase did not affect the originally specified functionality of components. This process is one of the costliest aspects of software development [121], as it should ideally be performed every time a code change is committed, and involves much repetition of previously performed tests. It is defined in [98] as "an activity which makes sure that everything is working correctly after changes to the system." That is, its primary objective is to assure that, after each change to the software, previously existing code continues to comply to specification (or simply to expectations, in case no formal specification exists).

The term *regression testing* (RT) has its origins in pre-agile days and, as a research topic, has been studied since the 1980s [80, 148]. At the time, release schedules were centered around a hard deadline, so RT was an activity that was only performed near the end of the cycle, after the important features of the release had already been developed. At that point, testers would check if any of the new changes interfered with previous functionality of the software; in some places this was a manual process, in others semi-automated. Doing so earlier was not advantageous — if a bug is detected in the middle of development but new features are not yet complete, it is possible that another bug will be detected on another round of testing. Since the software could only be shipped once all features were done, intermediary regression testing provided little benefits.

Continuously evolving software shifted this dynamic. With smaller and more frequent release cycles, regression testing too became a more frequent activity. At the same time, the incredible feature speed demanded by customers and consumers means that it is not viable to postpone testing until right before release — if a bug is detected at that point, it might be too late to fix it before delivery. Thus, with the development of Continuous Integration/Continuous Delivery (CI/CD) practices and tools, automated regression testing became commonplace, sometimes executed as frequently as new code changes are pushed into a repository.

Test automation mostly solves the problem in small projects, where it takes only a few seconds or maybe minutes to run a full test suite. Large-scale software demand additional attention because of two factors: the test suite is large and takes a long time to execute, and code commits arrive at such a high frequency that there is not enough time to run the test suite between each commit. Often, a combination of both factors become a major challenge in large-scale software development [95].

In order to maintain the health of the testing process and the availability of testing equipment, the execution time of a suite should be less than the average time between commits pushed by developers. In reality, this is difficult to achieve and maintain, as test suites tend to increase in scale (according to the necessities of an ever-growing software) and commit frequency remains stable or can even increase if new developers are added to the team. The straightforward solution is to increase the computational power of testing servers, so testing time reduces by brute force, although obviously this incurs additional costs.

The concept of software size and scale is fundamental for the motivation of this research. There are multiple ways of measuring software scale — it could refer to a large number of lines of code (LOC); it can also mean high-complexity algorithms that run for a long time, or software that needs to serve multiple users simultaneously. For this research, we are considered primarily with the number of test cases that the program needs to be reliable. Thus, other aspects of software scalability go beyond the scope of this thesis and, here, the term "scalability" itself refers primarily to the ability of managing an ever-growing number of test cases.

That said, in this work, we are interested in "industrial-scale evolving software". Understand "industrial-scale" as a general term for large-scale software in the real world. In practice, it can mean several different kinds of software, such as software developed as the primary product of a corporation (in the technology industry), software that provides essential features to other products (such as in the automotive or telecommunication industries), or open-source software that is developed by a community instead of a team within a company.

It is also noteworthy that software can exist in a multitude of contexts — e.g. embedded software, distributed systems, web or mobile applications, cloud-based solutions, and so forth. Each context is associated with unique challenges that inevitably alter how they are designed. For the most part, this thesis explores testing strategies that can be generalized into most contexts, as long as the software is continuously evolving in nature, although the ultimate implementation of these strategies might require adjustments.

2.2 Test Suite Orchestration

Given the challenges associated with ever-expanding regression testing suites of continuously evolving software, we define *test suite orchestration* as the art of generating, choosing, prioritizing and executing tests in order to maximize the effectiveness of testing while keeping costs within a desired budget. Today, research on test orchestration is quite granular, with individual researchers mostly focusing on specific challenges within this topic. While this is important for the continuity and advancement of research, it fails in addressing the practical concerns of software developers, who desire a complete solution to aid the development cycle.

Features such as *test case generation*, *test case prioritization*, handling of *flaky tests*, *mutation testing*, *test suite augmentation* and others can be considered under the broader scope of test orchestration. While improvements in each of these features can provide substantial benefits, it is their combination that can produce the desired solution.

In general, test suite orchestration can be thought of as a broad challenge with the ultimate goal of improving regression testing in multiple aspects, composed of several sub-challenges, which are in turned addressed by groups of techniques. These sub-challenges include, but are not limited, to the following:

- **Test case selection (TCS):** the challenge of determining a sub-set of tests that, when executed, provides sufficiently high confidence that recent changes have not introduced failures in the software, while substantially reducing the execution costs [32, 69].
- **Test case prioritization (TCP):** the challenge of ordering tests to detect potential faults as early as possible, prioritizing tests that are most likely to reveal faults or that cover critical parts of the program [32, 84].

- **Test suite reduction or minimization (TSR):** the challenge of reducing the test suite by finding and possibly removing redundant tests [32, 117]. Unlike TCS, which is change-aware, TSR can be used on a single version of a test suite.
- **Test suite amplification or augmentation (TSA):** the challenge of expanding and improving an existing test suite through various different means. A survey on test suite amplification is found in [29]; out of the categories presented, the synthesis of new tests with respect to changes is the most relevant for a continuously-evolving system.
- **Handling of unreliable/flaky tests:** a test that might pass or fail non-deterministically without changes to the SUT is designated as unreliable or flaky. This can happen due to poor test design, misconfiguration of the test suite or the testing environment, or timing errors in asynchronous tasks. These tests make it difficult for developers to identify true faults in the system and thus they should ideally be detected and flagged as such.

Individually, each of these challenges can be its own field of research, and indeed many works have been published on them. However, an ideal test orchestration solution should consider all or most of these challenges in unison, as solving each one alone is not sufficient to solve the problems faced by software developers in practice.

Due to the breadth of the orchestration challenge, for this thesis the decision was made to restrict the scope and focus primarily on four aspects: test case prioritization, test case selection, test suite reduction/minimization and test suite amplification/augmentation. Other topics remain tangential to the research and may occasionally be part of the discussion, but are not the focus of this work. The following subsection offers definitions for the four groups of techniques that this study focuses on. In Section 3.4.1, common approaches and metrics for each group are described in more detail, according to data extracted from the existing literature.

2.2.1 Test Case Prioritization

One challenge of regression testing is to detect failing tests fast. The objective of *test case prioritization* (TCP) is to re-order test cases according to some definition of priority, in order to get faster feedback from the test execution [148]. Given an SUT M and its test suite T, TCP can be described as a function $P(M, T)$ that provides a permutation of T, T' such that, given a metric function f, $f(T') > f(T)$. The optimal prioritization is one where $f(T')$ is greater than or equal to any other possible permutation of T.

Some criteria often used for TCP include: *(1)* similarity-based, which attempts to diversify the execution of tests; *(2)* coverage-based, with the objective of maximizing block[2] coverage with as few tests as possible; or *(3)* history-based, which prioritizes tests that have a history of failing or revealing faults [70].

Common metrics include: *(1)* average percentage of faults detected (APFD), which estimates how effective a prioritization is in detecting faults in fewer tests; *(2)* tests till first fault (TTFF), a count of how many tests were executed until one failed; or *(3)* developer feedback time, a measure of how long it takes for a developer to get a report if there is a failing test in the suite.

TCP is particularly useful in situations where the test suite is exceptionally large and detecting failures sooner allows for potential faults to be addressed quicker. It's also relevant in cases where the testing budget is limited but not consistent, so testing might stop at any time and only tests that failed until then can be added to a report.

A prioritized test suite still contains all test cases, so there is no loss of failures detection ability (assuming that test results are independent and the testing budget is sufficient) – what changes is the amount of time that it takes for one or more failures to be detected.

2.2.2 Test Case Selection

In regression testing, not all tests are relevant to a particular code change: if only a small part of one file was updated, it is unlikely that the entire project would be affected and the full regression test suite would have to be run. *Test case selection* (TCS)[3] addresses the challenge of selecting a subset of tests that is representative of the entire suite in a given situation [148, 122]. In other words, given subsequent versions of an SUT, M and M' and its test suite T, TCS can be described as a function $S(M, M', T)$ that selects a subset $T' \subseteq T$ to be used for testing M', considering the differences between M and M'.

[2]A *block* of the SUT can be one line of code, a branch, a function, etc. according to system design and other necessities.

[3]Also referred to as Regression Test Selection (RTS) in the literature.

We say that a TCS technique is *safe* if it guarantees that all tests whose outcome may be affected by a change are included in the selected subset [122]. That is, safe selection techniques output a subset T' while maintaining the output of a fault detection metric function $f(T') \geq f(T)$.

Examples of approaches for TCS are: *(1)* change-based, which executes tests that have some relation to modified files, classes, or methods; *(2)* model-based, which uses data extracted from models of the SUT to determine test execution; or *(3)* graph-based, which uses a graph representation of the SUT to detect control flow and select relevant tests [69].

Some metrics for TCS are: *(1)* selection count or percentage, which measures how many tests were executed in comparison to the original suite (e.g. $|T'| \leq |T|$); *(2)* testing time, or the time taken to execute the selected subset of tests; and *(3)* fault detection capability, used to determine the safeness of the proposed technique.

A potential drawback of TCS is that, depending on the size of the test suite and the execution time of individual tests, it may happen that the time needed to produce the subset T' is greater than the savings provided by executing T' instead of T.

2.2.3 Test Case Reduction and Minimization

Without considering subsequent versions of the SUT, *test suite reduction* (TSR) aims to find a minimal subset of test cases such that the testing requirements are still met [148]. Thus, given an SUT M its test suite T that satisfies a set of requirements $\{r_1, ..., r_n\}$, we describe TSR as a function $R(M, T, r)$ which outputs a test suite $T' \subseteq T$ such that each r_i is still satisfied.

There exists conceptual overlap between TCS and TSR, with the key differences being change-awareness and the objective of the result. While TCS uses a comparison between versions of the SUT and produces a set of tests meant to validate those changes, TSR can be performed on an isolated iteration of a program and is meant to detect tests that are no longer needed for full satisfaction of the requirements. TSR approaches are often coverage-based or requirements-based and, as discussed in Section 3.4.1, evaluation metrics for TCP and TSR are often shared, since both are concerned with running fewer tests and reducing the overall testing time; however TSR must ensure that there is no loss in fault detection capability in the long-term evolution of the suite.

Regarding the terms reduction and minimization, both options are used nearly interchangeably in the literature. According to Yoo and Harman [148], the difference in terminology is subtle: while both remove tests from the suite, *minimization* implies this change is temporary, while *reduction* stands for permanent removal of tests. Generally speaking, the same techniques can be applied for both ends so, from the perspective of researchers, the two terms are not distinct.

2.2.4 Test Case Amplification and Augmentation

As a program evolves and grows in scope, so must its test suite keep up with the additional features requirements. Today, this is mostly done manually by the development teams. In some cases developers are responsible for testing their own code; in others, developers test each other, or there can be designated testers whose job is to ensure other people's changes satisfy requirements and are error-free.

Unfortunately, tests do not add to the perceived value of a software product, since they do not provide direct functionality to end users. Thus, in a lot of cases, developers are encouraged to spend little time writing new tests or improving existing ones. As such, an automated solution can prove to be valuable to both reduce the manual work done by programmers and to improve the overall quality of new test cases. *Test case amplification* (TSA) is the technique to achieve that goal.

Given an SUT M and its test suite T that *partially* satisfies a set of requirements $\{r_1, ..., r_n\}$, TSA is described as a function $A(M, T, r)$ outputting a test suite $T' \supseteq T$ that satisfies more r_is than T. Much like TSR, coverage-based and requirements-based approaches are common, although the objective is naturally to increase coverage rather than maintain it. Metrics for measuring the output include the relative increase in coverage/requirements.

This problem is related to *test suite generation* (TSG) although, for the discussion of regression testing, TSA is a more useful concept. The difference is that TSA increases a pre-existing test suite, while TSG generates one from scratch using only the SUT as input. Since the latter does not presume an ongoing and evolving regression test suite, it was determined that it falls out of the scope of discussion of this thesis. That said, from a purely technical standpoint, both challenges are strongly related and, indeed, TSA is sometimes referred to as *incremental generation* of test cases.

There is a nuanced distinction between the usage of amplification and augmentation, and the two are sometimes conflated in the literature. Generally, *amplification* refers to any improvement of the test suite, which may happen by adding new tests or enhancing existing ones. On the other hand, *augmentation* implies the creation of new tests without modifying previous ones. By this definition, *test suite augmentation* is a problem embedded within test suite amplification, so for the purposes of this thesis, we refer as amplification the combined challenge.

Chapter 3

Literature Review

This chapter provides a Systematic Literature Review (SLR) with the purpose of reviewing the IR&A of RT approaches published in the latest years, i.e., since 2016. For the sake of comprehensiveness, we characterize as having IR&A not only those studies that report an evaluation on industrial applications (as was done by Ali et al. [4]), but also approaches that are explicitly motivated by industrial problems, or by related concerns, such as costs, scalability, or impact on the development procedures. Within this scope, we performed a systematic search over the five main digital libraries (ACM, IEEE, Springer, Scopus and Wiley) for RT studies mentioning industry or practice or applicability or scalability (or similar wordings) in their abstract, and completed this search with a snowballing cycle. We collected 1320 candidate papers published between 2016 and 2022 (780 via query, 540 via snowballing), and after applying a systematic selection process we identified a total of 78 primary studies that present IR&A approaches.

However, we understand that there is not a direct mapping between motivation and results, and approaches stemming from applicability concerns could end up with having low significance. In order to get a better assessment of the long-term impact of the papers after publication, we complemented our literature review with a survey sent to the authors of all the selected studies, asking them about the outcome of their research post-publication. We received responses from authors of 64% of the papers, reporting both positive and negative outcomes, including some of the reasons why an approach was unsuccessful. Some of the authors also signaled additional papers for consideration of the review, out of which we selected only 1.

The review includes a total of 79 primary studies. Based on a full reading of the selected papers and on the feedback received by their authors, we discuss the main characteristics of IR&A approaches, how they tackle applicability concerns, and whether they produced in impact in practice and why (or why not). We then also conducted a further survey among test practitioners to get their opinion in order to comment and possibly validate our conclusions. By applying a convenience sampling method we got answers from 23 practitioners who confirmed our findings and provided further useful insights in our study of investigating IR&A of proposed RT approaches.

This chapter provides the following contributions:

- A systematic literature review of recent RT techniques emphasizing IR&A;
- A survey among the authors of the selected primary studies that provides information about their impact;
- A survey among practitioners to validate and complement our findings.

The chapter is structured as follows: in the next section we overview related secondary studies. The formulated Research Questions (RQs) are defined and described in Section 3.2. In Section 3.3 we describe the study methodology, in particular the selection and data extraction process and the surveys with authors and practitioners. Section 3.4 then provides discussions and answers to the RQs: in Section 3.4.1 we answer RQ3.1, by overviewing and classifying existing techniques; in Section 3.4.2 we answer RQ3.2, which addresses IR&A concerns; in Section 3.4.3 we answer RQ3.2, about the impact obtained by the collected studies.

3.1 Related Works

The first recommended step before undertaking any new systematic review is to verify that such a study is actually needed [72]. Indeed, in view of the large set of papers published every year on RT techniques and related topics, it is natural that a good number of secondary studies reviewing the regression testing literature has also been produced.

The already cited study by Ali et al. [4] has previously verified whether existing reviews of literature regarding RT techniques took into consideration IR&A. After a systematic

Paper	Year	Techniques	Nº	Period	Systematic?	Context
Do [32]	2016	TCP, TCS, TSR	12	2010-2016		
Hao et al. [56]	2016	TCP	27	2010-2016		
Rosero et al. [120]	2016	TCP, TCS, TSR	25	2000-2014		
Kazmi et al. [69]	2017	TCS	47	2007-2015	✓	
Bajaj and Sangwan [9]	2018	TCP, TCS, TSR	15	1999-2016		Nature-inspired
Khatibsyarbini et al. [70]	2018	TCP	80	1999-2016	✓	
Mukherjee and Patnaik [101]	2018	TCP	90	2001-2018	✓	
Rehman Khan et al. [117]	2018	TSR	113	1993-2016	✓	
Bajaj and Sangwan [10]	2019	TCP	20	2006-2018	✓	Genetic
Ali et al. [4]	2019	TCP, TCS, TSR	38	2002-2017	Mix	
Danglot et al. [29]	2019	TSA	49[1]	1993-2017	✓	
Lou et al. [84]	2019	TCP	191	1997-2016	✓	
Hasnain et al. [60]	2020	TCP	65	2001-2017	✓	Web services
Prado Lima and Vergilio [114]	2020	TCP	35	2009-2019	✓	Continuous integration
Abdul Manan et al. [2]	2021	TCP	20	2011-2020	✓	Combinatorial
Hasnain et al. [61]	2021	TCP	24	2007-2019	✓	Ontology-based
Mohd-Shafie et al. [100]	2021	TCP	22[2]	2005-2018	✓	Model-based
Rosero et al. [119]	2021	TCP, TCS, TSR	40	2002-2020	✓	
Samad et al. [124]	2021	TCP	52	2007-2019	✓	
Ahmed et al. [3]	2022	TCP	21	2001-2019	✓	Value-based
Pan et al. [110]	2022	TCP, TCS	29	2006-2020	✓	Machine learning
Sadri-Moshkenani et al. [123]	2022	TCP, TCS, TSR	13[2]	2015-2019	✓	Cyber-physical

1: Ref. [29] covers test amplification, which is a wider scope than test augmentation, and the reported number of 49 primary studies includes the whole field. 2.: For Refs. [100] and [123] the reported number of primary studies also includes papers addressing test generation.

Table 3.1: Overview of existing secondary studies on RT.

search, they found eleven secondary studies spanning over 2008-2017, including [120, 42, 39, 152, 69, 59, 16, 148, 115, 131, 17]. After a thorough analysis of those reviews they concluded that at the time none of them addressed satisfactorily such aspects. The authors hence used such studies[1] as the start-set for a snowball sampling search, launched in August 2016. In order to verify if another review is needed, it is hence necessary to conduct a thorough examination of existing secondary studies on RT published since 2016.

We performed a search for secondary studies on RT over the same libraries queried for the primary studies (see Section 3.3.2) and complemented the search results with a snowballing cycle. We eventually identified 22 works published since 2016 that are listed in the first column of Table 3.1, whereas the second column includes the year the review was published.

In the third column of Table 3.1 we report which RT techniques are covered in the study. Most reviews only focus on TCP approaches [56, 101, 84, 70, 10, 114, 60, 2, 61, 100, 124, 3]. One study is dedicated solely to TCS [69], one other study to TSR [117], and again

[1]Actually 10 of them, as the authors explain that the 2017 survey [69] only appeared after they had concluded the analysis.

only one to TSA [29]. Finally, seven secondary studies investigate primary studies on multiple RT techniques [120, 32, 9, 4, 123, 119, 110].

In the 4th and 5th columns, we report the number of primary studies reviewed and the interval of years to which they belong, whereas the 6th column is checked if the review is conducted in systematic way. Finally, in the last column, we also report on the context of the review, if it focuses on techniques using a specific approach or covers a specific application domain. A version of this data is also included in our online repository (Chapter 7), with some additional notes; the intent is to update the table as more secondary studies are written and published.

For the sake of comparison, in the following paragraphs we briefly report the motivations behind the 22 reviews, grouped by the targeted technique (i.e., TCP, TCS, TSR, TSA, or multiple techniques).

TCP only The work by Hao et al. [56] aims at reviewing the advancements in TCP and identifying open challenges. Similar goals are pursued by Lou et al. [84], who analyze the primary studies along six aspects: algorithms, criteria, measurements, constraints, empirical studies, and scenarios. The objective of Khatibsyarbini et al. [70] was to review the experimental evidence relative to the most recent TCP approaches along with the metrics used for evaluating them. Mukherjee and Patnaik [101] generically aim to identify the most popular and useful TCP approaches. The review by Samad et al. [124] classifies existing work according to the algorithms or models adopted, the subjects of evaluation and the prioritization measures. A number of reviews focus on TCP for specific test approaches, namely: Bajaj and Sangwan [10] cover genetic algorithms; Abdul Manan et al. [2] address combinatorial testing; Hasnain et al. [61] consider ontology-based test methods; Mohd-Shafie et al. [100] cover model-based testing approaches, and Ahmed et al. [3] review TCP techniques that integrate value consideration, either in terms of fault severity or test case cost. Finally Hasnain et al. [60] investigate TCP approaches for web services, whereas Prado Lima and Vergilio [114] study how TCP has been adapted for Continuous Integration environments.

TCS only The only secondary study focusing on TCS work is Kazmi et al. [69], which aims at presenting the state-of-the-art in effective regression test case selection techniques.

TSR only The systematic review by Rehman Khan et al. [117] is motivated by the quality assessment of empirical studies employed to evaluate the test reduction approaches.

TSA only Danglot et al. [29] present the first review on test *amplification*, a novel term they introduce as an umbrella for various activities that aims at improving an existing test suite, including test augmentation, optimization, enrichment, or refactoring. The review is not specifically devoted to RT, but a subset of the primary studies they overview deals with creating new tests for assessing the effects of changes.

Multiple techniques Among the secondary studies that focus on multiple RT techniques, both Do [32] and Rosero et al. [120] aimed at generically providing an overview of recent research advances. Some authors instead were motivated to study more specific type of techniques: Bajaj and Sangwan [9] aimed at reviewing RT approaches leveraging nature-inspired algorithms, while Pan et al. [110] analyzed TCP and TCS studies that use Machine-Learning based techniques. The review by Sadri-Moshkenani et al. [123] characterizes the approaches and the open challenges for the generation, selection and prioritization of test cases for cyber-physical systems. Rosero et al. [119] provide a preliminary brief mapping of primary studies that report about industrial usage of RT techniques. Finally, the already mentioned study by Ali et al. [4] surveys RT research that has industrial relevance and applicability, and also creates a taxonomy useful for the communication between academia and industry.

Nearly all of the secondary studies express some concerns over IR&A of RT techniques, although in most cases these concerns are only mentioned in passing, and are not central to their motivations. Rosero et al. [120] report that only 16% of the surveyed primary studies experimented in industrial context. On the positive side, Do [32] observes that recently, more research is focusing on industrial software or open-source programs of different types. The review by Lou et al. [84] contains a subsection titled "Practical Values" in which they suggest researchers to consider TCP in practical scenarios and to develop usable TCP tools. The only two reviews that specifically target IR&A as this study are: *i)* the aforementioned work by Ali et al. [4]. However it selects only papers that performed evaluations with industrial subjects, and was motivated mainly by the goal of establishing a taxonomy for communicating RT research in a way that is accessible and relevant for practitioners; and *ii)* the preliminary work by Rosero et al. [119], but this is is a brief report that just classifies 40 selected primary studies found by searching the TCP literature for the term "industrial" without investigating in depth their characteristics and actual impact.

Considering the list of related secondary studies in Table 3.1, we conclude that a new secondary study specifically addressing progresses in latest years about IR&A is needed and can provide value to the research and development communities.

3.2 Research Questions

With these research questions, we aim to synthesize the current state of RT research in terms of most frequently used approaches and metrics as well as understand researchers' motivations and efforts regarding the practical implementation of their proposed techniques.

RQ3.1 What are the main approaches used for RT techniques and what are the metrics used to evaluate them? — We want to have an overview of the approaches most used in this field, including what information is required from the software, what algorithms are put into use and what goal are they trying to achieve. In addition, we want to know what metrics are widely accepted among researchers to evaluate such approaches and whether there is evidence to suggest that these metrics correlate to the technique's practical applicability.

RQ3.2 Is IR&A an explicit concern in RT research? — We want to find out if there is a meaningful number of papers that state IR&A as their motivation and include it as part of their problem description. Additionally, we want to understand what are the main steps that authors usually take towards addressing these concerns with the tools, techniques and solutions they provide.

RQ3.3 Is there evidence that techniques developed in academia make their way into software in practice? — In an effort to measure the extent of active industry-academia collaborations, we want to highlight examples of techniques that have been put into practice at some point during the development of the work. To provide a clearer picture, we asked authors of the selected studies to provide details of the state of their research post publication.

3.3 Methodology

This section elaborates the entire review process, from its conceptual phases to the list of selected papers and how we organized their contents, until conclusions drawing. First, we establish the research questions that drive both the selection of papers and the data presented and discussed in subsequent sections. Then, we explain the planning and design phase of the survey, followed by its actual execution. We also highlight the data that was extracted from each paper and the process of sending complementary questions to authors via e-mail. We present the survey with practitioners that we conducted in support of the study conclusions. Finally, we make available the relevant data needed to replicate this process, to the extent of possibility.

3.3.1 Planning and Design of the Review

To answer the questions above, we designed the following literature review process. For the purposes of this review, a "regression testing technique" addresses test case prioritization (TCP), test case selection (TCS), test suite reduction (TSR), or test suite amplification (TSA). Only papers concerning one or more of these four challenges should be considered. Due to the scale of the available literature and our focused interest in recent developments, we only look for papers published between January/2016 and July/2022. We also only consider papers written in English.

We want to focus on papers that either signify an advance of the state of the art in academia towards practicality, or includes data and discussions that might help guide future researchers to make their research more valuable for practitioners. Thus, to be included in the review, a paper must satisfy at least one of these inclusion criteria:

- It introduces a new regression testing technique and provides evidence that it addresses a real-world concern, or provides substantial benefits in experiments performed with real software;

- It introduces and/or discusses a metric for evaluating regression testing techniques with evidence that it might be valuable in practice; or
- It provides a case study of how regression testing is done in a certain team or company, and provides some insight into the actual needs of practitioners.

We also want to avoid certain topics that are related to regression testing but would increase the scope of the review beyond necessary. A paper should be excluded according to following criteria:

- It is regarding software testing education, as this is a completely different challenge;
- It proposes a technique for test suite generation, which is related albeit distinct problem from TSA[2];
- It is concerning security testing, because it typically requires specific types of techniques [41]; or
- It is concerning software product lines or highly configurable software, as these also present quite different challenges from typical regression testing [15].

After collecting the unfiltered set of papers, the inclusion and exclusion criteria are applied by each author to a random sample set based on their titles, abstracts and, if needed, superficial analysis of the text. In case of divergences in the analysis, the authors should discuss their conclusions. The Cohen Kappa agreement measure [26], a scale from -1 to 1, is used to determine if both authors are generally in agreement regarding this sample of papers. Upon establishing a satisfying agreement value, the analysis of remaining papers are split among the authors. If it is not clear whether a paper fully satisfies the criteria, it is brought for discussion among all authors until a mutual decision is reached.

After this initial analysis, full-text assessment of the remaining literature is performed. The following quality criteria are to be used to further narrow down the papers that are relevant to our research goals:

- The writing and presentation quality should not hinder comprehension;

[2]The primary difference is that *test suite augmentation* presupposes the existence of a test suite to enhance, while *test suite generation* can create a new test suite from scratch.

- A paper should provide evidence that they address a problem found in real-world software development and/or that the technique was evaluated on real-world software;
- The metrics used for evaluation should be clear and the authors should provide some reasoning as to why they are relevant; and
- In case there are multiple papers by the same group of authors that reference versions of the same work, we keep the most extensive one, avoiding, for instance, a conference paper and a journal paper that address the same research (if they are equivalent, we keep the most recent one).

The results from our queries are complemented by both forward and backward snowballing to improve the comprehensiveness of the review. The same date restrictions and criteria apply to papers found via snowballing.

Finally, a questionnaire with authors (Section 3.3.4) is used in order to clarify and update some details regarding the selected papers. The authors have the option of suggesting additional papers for consideration in this study; in that case, they should also be analyzed according to the established criteria.

3.3.2 Executing the Review

We began by assembling a list of keywords that form the basis of our queries, including potential variants of the same terms. These are: test/testing, evaluate/evaluation, metric, indicator, investment, cost, relevant/relevance, industry/industrial, practice/practical/practitioner, applicable/applicability, scale/scalability, regression, selection, prioriti[s/z]ation, amplification/augmentation, reduction/minimi[s/z]ation software. These keywords were used to manually experiment with the ACM and IEEE digital libraries, in order to have a general understanding of the relevance of the results. We found, for example, that the term "regression" would often bring papers on the broad topic of machine learning (even not related to RT), so we had to make sure the word "software" was also mentioned in the abstract.

Once the desired keywords were established, we built a query combining them. The query went through several iterations, in order to maximize the likelihood of finding all the

ACM	IEEE	Springer	Scopus	Wiley	**Total**
217	189	202	285	31	**865**

Table 3.2: Number of results for each search engine

papers that are relevant to our research, while also minimizing the number of papers in excess. The final query was structured as:

```
Title:(test OR testing) AND
Abstract:(evaluat* OR metric OR indicator OR investment OR cost OR relevan*) AND
Abstract:(industr* OR practic* OR applicab* OR scal*) AND
Abstract:(regression OR selection OR prioriti* OR augmentation OR
        amplification OR reduction OR minimi*) AND
Abstract:(software)
```

Queries were executed on five digital libraries: ACM, IEEE, Springer, Scopus and Wiley. The searches were performed on Nov. 4, 2021 and Jul. 27, 2022. Each of the five search engines uses a different syntax for queries, so we adapted the query to each syntax while keeping its overall meaning as similar as possible. We also attempted to include results from Science Direct into the study, but its search engine cannot handle all of the query details. In all of the search engines, we narrowed the results to papers published since January of 2016 and under the fields of Computer Science and Software Engineering.

The number of results were: ACM (217), IEEE (189), Springer (202), Scopus (285), Wiley (31). The total number was 865. Removing exact duplicates that were found in more than one digital library, the number of papers considered for the review was 780. We then assembled a spreadsheet with the year, author list, title, abstract and keywords of each paper in a shuffled order, to be reviewed by two of the authors of this study. A sample of 40 papers was used to calculate the Cohen Kappa measure and establish a consensus. From these, we achieved an agreement value of 0.89, which is considered very high, so we were satisfied with the criteria and the authors' interpretations of them. Processing of the remaining papers was split among us.

This initial filtering resulted in 180 remaining papers, on which we conducted full-text analysis to ensure topic relevance and satisfaction of exclusion and quality criteria. Like the previous step, a set of 20 papers was analyzed and discussed by all authors, and again we achieved a satisfactory agreement value; analysis of the remaining ones was split.

When there was uncertainty, some were discussed between the authors and we decided to be overly inclusive at this step, leaving the most rigorous filtering for last. Papers from the same groups of authors were also flagged to then determine if they were describing the same or similar work. Our list, before any snowballing, contained 86 papers.

Snowballing upon the selected primary studies was performed on Nov. 15 2021 and Aug. 15 2022, using the papers' own references for backward snowballing and Google Scholar for forward snowballing. This resulted in a further 540 papers published since 2016, after removing duplicates of papers already found in the previous review step. From the snowballing sample, we selected 108 candidates, forming a pool of 194 papers for analysis.

We performed full-text analysis of these papers, carefully extracting the information pointed out in Section 3.3.3 and using that to form the decision of whether or not the paper satisfied our inclusion and quality criteria. Again during this step, we divided the papers among the authors and, in case there was uncertainty regarding one paper, we made the decision together.

Later, when we received responses from the authors of the selected studies (Section 3.3.4), four papers were brought to our attention. We applied all our aforementioned criteria to these suggestions and decided to incorporate one of them into the review. It had not been caught by either the query or the snowballing process, but we understand that a single missed paper is evidence that our review process has been sufficiently comprehensive.

Finally, our survey, as is presented in this study, contains the 79 papers listed in Table 3.3: 46 found by the query; 16 from backward snowballing; 16 from forward snowballing; and 1 author suggestion. The entire selection process is illustrated in Figure 3.1.

It is noteworthy that other studies, not included in this review, are also important for the advancement of software engineering research. During the execution of this review, we came across several papers that provide meaningful contributions to the theory or practice of regression testing research, but exist in an isolated context. The focus of this collection of studies is to find techniques and approaches that are applicable in real software or are close to that - oftentimes, these papers are the result of a longer series of smaller contributions that ultimately culminated in a usable product.

ID	Year	Authors	Title	TCP	TCS	TSR	TSA

S1	2016	Srikanth et al. [135]	Requirements Based Test Prioritization Using Risk Factors	●	○	○	○
S2	2016	Noor and Hemmati [105]	A similarity-based approach for test case prioritization using historical failure data	●	○	○	○
S3	2016	Schwartz and Do [125]	Cost-effective regression testing through adaptive test prioritization strategies	●	○	○	○
S4	2016	Hirzel and Klaeren [64]	Graph-walk-based selective regression testing of web applications created with Google web toolkit	○	●	○	○
S5	2016	Lu et al. [85]	How does regression test prioritization perform in real-world software evolution?	●	○	○	○
S6	2016	Vöst and Wagner [141]	Trace-based test selection to support continuous integration in the automotive industry	○	●	○	○
S7	2016	Wang et al. [142]	Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search	●	○	○	○
S8	2016	Srikanth et al. [134]	Test Case Prioritization of Build Acceptance Tests for an Enterprise Cloud Application	●	○	○	○
S9	2017	Blondeau et al. [12]	Test case selection in industry: an analysis of issues related to static approaches	○	●	○	○
S10	2016	Pradhan et al. [113]	Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study	○	●	○	○
S11	2016	Buchgeher et al. [13]	Improving testing in an enterprise SOA with an architecture-based approach	●	●	○	○
S12	2016	Tahvili et al. [138]	Dynamic integration test selection based on test case dependencies	●	●	○	○
S13	2016	Öqvist et al. [107]	Extraction-based regression test selection	○	●	○	○
S14	2016	Magalhães et al. [92]	Automatic selection of test cases for regression testing	○	●	○	○
S15	2016	Aman et al. [5]	Application of Mahalanobis-Taguchi Method and 0-1 Programming Method to Cost-Effective Regression Testing	●	○	○	○
S16	2016	Busjaeger and Xie [14]	Learning for test prioritization: An industrial case study	●	○	○	○
S17	2016	Yoshida et al. [150]	FSX: A tool for fine-grained incremental unit test generation for C/C++ Programs	○	○	○	●
S18	2016	Tahvili et al. [137]	Cost-benefit analysis of using dependency knowledge at integration testing	●	○	○	○
S19	2017	Ramler et al. [116]	Tool support for change-based regression testing: An industry experience report	○	●	○	○

S20	2016	Strandberg et al. [136]	Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors	●	●	○	○
S21	2016	Marijan and Liaaen [93]	Effect of time window on the performance of continuous regression testing	●	○	○	○
S22	2017	Gotlieb and Marijan [50]	Using global constraints to automate regression testing	○	○	●	○
S23	2017	Chi et al. [23]	Multi-Level Random Walk for Software Test Suite Reduction	○	○	●	○
S24	2017	Bach et al. [7]	Coverage-Based Reduction of Test Execution Time: Lessons from a Very Large Industrial Project	●	●	○	○
S25	2017	Spieker et al. [133]	Reinforcement learning for automatic test case prioritization and selection in continuous integration	●	●	○	○
S26	2017	Vasic et al. [140]	File-Level vs. Module-Level Regression Test Selection for .NET	○	●	○	○
S27	2017	Celik et al. [18]	Regression test selection across JVM boundaries	○	●	○	○
S28	2018	Ouriques et al. [108]	Test case prioritization techniques for model-based testing: a replicated study	●	○	○	○
S29	2017	Kwon and Ko [74]	Cost-effective regression testing using bloom filters in continuous integration development environments	●	●	○	○
S30	2018	Garousi et al. [45]	Multi-objective regression test selection in practice: An empirical study in the defense software industry	○	●	○	○
S31	2018	Shi et al. [127]	Evaluating test-suite reduction in real software evolution	○	○	●	○
S32	2018	Haghighatkhah et al. [55]	Test prioritization in continuous integration environments	●	○	○	○
S33	2018	Zhang [154]	Hybrid regression test selection	○	●	○	○
S34	2018	Miranda et al. [99]	FAST Approaches to Scalable Similarity-Based Test Case Prioritization	●	○	○	○
S35	2018	Yilmaz and Tarhan [147]	A case study to compare regression test selection techniques on open-source software projects	○	●	○	○
S36	2018	Chen et al. [20]	Optimizing Test Prioritization via Test Distribution Analysis	●	○	○	○
S37	2018	Celik et al. [19]	Regression Test Selection for TizenRT	○	●	○	○
S38	2018	Zhu et al. [158]	Test re-prioritization in continuous testing environments	●	○	○	○
S39	2018	Azizi and Do [6]	Retest: A cost effective test case selection technique for modern software development	○	●	○	○
S40	2019	Guo et al. [54]	Decomposing Composite Changes for Code Review and Regression Test Selection in Evolving Software	○	●	○	○
S41	2019	Zhong et al. [155]	TestSage: Regression test selection for large-scale Web service testing	○	●	○	○
S42	2019	Fu et al. [43]	Resurgence of Regression Test Selection for C++	○	●	○	○
S43	2019	Eda and Do [34]	An efficient regression testing approach for PHP Web applications using test selection and reusable constraints	○	●	●	○

S44	2019	Goyal et al. [51]	Test suite minimization of evolving software systems: A case study	○	○	●	○
S45	2019	Yu et al. [151]	TERMINATOR: better automated UI test case prioritization	●	○	○	○
S46	2019	Correia et al. [27]	MOTSD: A multi-objective test selection tool using test suite diagnosability	●	●	○	○
S47	2018	Machalica et al. [90]	Predictive Test Selection	○	●	○	○
S48	2019	Najafi et al. [102]	Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report	●	●	○	○
S49	2019	Leong et al. [78]	Assessing Transition-Based Test Selection Algorithms at Google	○	●	○	○
S50	2019	Cruciani et al. [28]	Scalable Approaches for Test Suite Reduction	○	○	●	○
S51	2019	Philip et al. [112]	FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services	○	○	●	○
S52	2020	Magalhães et al. [91]	HSP: A hybrid selection and prioritisation of regression test cases based on information retrieval and code coverage applied on an industrial case study	●	●	○	○
S53	2019	Wu et al. [143]	A Time Window Based Reinforcement Learning Reward for Test Case Prioritization in Continuous Integration	●	○	○	○
S54	2019	Land et al. [75]	An Industrial Evaluation of Test Prioritisation Criteria and Metrics	●	○	○	○
S55	2020	Noemmer and Haas [104]	An Evaluation of Test Suite Minimization Techniques	○	○	●	○
S56	2020	Lübke [86]	Selecting and Prioritizing Regression Test Suites by Production Usage Risk in Time-Constrained Environments	●	●	○	○
S57	2019	Yackley et al. [145]	Simultaneous refactoring and regression testing	○	●	○	○
S58	2019	Shi et al. [129]	Understanding and improving regression test selection in continuous integration	○	●	○	○
S59	2022	Lima and Vergilio [83]	A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments	●	○	○	○
S60	2020	Zhou et al. [157]	Beating Random Test Case Prioritization	●	○	○	○
S61	2020	Peng et al. [111]	Empirically revisiting and enhancing IR-based test-case prioritization	●	○	○	○
S62	2020	Bertolino et al. [11]	Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration	●	●	○	○
S63	2021	Chen and Chen [22]	Multi-objective regression test selection	○	●	○	○
S64	2021	Rosenbauer et al. [118]	An Artificial Immune System for Black Box Test Case Selection	○	○	○	○

S65	2022	Bagherzadeh et al. [8]	Reinforcement learning for test case prioritization	●	○	○	○
S66	2021	Elsner et al. [37]	Empirically evaluating readily available information for regression test optimization in continuous integration	●	●	○	○
S67	2020	Pan et al. [109]	Dynamic Time Window based Reward for Reinforcement Learning in Continuous Integration Testing	●	○	○	○
S68	2021	Mehta et al. [94]	Data-driven test selection at scale	○	●	○	○
S69	2021	Xu et al. [144]	A Requirement-based Regression Test Selection Technique in Behavior-Driven Development	○	●	○	○
S70	2022	Zhou et al. [156]	Parallel Test Prioritization	●	○	○	○
S71	2021	Sharif et al. [126]	DeepOrder: Deep Learning for Test Case Prioritization in Continuous Integration Testing	●	○	○	○
S72	2021	Li et al. [82]	AGA: An Accelerated Greedy Additional Algorithm for Test Case Prioritization	●	○	○	○
S73	2021	Chen et al. [21]	Context-Aware Regression Test Selection	○	●	○	○
S74	2022	Zhang et al. [153]	Comparing and Combining Analysis-Based and Learning-Based Regression Test Selection	○	●	○	○
S75	2022	Abdelkarim and ElAdawi [1]	TCP-Net: Test Case Prioritization using End-to-End Deep Neural Networks	●	○	○	○
S76	2022	Çıngıl and Sözer [25]	Black-box Test Case Selection by Relating Code Changes with Previously Fixed Defects	○	●	○	○
S77	2022	Yaraghi et al. [146]	Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts	●	○	○	○
S78	2022	Omri and Sinz [106]	Learning to Rank for Test Case Prioritization	●	○	○	○
S79	2022	Greca et al. [53]	Comparing and combining file-based selection and similarity-based prioritization towards regression test orchestration	●	●	○	○
			Totals	46	41	8	1

Table 3.3: Selected papers.

3.3.3 Data Extraction

To collect the data, each of the selected papers was assigned to one author to lead the data extraction, and to another author to review the data afterwards. Thus, each paper was thoroughly reviewed by at least two authors. At the end of this phase, the three authors

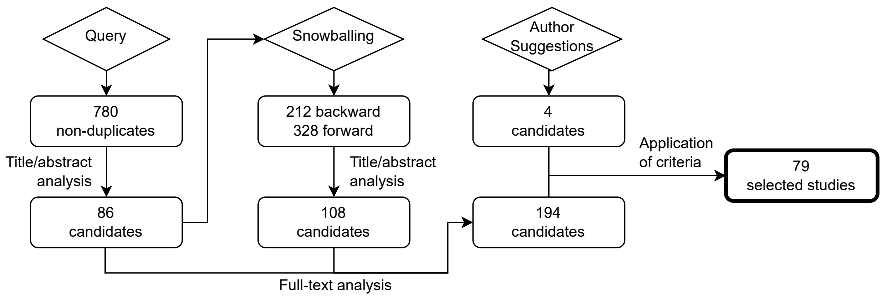


Figure 3.1: Diagram of the literature review process.

performed a broad review of the collected information in order to ensure consistency of the results.

During the full-text analysis of the selected papers, we took notes of four groups of properties we wished to extract from each paper. First, we wanted to have the core bibliographical information of the paper. Then, we categorized the papers according to the RT challenge being addressed and contextual factors such as software type and development environment. We also took note of eight properties we considered important regarding IR&A of each proposed technique or case study. Finally, we synthesized the results of the papers by highlighting the types of approaches and metrics used and, if available, the open challenges/future work discussed by the authors. The properties we collected are listed in Table 3.4.

Some further explanation is needed regarding the "applicability concerns" properties. During the data extraction, it became clear to us that there is some ambiguity regarding industrial motivation; among the non-selected papers, we also saw a great number of them briefly mentioning industry needs in the abstract and introduction, but not forming a connection between those needs and the technique being proposed. So, for our criteria, industrial needs must not only be mentioned, but clearly stated with motivating evidence and/or references, and serve as the actual principle behind the idea of the study.

Regarding the industrial evaluation of results, this property is satisfied when experiments are performed directly on industrial software, usually through collaboration with a technology company. However, there are plenty of papers that have robust experiments performed on notable open-source software, such as those from the Apache and Mozilla foundations. Thus, we collect the following information about the subjects:

- Its openness, which can be industrial proprietary, industrial open-source, fully open-source, or an academic dataset;
- Its testing scale (small up to 500 TCs, medium up to 2,000 TCs, large up to 10,000 TCs, or very large if more than that)[3];
- The language used for writing tests, which can be a programming language, natural language, a domain-specific language or a combination; and
- Its origin (the company that wrote it, or the dataset it is from).

We also checked to see if there is feedback from practitioners in the text of the paper. Relatively few authors include feedback and, often times, it is only a brief passage. Sometimes feedback seems to be implied, but we only considered explicit references to comments from practitioners (in either direct or indirect quotes).

The remaining properties are more straightforward: For experiment subject(s) and industry partner, we merely point out the kind of software (or the specific software, if possible) used for evaluation, as well as any collaboration received from a company. For industrial author(s), we check if the authors of the paper come from an academic, industrial or mixed background. "Available tool" is a URL pointing to an implementation of the technique, if it exists (regardless if it is source code, a plug-in, or a robust replication package). Finally, "put into practice" indicates whether the technique was actually incorporated into the development workflow of a software, to the extent of the information contained in the paper.

Most of the data extracted according to the form can be found in this document under Table 3.3 (bibliographical data); Figures 3.2 and 3.3 (approaches); Figures 3.4 and 3.5 (metrics); and Table 3.7 (IR&A relevance properties). Additional properties, such as abstracts, DOIs, details on the experiment subjects, and links to supplementary material, can be found in the replication package (Section 3.3.6) and the live repository (Chapter 7).

[3]It is also important to note that the number of test cases is only one dimension of scale: on S76, for example, evaluation was performed on Smart TV apps with only 38 test cases, but the testing time was over 7 hours.

Bibliographical data	Basic information about the publication.
Date	The date the paper was made available online.
Authors	The list of authors.
Title	The title of the paper.
Abstract	The abstract of the paper.
Venue and Publisher	The conference or journal where it was published and its organization.
DOI	The Digital Object Identifier of the paper.
Categorization	Details regarding the problem addressed by the paper.
RT challenges	Whether the paper covers TCP, TCS, TSR, TSA or a combination.
Context	The type of software targeted by the approach.
Applicability concerns	Properties of the paper related to its IR&A.
Industry motivation	Whether the paper is clearly motivated by an industrially relevant problem.
Industry evaluation	Whether the technique is evaluated in industrial software or sufficiently large-scale open-source projects.
Experiment subject(s)	Which software or kind of software was used for the experimental evaluation of the technique, including the testing scale, the availability and the language in which tests are written.
Industry partner	Which, if any, industrial partner collaborated with the development and/or evaluation of the technique.
Industrial author	Whether one or more of the authors of the paper come from industry.
Practitioner feedback	Whether practitioners were consulted to provide feedback to the results of the paper.
Available tool	Whether the technique introduced in the paper is available to be used, either as a prototype or as a complete tool. If true, we also stored the relevant URLs.
Put into practice	Whether the proposed tool has been adopted into the development process of a certain software.
Findings	Details of the proposed technique and remaining challenges.
Approach	What sort of algorithm and information the technique is using.
Metrics	What criteria are being used for evaluating the techniques.
Open challenges	What the authors list as next steps and unsolved issues related to the problem they addressed.

Table 3.4: Data extraction form.

3.3.4 Questionnaire with Authors

As we collected the data needed to answer the research questions, we realized the need of a perspective beyond what is possible to extract solely from the papers, particularly regarding the ongoing usage of the described techniques. This happens because the information contained in the papers themselves might be out-of-date or unclear. For example, the authors of S11 mention that their tool, TePSEx, was in use at the time of the publication; however, it is impossible to tell from the paper itself whether the situation has changed since 2016. Conversely, there are also several papers that mention a practical implementation among their future work [S7, S12, S22, S37, S51], but we were not able to find follow-up papers clarifying whether that actually happened.

In order to provide a satisfactory answer to RQ3.3, we reached out to the authors of the papers via e-mail. Our initial objective was to discover if techniques were ever put into practice and, if so, if they continue to be used to this day. We also realized that the authors could also provide fruitful insight into RQ3.1 and RQ3.2, so it ultimately became an important pillar of this work.

We were able to contact authors from most of the papers; in some cases we were not able to locate the author's e-mail address, or the address is no longer valid. The authors were given 12 days to respond and we received replies related to 51 out of the 79 papers — in some cases, one author answered for several papers, in others several authors of the same paper provided answers. The total number of responding authors was 45, although five of them did not provide meaningful answers (i.e. asked us to contact another co-author or said they were not able to answer our questions).

The e-mail sent out to the authors had the following questions:

1. Is there a functional version of your technique (tool, prototype, source code, etc.) available online? If so, please share with us the URL.
2. Was there an attempt to implement your technique in industrial or large open-source software? Is the technique currently in use with the software?
3. If the technique was put into practice, were the metrics used in the paper relevant for the technique's applicability? If not, were there other metrics that proved to be useful?

In addition, we also asked if the authors authorized their answers to be used in this study, if we could link them to their answers, and if they wish to be contacted about updates to this study. The full template of the e-mail is found in Appendix A.1.

All responding authors authorized the use of their answers, but several of them asked not to be linked directly to specific answers due to non-disclosure agreements with industrial partners; therefore, we use the received answers broadly, collecting quantitative and qualitative data from them without specifying which piece of information came from which author. Whenever a direct quote is significant, we transcribe it anonymously; for readability we use an arbitrary ID numbering.

3.3.5 Survey with Practitioners

In addition to the questions we sent to the authors of reviewed papers, we also prepared a survey destined to practitioners. The objective of this survey was to complement and verify some of the conclusions we draw from the literature, and help align the interests of academia and industry.

The survey was disseminated using a convenience sample, including contacts we personally know in industry and people who participated in software testing centric events. We considered also putting the survey in public online forums centered on software testing/engineering, but ultimately decided against that for fear of low-quality responses and data pollution.

We received 23 responses from practitioners in six different countries (Brazil, Italy, Finland, Hungary, Portugal and Sweden). Obviously this survey covers an extremely small part of software testing practice, but it is possible to trace some common elements pointed out by the respondent that corroborate some of the findings and conclusions we had extracted from the literature. When relevant, these responses are used in Section 3.4.3 and also contribute to Chapter 6.

Due to space concerns, we cannot include the full questionnaire here, but it is available online, along with the anonymized responses we received. The main points covered in it were:

1. What are the most common pain points when it comes to regression testing?

2. Do you know of, or have you ever used, a regression testing tool originating in academia?

2.1 If so, how was the experience of using it?

3. Do you stay informed on current advances in software engineering research? Are there attempts of collaboration between your company and academia?

We also asked for their company, country and role, which we used to assess the diversity of respondents. This will not be published for privacy reasons. Again, the full transcript of the survey questions is available in Appendix A.2.

3.3.6 Replicability

To allow replicability of our review and clearly describe the thought process behind the choice of included studies, we make a replication package available online[4]. This package includes the original search queries, the list of papers that we included or excluded via the criteria, the full contents of the data extraction form, and the data used to generate

[4]Available at: https://bit.ly/3gCKh7V. For the purposes of review, we make the online material available through Google Sheets. This will be updated for the final release of the paper.

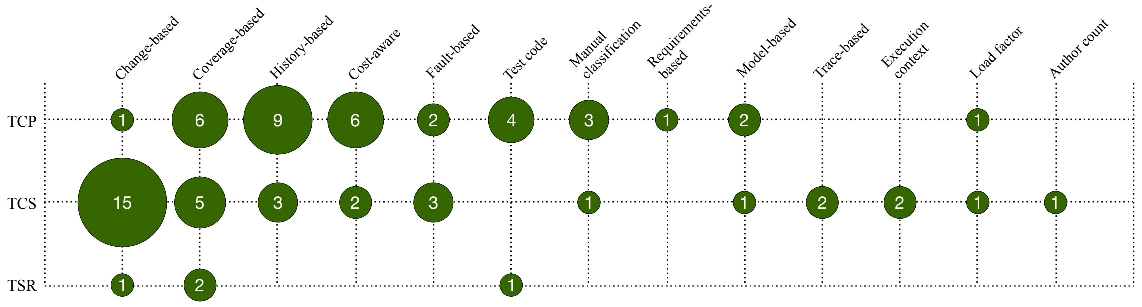


FIGURE 3.2: Distribution of information approaches.

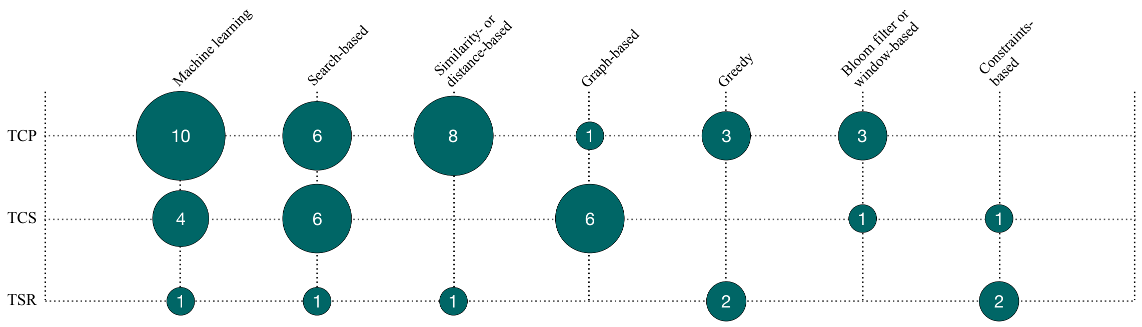


FIGURE 3.3: Distribution of algorithm approaches.

the figures. It also includes the full version of the e-mail template sent to the authors and the full questionnaire sent to practitioners.

3.4 Discussion

3.4.1 RQ3.1: Common Approaches and Metrics in RT research

A summary of the main approaches used to tackle RT challenges is presented in Table 3.5. We observe that the approaches adopted by a technique may serve to two different purposes: one is regarding the source from where the information used as input for a technique is collected[5]; a second purpose refers to the actual algorithm used to address the problem to solve. Correspondingly, the main approaches used to tackle RT challenges are presented in Figure 3.2 and in Figure 3.3, respectively. Regarding information, change-based, coverage-based, history-based and cost-aware approaches are the most common; while machine learning-based, search-based, similarity-based and graph-based are the popular algorithmic approaches.

[5]This is also referred to as criteria in the literature [84].

Information	**TCP**	**TCS**	**TSR**	**TSA**	**Description**
History-based	S8, S15, S16, S20, S21, S29, S32, S45, S48	S29, S47, S48			Uses information from previous testing cycles to decide about test case relevance.
Change-based	S20	S13, S19, S26, S27, S33, S35, S37, S40, S42, S43, S58, S73, S74, S76, S79	S43		Uses changes between versions to identify the relevant test cases.
Coverage-based	S16, S24, S28, S45, S52, S56	S19, S24, S41, S52, S56	S31, S55	S17	Uses structural coverage information, whereby coverage can be of statement, method, class, file, etc.
Cost-aware	S5, S7, S12, S18, S45, S70	S12, S63			Utilizes test case cost or time information to assess test relevance.
Requirements-based	S1		S44		Relate tests with project-sensitive information, such as requirements and risk assessments.
Manual classification	S12, S18, S45	S12			Requires at least some information that must be manually inputted by an expert.
Model-based	S11, S28	S11			Informs the test technique using behavioural or architectural models.
Trace-based		S6, S41			Provides inputs and keeps track of the effects of those inputs throughout the program.
Fault-based	S21, S38	S49, S63, S76			Utilizes information related to fault detection or failure beaviour.
Test code	S2, S34, S61, S79		S50		Uses the plain text source code of the test cases.
Load factor	S11	S11			Indicates what parts of the SUT are most used by different services and components.
Author count		S49			Number of authors associated with a certain part of the SUT.
Execution context		S27, S73			Considers environment data such as libraries, APIs, databases, operating system, etc.
Algorithm	**TCP**	**TCS**	**TSR**	**TSA**	**Description**
Similarity or distance-based	S2, S15, S16, S28, S32, S34, S60, S79		S50		Assesses test cases based on their similarity, with the objective of diversifying the suite.
Search-based	S5, S7, S46, S52, S61, S70	S10, S14, S30, S46, S52, S64	S23		Utilizes search-based algorithms, such as genetic or nature-inspired ones
Machine learning-based	S16, S25, S53, S59, S62, S65, S66, S75, S77, S78	S47, S66, S68, S74	S51		Trains a ML model using historical or other data. Includes supervised, unsupervised and reinforcement learning methods.
Graph-based	S28	S4, S19, S35, S37, S39, S62			Creates a graph representation of the SUT and utilizes graph theory algorithms to solve problems.
Greedy	S5, S70, S72		S24, S55		Utilizes greedy algorithms and heuristics (usually based on coverage or similarity information).
Constraints-based		S43	S22, S43		Utilizes constraint programming paradigm.
Bloom filter, window-based	S3, S29, S67	S29			Utilizes Bloom filter data structures and time windows to filter out tests that fail only once.

Table 3.5: Information- and Algorithm-based Approaches

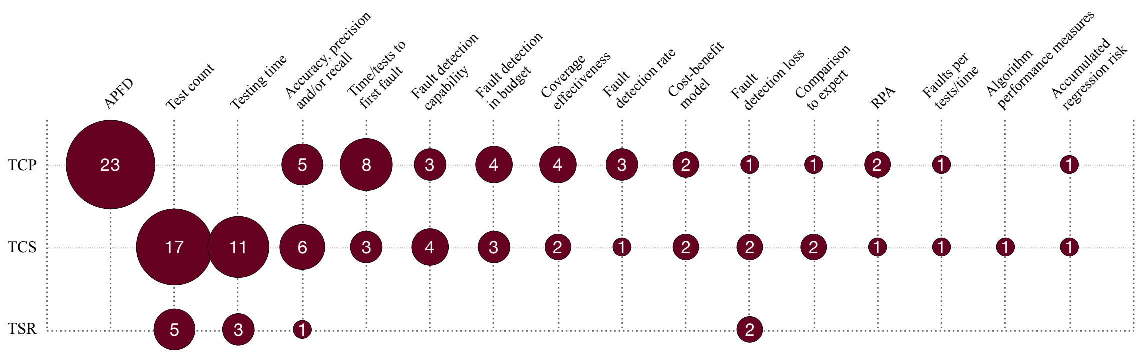


FIGURE 3.4: Distribution of effectiveness metrics.

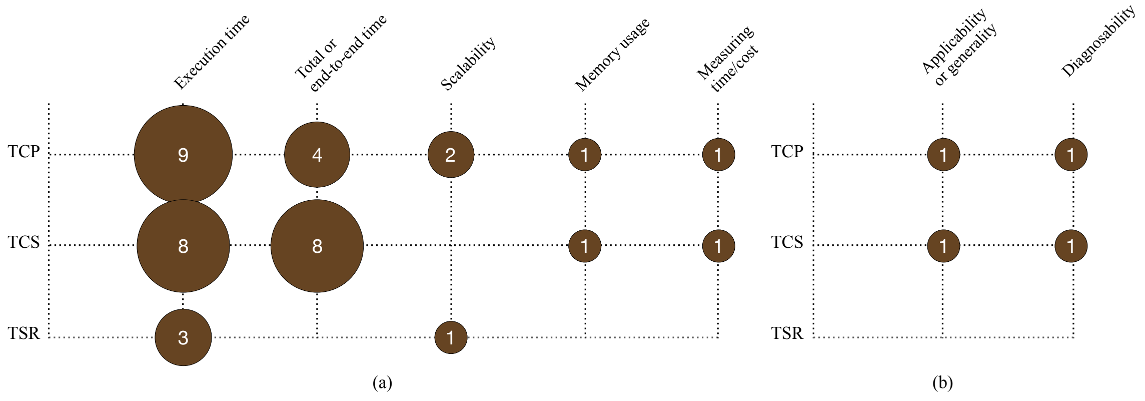


FIGURE 3.5: Distribution of (a) efficiency and (b) other metrics.

The main metrics reported in the literature are shown in Table 3.6, Figure 3.4 and Figure 3.5, grouped according to their main goal[6]. The reported metrics primarily focus on effectiveness (how good a solution is at accomplishing its task) or efficiency (the time and cost of using the solution), but two metrics were identified that are neither — namely, applicability/generality and diagnosability.

APFD is the most widely accepted metric for assessing TCP approaches. Because TCS and TSR both have the goal of running fewer tests than an original test suite, their metrics are mostly shared: testing time, selection count and fault detection ability are the most common ones. The set of accuracy/precision/recall appears to be the effectiveness metric that covers the most situations. For efficiency, the execution time of a technique is both widely used and is useful for any kind of solution.

In our questionnaire to the authors, we included a question focused on the choice of metrics. We asked authors who had successful or attempted attempts of implementing their technique whether the metrics described in the paper proved to be relevant in practice, or if additional measures were needed. We received 27 meaningful responses to that question,

[6]In each figure, we omit TSA due to space concerns, as only one paper (S17) covers it.

Effectiveness	**TCP**	**TCS**	**TSR**	**TSA**	**Description**
Selection/reductio count/percentage		S4, S6, S9, S24, S26, S27, S33, S35, S37, S39, S42, S43, S47, S58, S73, S74, S76	S22, S23, S43, S44, S55		Absolute or relative size of the resulting test suite compared to the original.
Average Percentage of Faults Detected (APFD)	S1, S5, S8, S16, S21, S25, S28, S32, S34, S36, S45, S53, S59, S61, S65, S67, S70, S72, S75, S77, S78, S79				A measure of how quickly a test suite detects faults, on average. Includes many variations, such as APFDc and NAPFD.
Testing time		S12, S19, S27, S30, S35, S37, S41, S58, S68, S74, S76	S44, S51, S55		Time required to execute the prioritized/selected/reduced test suite as opposed to the original suite.
Accuracy, precision and recall	S16, S29, S67, S75, S78	S9, S14, S29, S40, S47, S69	S51		Measures of correctness and completeness of the resulting test suite (e.g., count of false positives and false negatives).
Fault Detection Capability	S3, S7, S21	S29, S64, S73, S76			Number or proportion of faults detected by the resulting suite compared to the original.
Fault Detection Rate (FDR)	S15, S20, S45	S39			Time to detect faults compared to the optimal RT suite.
Coverage Effectiveness (CE)	S2, S45, S52, S56	S52, S56		S17	Measure of the tradeoff between cost of the test suite and structural coverage of the SUT.
Time/tests To First Failure	S2, S36, S38, S59, S60, S67, S70, S79	S9, S64, S79			Number of tests or amount of time needed to reach the first failure.
Fault detection within a budget	S7, S24, S59, S79	S10, S24, S79			Faults still detected when restricting the testing time budget.
Cost-benefit model	S3, S18	S30, S68			Mathematical models considering costs and benefits of applying a technique throughout development.
Fault Detection Loss	S48	S48, S63	S31, S50		Number or proportion of faults undetected by the selected/reduced test suite compared to the original.
Comparison to expert	S11	S11, S14			Compares the output of the tool with a list of tests selected by the project architect.
Faults per tests or time	S29	S29			Number of faults deteted per number of tests or testing time.
Number of tests added				S17	Number of tests added to the test suite.
Algorithm performance measures		S10			Fitness value or hypervolume metrics applied to search-based algorithms
Accumulated regression risk	S56	S56			How much of the ”regression risk” is covered by the tests.
Rank Percentile Average (RPA)	S62, S65	S62			Comparison between the predicted ranking and the actual ranking (from the dataset).
Efficiency	**TCP**	**TCS**	**TSR**	**TSA**	**Description**
Execution time	S7, S32, S34, S48, S53, S59, S70, S72, S79	S4, S26, S27, S41, S48, S69, S74, S79	S22, S23, S50	S17	Time required to run the tool (e.g., selection time, prioritization time, etc).
Total/End-to-end time	S7, S34, S62, S79	S13, S26, S33, S37, S42, S62, S74, S79			End-to-end time, combining measuring time, execution time and testing time. Due to this, it is a measure of both efficiency and effectiveness.
Memory usage	S7	S4			Measures the amount of memory used by the tool.
Scalability	S34, S77		S50		How well the tool performs on subjects of different sizes.
Measuring time/-cost	S66	S66			Measure of how costly is the information needed by the technique (e.g. compiling tests, collecting coverage, training a model).
Other	**TCP**	**TCS**	**TSR**	**TSA**	**Description**
Applicability/Gen	S60	S69			The variety of SUTs upon which the tool can be applied.
Diagnosability	S46	S46			Cost of diagnosing a fault upon detection.

Table 3.6: Effectiveness, Efficiency and Other Metrics

out of which 24 were satisfied with the chosen metrics. We quote some of the answers received: "*The metrics directly influenced decisions of the industrial partner*" (respondent author #16); Respondent author #8 stated that "*[the] metrics were at the heart of the approach*" and that the provided metrics were "*always perceived as necessity by developers to support them in their work*"; "*The technique was put in practice for subsequent release and the metrics were useful and effective*" (author #17); Respondent author #23 answered that "*the metrics presented in the paper were critical for adoption and to measure ongoing improvements*"; "*they [the metrics] were relevant - they were also collected in the same environment in which the technique ended up being used*" (author #19).

Out of the three divergent responses, one suggested that the metrics were not a problem, but the dataset they used for the experiments was too small to provide meaningful evidence (author #42). Curiously, the remaining two complement each other. Author #45 said that they proposed a new metric, which is believed to be relevant but has not been experimented in practice yet; while author #25 claimed their own choice of metrics was not relevant to applicability, and is considering using the same metric proposed by #45.

After analyzing all the answers to this question, two very interesting things emerged: 1) One author (#14) reflected that although the metrics used were relevant at the time, looking back in retrospect other relevant metrics should have been used — "*Now, 7 years later, we have realized that some metrics were not included that should have been included*". The author was referring to the use of a metric for test case diversity as this could have helped them to tune the approach to avoid putting together many test cases targeting the same functionalities. This reinforces the importance of following up the adoption of a proposed approach in its application environment: even if we strive to anticipate all the possible uses of a proposed approach, observing its adoption in a real industrial context may reveal details and needs that were not captured while the approach was being conceived. 2) Two respondent authors (#23 and #36) reported that their approaches were evaluated with some additional metrics relevant to industry — "*the company has also developed their own metrics*" (respondent author #36) — that were not reported in their papers. The answers do not make it clear if the metrics were omitted because the measurements were not available at the time the paper was published or if they were omitted on purpose (e.g., because they could reveal sensitive company data).

ID	Ind. Mot.	Ind. Eval.	Ind. Auth.	Prac. Feed.	Avail. Tool	In Practice	ID	Ind. Mot.	Ind. Eval.	Ind. Auth.	Prac. Feed.	Avail. Tool	In Practice	ID	Ind. Mot.	Ind. Eval.	Ind. Auth.	Prac. Feed.	Avail. Tool	In Practice	ID	Ind. Mot.	Ind. Eval.	Ind. Auth.	Prac. Feed.	Avail. Tool	In Practice
S1	●	●	●	○	○	○	S21	●	●	●	○	○	○	S41	●	●	●	●	○	●	S61	●	○	○	○	◐	○
S2	●	○	○	○	○	○	S22	●	●	○	●	○	◐	S42	●	○	○	○	○	○	S62	●	○	○	○	●	○
S3	●	○	○	○	○	○	S23	●	○	○	○	◐	○	S43	●	○	●	○	○	○	S63	●	●	○	○	○	○
S4	○	●	○	○	●	○	S24	●	●	●	●	○	●	S44	●	●	●	○	○	○	S64	●	●	●	○	●	○
S5	●	○	○	○	●	○	S25	○	●	●	○	●	○	S45	●	●	●	●	◐	●	S65	●	○	○	○	○	○
S6	●	●	●	○	○	○	S26	●	●	●	○	●	○	S46	●	●	●	○	●	●	S66	●	●	●	●	●	●
S7	●	●	●	○	○	◐	S27	●	○	●	○	○	○	S47	●	●	●	○	○	●	S67	●	○	○	○	○	○
S8	●	●	●	○	○	○	S28	●	●	○	○	○	○	S48	●	●	○	○	○	○	S68	●	●	●	●	○	●
S9	●	●	●	○	○	○	S29	●	●	○	○	○	○	S49	●	●	●	○	○	○	S69	●	●	○	○	○	○
S10	●	●	○	○	○	○	S30	●	●	○	○	●	●	S50	●	○	○	○	●	○	S70	●	●	○	○	●	○
S11	●	●	●	●	○	●	S31	●	○	○	○	○	○	S51	●	●	●	○	○	◐	S71	●	●	●	○	●	○
S12	●	●	○	●	○	◐	S32	●	○	○	○	○	○	S52	●	●	●	○	○	●	S72	●	●	●	●	●	◐
S13	●	○	○	○	●	○	S33	●	○	○	○	●	○	S53	●	●	○	○	○	○	S73	●	●	○	○	○	○
S14	●	●	●	●	○	●	S34	●	○	○	○	●	○	S54	●	○	○	●	○	○	S74	●	○	○	○	●	○
S15	○	●	●	○	○	◐	S35	●	○	○	○	○	○	S55	○	○	●	○	○	○	S75	●	●	●	○	○	○
S16	●	●	●	○	○	○	S36	●	●	○	○	●	●	S56	●	●	○	●	○	○	S76	●	●	●	○	○	○
S17	●	○	●	●	○	●	S37	●	●	●	○	●	◐	S57	○	○	○	●	○	○	S77	●	○	○	○	●	○
S18	●	●	○	○	○	○	S38	●	●	○	○	○	○	S58	●	○	○	○	○	○	S78	●	●	○	○	◐	○
S19	●	●	●	●	○	●	S39	●	○	○	○	○	○	S59	●	●	○	○	◐	○	S79	●	○	○	○	●	○
S20	●	●	●	○	○	●	S40	●	○	●	●	●	●	S60	●	○	○	○	○	○							

Ind. Mot.: Industrial Motivation. **Ind. Eval.**: Industrial Evaluation. **Ind. Auth.**: Industrial Author(s). **Prac. Feed.**: Practitioner Feedback. **Avail. Tool**: Available Tool. **In Practice**: Put into Practice. A half-filled circle indicates a partially satisfied property. For example, a paper that provides its dataset but not its source code, or one that has some indication of having been implemented without explicitly stating so.

TABLE 3.7: Relevance properties found in the papers.

Summary of RQ3.1. The data reported in the figures show what are the most common approaches and metrics according to the objective of the RT techniques. For example, we see that TCP often relies on history-based and similarity-based approaches and uses APFD for evaluation, while TCS is usually change-based with a focus on the number of selected tests. We can also see that some overlap occurs and there are authors who choose unconventional but potentially promising combinations of techniques and metrics. From the author responses we received, it appears that many authors are satisfied with their selection of metrics but a few indicate that more were discovered in the process of implementing the tool with their industrial partner.

3.4.2 RQ3.2: Applicability Concerns in Regression Testing Research

To answer this research question, we look carefully at the applicability concerns extracted according to Table 3.4. The full mapping of the papers with the properties they satisfy is available in Table 3.7. It is worth observing that our conclusions here, as well as in the next section, are only relative to the set of primary studies that we retrieved; we cannot exclude the possibility that works that we did not select could eventually find application in practice. For instance, a paper with no obvious practical motivation could be the theoretical foundation for a tool later adopted by practitioners.

Most of the selected papers satisfy the properties of having a clear industrial motivation: out of the 79 papers, only five [S4, S15, S25, S57, S58] did not have a clear IR&A motivation. Regarding evaluation, 50 of the papers contained experiments on industrial (or industrial-scale) software. In other words, it is quite clear that IR&A is frequently a concern that motivates researchers to develop novel RT techniques. While providing adequate experimentation and evaluation to these techniques can be a tough challenge, it is one that researchers are indeed attempting to address.

Out of the 74 papers with relevant evaluation, 44 perform experiments with the direct collaboration of an interested partner — in most cases a corporation, in one case a government department (S30), indicating that such collaborations can play an important role in improving the relevance of experiments. Curiously, there are also four papers that have industrial collaborations, but the experiments are not performed with software from that partner [S17, S27, S40, S55]. Finally, there is one paper with an industrial partner but the objective of the work was not to develop a tool, so there are no experiments (S54).

In our retrieved literature, the industrial background of the authors is significant in a few ways. The papers with primarily industrial authors are the most likely ones to be relevant in practice, because these are generally designed with the application on a specific software product in mind; these papers usually provide insight into the testing workflow at large companies and share the lessons learned from applying a certain technique to a specific scenario. Examples include S47 with Facebook; S49 with Google and S51 with Microsoft. There are also some cases of companies whose main product is not software, but software is an important part of their products (e.g., transportation manufacturers, as S6 with BMW).

Papers with a mix of industrial and academic authors also represent good progress in enhancing industry-academia collaborations, such as the collaborations between University of Texas at Austin and Microsoft [S26, S37] or between the Federal University of Pernambuco and Motorola [S14, S52].

Finally, we want to highlight the papers that have tools available online. This is important for replicability and ease of access, but is still lacking in many publications. To facilitate comparisons by other researchers and simplify experimentation by software developers, it is fundamental that a version of the technique exists, either in binary or source code format. Only 22 of the surveyed papers made their tool available in some form (usually source code repository), making it improbable that any of the other tools were used by practitioners without direct contact with their developers. Notably, there appears to be a change in this trend: between 2016 and 2020, only 14 papers had any sort of replication package or tool available. In 2021 and 2022 (up until July), we found 8 papers satisfying this criteria. The likely explanation to this is that noteworthy Software Engineering conferences have given more value to easily-replicable research in recent years and this has caused authors to make it a priority. However, there are some cases where the code is made available with little to no documentation or explanation of how it works; on the bright side, there are also examples that stood out for having clear and detailed steps on how to use the code and replicate the experiments. Among the e-mail responses from the authors, we received the source code repository URL for four additional papers, confirming that at least 26 papers have material available online — whenever possible, the relevant URLs can be found in our live repository (Chapter 7).

Out of the investigated URLs, only S4 and S33 provide clear usage instructions for arbitrary software projects; they are available as plug-ins for the Eclipse IDE and the Maven build system, respectively. S40 also mentions the tool is available as an Eclipse plug-in, but we were unable to find a URL pointing to it. The remaining papers provide their source code primarily for study replication, not necessarily intended for actual usage by developers, meaning that the tool is likely not sufficiently robust for practical usage beyond experiments. It also happens frequently that tools developed in conjunction with an industrial partner end up becoming proprietary software and cannot be easily distributed (e.g. S14, S16). Authors of 14 papers said in their responses that the code or the tool could not be shared, since the resulting software is completely or partially proprietary or confidential.

An issue we identified is regarding the programming languages of the SUTs targeted by the experiments. Figure 3.6 shows that there is a heavy bias towards Java, with 23 papers targeting software written in that language. On most of the papers focused on a specific language, it is not clear if the same approach would be easy to adapt and would produce

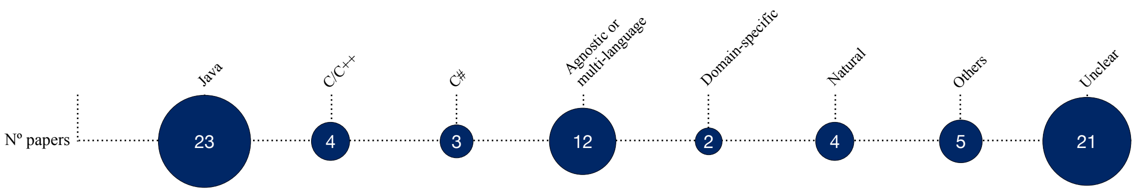


FIGURE 3.6: Distribution of the targeted programming languages.

equivalent results on software developed using other widely used languages. However, 12 papers target systems written in multiple languages, or explicitly state that the approach is language-agnostic, which highly increases its applicability. Unfortunately, it was not possible to identify the target language of 21 papers; this creates a substantial challenge for both the replicability of the experiments and applicability of the technique.

Summary of RQ3.2. Our survey shows that a large number of papers exhibit IR&A concerns in their motivations, and a smaller albeit still significant amount contains experiments at relevant scale. Most of the times, the techniques that are implemented into a software workflow are also papers that have authors from an industrial background. Unfortunately, few authors share their tools in a well-documented, open-source fashion, which hampers both future researchers, who wish to compare their solutions against the state-of-the-art, and practitioners, who might want to see how existing RT tools can help their software.

3.4.3 RQ3.3: Evidences of Real-world Application of Regression Testing Techniques

Our study is motivated by the concern that there is potentially valuable technology being proposed in academia that does not always make its way into usage in industry. The difference between the state-of-the-art techniques proposed in academia and the ones actually used in real-world software is what we call the *academia-industry technology transfer gap*. Expressing concerns over IR&A of RT techniques is an important step towards awareness of the gap, although not sufficient *per se* to solve the problem of actually putting these techniques into practice. The focus of this section is to discover if and how much evidence exists of techniques developed by the research community being adopted by real-world software development. As previously stated, there might be studies that have been put into practice, but escaped our review because they were not

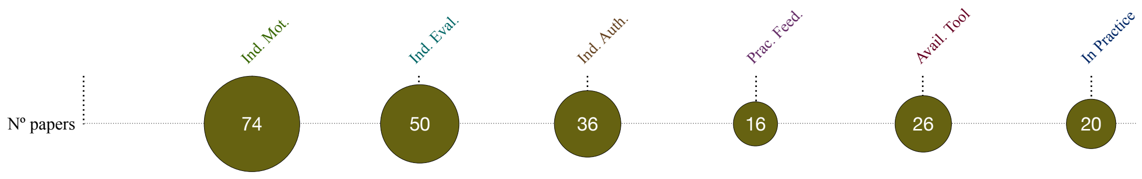


FIGURE 3.7: Quantitative analysis of the satisfied criteria.

explicitly motivated by IR&A; we hope that, in the future, our live repository solution will eventually find them and potentially widen the conclusions described here.

Table 3.7 contains only data extracted from the papers themselves; since the author responses are anonymous, we cannot map them directly to the table. Thus, Figure 3.7 displays the total number of papers that satisfy each of our applicability criteria, including updates from the author responses. In other words, we consider the author response if it updates the information retrieved from the paper; otherwise the data extracted from the paper remains.

Regarding the adoption of the proposed approaches, Table 3.7 shows that 16 out of the 79 selected papers explicitly state that the proposal is applied with a partner, or suggest that implementation was ongoing at the time of publication, out of which six are confirmed to still be in use by their authors, while four say it fell out of use (the remaining six did not respond, so we assume no change). Eight other authors claim their approach was implemented after publication, so the count in Figure 3.7 is 20 (16-4+8).

We can observe that having a practitioner as a co-author helps to provide a direct line from the founding theory of the technique to its application in practice: indeed, 14 of these 20 papers have at least one author from industry. This is not surprising, because such collaborations often originate directly from a need expressed by the practitioners.

However, we also see that only 8 out of those 20 papers featured feedback from the practitioners who actually used the developed tool. That is, although the tool was incorporated into the production workflow, in many cases an assessment of long-term benefits and acceptance by its users is either not done or not reported. Ultimately, the authors were our best chance of understanding the story behind each tool, revealing whether it is still being used by a partner and the reasons it might have fallen out of use.

From the respondent authors, we received six confirmations that the tool continues to be in use by their industrial partner in some form, e.g. "*The tool was implemented at* *a company [...] and it is still in use at the company [with significant changes].*" from respondent author #14. Authors of another two papers stated that the technique is undergoing an implementation process at the time of the response. Author #37 claims that their work on a newer paper is seeing adoption by an industrial partner; however, at the time of writing, that paper remains in pre-print and cannot be formally included in this review.

Interestingly, eight authors say that the tool was successfully incorporated into an industrial partner's development cycle after the publication of the paper: "*the technique has been adapted and embedded into a random data selection tool by the [company]'s testing team, for purposes including but not limited to regression testing.*" (author #36); "*the [technique] has been in use at [company since roughly the date of publication. [It] is used to run relevant test cases for every code review in [company]'s main code repository.*" (author #23). However, the details are not always known to them: "*We were told it was put into practice but we were not given any information, due to confidentiality rules.*" (author #44).

To the extent of the authors' knowledge, 12 papers were never put in practice, although some say there was a discussion to do so at some point. From author #35: "*We discussed the possibility of conducting a research visit at one of the corporation branches to experiment with the technique* in vivo*, but in the end it did not go through.*"

Authors of ten papers (out of which four were tagged as implemented in Table 3.7) said that the tool saw usage but fell out of use after a few years; an additional three claimed some sort of attempt, but the current status is unknown. What this means is that, even if a technique is incorporated into a software, a lot of work must still be done to ensure that the approach remains viable in a longer term. Some challenges mentioned by these authors include:

The tool became outdated and it was not updated to remain relevant "*It was implemented in an industrial setting, but this work is several years old and has to be evolved to stay relevant for business.*" (author #20). This can be either due to a technical issue, e.g. the tool was designed for an older version of a programming language or platform and would require some effort to be updated and be used on newer software, or because the tool does not consider newer requirements of its subject software.

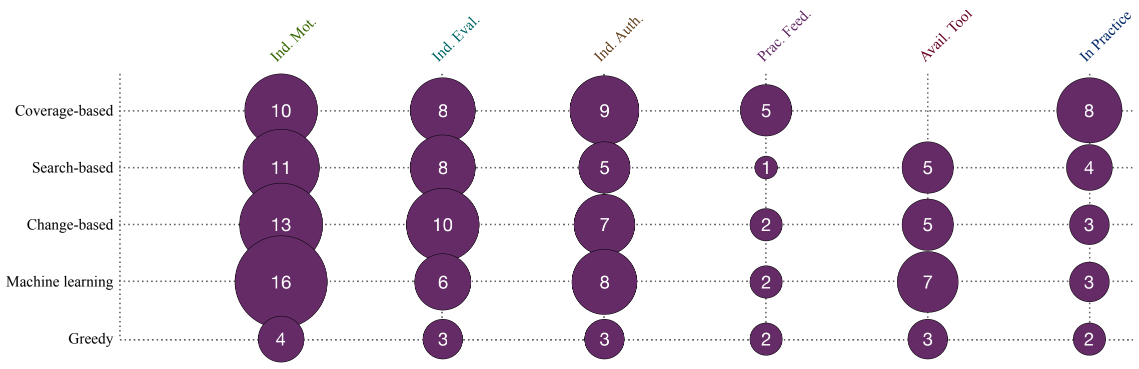


FIGURE 3.8: Mapping of approaches and techniques that have seen practical application on at least 2 papers.

The authors noticed that adapting an academic prototype into an industry-strength tool required more time and budget than the project permitted "*There is a gap between developing a research prototype and an industrial-strength tool. Evolving research prototypes towards industry-ready tools was beyond the project budget.*"(author #8) It can happen that a technique seems promising in initial experiments, but an enormous amount of work would be needed to actually incorporate it into a workflow. The technique might require data that is not currently being collected, or use some manual process for the evaluation that would need to be automated. The tool must also be verified for correctness and robustness before practical usage.

Authors lost contact with their partners and no longer follow development of the tool "*[The tool] was supported by [our partner]. We have no input if the tool has been used.*" (author #26) There are cases where the partnership does not continue after the publication of the paper or some other condition occurs. The industrial partner is likely free to continue using the developed tool, but the authors from the academic side are no longer part of its evolution and do not receive updates and feedback regarding the subsequent challenges and achievements.

The cost-benefit ratio was off "*We tried to use it within [our partner]. It seemed to work fine but the cost associated with the 1% bugs that were missed is too high*" (author #43). Even if a TCS technique detects 99% of bugs by running a very small set of tests, practitioners will be skeptical of using as a replacement for TestAll strategy. After all, although testing is a costly procedure, it is still much cheaper to detect an error during testing than after the software has been shipped to customers.

Figure 3.8 shows the relationship between the applicability criteria and the approaches that have seen real-world usage. The figure shows approaches with at least two papers put into practice[7]. Unsurprisingly, the most common information-type and algorithm-type approaches are the ones that see the most real-world usage. Coverage-based approaches dominate the implementations of techniques, despite previous concerns regarding the cost of measuring coverage [63]; although time-consuming, coverage measurements are easy to obtain in most programming languages. Conversely, there are 16 papers proposing machine learning approaches, but only three were implemented, likely because machine learning models are only as good as the data they are fed; often, obtaining data of enough volume and quality is more difficult than implementing the method itself.

From the practitioners' point of view, one possible source of information is grey literature — that is, material produced by experts and published without peer-review. However, this data is decentralized and unstructured, making it difficult to locate useful information. We did find one example: Netflix has a post on their blog [71] describing a system they developed inspired by S25. This indicates that grey literature might be worthy of investigation, but such an effort would fall beyond the scope of the current study.

To provide some insight into the state of practice, we surveyed 23 practitioners who are involved with software development and/or testing at their workplace. 60% of respondents claim they do not know about RT tools that originated from research, which corroborates the well-known lack of communication. 35% say they use or have used a tool to aid RT; however most of these claim the tool was developed specifically for their needs, so it is not clear that their origins can be traced back to Software Engineering research.

Summary of RQ3.3. From the papers and the responses we received, we have evidence that 20 papers propose techniques that are still being used in practice. It is a relatively small number, but it shows that RT research can have concrete positive impact on real-world software development. Unfortunately, many of the techniques that are implemented fall out of use after some time, as an ongoing effort is needed to motivate their usage and keep the tool relevant and updated. There is a hint of evidence stemming from grey literature, although practitioners themselves, when surveyed, mostly claim to be unaware of RT techniques originated in academia.

[7]Constraint-based, graph-based, similarity-based, trace-based, manual classification, cost-aware and history-based approaches have one paper each implemented in practice.

3.5 Threats to Validity

Construct validity Despite our efforts to comprehensively find all primary studies that meet our selection criteria, we might have missed some. To mitigate this threat, we performed a systematic search over five broad digital libraries and complemented the search with a snowballing cycle and a check with authors of all found studies, who in fact suggested a few additional entries.

As usual for this kind of study, our selection of papers was performed through queries, followed by manual filtering. To diminish potential bias of the latter step, the filtering process was systematically reviewed and agreed upon among all the three authors.

Internal validity The internal validity of this study is strongly dependent on the three research questions that guided all our analysis as well as the data extraction form we built. We took great care in ensuring that they properly reflect our objectives, although it is unavoidable that, by formulating different questions or using other data extraction forms, we could have obtained other results. We might also have overlooked or misinterpreted some important information or arguments in the primary studies, beyond our best efforts and accuracy in the full reading of all selected papers. To mitigate such threats we provide all extracted data in traceable format, highlighting the main points we extracted from each primary study. Furthermore, the responses we received directly from authors often provide additional context that reduce the risk of misinterpretation. That said, we cannot make the full responses available due to non-disclosure requests from some authors.

Conclusion validity The conclusions we drew in terms of the information we summarize from the primary studies, the detected challenges we discuss in the above section and the recommendations we formulate in the conclusions might have been influenced by our background, and other authors might have reached different conclusions. Such potential bias is unavoidable in this type of study, however we tried to mitigate it by aiming at full consensus of all authors behind each conclusion. Furthermore, by documenting in detail the data extraction process, we ensure a fully transparent study that can be verified and replicated. The survey sent to practitioners helps to validate our conclusions. Although the sample of 23 responses is very small, it shows a degree of alignment among people working in six different countries. A convenience sample was used to distribute the survey; thus, the practitioners we reached are more likely to have some contact with ongoing research. To avoid excessive bias in that direction, we did not contact members of industry who are known to regularly publish in Software Engineering events.

External validity We do not make any claim of validity of our conclusions beyond the 79 papers analyzed. As more primary studies are published, they should be read and analyzed on their own, and our conclusions should be revised accordingly. In consideration of this threat, in the aim of ensuring validity even in future, we are committed to keep the live repository up-to-date, taking into account the community inputs. Moreover, we believe that the framework we developed consisting of the three research questions, the data extraction form and the structured tables for summarizing the approaches and the metrics could be still applicable also by other external authors.

Chapter 4

Test Suite Orchestration

We understand *test suite orchestration* as a series of steps that can be performed before the execution of a test suite, with the objective of improving the effectiveness and efficiency of the suite. This can have an impact in the moments that immediately follow the orchestration, as well as in the long-term evolution, health and usefulness of the suite, as it adapts to an SUT that is also continuously evolving.

As a step towards the goal of fully automated orchestration of test suites that can be useful in real-world software, we focus here on regression testing techniques extracted from the existing literature that have been conceived for practical relevance and scalability. Specifically, as a representative TCS approach we adopt Ekstazi [48] while for TCP we use FAST [99]. The criteria used for selecting these tools were: their cost-effectiveness and simplicity of application; their availability as open-source programs; finally, also for convenience as authors of both tools were also involved with the development of the current study.

Concerning TCS, in an empirical study conducted in 2014 [49], the authors observed that many techniques were not adopted in practice and developers mostly continued to perform manual selection of test cases. Motivated by this study, Gligoric et al. [48] proposed Ekstazi, a lightweight TCS technique that leverages file dependencies. Besides the original paper on Ekstazi, several follow-up studies showed the benefit of file-based selection over other approaches [76, 154].

Concerning TCP, in a recent study Miranda et al. [99] showed that many existing techniques do not scale-up to large test suites. They hence proposed the FAST approach that applies Locality-Sensitive Hashing (LSH) techniques [79] for similarity-based prioritization. In the original work, the authors assess FAST against several competing TCP techniques, showing that it gives comparable effectiveness but with higher efficiency.

This work stems from the simple yet powerful idea of comparing these two approaches—TCS by Ekstazi and TCP by FAST—and possibly taking the advantages of each while overcoming their potential shortcomings. We make the following two observations:

- Ekstazi comes with no notion of test case priority: it assumes that all the selected test cases are run and makes no distinction about whether a failure is found by the first or the last executed test case;
- FAST reorders tests with the goal to detect failures early, but does not consider recent code changes, whereas we know from practice that these are related with failures, e.g., [73, 36].

By combining Ekstazi and FAST, we aim at developing a *practical* and *effective* approach to regression testing that we call *Fastazi*. This is meant to be practical because it combines two scalable techniques, and effective because it overcomes the above shortcomings of each. In particular, this combined approach aims to decrease *developer feedback time*, which is the time it takes for a developer to receive a test failure notification once testing begins.

Clearly Fastazi is one instance within a plethora of possible combinations of many existing TCS and TCP approaches, and further studies should be conducted to evaluate different combinations. Indeed, following the case made by Harman [57], research in combining multiple criteria in the context of one regression technique is very active, e.g., [40, 47]. Much less attention has been devoted so far to using multiple criteria while combining different regression techniques, which we see as an essential part of test suite orchestration. Di Nardo et al. [30] applied and assessed minimization, selection and prioritization techniques on a single industrial case study, but only considering coverage-based criteria; Silva et al. [130] proposed to combine prioritization and selection based on function criticality (assessed manually); Najafi et al. [102] evaluated selection and prioritization based on test execution history on a large industrial system; Shi et al. [128] combined and evaluated test reduction (based on coverage) and selection (based on changes). Fastazi is the first regression test orchestration approach that combines file-based TCS with similarity-based TCP.

We compared Ekstazi, FAST and their orchestration through Fastazi using a set of 12 projects (from the Defects4J repository [68]). Our results shows that for most subjects, executing a change-aware selection of test cases (in random ordering) detects the first failure faster than executing the whole prioritized suite (based on similarity). However, we also observed that adding FAST ordering on top of Ekstazi selection further improves effectiveness at negligible additional cost.

To conclude this study and provide a direction for future evolution of test orchestration strategies, we provide a discussion on methods of incorporating other TCS or TCP techniques, along with potential combinations with TSR and TSA approaches.

In summary, our contributions include:

- an empirical study comparing TCS against TCP, and their orchestration against each technique alone;
- the novel Fastazi approach to regression testing that combines filed-based TCS and similarity-based TCP;
- a replication package[1] including Fastazi implementation and all data from the study.

For practitioners our results signify not only a further confirmation of change-aware selection validity, but also the convenience of executing the selected test cases in prioritized order based on their similarity. In fact, using state-of-art scalable techniques as FAST over the selected test subset can help detect failures faster at virtually no cost. For researchers, this paper signifies the importance of studying regression techniques as an orchestration rather than individually, and opens up the floor for many potential experiments in which various TCS techniques are compared against, or combined with, various TCP techniques.

In Section 4.2 we provide a short summary of the TCS and TCP approaches that we compare and combine, while in Section 4.3 we present Fastazi. The study methodology is described in Section 4.4 and the results are discussed in Section 4.5. Finally, in Section 4.6 we draw brief conclusions derived from this study and discuss the possible next steps in the evolution of this work.

[1]Available at: https://doi.org/10.5281/zenodo.5851288

4.1 Research Questions

We evaluate Ekstazi against FAST, and their combination (Fastazi) against either of them, considering first their *effectiveness* in failure detection (RQ4.1). Then, based on the real example shown in Figure 4.1, we hypothesize that the potential gain in effectiveness of a combined approach could be better observed under a *limited test budget* (RQ4.2). Finally we also compare their *efficiency* (RQ4.3). Precisely, we formulate the following research questions:

RQ4.1 How do Ekstazi, FAST, and Fastazi compare in terms of effectiveness? For the scope of this study, the comparison between the respective effectiveness of the three approaches can be based on how quick they are in detecting the failures. As FAST uses the whole test suite, we know it will detect all regression failures as a retest-all technique. Also, Ekstazi is developed as a safe TCS technique, thus it should, as well, detect all failures found by retest-all. Consequently, Fastazi too detects all failures. Thus, we refine the above question into the following two sub-questions:

RQ4.1.1 Between Ekstazi and FAST, which tool detects failures running fewer tests? While both Ekstazi and FAST have been shown to be effective in failure detection, we do not know whether when a new project version is released, potential regression failures would be revealed earlier by selecting those test cases that are affected by the changes (and randomly ordered) or instead by prioritizing test cases based on their similarity.

RQ4.1.2 How does Fastazi compare against Ekstazi and FAST with respect to feedback time? It is unclear if, and by how much, a combination of both techniques would provide lower feedback time from a test suite. With this question, we aim to discover if the orchestration of TCS and TCP has a positive and substantive impact to the regression testing workflow.

RQ4.2 How does a limited testing budget affect the effectiveness of the three approaches? While in RQ4.1.1 and RQ4.1.2 we compare Ekstazi, FAST, and Fastazi without considering possible time constraints, with this RQ we aim at assessing whether, and how, testing under limited resources impacts each of the three approaches. This problem is similar to cost-bounded selection [24] (i.e., selecting test cases according to a predetermined budget), which can be a concern in large-scale industrial projects [36]. TCS and TCP each provide benefits when it is not possible to test 100% of the test suite in each execution, but they cannot assumed to be safe in these circumstances. Perhaps an orchestrated test suite would viable at even stricter testing budgets.

RQ4.3 How do Ekstazi, FAST and Fastazi compare in terms of time efficiency? With this question, we aim to discover what is the additional cost in terms of time required by either technique alone, and then by their orchestration. Inevitably, the orchestration increases total testing time, and we aim at assessing such drawback.

4.2 Background

4.2.1 Ekstazi

Ekstazi [48] is a *change-based* and *coarse-grained* approach to TCS. It works by collecting test case dependencies (i.e., set of used classes by each test case) during an initial run of the entire test suite, then by selecting the test cases based on the changes applied to those dependencies from one version of the software to another. In doing this Ekstazi applies a *file-level granularity*: any code changed within a file that is related to a test case will result in that test being selected. To compare two versions of a file, Ekstazi uses cyclic redundancy check (CRC). For example, consider a test t that invokes a function a. If a change is made to another function b located in the same file as a, t will be selected (as the CRC of the file changed).

The result of this approach is an over-approximation of the subset of selected tests. Although Ekstazi selects, on average, more tests than fine-grained TCS solutions (e.g., those that track dependencies on methods), the authors demonstrated that the actual selection time is much faster than the alternatives. Consequently, the total end-to-end time (i.e., time to select tests + time to execute selected tests) tends to be lower, even if more tests are selected.

We chose Ekstazi for our study for its efficiency and ease of use: Ekstazi is publicly available as a plug-in for various Java build systems. Furthermore, aiming eventually at an orchestration of TCS plus TCP with the objective of reducing feedback time, we considered that a prioritized test suite could mitigate the drawbacks of over-selecting test cases.

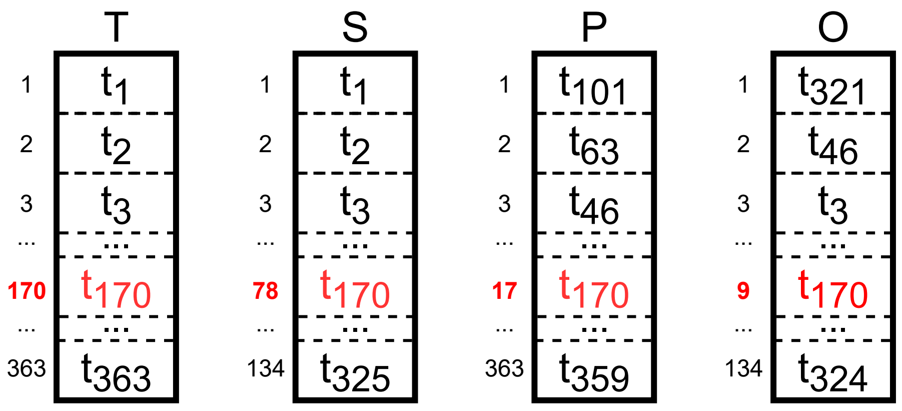
4.2.2 FAST

FAST [99] utilizes test source code as input for a *similarity-based algorithm* to prioritize the tests. Inspired by big data techniques, string representations of test cases are transformed using minhashing signatures, which are then ordered according to their similarity. The benefits of FAST are low overhead and scalability, which make it usable for large software projects. We chose it because of its low running times and relatively simple implementation.

FAST authors [99] examined several possible variations of it that trade off efficiency for accuracy when choosing the next test(s). These are all stochastic by nature; as the authors point out, if two test cases are ranked equally, the tie is solved randomly. In our experiments with FAST we observed that FAST-pw (which is one of the variations) produced consistently similar permutations when executed more than once with the same test suite. This was an expected result given that FAST-pw is designed to always select the test case that is the furthest away from the set of already-prioritized tests. It does so by computing the similarity between each candidate test and the set of already-prioritized tests in a pairwise fashion. Furthermore, FAST-pw was able to rank failing tests higher than other variations. Therefore, in this paper we consider the FAST-pw variant, and in the following we refer to it simply as FAST.

4.3 Fastazi

Many researchers have shown that TCS and TCP provide substantial benefits to regression testing [4, 69, 70, 122]: a good selection decreases the overall testing time, while a good prioritization allows for detecting failures faster. However the two concepts are not mutually exclusive, and an orchestration of both may provide even further improvements, e.g., [133, 36].



T: complete test suite; S: selection by Ekstazi;
P: prioritization by FAST; O: orchestration by Fastazi.

FIGURE 4.1: Sample outputs of Ekstazi, FAST and Fastazi

If a test suite T is selected *and* prioritized, both testing time and feedback time can be decreased. Recall that, as seen in Section 2.2.1, TCP can be defined as a function $P(T)$ that outputs a permutation of T for a given SUT. Furthermore, as detailed in Section 2.2.2, TCS can be seen as a function $S(T)$ that produces a subset of T considering the differences between versions of an SUT[2]. Then, the goal of an approach that orchestrates TCS and TCP is to generate another function, $O(T)$, whose output is smaller than T and ordered to speed up failures detection. When discussing possible ways of orchestrating TCS and TCP, two approaches stand out.

Parallel execution. One approach is to independently perform the prioritization and selection of the entire test suite, and then arranging the selected tests according to the ordering given by prioritization. This approach has the advantage of allowing parallel execution of $S(T)$ and $P(T)$ and merging their outputs, instead of having one depend upon the other. To combine the outputs, it is sufficient to go through the prioritized list of tests and remove the ones that are not included in the selection.

Sequential execution. Another possible approach to the idea is performing selection first and then prioritizing the output. The advantage of this approach is reducing the running time of the prioritization, which would focus on the tests impacted by the changes

[2]For simplicity of notation, in this section we omit the SUT and its subsequent versions as parameters to the S and P functions as, for the purposes of experimentation, they can be thought of as information inextricably linked to the test suite T.

and thus more likely to fail. However, this also means that the selection and prioritization steps cannot be performed simultaneously (although it is still possible to parallelize the preparation steps). Intuitively, it is not clear which option should be more effective or efficient than the other. Indeed, our experiments show that the effectiveness and efficiency of the parallel and sequential approaches are statistically equivalent (according to the same analysis detailed in Section 4.5). For lack of space, henceforth Fastazi results always refer to the sequential execution, while the results of the parallel combination are available in the replication package.

As an example, Figure 4.1 contains sample outputs from Ekstazi, FAST and Fastazi[3]. Colored red, t_{170} is the failing test within a test suite T of 363 test cases. In S, the output of Ekstazi, this test is found in the 78th position, because several tests were excluded during the selection, while in the output P of FAST, it is moved up to the 17th position. Finally, the output of Fastazi, O, which is selected and prioritized, promoted the test to the 9th position.

Algorithm 1 provides an abstract view of Ekstazi and FAST[4], and outlines how Fastazi works in practice. Ekstazi requires tests to be compiled before performing selection, while FAST needs the hash signature of each test before prioritizing the suite. These two steps are independent and can be performed in parallel (they are both abstracted by the function GetHashesAndModified). After that, Ekstazi can perform its selection normally, and FAST prioritizes the resulting list of tests.

It is important to observe that, while we utilize Ekstazi and FAST as representative implementations of TCS and TCP, the idea behind Fastazi could be attempted using different approaches. There are many proposed techniques that address TCS and TCP; combining different ones would inevitably result in changes to effectiveness and efficiency.

4.4 Experiments

[3]This example is based on results of the experiments on Chart v26. Actual names of test cases are omitted for clarity.

[4]For a complete understanding of Ekstazi and FAST, refer to [48, 99].

Algorithm 1 Ekstazi, FAST and Fastazi overview

```
 1: function GetHashesAndModified(files $F$)
 2:     $M, C \leftarrow$ ExistingHashes(F)
 3:     $M' \leftarrow \emptyset$                    ▷ Minhashes for FAST and CRC for Ekstazi
 4:     $F' \leftarrow \emptyset$
 5:     for $f \in F$ do
 6:         $M'[f] \leftarrow$ ComputeHashes($f$)
 7:         if $M[f] \neq M'[f]$ then
 8:             Append($F'$, $f$)
 9:     return $M', F'$                          ▷ Updated hashes and modified files.
10: function Ekstazi(test suite $T$, files $F$)
11:     $S \leftarrow \emptyset$
12:     for $f \in F$ do
13:         for $t \in T$ do
14:             if TestDependsOn($t$, $f$) then
15:                 Append($S$, $t$)
16:     return $S$                               ▷ A selected test suite.
17: function FAST(test suite $T$, hashes $M$)
18:     $P \leftarrow \emptyset$
19:     while $|P| \neq |T|$ do
20:         $t \leftarrow$ PickNextTest($T$, $P$, $M$) ▷ Pick the test that is furthest away from the
    so-far-ordered tests $P$ based on $M$.
21:         $P \leftarrow$ Append($P$, $t$)
22:     return $P$                               ▷ A prioritized test suite.
23: function Fastazi(test suite $T$, files in the project $F$)
24:     Compile($T$) • $M, F \leftarrow$ GetHashesAndModified($F$) ▷ Compiles the test suite
    using the build system and, in parallel, computes hashes and detects modified files.
25:     $S \leftarrow$Ekstazi($T$, $F$)
26:     $P \leftarrow$FAST($S$, $M$)
27:     return $P$                               ▷ A selected and prioritized test suite.
```

4.4.1 Evaluation Metrics

The primary objective of TCS is to reduce the total number of tests executed per run, while TCP, on the other hand, has the goal of detecting failures quickly and reduce the feedback time of the test suite. Thus, the metric for an orchestration should somehow measure both of these objectives.

For RQ4.1, we utilize a metric called *Time To First Failure* (TTFF) [149]. Given a test suite T, its TTFF indicates the position of the first test to detect a failure. A low TTFF indicates that the test suite provides quick feedback. TTFF is a useful metric to evaluate both TCS and TCP, because it simultaneously encourages a tight selection of truly relevant tests and a prioritization that puts a failing test at the top of the list.

However, since the output S of TCS is a subset of T, its size might be smaller than the output P of TCP. Therefore, for fairness, all TTFF results in this paper are normalized according to size of T. For example, if $|T| = |P| = 1000$, $|S| = 100$ and a failing test is in the 100th position of P but the the 50th position of S, then $TTFF(P) = 0.10$ and $TTFF(S) = 0.05$.

We also utilize *Average Percentage of Faults Detected* (APFD), the most popular metric for evaluating TCP solutions [70]. It is not designed to evaluate TCS and thus may not provide a fair comparison for Ekstazi; however, as previously explained, we assess here effectiveness in terms of how fast failures are detected by the compared techniques, and for this APFD provides an intuitive, well known assessment.

Regarding RQ4.2, when considering a limited testing budget, we use the output from RQ4.1 and create versions of the suites that are cut off at certain points, according to the budget restriction. This data is analyzed in two ways: first, we observe, for each version of each subject, the proportion of the 30 variations that were able to detect the failure or not. Then, we also reduce this number into a binary form: 1 if the test suite detects the failure in all of its 30 variations, and 0 otherwise. This has the effect of punishing suites that are somehow inconsistent, rewarding those that catch the failure every time, since it can be important that an approach is consistent and reliable.

Finally, for RQ4.3, we measure the time taken to execute the discrete steps of the approach. For this, we use the GNU time utility (user+sys CPU time) to measure each step of the experiment individually, allowing us to understand where are the bottlenecks of the approaches.

4.4.2 Experiment Design and Execution

The goal of the experiment is to compare four possible arrangements of the test suite: the tests selected by Ekstazi; the test suite prioritized by FAST; the orchestration of both with Fastazi; and a random ordering of the test suite to provide a base case. Considering that both Ekstazi and FAST have been previously compared to several competing TCS and TCP approaches [76, 154, 99], we deemed it not necessary to add further alternatives in a direct comparison between the two tools.

Subject	# Versions	Min. # Tests	Max. # Tests
Chart	26	303	363
Cli	30	24	85
Closure	168	236	258
Codec	8	34	52
Collections	4	157	165
Compress	39	44	133
Gson	18	77	119
Jsoup	93	12	39
JxPath	4	27	33
Lang	28	87	178
Math	100	137	821
Time	23	121	123
Total	541	n/a	n/a

Min. # Tests and Max. # Tests show the smallest and largest test suites, respectively, among all versions of a certain subject.

Table 4.1: Subjects Used in the Evaluation.

We utilize as subjects 12 projects available as part of the Defects4J repository [68] that contains multiple versions of Java-based open-source software projects of different sizes. Each version is comprised of one commit containing a bug, the commit that fixed the bug, and metadata such as the files related to the bug, and which tests would detect it. Table 4.1 shows basic properties about each project used in our evaluation. For each project, we show the number of versions used and minimum and maximum number of test cases (across versions). A few versions were skipped, either because their bugs are listed as deprecated by Defects4J, or because we ran into compilation issues for them (e.g., due to Java version incompatibility).

We use Ekstazi version 5.3, available on the project's website[5], as a plug-in for the Maven and Ant build systems. A script is used to automatically incorporate the Ekstazi task into a project's build script, allowing us to easily perform test selection over multiple versions of different subjects.

In the case of FAST, we use the source code from the replication package of the original paper[6]. This code was modified by us with two purposes. The first was to make FAST version-aware by storing the hash signatures of test cases between versions so they do not

[5]http://ekstazi.org

[6]https://github.com/icse18-FAST/FAST

need to be re-computed unless there is a modification. This is important because computing the hashes is the most time-consuming part of FAST, so storing these representations for unchanged tests greatly reduces overhead after an initial execution. In addition, it was updated to guarantee that the input and output of both Ekstazi and FAST are in the same format.

Fastazi was not incorporated into the build system, but its results can be easily generated by using the output of Ekstazi as input for Fastazi, as shown in Algorithm 1. Observe that change-based TCS provides no benefit in the initial version of a project, since there are no changes to be detected; thus the first output of Fastazi, for each experiment subject, is identical to using FAST in isolation.

To collect the metrics, we did not actually execute the test suites given by each approach. First we collect the outputs of the approaches as text files containing lists of tests and then we calculate the metrics according to the position of the failing test(s) (ground truth given by Defects4J).

When measuring TTFF, the default order of test executions could have a large impact on (unprioritized) test suites; hence, for fairness we shuffled the output of Ekstazi 30 times and reported the average of these repetitions. Similarly, to account for the nondeterministic behavior of FAST, Fastazi and random, their outputs are also generated 30 times to reduce any potential noise in the data[7].

The experiments were executed in a Docker container running Ubuntu 20.04 LTS, using Java OpenJDK 1.8.0, Apache Maven 3.6.3, and Apache Ant 1.10.7. On all the projects, JUnit version was set to 4.12. The host computer was running macOS 11.0.1 on a 6-core Intel Core i7 processor, with 32GB RAM and SSD storage.

4.5 Results

4.5.1 RQ4.1: Effectiveness

The answer to RQ4.1 contains two parts: first, we compare the effectiveness of Ekstazi and FAST against each other (RQ4.1.1), and then, we assess whether orchestration TCS

[7]We experimented with values between 10 and 50 and found that 30 provided a good amount of data without severely impacting the running time.

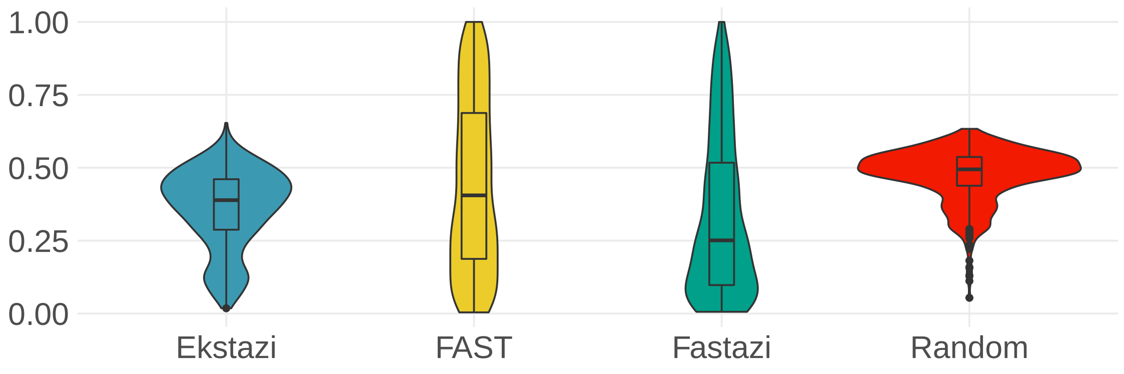


FIGURE 4.2: Normalized TTFF of different approaches

and TCP ultimately improves effectiveness (RQ4.1.2). For the sake of space we show the results for both subquestions within unified plots and tables.

The TTFF results are displayed as violin plots in Figure 4.2, in which each version of each subject is one data point (totaling 541). The violin plots display, in addition to the median and interquartile ranges, the full distribution of the data, which allows us to identify the different peaks in a distribution. For the TTFF metric, the lower the result, the better.

The visual assessment of the data shows us that the median TTFF achieved by Ekstazi and FAST are both close to 45% (the two leftmost plots in Figure 4.2), although there is a large difference in the distribution of the results. This can be explained in part by the experiment design – since Ekstazi's TTFF is an average of 30 permutations of S, the value tends to be close to the center. Indeed, we can see that the median for Random is very close to 50%, while Ekstazi is lower than that because S is frequently smaller than T.

When adding Fastazi to the comparison, we can see that its median TTFF is much lower, at around 25%, which is slightly over half the medians of Ekstazi and FAST. Both FAST and Fastazi can, in some instances, produce a TTFF close to 100%, meaning that the failing test is found at the very end of the test suite. In the case of FAST, this is explained by the fact that similarity-based TCP can occasionally produce poor results if there are multiple similar test cases out of which only one reveals the failure.

With Fastazi, this happens less frequently; when it does, it is caused by performance of both Ekstazi (selecting nearly 100% of the test suite) and FAST (ranking the failing test low) in specific subject versions.

After the visual inspection we proceeded with the statistical analysis of the data. As we could not assume our data to be normally distributed, we adopted a non-parametric statistical hypothesis test, the Kruskal-Wallis rank sum test[8]. We assessed at a significance level of 5% the null hypothesis that the differences in the TTFF values are not statistically significant. The observed differences in TTFF were statistically significant at least at the 95% confidence level (*p-value* $< 2.2e\text{-}16$).

Provided that significant differences were detected by the Kruskal-Wallis test we performed pairwise comparisons to determine which approaches are different[9]. The results are displayed in Section 4.5.1 (column *Group* for TTFF). If two approaches have different letters they are significantly different (with $\alpha = 0.05$). If, on the other hand, they share the same letter, the difference between their ranks is not statistically significant. The approach (or group of approaches) that yields the best performance is assigned to the group (a). Looking at the results in Section 4.5.1, we can tell that Fastazi is different from (better than) Ekstazi (b). Ekstazi, on its turn, is different from (better than) FAST (c), and all the approaches are different from (better than) Random (d).

Approach	TTFF			APFD		
	Med	*SD*	*Group*	*Med*	*SD*	*Group*
Fastazi	0.25	0.27	(a)	0.75	0.27	(a)
Ekstazi	0.39	0.14	(b)	0.62	0.14	(b)
FAST	0.41	0.29	(c)	0.60	0.29	(c)
Random	0.49	0.09	(d)	0.51	0.09	(d)

Med is the median, *SD* is the standard deviation, and *Group* displays the result for the pairwise comparisons after the Kruskal-Wallis test.

Table 4.2: TTFF and APFD for the different approaches.

To understand the effect of choosing one technique over another on the effectiveness of the test suite, we measured the effect size using the Vargha and Delaney $\hat{A}_{12}$ measure [139], which tells us the probability of an observation from one group being larger than an observation from the other group. The results are displayed in Section 4.5.1. For interpreting the results, the $\hat{A}_{12}$ measure ranges from 0 to 1, and when the measure is exactly 0.5 the two techniques (in the column name) have equal performance. When $\hat{A}_{12} > 0.5$, the first technique outperforms the second, and when $\hat{A}_{12} < 0.5$, the second technique outperforms the first. Vargha and Delaney suggest that the effect size is *small* if the measure is over

[8]We used kruskal.test() from the Stats package in R.

[9]A significant Kruskal-Wallis test indicates that there is a significant difference between approaches, but does not identify which pairs of approaches are different.

Subject	Fastazi vs Random	Fastazi vs FAST	Fastazi vs Ekstazi	Ekstazi vs FAST
Chart	0.82 (L)	0.79 (L)	0.57 (S)	0.78 (L)
Cli	0.85 (L)	0.56 (S)	0.81 (L)	0.23 (L)
Closure	0.62 (S)	0.55 (N)	0.56 (S)	0.51 (N)
Codec	0.88 (L)	0.66 (M)	0.66 (M)	0.52 (N)
Collections	0.50 (N)	0.66 (M)	0.44 (N)	0.63 (S)
Compress	0.82 (L)	0.65 (M)	0.59 (S)	0.58 (S)
Gson	0.69 (M)	0.58 (S)	0.60 (S)	0.47 (N)
Jsoup	0.63 (S)	0.64 (M)	0.51 (N)	0.66 (M)
JxPath	0.50 (N)	0.56 (S)	0.44 (N)	0.57 (S)
Lang	0.66 (M)	0.63 (S)	0.65 (M)	0.58 (S)
Math	0.83 (L)	0.66 (M)	0.64 (M)	0.57 (S)
Time	0.60 (S)	0.61 (S)	0.52 (N)	0.61 (S)

L, M, S and N indicate large, medium, small and negligible effect size, respectively.

Table 4.3: Effect size per subject.

0.56, *medium* if over 0.64, and *large* if the measure is over 0.71. As an example, when comparing Fastazi against Random for the subject Chart, Fastazi outperforms Random with a *large* effect ($\hat{A}_{12} = 0.82$) on the testing effectiveness. We can see that Ekstazi generally outperforms FAST, most of the time with a negligible or small effect, but there are cases where FAST outperforms Ekstazi. Fastazi, on its turn, outperforms Ekstazi and FAST with a non-negligible effect in the vast majority of the cases (18 out 24). The effect of choosing Fastazi over Ekstazi or FAST on the test effectiveness is large or medium in 11 cases.

While TTFF captures how many test cases are required to reveal the first failures, the APFD metric measures the speed at which failures are revealed.

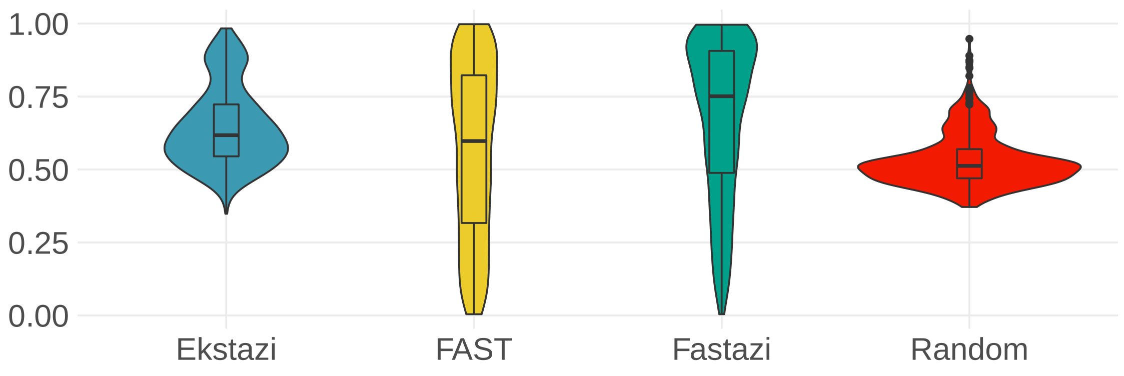


Figure 4.3: APFD of different approaches.

The observed APFD results are displayed as violin plots in Figure 4.3. For the APFD metric, the higher the better. Visual assessment of the results lead to the same conclusion as for TTFF: Ekstazi and FAST have similar medians, although FAST sometimes performs very poorly, while Fastazi has a higher median than both and mitigates most instances of poor performance from FAST. It is also visible that the peak of the distribution of Fastazi leans towards the highest possible values, while Ekstazi peaks at around 0.6.

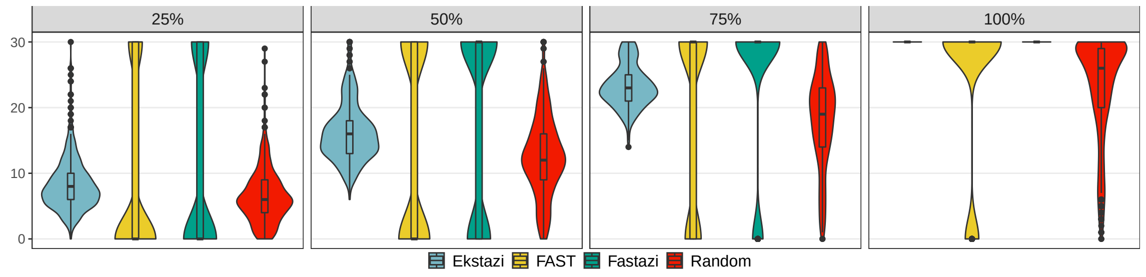
Statistical analysis results are reported in Section 4.5.1 (right side). We performed again the Kruskal-Wallis rank sum test, followed by the pairwise multiple comparisons. All results in Section 4.5.1 are statistically significant at the 5% significance level. Both the groups assigned to each approach and the results of the effect size analysis were the same as the ones observed for the TTFF metric.

Summary of RQ4.1. While statistically significant differences were observed for the comparison between Ekstazi and FAST, a further investigation of the effect size revealed that the effect of choosing Ekstazi over FAST is either small or negligible in almost all the cases. Fastazi, on the other hand, outperformed Ekstazi and FAST with a non-negligible effect in the vast majority of the cases, suggesting that adopting Fastazi can help improving the testing effectiveness.

4.5.2 RQ4.2: Effectiveness Under a Limited Budget

To answer RQ4.2, we proceeded with a detailed analysis of the impact of limiting the number of test cases with respect to those that would be run by Ekstazi. We investigated the impact on the failure detection capability of all the approaches when the testing budget is gradually reduced from 100% (no budget restrictions) to 25% of the test suite selected by Ekstazi, at steps of 25%. We discuss our findings first at a higher level, then with a more in-depth analysis of the results for each of the subjects considered in our study.

Figure 4.4 depicts the impact on failure detection capability on the different approaches. The results are grouped per budget (25% to 100%) and each approach is represented by a violin plot. For each version of each subject we counted how many times, out of the 30 repetitions (see Section 4.4.1) each approach would be able to reveal the failure under the different budget restrictions (the number of observation in each violin plot is thus the



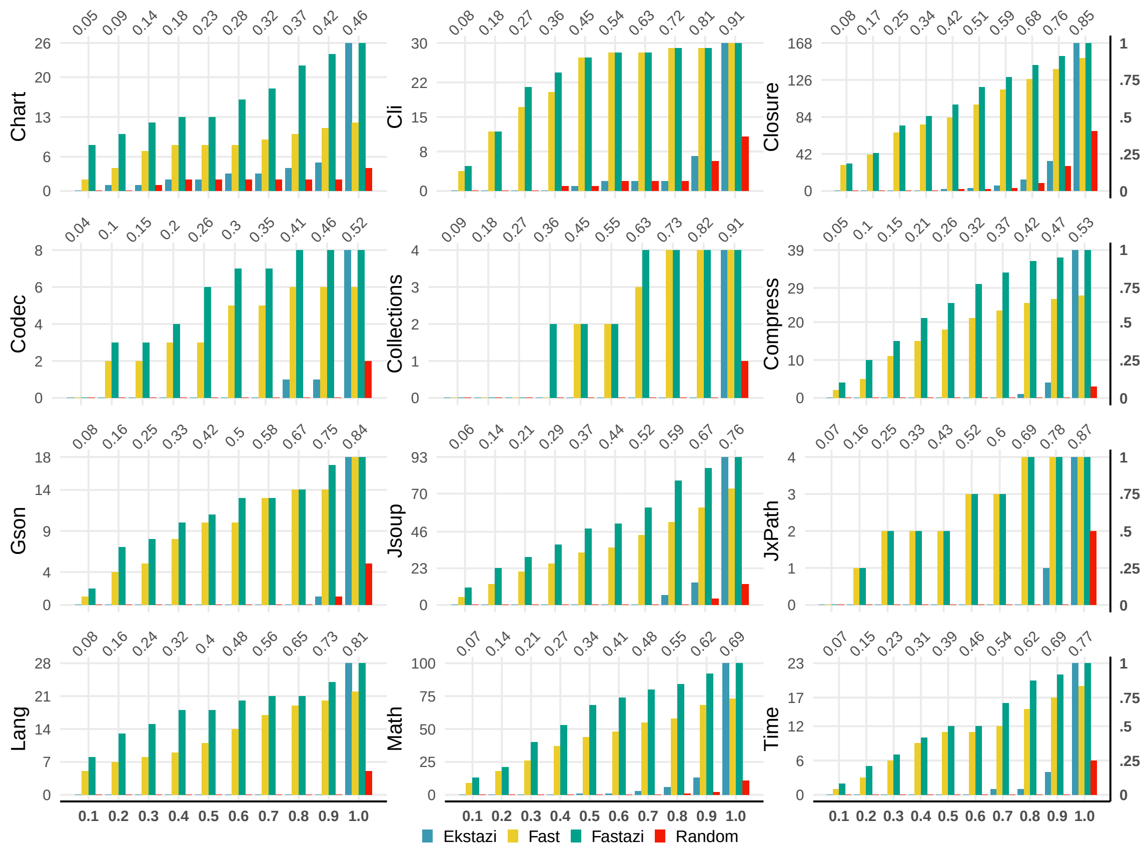
Each panel represents a different budget constraint (100% is defined as the percentage of the test size selected by Ekstazi). The vertical axis shows how many times, out of the 30 repetitions, each approach is able to reveal the failure.

FIGURE 4.4: Impact on failure detection capability in a budget-constrained scenario.

same as the total number of versions, i.e., 541). The vertical axis varies from 0 to 30, respectively the minimum and maximum number of times an approach could reveal the failure across the 30 repetitions. Notice that for this RQ it is not a concern whether the failure is revealed by the first or the last test case, as this was already answered by RQ4.1; the concern here is whether the failure is revealed.

We can draw several observations from Figure 4.4: *i*) the median number of times the random approach can reveal the failure decreases almost uniformly as the budget becomes stricter; *ii*) because Ekstazi is the result of Ekstazi selection with random ordering, the observed medians and distributions are always slightly better than random, but following a similar trend as the one observed for random; *iii*) Fastazi outperforms the other approaches up to a budget restriction of 50%; *iv*) for the more restrictive budget of 25% the median of Ekstazi and even random are better than those of the Fastazi approach. Looking at the shape of the violin plots, however, we can see that even with a lower median Fastazi appears to have more observations leaning towards the maximum possible value.

To better understand such a behavior we analyze the data again from a different perspective in Figure 4.5, in which we observe the impact on failure detection capability on a per subject basis. This time, however, instead of counting how many times the failure would be revealed across the 30 repetitions, we are interested in the cases where the approach would consistently reveal the failure across all the repetitions for a given version. In this way we do not reward the cases where an approach would be able to reveal a failure by pure chance. Each subject is represented by a grouped bar plot and the height of each bar represents the number of times the approach was able to consistently reveal the failure,



The vertical axes represent the number of failures revealed in absolute (left) and in relative terms (right), whereas the horizontal axes show the budgets w.r.t the number of tests selected by Ekstazi (bottom) and w.r.t the total number of tests in the subject's test suite (top).

FIGURE 4.5: Impact on failure detection capability grouped by subject and by budget.

both in absolute (left vertical axis) and in relative terms (right vertical axis). For example, the maximum value in the left vertical axis for Closure is 168, which is the number of versions we considered for that subject and, at the same time, the maximum number of failures that can be revealed (one per version).

The primary horizontal axis (bottom) represents the budgets, from 10% to 100%, whereas the secondary horizontal axis (top) shows what a given budget restriction would mean with regards to the whole test suite. This is important because the size of the test suite varies greatly across the subjects. For example, while a budget restriction of 50% for Collections means that 45% of the whole test suite is selected, only 23% of the whole test suite would be selected for Chart under the same budget restrictions (we recall that the budget restriction is calculated over the size of the test subset selected by Ekstazi).

By analyzing Figure 4.5 we can draw the following observations: *i*) with no budget restrictions (budget = 100%), Ekstazi and Fastazi were able to consistently reveal all the failures across the 30 repetitions; *ii*) for any other budget value below 100% Fastazi outperformed Ekstazi alone and FAST alone — in a very few cases FAST appears tied to Fastazi; *iii*) Ekstazi can consistently reveal some failures for almost all the budgets for Chart. For all the other subjects, it cannot reveal any failure for budgets restricted below 50%. For the particular cases of Collections and Lang, Ekstazi cannot reveal any failure consistently in the constrained budget scenario; *iv*) with the exception of Codec, Collections, and JxPath, Fastazi was able to consistently reveal some failures across the 30 repetitions for all the budgets, including the more restrictive budget of 10%.

Summary of RQ4.2: Without controlling for the differences across subjects, Fastazi exposes the best failure detection capability even under restricted budgets, except for under 25% reductions in which Ekstazi and even random appear to show better median values. However, when we look from a per subject perspective and reward the approaches that consistently reveal failures, Fastazi outperform Ekstazi alone (with random ordering) and FAST alone (without TCS) for all the budgets considered.

4.5.3 RQ4.3: Efficiency Comparison

To compare the time efficiency of Ekstazi, FAST, and Fastazi, we isolated the individual steps of each approach and measured the average time each step took, across the different versions of each subject program. In our measures, displayed in Section 4.5.3, the average build time (column 2) for each project was substantially longer than any cost added by Ekstazi, FAST, or Fastazi. This is an important observation because FAST can run its preparation phase (column 3), i.e, computing hashes of added/modified test cases, in parallel with the building process as it requires only test code. Fastazi takes advantage of this aspect to minimize the time overhead. Ekstazi, on the other hand, requires the code to be compiled before it can perform selection, so it cannot be run in parallel with the build.

Looking at the average execution times for FAST, Ekstazi, and Fastazi (the three rightmost columns in Section 4.5.3) the two main things we can observe are: *i*) overall, FAST

Project	Build	FAST (setup)	FAST (TCP)	Ekstazi (TCS)	Fastazi (TCS + TCP)
Chart	4167	1105	112	3224	3259 (35)
Cli	2997	165	10	137	147 (10)
Closure	6627	1403	83	1571	1637 (66)
Codec	4581	551	6	163	166 (3)
Collections	6627	1043	29	237	259 (22)
Compress	4986	326	10	309	314 (5)
Gson	4901	221	20	297	313 (16)
Jsoup	6098	195	3	222	224 (2)
JxPath	3643	67	3	227	230 (3)
Lang	5032	621	37	262	287 (25)
Math	6903	1129	265	747	907 (160)
Time	11521	1439	20	500	515 (15)

Table 4.4: Average Running Times (in ms).

Comparison	*p-value*	Significance	Effect Size ($\hat{A}_{12}$)
FAST-Ekstazi	0.000462	***	0.04 (large)
FAST-Fastazi	0.000366	***	0.03 (large)
Fastazi-Ekstazi	1	ns	0.55 (negligible)

ns = not significant, *** means *p-value* ¡ 0.001

Table 4.5: Time Efficiency Comparison.

is the technique that incurs the least time overhead; and *ii*) the overhead of Fastazi with respect to Ekstazi running time is generally very small.

To confirm our observations we performed the non-parametric Kruskal-Wallis rank sum test, and the result (*p-value* = 4.5e-05) confirmed that at least one of the approaches was different from the others with respect to the time efficiency. Provided that significant differences were detected, we proceeded with pairwise comparisons to determine which approaches were different and the results are displayed in Section 4.5.3. Statistically significant differences were observed when comparing FAST with Ekstazi and Fastazi, but not when comparing Fastazi with Ekstazi. Finally, to understand if the observed differences in time efficiency are not only statistically significant but also meaningful to support practitioners in the decision of whether Fastazi should be adopted, we measured the effect size. The results can be interpreted in an analogous way of that explained in Section 4.5.1. The effect size for the comparison of FAST with Ekstazi and Fastazi was $\hat{A}_{12} = 0.04$ and $\hat{A}_{12} = 0.03$, respectively, indicating that *the effect on the time overhead* *when running Ekstazi or Fastazi is large.* On the other hand, the effect size for the comparison between Fastazi and Ekstazi was $\hat{A}_{12} = 0.55$, indicating that *the additional time overhead incurred by Fastazi when compared with Ekstazi is negligible.*

It is important to notice that such results concern the overhead time required by the studied techniques, which are anyhow one or two orders of magnitude shorter than the time required for actually running the whole test suites.

Summary of RQ4.3: When considering the three approaches in isolation, FAST is the most efficient one and the difference with respect to the time overhead incurred by the other approaches is *large.* The additional time overhead incurred by Fastazi for prioritizing the test cases selected by Ekstazi is not statistically significant and the effect size is negligible.

4.6 Discussion

Software regression testing has undergone extensive research in the last several decades. The largest part of solutions, though, addressed separately one dimension of the problem at a time. While many TCS and TCP techniques have been proposed, they have not been directly compared, only few authors look into integrated approaches for combined selection and prioritization, and no work empirically assessed the advantages of using TCS and TCP in combination over their individual application. In contrast, we believe that, by merging differing criteria for selection and prioritization, we can achieve the most from the restricted subset of test cases that can be executed at each new release.

Towards this direction, we presented a study directly comparing two recent practical and effective approaches to TCS and TCP, namely file-based selection (by Ekstazi) and similarity-based prioritization (by FAST). Our results show that Ekstazi generally outperforms FAST, although the effect size is negligible or small; however, their orchestration by Fastazi outperforms both with a non-negligible effect. Moreover, considering a limited test budget, Fastazi exposed a higher effectiveness in consistent way. After assessing the overhead imposed by each of the studied approaches, we can conclude that Fastazi is quite practical: if we parallelize the preparation steps, the additional cost of similarity-based prioritization of the test cases selected by Ekstazi is negligible.

We aim at further improving the effectiveness and efficiency of Fastazi by refining several technical aspects. In particular, to make the approach more easily usable, it should be integrated into build systems as a plug-in as Ekstazi is now. In addition to that, we would also like to try orchestrating other TCS and TCP techniques from the literature to understand the resulting challenges and outcomes.

4.6.1 Existing Examples of Test Suite Orchestration

Although the nomenclature of Test Suite Orchestration is recent, the concept behind it is not entirely new. The reader might have observed that the literature review in Chapter 3 contains some examples of papers that address some combination of TCP, TCS, and TSR— thus wondering if these are previous examples of orchestration in the literature. This is true in certain cases, although not always.

Particularly, orchestration refers to strategies combining different approaches to certain aspects of RT. Conversely, most of the papers in the review covering more than one RT challenges (in most cases, TCP and TCS) do so by applying a common algorithm to both problems. Thus, they do not strictly fall under the definition of Test Suite Orchestration, although they could still be considered as a component of an orchestration strategy along with other approaches.

Aside from our own publication introducing Fastazi [53], a notable example of test suite orchestration is from Shi et al. [128], which is not discussed in Chapter 3 due to its publication date of 2015, who compare empirically TSR and TCS. The authors observe that both techniques aim at running only a subset of the test suite, thus asking which one is better considering the size of the reduced test suite, and the loss in terms of detection capability of change-related faults. For TCS their study adopts the Ekstazi tool above mentioned, whereas for TSR they remove redundant test cases using a greedy heuristic based on statement coverage. From the comparison they conclude that TCS on average returns a smaller test suite size, with no loss in change-related faults detection for safe TCS techniques. In comparison, TSR can miss a small percentage of change-related faults. They also evaluate a combination of the two techniques, specifically "selection of reduction" in which TCS is applied on the reduced suite obtained by TSR: this can further reduce the number of tests yielding the same loss in fault detection observed for TSR alone.

We were unable to detect further examples of publications fitting the mold of Test Suite Orchestration. This signifies that addressing more than one challenge with the same technique is an existing — even if infrequent — concern among researchers, but combining distinct approaches together is not.

4.6.2 Future Directions for Test Suite Orchestration

As previously discussed, the experiments we performed with Fastazi is a starting point for test suite orchestration. This work paves the way to exploring a full range of potential strategies of combining differing criteria for selection and prioritization. It can be worthwhile to also expand the study to the orchestration of techniques along other dimensions of regression testing, e.g., also test suite reduction (TSR) or test suite amplification (TSA).

When combining multiple techniques into a cohesive orchestration strategy, the first and perhaps most important aspect to consider is the sequence of operations. We see in Section 4.3 that there are two ways of using TCS and TCP together: we can either select a set of test cases and the prioritize these, or prioritize the entire test suite and run the selected tests in that given order. Including more techniques in the orchestration inevitably leads to more possible sequences.

For example, if we add TSR to the orchestration, the operation could be performed before or after the selection and prioritization. By using it before, we already restrict the number of test cases the other techniques must deal with; doing it afterwards, the results of the reduction will only be used in the next execution of the test suite.

The combination becomes more interesting when adding TSA to the strategy. TSA could be the first technique to run, updating or adding test cases that will then serve as input for selection, prioritization and reduction. Or, it could be placed in between selection and prioritization, modifying the suite only according to the results of the selection. This could be desired if the TSA process is costly and running it with fewer targets greatly reduces the time it consumes.

Continuing this line of thought, Figure 4.6 shows an example of a fully orchestrated test suite execution. In it, we consider three subsequent versions of the SUT ($\mathbf{v_{i\text{-}1}}$, $\mathbf{v_i}$, $\mathbf{v_{i+1}}$). The chevron boxes represent some process being applied to the tests, while the cut rectangles represent variations of the test suite (e.g., a list of test cases).

The target of the orchestration is **T**, which is the test suite corresponding to version $\mathbf{v_i}$ of the SUT. The first technique to be applied is TCS, generating a subset of tests **S**. Additionally, from previous test execution logs, historical data, such as test that have recently failed, can be extracted, forming the set **H**.

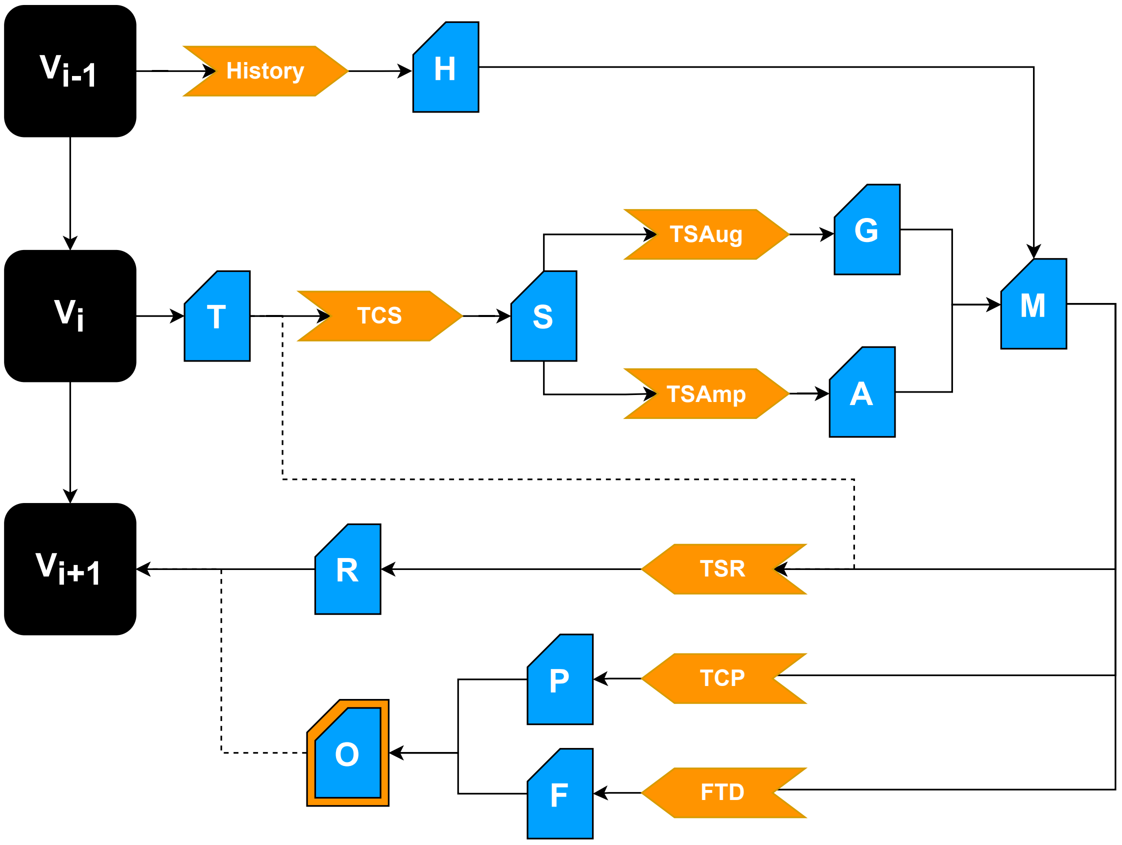
This subset is then used as input for TSA techniques, in this example displayed separately as augmentation and amplification. The results are one set of *newly generated* tests **G** and one set **A** containing the amplified versions of the tests in **S**. At this point, information from **H**, **G** and **A** is merged into a list of tests **M**.

M is then used as input for three different techniques. On one side, TSR is used, using information from **M** and **T** to eliminate excessive redundancies in the suite and produces a tighter suite **R** that can be used as a starting point for the next cycle of orchestration (when it is time for version $\mathbf{v_{i+1}}$ of the SUT to be tested). On the other, TCP prioritizes the test cases to **P** and a test flakiness detection technique provides a list **F** of potentially unreliable tests, which should be handled differently during execution.

Finally, the orchestrated test suite **O** is produced, which can be used to test the SUT version $\mathbf{v_i}$.

It is worthy of reiteration that this is simply an example, built upon the goals of each RT technique and considering how they can be used to each others' advantages. Validating such a model requires extensive experimentation, which unfortunately poses a technical challenge, as not every technique has an available and easily usable implementation. Even when the tools exist, the way each one handles inputs and outputs can be incompatible, so some alteration is needed.

Some questions remain unanswered regarding a fully orchestrated strategy. The possibility of executing all RT techniques at each new version of the SUT largely depends on the intervals between versions; if new versions are committed frequently, there might not be enough time to execute the full process. In such cases, an additional point to consider is which techniques are important for frequent execution, and which ones can become part of a nightly testing cycle.



Legend: $\mathbf{v_{i-1}}$, $\mathbf{v_i}$, $\mathbf{v_{i+1}}$: previous, current and next version of the SUT; **H**: output of history-based criteria; **T**: the test suite as of version $\mathbf{v_i}$; **S**: the output of TCS; **G**, **A**: the outputs of test suite augmentation and amplification, respectively; **M**: a *selected and enhanced* test suite combining the outputs of the previous steps; **R**: the output of TSR; **P**: the output of TCP; **F**: the output of a flaky test detection technique; **O**: the *orchestrated test suite* that should be executed for $\mathbf{v_i}$.

FIGURE 4.6: Diagram showing an example of a fully orchestrated approach to the test suite execution and evolution.

4.7 Threats to Validity

We evaluated Fastazi using faults available in Defects4J. Our results and conclusions could be different had we used another bug repository. However, Defects4J is among the most popular bug repositories and is heavily used in research on regression testing. Additionally, it includes real faults, which strengthens our findings.

The fact that we use Defects4J means that we were running experiments on project versions that are potentially very far apart (e.g., years). In this setup, Ekstazi might select a very large number of tests, because it was designed for small code changes between two consecutive commits [48, 140]. However, Ekstazi ended up performing well even in our setup.

We defined the testing budget as the number of tests that one can run at each project version, which does not take into account the differences in individual test execution time. As we focus on unit tests, we do not expect that there would be substantial differences in execution time across tests.

To measure effectiveness, we used TTFF and APFD. As known the Defects4J subjects contain only one fault per version and hence the two measures behave similarly. To mitigate this issue, we need to perform more studies on subjects containing multiple faults, for which the APFD measure becomes more valuable.

In our experiments we assume that test execution is deterministic, which we know does not always hold in practice, i.e., tests are flaky [88, 58]. We have not observed any flaky behavior in our experiments: only the expected set of tests was failing in each run.

Chapter 5

Insights from Industry

As seen in the discussion from Chapter 3, not many RT approaches and techniques proposed in academia make their way into practical usage. To dig deeper into this problem and understand the underlying challenges, a crucial part of this research involves direct communication with members of industry who work on software testing.

Upon contact, members from a large technology company granted the opportunity to spend some time at their offices to observe practices, collect data and interview team members. This chapter synthesizes the findings of a seven-week period which was spent in direct contact with the company.

The interacted team is responsible for a software system which is an integral component of the company's delivered product. Specifically, practices related to multi-component testing (MCT) were investigated; at the time, there were thousands of MCTs in the system. Our focus is on the regression testing procedures (which was at times referred to as *legacy testing*); although naturally we were also interested in knowing where RT fits in the overall test strategy of the team.

The interviews cover a variety of topics, including education in software testing, current practices and procedures employed by the team, their relationship with members and research from academia, and the most notable challenges they face. Analysis of the responses show that there is a strong desire to improve processes, which is hampered by reasons including technical challenges, bureaucratic hurdles and even skepticism by some team members of automated solutions.

Section 5.1 describes the questions that drive the discussions brought here. Section 5.2 provides an overview of the testing process at the company and of the system for which the interviewed team is responsible. Section 5.3 brings quotes from the interviews to form a picture of the state of practice from the perspective of the practitioners themselves. Finally, Section 5.4 contains the answers to the research questions, in the form of a series of observations derived the interviews.

5.1 Research Questions

The following research questions represent the main objectives of the collaborative interaction with the industrial partner. Generally, the contribution of this period is the identification and explanation of notable challenges involved with industry-academia collaboration and the advancement of practical software testing techniques.

It is worth noting that these are not the questions asked directly to the practitioners (which are found in Appendix A), but they were used a starting point to design the interviews.

RQ5.1 What are the regression testing issues most frequently encountered by software testing practitioners? — We want to hear from the practitioners themselves what are the regression testing challenges they face on a daily basis. It is important to understand whether these challenges are the same that motivate software testing research, or if there are concerns from industry that are yet unaddressed by researchers.

RQ5.2 What are the challenges that arise when trying to incorporate academic insight in practice? — Based on their previous experiences with regression testing techniques, we would like to know what worked and what didn't, and for what reasons. We know beforehand that it is not trivial to convert academic knowledge and insights into a company environment, but we should understand with more details why this conversion is so difficult to accomplish.

RQ5.3 What are potential paths to improve collaboration between academics and practitioners? — With this we seek to capture what the practitioners are looking for when they are approached by academics. This discussion should help researchers propose more relevant collaborative projects, as well as set the expectations of practitioners as to what is feasible to achieve when working together with academics.

5.2 Overview of Testing at the industrial partner

Since the mid-2000s, the company in question has adopted agile development principles. This had a notable effect on testing practices, as previously there were distinct roles for developers and testers and now many developers are responsible for testing their own code, although the company also continues to employ a substantial number of dedicated systems testers.

Currently, their test design is based on a multi-layer "test strategy model" that provides the hierarchy of tests for a given system, starting with unit tests, which should be the most atomic and numerous tests. At each layer, the complexity of tests increases, as each test becomes responsible for covering a larger amount of source code, leading up to extensive end-to-end tests.

The discussions in this chapter refer mostly to multi-component testing (MCT), which is among the lower layers of the testing strategy model. Initially, developers run the MCT suite on their development machine as a mechanism to aid the writing of new code. At this point, new tests can also be written, or old ones can be updated, to account for changes in requirements. During the day, multiple developers commit changes that should be merged into a component's main branch and, at that point, a test suite run is queued for execution on a testing server, running a limited number of tests to ensure critical features are functional. For the purposes of this discussion, we shall name these executions, which are triggered after each change to the source code, Short Interval Regression Testing (SIRT). Finally, at certain longer intervals, another test execution is started, which runs all test cases and ensures none of the day's updates caused a system-breaking error. Here, we shall name this procedure, which happens at a fixed schedule, as Long Interval Regression Testing (LIRT)[1].

[1]SIRT and LIRT are not terms used by the company, who asked to not disclose internal process names. These two acronyms are used here as a proxy, aiding legibility.

Generally speaking, although the unit tests are more numerous, the higher-level tests are responsible for a great part of the testing costs, as they involve multiple software components along with hardware simulators and, in some instances, actual physical hardware. Additionally, when a failure is detected at higher levels, it is more challenging and time-consuming to identify the cause of the issue. In comparison, unit tests are often executed completely by a developer at their local computer and checked again while merging the code in continuous integration; as each test covers well-defined pieces of code, a failure in this level leads to a quicker understanding of what could be going wrong.

For this reason, certain teams have been implementing a so-called "shift left" policy for testing. The objective is to bring as much fault-finding capability as possible to the lower levels of the test strategy model. One notable way of doing this is by writing new unit or component tests whenever a failure happens in a complex test. However, there is also a plan to incorporate this policy into the test strategies of software still under design and development.

5.2.1 Overview of the system

For this study, a period of seven weeks was spent at an office of the company in Europe, in order to understand fundamental aspects of the testing procedures at the company. It cannot be said that this is a comprehensive account, because the data is extracted from only one small part of the entire corporation, and the overall scope of the projects being conducted is too large for full comprehension in such a short time frame.

The investigated system is part of a software stack deployed to a delivered product. Currently, there are two versions of the software in active development/maintenance mode and in use by customers.

Furthermore, the team we interacted with is mainly concerned with MCT, which involves the integration of multiple software components in addition to hardware and infrastructure simulators. Based on reports from the team members, generally speaking the tests are written in a different programming language than that of the SUT. Reports indicate that the two versions have thousands of tests each.

ID	**Role**
R1	Functional Area Domain Tester
R2	Functional Area Domain Tester
R3	Functional Area Domain Tester
R4	Continuous Integration Test Manager
R5	Module Test Manager
R6	Module Test Manager
R7	Senior Test Specialist
R8	Senior Test Specialist

Table 5.1: Team members interviewed for this study.

5.3 Interviews

Information about how testing is performed at the industrial partner and regarding the challenges still faced at the company was gathered via a series of interviews. This was initially in the form of unstructured conversation, while the interviewer understood the central details. After the basics were covered, we performed a series of 30- to 60-minute sessions with 1 to 3 people at a time asking more focused questions.

Table 5.1 lists the roles of the interviewees, who are anonymized for this study.

5.3.1 Roles and Experience

R1, R2 and R3 are members of the Functional Area Domain team[2]. They describe themselves as "*the owners of the test suite*" (R2) and are responsible for "*monitoring the test results*" (R2). In particular, they "*are monitoring the nightly runs*" (R1), i.e. the LIRT or non-blocking tests.

R4, R5 and R6 are test managers, albeit R4 has a different responsibility. The module test managers, R5 and R6, are responsible for the testing of particular modules, meaning "*not working on specific features*" (R5). "*I also guide the teams on how to write test cases [...], how we organize test suites and so on*" (R6). Meanwhile, the CI Test Manager R4 is "*responsible for the machines and environment and some of the test framework parts. To simplify, we are doing the framework, the developers are writing the tests, the managers are designing the strategy*" (R4).

[2]These team members were interviewed jointly; others interviews were one-on-one sessions.

Finally, R7 and R8 are designated as test specialists, meaning they handle longer-term strategy. "*I work a lot with test strategies. How we should test features and products [...] from now until 2025*" (R7). Despite the similar titles, there is a key difference between the two specialists: R7 is responsible for a long-term strategy of a specific core of products, while R8 has an overview of the entire company, so their job also includes sharing technologies and strategies among distant teams.

Aside from R8, who has a PhD on the topic of software testing, most of the interviewees did not have testing as a focus of their education while at university. When asked about their education, R4, R5 and R6 are computer scientists and/or engineers, and most of the testing knowledge they had prior to working at the company was "*standard university [curriculum], which contains tests*" (R5). Meanwhile, R7 "*graduated in media and communication*", so testing was "*not covered in university*". R4, R5 and R7 mention having a certification by the ISQTB[3]. Additionally, R4 also mentions taking testing courses led by R8 internally, while R5 reports having attended workshops from "*people [from the main office] presenting how testing should be done, but it was so high level that I couldn't understand how to apply it to my level*" (R5). It is unclear if the testing courses/workshops mentioned by R4 and R5 are the same.

5.3.2 Current Practices

Regarding the current testing practices, we want to understand what are the day-to-day activities performed by the team. We would also like to know if there are any implementations of the regression testing techniques classically studied in research (selection, prioritization, reduction, amplification).

The interviewed team members are mostly working on MCT and have all mentioned two key processes for this layer: "*[one] for delivery, [another] for less frequent runs*" (R1). As previously discussed, we use the acronym SIRT to refer to a short execution of the test suite that happens whenever a developer pushes changes to be merged into the main branch of a module. Due to time and resource constraints, "*for [SIRT] there is selection, in nightly we run everything*" (R1). This is also called the "gating loop", since a failing test prevents the change from being merged. At the time of the interviews, "*it's manually decided what goes in [SIRT]*" (R6) and "*the running time [...] for MCT is 15-20 mins*"

[3]International Software Testing Qualifications Board

(R5). The LIRT is also known as "assessment loop", which includes all tests in the suite, and can take up to 10 hours. As a general rule, the tests in SIRT are a subset of the LIRT suite.

Selection. In MCT, it is manually determined whether a test should be included in the SIRT. As an example, "*yesterday, we got the question, 'should we include this in gating?' I don't know, I just go by feeling. We have to see if the feature seems fundamental or important somehow*" (R5). Despite this practice, most of the interviewees are aware that there could be a better way of doing things: "*I think there should be more strategy than just me thinking*" (R5). When this finding was reported to R8, who oversees testing all over the company and does not work with R1-7 on a daily basis, they mentioned that the company does have an internal tool for test selection and was surprised to find out this team did not use it. R4 explains: "*we had attempts to integrate [the test tool mentioned by R8] but it did not work out so well. I think we never built a business case for [the new version of the system] and it ended up way in the backlog. In 2017 or 2018 we said 'this is something we should add to our plan', but we did not do anything.*" However, "*it might be deployed in other parts of the company*" (R7). That said, R8 questioned the necessity of TCS in a well-designed agile production: "*why do you need to select? The agile principle says we should test everything*" (R8).

Prioritization. This does not appear to be an active concern for the team and there are no techniques in place for advanced prioritization: "*for prioritization, nothing specific*" (R1). "*We have suites, which are groups, e.g. for sub-modules, gating or not. Then I think it's just the order they're written.*" (R5). However, "*now we have introduced shuffling for the assessment*" (R5), i.e. random ordering of the test cases. The motivation for this is not decreasing the feedback time, but rather "*to help find problems with unstable tests*" (R1), which might be flaky according to the execution order. R8 again expressed a concern: "*using the same TCP approach every time, wouldn't the same test be top priority every time?*" (R8).

Reduction and Amplification. These techniques are absent in the workflow. "*We don't have any tools that helps us in any way in shrinking or expanding the tests*" (R5). "*No, we don't have anything like that*" (R6). Regarding reduction, "*if it happens that* *there are too many tests for overnight, there would be an initiative to reduce*” (R1), but “*we can put more machines and we can run more*” (R6). Interviewees generally agree that it is more cost-efficient to increase computing capacity of testing servers than to spend human time determining which tests to remove. For amplification, there appears to be no interest in automating the process. There is a protocol, however, to manually augment the test suite in the situations where a fault slips through a layer of testing: “*if a fault was found later in test or in the field, we try to investigate why did it slip through, and we should have a test for that*” (R1); then, “*after the fact, if there is time, we can go back and understand why a needed test was not in the plan*” (R2).

5.3.3 Common Issues With Regression Testing

Part of the questions asked interviewees to detail the common issues they notice with the current state of the test suite and the testing practices at their team. They were asked to elaborate on five key points: fault detection, flaky tests, running time, increasing test scope and reducing test scope.

Fault detection. Most respondents have reasonable confidence on the fault detection capability of the test suite, although several admit that they might not have the data needed to be sure and it mostly relies on the fact that few faults are detected in the delivered product. Since most of the monitoring of the test suite happens within the CI/CD pipeline, there is an understanding that most of the fault detection actually happens before that, on the developers' local environments. “*Perhaps a suite is going green always, but it's being used by a developer while they are writing*” (R5); “*the very good test cases [...] find faults when running locally at the developer's computer*” (R1). “*[Developers] are quite happy doing MCT testing locally. I think they find a lot of bugs there. They rely on SIRT to feel confident they won't break everything when delivering*” (R2). Thus, most of the faults detected during the SIRT would be a result of conflicts between changes merged by two developers asynchronously, rather than the result of poor code submitted by a single developer. That said, the team recognizes that the current version of the test suite is not 100% effective in avoiding faults or “slips” (when a failure is detected in a higher layer of testing than it should have been), especially during the gating loop. “*We can't have all the test scope in the gating loop. There is a discussion of having specific test* *cases for gating in order to have more coverage, instead of just a subset*" (R6). There is a back-and-forth discussion between two sides of the team: one wants the gating loops to pass more often and let the changes through, the other wants them to be stricter. "*The flow guys remove everything to get the flow going, but we are like I don't know, this test might be important*" (R5). Once a fault slips and it is detected at a later testing phase or in the field, there is "*a systematic way of processing trouble reports (TRs) and we can identify where it is needed to add tests*" (R5). Regarding that, R8 brings an analogy: "*if you only take care of the shootings, you do not solve crime*", implying that the above is not an effective strategy for delivering a fault-free product.

Flaky tests. Although techniques for addressing flaky tests falls beyond the scope of this thesis, it was a topic brought up frequently during the discussions — "*that's the worst, we have a lot of those*" (R5) — it is strongly related to the above point regarding fault detection. "*The tests are run distributed in several computers connected to a network, so when the load gets really high, running thousands of tests at night, we can get different behavior compared to running the tests manually*" (R2). To account for this, "*we started shuffling the order of the tests*" (R3), although sometimes the developer who wrote the test case admits "*'I don't know why it fails if it comes later in the suite'*" (R3). There is an agreement that "*creating stable [non-flaky] tests is cumbersome*" (R1) and, "*if it's due to the code, it's usually timing issues*" (R2). Some team members believe this could be relieved with better test design: "*most of the times, flaky tests are caused by poor testing*" (R8); "*a lot of people develop tests but are not very familiar [with test design principles]*" (R1). In fact, upon investigation, testers in the functional team "*find more problems with the tests than with the products*" (R1). The test managers share this perspective: one says "*less than 50% of our failures are actually a product fault*" (R6), while another thinks "*it's 30/30/30 [actual error/test design/environment issues]*" (R5). Despite being aware of this, the team has learned to deal with the issue. For example, "*we have one test that is mostly working, but it's failing once a week*" (R1). At first glance, it might appear that this test is not adding value to the suite, but actually, "*we can double and triple check that it's not actually a bug in the product*" (R2) and, "*if it starts failing every day, we would look into it, so there is some value*" (R1). Due to situations like this, "*every day there's at least one failure*" (R6) in LIRT runs, but the team recognizes the troublesome tests. That's not to say these tests are ignored: "*usually, we are currently working on a* *solution but it's not done yet*" (R2). Since these tests are particularly difficult to fix and they still provide some value to the test results, they are left as-is unless a fault related to it slips to another layer of testing.

Running time. Test managers claim that "*the running time is fairly okay, [...] for MCT 15-20 mins*" (R5). More important than the running time itself (which is offloaded to virtualized testing server) is the queueing time, which increases during the day as more test jobs are dispatched to the servers by the developers. For this, "*we can also test in parallel a lot of things [...] and spin up more virtual machines as needed*" (R5). Naturally, "*the main pain point is the gating loop, since we want to have shorter feedback, it's more loaded during the day as people are working and at night it's mostly free*" (R6). The parallelization is limited, however, at least in some layers of testing. Since the main branch of a module is updated linearly, separate commits cannot simply be tested in parallel and then merged without being tested together, so what may happen is that a certain number of commits is tested in parallel and, if the tests of at least one passes without issue, only one is merged into the branch. Failing tests are sent back to the developer, and other passing commits are queued for execution again, using the newly-updated branch as the base.

Increasing test scope. Although simply increasing the number and even coverage of tests is not a challenge on its own, doing so maintaining quality and consistency might be. Currently, "*it's the feature developers who write tests*" (R6) or, in other words, developers mostly write tests to cover their own code. "*The problem is that cross-functional teams (XFTs) are racing for product delivery*" (R6) but, since many developers are not necessarily proficient in good test design, "*it's easier to copy paste code. Tests are not always great. I would say it's something we need to work on.*" (R6). As the product itself grows in scope, "*it makes sense that the tests increase at the same rate, but to me we are not increasing the test scope at fast as the product*" (R7), so there is still a feeling that the tests are seen as secondary compared to the product itself.

Reducing test scope. Many interviewees agree that this is a difficult challenge that is not currently well addressed. "*Cleaning and updating old tests is most difficult*" (R1), but "*who can decide if a test case should be removed?*" (R5). Even if refactoring or reducing the test suite is discussed, "*it's cheaper to leave [the test] there and run it than spend money to have people looking at it*" (R5), especially in LIRT. On the more restrictive "*SIRT, if new tests are coming in, we have to shift out legacy tests*" (R6), so in this situation the tightening of test scope is essential. Although this perspective has changed over time: "*in the beginning of [the new version of the system], we said 'everything should be gating', but now we have too many test cases*" (R6). Often, a test "*might make sense when it's first written, but many developers go into the same same suite, so over time it might degenerate a bit*" (R7), which would call for refactoring if not outright removal. The specialists bring a complementary insight: "*even if tests are obsolete in terms of fault detection, they might still be useful for root causing*" (R8), meaning that a test that doesn't find faults might help narrow down the causes why other tests are failing.

Testing costs. Testing costs is often discussed in the literature when it comes to the applicability of solutions, but it is not a major concern to most of the interviewees. The team uses cloud-based testing, and "*the cloud infrastructure is completely ours*" (R4), not outsourced to other companies. The functional testers "*don't know [the cost]*" (R1), but it is "*probably a lot*" (R2). R6 also says "*I don't know how much kilowatts it's drawing*", but, according to R5, "*I don't think it's more expensive than watching a youtube video*". It's understood by several team members that resources can be increased if required, and one manager goes on to ask "*why not run everything more and faster, instead of selecting things? It might not be cheaper, but it seems like the company is more willing to pay for equipment than employees*" (R5). When asked specifically who would know this, the answer is "*the area product owner*" (R4), who "*handles the budget with inputs from us, according to how much we use and how much we'll need*" (R4). The specialists highlight that "*earlier test loops are much cheaper*" (R7), so it is important to detect faults as early as possible. Furthermore, even if "*electricity is so cheap*" (R2), "*we have some hardware that is super expensive, it costs as much as house*" (R7). For this reason, "*we want to make sure the test cases that execute on that hardware are only the most important ones*" (R7) and this type of load balancing is still being worked on by the cloud infrastructure team.

5.3.4 Collaboration with Academia

Another point relevant to this study is the practitioners' current relationship with academia. Firstly, they were asked if they keep themselves informed about recent research trends in software engineering. Most respondents answered in the lines of no, "*I don't find the time or I don't prioritize it*" (R1), "*our level, we don't really have time*" (R5). R2 occasionally reads books on the topic, although these are more practical in nature rather than research works. Time is the main issue, because "*because the research world is huge and there's so much happening*" (R7). R5 thinks "*it would be interesting to see how other big companies do it*", which alludes to initiatives such as the Google Journal Club [103] that have seen success in some companies but does not appear to be a practice in this case. "*I suppose someone at the company has this assignment somewhere*" (R5) and, indeed, this someone is likely R8: "*I read the literature and try to apply it*". However, "*academia is both ahead and behind [the state of practice]. It's ahead because it's looking into techniques that might be useful many years from now. It's behind because it doesn't keep up with the current challenges in the state of practice*" (R8). There is a perception that academic work is important, but ultimately disparate from the immediate needs of practitioners.

A follow-up question asked if the team member were aware of any current tool or practice at the company that had origins in academic research. The general response to this was uncertain. Nobody could point to a specific and concrete example, but there are some possibilities, for example "*before we used to run everything manually, then this scheduler was developed that can define when and what to run*" (R5), but "*I don't know if that's from academia*" (R5). Furthermore, "*over the years we have had much cooperation with research, so I would assume there is something*" (R7), but again no clear example was given.

When asked if they still keep in touch with a friend or colleague in the academic world, it appears this is not common. "*We always have old colleagues that are PhD, but unfortunately we don't hang out so much*" (R5), and "*I think all my friends from college are now in industry*" (R4).

Continuing on this line of thought, some respondents mention there is some collaborative projects that happen involving the company and a nearby university. "*At least in [this city], we have been keeping a close relationship with the university*" (R4), although the main objective is not necessarily to develop new research or technology, but rather "*to vacuum the competencies from the university towards us*" (R4). This mostly occurs through co-sponsorship of Master degree students: "*we announce some thesis work and try to keep them here*" (R4). R7, however, mentions volunteering "*a study by another company [about] roles for testers, [...] I think [the researcher] is employed by both that company and the local university*", indicating there is some cooperation happening between the two companies and the local university. R1, R2, R3 and R6 provide a negative answer: "*no, there hasn't [been collaboration] from what I know*" (R6). Regarding specifically an experience similar to the present work, e.g. a visiting researcher collecting data and performing a study on the current practices of the company, "*no, I think you're the first PhD I'm having contact with*" (R7).

When asked what would motivate the team to incorporate a technique from academia into their workflow, the main factor is the ease of use and the resulting reduction of workload from the team: "*how easy to implement is it? Does it remove manual work?*" (R6). However, regarding risks and challenges of doing so, there are several relevant points. A major one is the matter of security. It is recognized that "*external tools can be security risks*" (R8) especially because "*we don't know if the researchers are rigorous with their implementation*" (R8); indeed, it appears that "*something happened some years ago and, for security reasons, we don't want to incorporate open-source tools into the process without making sure what they are*" (R5). Due to this, "*we have this [free and open-source software] (FOSS) process. If you want to use open-source you need to submit a ticket, then it needs to be scanned and approved*" (R5). Furthermore, it is hardly the case that general open-source tools can be incorporated without bespoke modifications: "*it usually doesn't fit*" (R8). For example, "*researchers might assume a certain level of quality or standardization among test cases, but in reality test design is often bad*" (R8), which could explain in part the existing gap between academic experiments and real-world usage. In addition, "*we have to figure out how to feed data to the tool*" (R2), which is not always trivial: "*the algorithms are usually fine, but it's hard to provide the data that is asked for*" (R7). One possible reason for this lack of data is that many of the test executions are triggered by the developers themselves while they are still working on a new feature or bug fix. A failure at this point might be a legitimate detection of an error, or it might be expected by the developer who triggered the test while being aware that the code was unfinished; it is difficult to filter out this type of data and provide a "clean" input to an algorithm.

Despite the above concerns, the team members expressed a drive to make changes to their workflow and say that "*we are quite good at solving technical difficulties*" (R2) and performing the necessary modifications and experiments is not the main obstacle to overcome. Rather, "*the big problem is that there are unclear responsibilities*" (R1), so "*It's not clear who we would need to convince to make changes to the process*" (R1). This opinion is corroborated by other team members: "*I always see bureaucracy as a part of the difficulty of getting things done*" (R5); "*I think the main challenge would be convincing people to spend time and money on it*" (R6); "*the problem [...] is to find out who is going to pay for it*" (R4). Considering the scale the company, however, "*I think it's good we have this organization, the company is really big so we need structure, but it can make it harder to make these smaller changes*" (R4), but it is clear that "*programs are measured by how many features they are developing, especially in the customer end. You're measured more on feature throughput than product care*" (R4), so software quality activities such as testing and refactoring, which are "invisible" to the customer, are hard to prioritize. R7 mentions a dedicated internal group "*who can work with anyone within the company*" (R7). Together, they performed a study to improve test case selection in their next-generation products, but it is "*challenge to convince someone to do it*" (R7). "*The project [with the internal group] was a theoretical experiment, but we did not try it in practice. There's a limited budget to how many improvements we can do in parallel and other things were prioritized*" (R7).

5.3.5 Metrics

In conversations with R4, the topic of metrics for measuring test suite effectiveness was discussed. Metrics familiar to researchers, such as APFD, is not part of the common language at the company, but there is a current initiative to add some measurements to the testing flow and extract useful metrics from them. Specifically, three metrics, all under consideration but not in active use, were explained:

Product Fault Finding Capability (PFFC). The objective of this metric is to determine whether faults are being detected in the "correct" layer of testing, or if they are slipping to more complex and costly testing. It is measured by two variables: (1) the number of *new product faults found* ($npff$), which counts the number of new faults after a test execution and analysis of failures, and (2) the number of *slips* ($slips$), which is the number of faults detected in a certain layer of testing that, upon inspection, should have been found earlier in the testing stack.

$$PFFC = \frac{npff}{npff + slips}$$

Cost to Fault Ratio (C). It is particularly difficult to measure the cost of an individual test, but it should be possible to at least estimate the cost of an entire test suite, or the cost of each detected fault. This metric does that by multiplying the *machine costs* (MC_n) with the number or sizes of *system test plans* (STP_n) meant for execution in that machine. The resulting value is then divided by the number of detected faults in order to get the cost per fault. Ideally, this should ensure only critical tests are executed in the most costly testing servers.

$$C = \frac{MC_1 \times STP_1 + MC_2 \times STP_2 + ... + MC_n \times STP_n}{faults\ detected}$$

The obvious flaw with this metric is that it might disfavor products with a low number of faults — as fewer faults are detected, the cost per fault ratio will greatly increase.

Reliability (R). This metric targets specifically flaky tests, providing a value to their reliability. It is simply a division of the number of *faults detected* (fd) by the test suite by the number of *failed runs* (fr) of that same suite.

$$R = \frac{fd}{fr}$$

Suites with low reliability are ones that fail often, but rarely lead to a true detection of errors in the SUT.

5.4 Observations

5.4.1 RQ5.1: Common Issues

From the literature examined in Chapter 3, it is possible to observe that test case prioritization is the regression testing technique most widely addressed in academic research. The conversations with the team, however, show that it is not a topic of active concern for the practitioners. Indeed, when questioned about methods of determining the execution order of tests, respondents claim that, until recently, it was "*just the order in which they were written*" (R5). Due to concerns with flaky tests and developers who were explicitly asking to run their tests before all others, shuffling (random prioritization) was introduced. This helps the identification of tests that are flaky due to interdependency — e.g. one test that presumes another has been executed previously — or due to improper cleanup — tests that affect the global state of the program and do not revert that change before concluding. However, it is a far simpler approach than what is studied in the literature and has an ultimately different goal: detecting flaws in the test suite itself rather than speeding up feedback time for faults detected in the SUT itself.

Indeed, the detection of poor test case design appears to be a common problem to the test managers and specialists. Although they did not provide raw data to analyze this claim, their intuition indicates that 30% to 50% of test failures are caused by poorly implemented tests and do not lead to a true error in the product. It's well-accepted that most developers are not adequately educated or trained in software testing principles and learn by imitating previously-existing tests written by people who might be in different roles by that point.

The practice of running shorter test suites during the day and the full range of tests at night is common at the company and, for that reason, test case selection can be a valuable asset. In the unit test level, "configuration-based selection" is used, which is similar to the file-based selection frequently discussed in the literature. However, for multi-component tests, it is manually decided what tests should be executed in SIRT, following a few guidelines and procedures. Each test in this level is certainly more complex than a unit test, and perhaps techniques for addressing each type of test need to be distinct. There is an internal tool used for selection, but the team interviewed for this study claim that attempts to integrate it into their workflow were unsuccessful.

Since most software products are in constant evolution and growth (meaning new requirements and features), it is expected that the test suite will also grow accordingly. In the long term, this means thousands of test cases for a product, sometimes spread across multiple modules. It is often the case that, in the early stages of development, tests will be designed to cover the most basic functionality of the software, which serve as the foundation to more complex features later on. Then, as these complex features are introduced, more specialized tests must also be added, covering intricate details of the functionality. Many of these specialized tests must also cover basic functionality by definition, even if indirectly through the higher-order features that depend on core elements[4]. Thus, in reality, a lot of older tests are made obsolete, at least in purely fault-finding terms. Detecting which tests are redundant is a challenge that was mentioned by several of the interviewees, as currently this would need to be performed manually and the time/budget restrictions do not allow this. An automated solution could be helpful, although it is not desirable to completely delete test cases — when a fault is detected, tests that cover similar parts of code can be analyzed together to help identify the cause of the issue. Thus, there are tests that do not need to be executed daily but could be added to a weekend-only testing schedule, for example. That said, despite acknowledging the challenge of reducing the size of a test suite, it is much easier to simply add resources to the testing server than to spend human time and effort into analyzing tests or even implementing an automated solution.

There did not appear to be a strong desire for automated amplification or augmentation of the test suite. As it is, the suite already grows substantially over time, so adding machine-generated tests to it could create more problems than it solves. Regarding new tests, test managers are more concerned with their quality than quantity, so perhaps methods for aiding developers in manually writing good tests would be better accepted than simply offloading that task to an automated tool.

[4]This is not always true for unit tests, which attempt to isolate functionality as much as possible for testing, but is common in component and multi-component testing.

Summary of RQ5.1. The issue most frequently brought up by respondents has to do with test flakiness, which is very common in their regression test suite; testers and managers have found ways of dealing with them, but it still makes it more difficult to detect true errors in the test suite. More generally, the main challenge is detecting and improving poorly written tests, which is the leading cause of test flakiness. There is a manual process for deciding which tests should be executed at each new commit, while removing redundant tests is something that is talked about, but not usually done because it's cheaper to simply add more compute power to testing servers.

5.4.2 RQ5.2: Challenges of Incorporation

The interviews make it possible to identify a few notable obstacles that prevent the usage of state-of-the-art research techniques in the corporate environment.

From a technical perspective, the first challenge is that academic tools can barely ever be used as-is. As discussed in Section 3.4.3, a lot of additional effort must be exerted in order to go from an algorithm to a replication package, to a functional prototype and, finally, to a commercially viable tool that can be used by developers. Even if the tool does exist and is available as FOSS, it might be geared specifically to a certain programming language or require certain environment characteristics; if these don't match, the interested party would need to re-implement the algorithm for a new target. Those conditions being met, the testing tool must still go through extensive security screening, which adds to the time and cost needed to implement a technique.

Assuming that the above requirements are met, there are additional barriers that might not even be a matter of time and cost. One issue that was pointed out by the test specialists is that academic tools generally assume that the SUT is perfectly designed and has plenty of data available to work with, but that might not always be the case. Even the best designed software will have the occasional oddity, peculiar characteristics that humans might be used to handling, but could produce unexpected results in automation. Regarding the data, there was a discussion during the execution of this study about collecting historical test execution data in order to run experiments; as it turns out, it is not a simple task. Many of the test executions happen in developers' local computers and this data is not included in the CI/CD history. Since these local executions are likely to have frequent failures, their execution history can provide important data about which tests are most important in the initial stages of testing, for example. Furthermore, even considering only the data available from the CI/CD tools, logs are not meant for long-term storage and not organized in a way that can easily be fed into, for example, a selection or prioritization algorithm.

Considering these challenges above, it is not exactly tackling them that is the greatest obstacle. Instead, many interviewees say they would be excited to try new techniques and perform experiments with their software to try and find more efficient ways to handle their testing workflow. However, all this would require a time investment from the team members, which means setting aside other tasks, such as feature delivery. Since new features are the most desirable output in the perspective of customers, it is difficult to convince people in managerial and decision-making positions to slow down deliveries in order to perform experiments that, realistically, might not be successful. An argument to managers would need to include time and cost estimates, including forecasts showing that, in the mid- to long-term, performing these experiments now will lead to notable cost savings in the future. To avoid slowdown, an alternative would be to hire people dedicated specifically to experiment with new techniques and identifying avenues of improvement; however, hiring employees is an expensive process by itself and, given the choice, managers would prefer buying new computers to run more tests before hiring an employee to optimize the existing ones.

Summary of RQ5.2.: Practitioners describe several technical challenges involved with the implementation of new techniques into their workflow. Notably, adapting algorithms to their environment, screening for security risks and collecting appropriate data for input are frequently cited. Regardless, bureaucratic hurdles are more difficult to address than technicalities and, in order to convince managers to invest time and money into the effort, there needs to be robust evidence that a technique will provide meaningful benefits.

5.4.3 RQ5.3: Paths to Improve Collaboration

The interviews make it clear that, at least within the interacted team, there have not been many attempts of collaboration with academia, in the sense of bringing techniques from theory into practice and/or attempting experimentation in realistic software. It should be noted that this is not a general assessment of the company, which may have other examples of collaboration with academia, but only of the team that interacted with this study. Nevertheless, the conversations highlights the current desire for such collaboration and some avenues for improvement.

The team is not completely isolated from academia, as the company frequently funds Master's programs jointly with the local university, although the motivation for this is not necessarily scientific advancement nor development of novel techniques. Students in this program get the opportunity to interact more closely with the inner workings of the team and acquire specialized knowledge before graduating.

Less common are collaborations with PhD candidates, post-doctoral fellows or professors. Occasionally a researcher will invite an employee to participate in a study, such as the aforementioned study with a neighboring company, or even this present study. In such cases, there is usually some data provided by the company or by one of its employee that are used by the researchers in some work meant for academic publication. The next step of this type of collaboration would be to bring the results of that study back into the company and attempt to internal experimentation and potentially implementation of the technique. As far as the interviewees are aware, this either has not happened, or happens very rarely.

With few exceptions, practitioners admit that they do not have the time to keep up with Software Engineering research and no longer maintain contact with former colleagues who might now be professors. There is no simple solution to this problem, as following research trends is not only time-consuming; it can be mentally draining for someone preoccupied with other tasks and it is rarely obvious how the results of a paper can be beneficial for an individuals' workflow.

Internally, there is a team whose responsibility is to interact with different parts of the company and update workflows with new techniques. This dynamic is similar to the "ideal" scenario of industry-academia relations, cutting down some major obstacles (the company maintains control over the developed tool, there is a budget allocated for this, etc.). However, even a tool proposed and experimented with this team did not see adoption by the team that was interviewed, due to the limited benefits observed in the multi-component testing layer.

Summary of RQ5.3.: There is interest and desire of collaboration among interviewed practitioners, but it is something they find difficult to take initiative upon. Most of their time is consumed performing day-to-day tasks and ensuring the delivered product is constantly improved, and there is little time or energy left to keep up with the quickly growing academic literature. They often fund Master's programs and there are examples of academic papers using data from the company, but it is difficult to find situations where research results directly impacted the current state of practice.

5.5 Threats to Validity

Construct validity The observations in Chapter 5 were only possible due to a collaboration with an European technology company, facilitated by a person who is in frequent contact with both the company and the academic world. It is important to note that the findings relate to an experience of only seven weeks at only one office of the company. We did our best efforts to understand the internal testing strategies and workflows, but the overall complexity of the processes is too high to be completely comprehended in a short timeframe.

Internal validity The interviews that were conducted for this chapter were based on the research questions presented here. These research questions were designed to reflect our goals when designing that part of the research, and inevitably narrow the possible findings and guide the general outcome of the study. The quotes included in this chapter are transcribed from verbal interviews with practitioners and edited by the author for legibility, clarity and cohesiveness with the text. It is possible that a response was misinterpreted by the interviewer or that the meaning of a quote is not fully clear to a reader. We took great care of transcribing the interviews as accurately as possible, and of ensuring that their meaning was not altered by cherry-picking quotes or adjusting their wording. Mitigating this risk, the chapter has been read and approved by two people from the industrial partner, ensuring their point of view is correctly expressed. Due to a non-disclosure agreement with the industrial partner, we cannot provide the full unedited transcripts of the interviews.

Conclusion validity The conclusions drawn from the interviews are a result of our own interpretation of the situation, based on the observed data, on discussions with members of the company, and related information extracted from the software engineering literature. Inevitably these conclusions are influenced by the background of the researchers who, to the degree of possibility, allowed the newfound information to shape the conclusions, not the opposite. Unfortunately, this study is not easily replicable, as core components of it are left undisclosed due to confidentiality concerns by the industrial partner.

External validity These observations relate to one team at one office at a large company. The conclusions we draw relate solely to that team and might not generalize to the practices observed in the rest of the company. Furthermore, we make no claims that these findings generalize to software industry as a whole. This threat can only be mitigated if more companies are willing to allow researchers to interview their employees and understand their testing processes and challenges.

Chapter 6

Challenges Between Industry and Academia

While the review in Chapter 3 indicates that IR&A is a growing concern among RT researchers, it's still only being addressed with any depth on a minority of secondary studies. It is clear that several authors believe IR&A is a challenge worth addressing in research, but there is not a lot of available RT literature focusing on the steps that need to be taken in order to improve academia-industry communication and shorten the technology transfer gap.

We conclude this work by highlighting some key challenges that we have identified, combining data found in the literature itself, in the authors' responses and in the practitioner survey. These are challenges that may have been addressed in certain circumstances but remain unsolved in a broad sense, as they are still present in several recent works. Along with each challenge, we make some suggestions that could be applied by Software Engineering researchers and/or Software Testing practitioners — these could be actionable steps for upcoming primary studies, or further avenues of investigation for secondary or meta studies. Table 6.1 provides the summary of the challenges we identified, indicating the primary source of our observation (i.e. the literature, the authors and/or the practitioners).

6.1 List of Challenges

ID	Title	L	A	P	I	ID	Title	L	A	P	I
CH1	Alignment of motivations			●	●	CH6	Absence of TSR/TSA	●		●	●
CH2	Realistic experimentation	●				CH7	Clarity of target	●			
CH3	Scalability	●				CH8	Skepticism				●
CH4	Relevance of metrics	●	●			CH9	Data quality/availability				●
CH5	Developing usable tools		●	●	●	CH10	Communication		●	●	●

Source(s): **L**: Literature; **A**: Author questionnaire; **P**: Practitioner survey; **I**: Industry partner.

Table 6.1: Summary of main challenges identified by this study.

6.1.1 CH1: Alignment of Motivations

When asked what would convince them to implement and use an RT tool, eight surveyed practitioners gave responses that can be synthesized into "*it would make my work easier*". This general sentiment matches the responses received during interviews at our industrial partner.

There exists a mismatch between academic motivations and industrial needs: research is concerned with discovering novel techniques that might provide marginal effectiveness gains over the state-of-the-art, while practitioners are mostly concerned with any solution that simplifies their workflow. In other words, even if an RT technique has the potential to greatly reduce the testing time of a suite, practitioners will weigh those benefits against the effort required to implement the technique and adapt/maintain it for their needs. This is not to say that the current research motivations are ill-informed: it is the role of academia to push the boundaries of what is possible in theory first, and sometimes this theory takes many years to find relevance in practice. One interviewee from Chapter 5 gave an example of this: "*mutation testing*[1] *has been in the literature since the 1990s, and it is starting to see adoption in industry today*" (R8).

If the researchers have the objective of implementing their approach, they must be certain that it is addressing the current needs of practitioners. An obvious way to achieve this, which is also confirmed by our literature review, is by developing techniques through direct partnerships between academic researchers and industrial practitioners (or even open-source communities). In these cases, the practitioners bring realistic examples of the challenges they face and, as a result, these collaborative works tend to produce results

[1]Mutation testing refers to a set of techniques that introduce artificial errors in the SUT in order to validate the efficacy of a test suite [67].

suitable for practical applications and could serve as a guideline for other, purely academic, approaches.

Naturally, not all research can be done with industrial partnerships, and in these cases there is difficulty in finding what exactly is relevant to current practitioners. One possible source of this information is grey literature: information produced by experts in a field, but without necessarily following academic guidelines, in the form of blog posts, videos, magazine articles, talks etc. Practitioners who produce grey literature can help inform researchers about the current state of practice, the main existing challenges in software development, and successful implementations of techniques (e.g. the aforementioned Netflix blog [71] which details the process of bringing a technique from a paper into their workflow).

Ultimately, the unavoidable reality is that academics and practitioners work towards different goals. Researchers are motivated and driven by publication; indeed, the point of research is to aggregate information into a body of knowledge that grows gradually over time. Historically, academia does not encourage researchers to continue working on a project after the related paper is published and "see it through" to an eventual application of a technique. In fact, the pursuit for novelty can diminish the perceived value of a researcher who is willing to perform experiments and do the additional work to implement a technique.

This is in direct contrast with the desires of industry. Few companies are willing to commit time, money and human resources into developing novel techniques that may or may not provide value or savings in the long term. Rather, they would rather adopt practices with a proven and predictable outcome.

6.1.2 CH2: Realistic Experimentation

It is clearly not possible for every research paper to feature practitioner co-authors or to rely on an industrial partnership for experimentations. Selecting the right subject for experiments is a decisive point when writing a paper about a technique. Older studies on RT would often rely on the "Siemens programs" [65], which is believed to have caused an overfitting of results to a particular kind of software [32]. More recently, the Software Infrastructure Repository (SIR) [33] (e.g. (S3)) and Defects4J [68] (e.g. [S2, S39]) have been used to similar ends. Having common subjects can provide replicability benefits when directly comparing techniques, although is not always clear if they approximate the difficulty of testing real software. Authors who are able to collaborate directly with members of industry gain an enormous advantage if they are allowed to run experiments on production code, but it is also clear that not every paper will have that opportunity.

The most obvious alternative is to use large-scale open-source software (e.g. from the Mozilla (S60) and Apache [S13, S62, S67, S65, S73] foundations) as subjects, since the communities developing these programs follow procedures much like the developers working for corporations . This is also far from trivial. The larger the software, the more time a researcher will need to dedicate in order to understand it and to adapt the technique to it, sacrificing the possibility of experimenting on a larger variety of subjects and thus again bringing the risk of overfitting. Additionally, there is no established consensus regarding which properties an open-source program must satisfy in order to be a satisfactory subject.

Alleviating this issue would require effort from both researchers and practitioners. For example, Google has an open dataset of testing results [35], and S25 combined it with one from ABB Robotics. As a result, this combined dataset has already been used by other papers covering machine learning [S53, S59, S67, S71, S78]. Two practitioners mention that "*open source code/data is not provided*" due to confidentiality reasons. In those cases, our suggestion would be to provide some opaque information regarding the system, e.g. its programming language, the number of lines of code and/or tests, how many developers work on it, how frequently is the code updated, etc. At the very least, this would help researchers choose subjects with similar characteristics.

6.1.3 CH3: Scalability

RT techniques provide the most savings when applied to large-scale software projects, which can have multiple thousands of test cases. Therefore, it is important that techniques are designed to scale up to any size of test suite, but few papers tackle this issue directly. The trouble is that scalability is very hard to measure unless multiple subjects of different sizes are used. One way to demonstrate scalability, beyond relying on industrial partners or large-scale open-source projects, is to artificially generate large datasets (e.g. [S34, S50]), which are useful from the algorithmic perspective, but might not address other issues that arise in large-scale software development. It is also worth mentioning that many RT techniques can become *disadvantageous* when applied to small test suites, as the cost of running the technique does not outweigh the savings in testing time. So selecting the size of the experiment subject is important both to highlight the scalability of the tool in large software and also to consider whether the necessary overhead is a deal-breaker on small or medium projects.

6.1.4 CH4: Relevance of Metrics

Section 3.4.1 shows that a wide variety of metrics has been used to evaluate the effectiveness of RT techniques. Some are used almost universally for a certain kind of challenge (e.g. APFD for TCP), while others have nearly no presence beyond the paper that introduced them.

The abundant use of APFD and its variants indicate that, at least among researchers, there is a consensus of its utility and importance when evaluating TCP approaches, although the usage of specific variants might hamper that benefit. At the same time, it is not clear that a technique optimized for only APFD is sufficient to satisfy the needs of software developers in practice. Still, APFD has been in use for over 20 years and it cannot simply be dismissed: at the very least it provides an agreed-upon method of directly comparing different techniques.

For the cases of TCS and TSR, there is less controversy on what are the most important metrics; reduction rate and fault detection loss appear to be the consensus among researchers, and there are fewer novel and single-use metrics. As an example, S68 interviewed practitioners at Microsoft before deciding on their TCS metrics, obtaining three main targets: reduction of cost, reduction of time and the failure detection rate. We can observe in Section 3.4.1 that these concerns are reasonably addressed by TCS techniques, although researchers still appear to prioritize reducing the selected set rather than ensuring all failures are detected.

The metrics of applicability and diagnosability (S46, S60) are interesting propositions that consider other degrees of usefulness of a tool to developers. Their existence indicates that some researchers still believe there is room for improved metrics that, perhaps, better map the requirements of real-world software, although these are rarely found in the literature.

Furthermore, ease-of-use is an important point to consider and, as far as we could detect, there is no established method of measuring it.

One practitioner stated: "*I don't think that academic tools are the best in a professional environment, I prefer commercial tools,*" implying they believe academics are not measuring the results that matter most to them. Indeed, managers allocating development funds will usually focus on the dollar savings a technique can bring, regardless of its theoretical effectiveness in fault-finding (as mentioned by respondent author #43: "*the cost associated with the 1% bugs that were missed is too high*").

6.1.5 CH5: Converting Research into Usable Tools

When techniques are designed in an academic context, they are normally developed as proof-of-concept works. That is, the purpose is to show that the technique works and provides significant results according to some metrics. However, this leads to two issues: either primary studies do not make their solution available for implementation, as we discussed in Section 3.4.2, or their experiments do not thoroughly consider practical concerns such as efficiency or the data requirements of a proposed approach. Finally, what seems to matter the most is time and budget for developing a tool. Papers are usually written targeting a hard deadline and their prototypes often do not see further work past publication. It is inevitable that researchers will move on to new challenges, but their contribution would be amplified if the tool is, at the very least, open-source and well-documented so that other interested parties can continue the work in the future if desired.

If an RT technique is implemented as a prototype that is shown to work on a certain kind of software, it is much easier to get the attention from a practitioner and convert the solution into something used in practice. If feasible, an available prototype with solid documentation and usage instructions can be valuable both for study replicability and as a way to get developers interested in using it. That said, the responsibility of developing fully functional tools should not fall solely upon researchers. One practitioner stated that "*[RT tools] need full security screening*", and other said "*it requires an adaptation*"; these steps are not actionable by researchers in isolation. As industry stands to benefit from scientific advances, it should be in its best interest to promote and fund the collaborations needed to continue development of promising prototypes.

This strongly relates to **CH1** regarding the motivations of academics and practitioners. Even if a researcher has the desire to see a technique through to its applicability, or even just to provide a robust source code for the implementation of an approach, there is little incentive from the academic side for doing so. Certain software engineering conferences have started to encourage or even require the inclusion of replication packages and source code of studies with an empirical component, which is a step in the right direction.

6.1.6 CH6: Absence of TSR/TSA

Out of 79 papers, only 8 are about TSR and, surprisingly, only one covers TSA (S17). 60% of the surveyed practitioners claim that "creating or updating tests" is a major challenge in real-world RT, so the desire for TSA exists and there appears to be ample room for experimenting with new approaches and metrics. However, most practitioners interviewed at the industry partner claim that increasing test scope is not a major concern, as even the manually-written tests can be too many.

On the other hand, they do mention the difficulty of refactoring and removing obsolete test cases as a frequent challenge, which aligns with 47% of the surveyed practitioners. TSR could prove valuable to testers who need to manage ever-growing test suites and hardly find the time to manually assess tests that are obsolete or in need of refactoring. In reality, this is often addressed by simply increasing the capacity of testing servers, but this is a costly and non-scalable solution.

This assessment of TSR and TSA techniques indicate an opportunity for researchers to develop novel methods for these challenges and to progress in directions that are in need of improvement in software development workflows.

6.1.7 CH7: Clarity of Target

Several of the papers we reviewed don't clearly state key characteristics of their SUT, such as its programming language or its scale (either in lines of code or test cases). For practitioners and other researchers to consider a paper worthy of investigation, it is important to know for which kind of system a piece of research was designed.

As mentioned in Section 3.4.2, few RT techniques are language-agnostic and many do not inform the target language at all. Similarly, the type of software (web, mobile, embedded, distributed, etc.) or its development paradigm are important factors to mention, seen in studies such as S41 for web services and S59 for software developed and delivered through continuous integration. Not every tool can be used in any type of software, and it is likely that specific types of software might require specific solutions, so it is important to state the particularities of certain subject programs. This is akin to the point of "context factors" brought up by Ali et al. [4], which helps to alleviate the issue by introducing a base taxonomy that can be used to categorize techniques.

Critically, there is often ambiguity on the very definition of test case. Software testing can include unit tests, integration tests, multi-component tests, system tests, end-to-end tests and so forth. Most papers do not make it explicit which layer of testing it is addressing. While it can sometimes be inferred with some domain knowledge, it is difficult to be certain for most readers. This information would be valuable for interested practitioners and also for researchers who are looking to identify gaps in the literature. On top of that, some papers use the term "test case" to refer to test methods, while others use it when referring to test classes/files (which contain several test methods), so the granularity of the technique is not always clear, and this can impact both effectiveness and efficiency analysis. This challenge can be solved by having a paragraph dedicated to explicitly describing the properties and context factors of the experiment subjects.

6.1.8 CH8: Skepticism

In general, there is some degree of skepticism from practitioners regarding automated solutions. For example, some interviewees at the industrial partner expressed concerns that a TCS solution might leave out an important test, or that TCP algorithms might always prioritize the same tests. Their intuition is that performing selection (or even prioritization) manually allows for easier troubleshooting in case something goes wrong. As for what can go wrong, the fear is to detect a fault later on, only to find out the test to detect it already existed, but was not selected or prioritized by the automated tool.

This calls back to a comment made by one of the responding authors in Section 3.4.2: even if 99% of faults are detected at a fraction of the testing costs, the remaining 1% of slips is unnerving to the people responsible for the tests. And even if a tool is shown to be safe, there is still a feeling that there *might* be a slip that will only be revealed much later than it should have been.

Regarding TSR, most developers are also opposed to outright removing tests, but are willing to put these tests in less-frequent rotations, e.g. weekly instead of daily. Finally, for TSA the practitioners are also doubtful that automatically generated or enhanced tests are as good as human-designed ones, but wouldn't be against to trying it if the need arises.

6.1.9 CH9: Data Quality and Availability

Another point raised by the interviewed practitioners at the industrial partner is their own ability to provide the data required by an automated tool. History-based approaches, for example, are more effective when there are many test logs archived for analysis, but in reality these logs are only stored temporarily.

Furthermore, even if the data exists it might not be of sufficient quality to, for example, serve as a training set for a machine learning model. This extends to the test code itself, as highlighted by one of the interviewees: "*researchers might assume a certain level of quality or standardization among test cases, but in reality test design is often bad*" (R8). If the existing test cases are not standardized and high-quality, automated attempts to improve them are unlikely to yield the desired benefits.

6.1.10 CH10: Communication

The main challenge, which connects most of the previous ones, is communication. Researchers and practitioners both lead busy lives, focusing on their day-to-day affairs, and ultimately communication between the two realms suffers.

There are some steps that can be taken to improve this. Companies can start by having round-table discussions on recent research publications (e.g. the Google Journal Club [103]) and, if possible, they should invite the author(s) to participate. On the other side, universities can host lectures by practitioners in addition to other researchers. This can start small — find people in the same city, perhaps alumni of the university, who are working on something interesting and have a conversation.

Out of the practitioners we surveyed for Chapter 3, 56% claimed they keep contact with a friend or colleague who is a researcher in Software Engineering. After all, most academics have interacted with people who are currently practitioners during their education process, and vice-versa. This means that both sides have an opportunity to network and communicate beyond their current professions, giving each other ideas of what is currently relevant in industrial software development and what is the latest state-of-the-art in academic research. Unfortunately, the practitioners interviewed for Chapter 5 have mostly lost contact with their academic colleagues and today struggle to keep up with the advancement of research.

It can be a daunting idea to catch up to latest research trends, so larger companies could consider having employees dedicated to understanding the internal processes and challenges while searching for collaborations with academics, or at least promote internal reading club sessions. Many researchers would be thrilled to receive a message inviting them for a joint effort with palpable outcomes.

6.2 Threats to Validity

The list of challenges assembled in Chapter 6 is based on our own observations of the current state of industry-academia relations from a multitude of sources: the literature, communications with other authors, feedback from practitioners and our own experiences developing techniques. It is not meant to be a comprehensive and end-all checklist of challenges to solve, as other researchers following different sources would likely come to a divergent set of conclusions. That said, we believe that most researchers performing a similar study would agree upon the majority of the listed challenges. To the extent of our knowledge, these challenges are real and in need of further study, but there is no guarantee that addressing each one of them will solve all the problems with software engineering research.

Chapter 7

Live Repository

In the analysis of recent secondary studies, we notice a gap of one or even two years elapses between the covered period of literature and the year the review appears. This is understandable, because the authors may need substantial time for analyzing the selected studies and then writing the article, followed by several months for the peer review process. Moreover, even if this temporal gap is reduced to a minimum, as long as the investigated topic remains active, new primary studies will always appear, soon distancing any systematic literature review (SLR) from the status of literature. If the purpose of an SLR is to provide an up-to-date summary of existing work on a topic, frequent updates are required to include newly appearing relevant studies. Also, it can happen that the original research questions and findings lose relevance, or are superseded by newer results. As previously questioned in other disciplines [44], Mendes et al. [96] have recently investigated the issue of SLRs in software engineering becoming obsolete and of when and how they would need to be updated. Specifically, in line with [44], they conclude that SLRs should be updated based on two conditions: *i)* new relevant methods, studies or information become available; or *ii)* the adoption or inclusion of previous and new research cause an impact to the findings, conclusions or credibility of the original SLR.

Thus, it is inevitable that a literature review becomes outdated after some time, as new research comes out that cannot be included in the published paper, limiting the long-term value of the work, since the text will no longer reflect the ongoing research in the field. In consideration of these challenges, rather than providing a static list of the studies found while conducting the review, along with this work we release an *open and updatable* *repository*, which is an integral contribution of this study, and the review from Chapter 3 is enhanced to become a *live* systematic review[1].

Aggregating long-term value to this work, we have made the list of papers and the information extracted from them available as an online live repository[2]. The papers listed in Table 3.3 and discussed throughout Chapter 3 serve as the starting point for a bibliography that will continue to grow year over year, through updates to the review and submissions by authors. We hope this website will serve as reference to anyone who is interested in practical applications of regression testing techniques in the coming years.

The intent is to invite the community to contribute with newly published works with approaches to RT that are relevant in IR&A. Additionally, it could also be incremented with works already published but that, for some reason, escaped our selection. This will allow us to keep track of newly published studies, as well as recover papers that had theoretical motivations and unclear applicability, but that eventually provided benefits in practice. At periodic intervals (planned to be once per year), we will check new results for queries and suggestions from the community and decide how to update the collection of studies. With this, another SLR covering the same topic should not be needed within the foreseeable future.

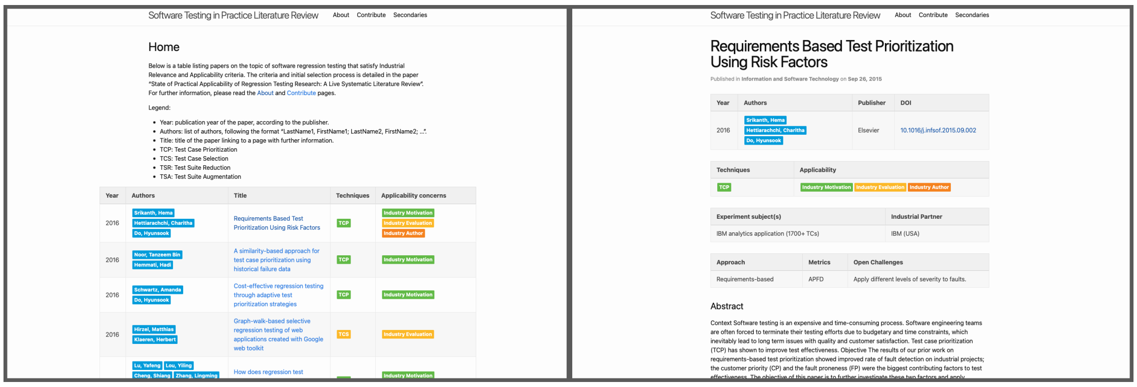
The repository also contains a separate section for relevant literature reviews, initially populated by the reviews mentioned in Section 3.1. This provides a starting point for new researchers and a place to gather the overarching themes of the field.

7.1 Implementation

The website is implemented using the Jekyll static website generator and the associated Liquid templating language. The data that populates the webpages is initially extracted from the spreadsheets originally used to manage the SLR from Chapter 3. The spreadsheets are exported as CSV files, then converted to JSON files for better readability, also making it easier for each paper to have its own data file. The fields for each paper mostly

[1]We notice that our concept of a “live” systematic review, while inspired by similar aims, should not be confused with the much more formalized approach for conducting *living systematic reviews* recently adopted in medicine, as illustrated by https://community.cochrane.org/review-production/production-resources/living-systematic-reviews.

[2]Available at: https://renangreca.github.io/literature-repository.



Software Testing in Practice Literature Review About Contribute Secondaries

Home

Below is a table listing papers on the topic of software regression testing that satisfy Industrial Relevance and Applicability criteria. The criteria and initial selection process is detailed in the paper "State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review". For further information, please read the About and Contribute pages.

Legend:

- Year: publication year of the paper, according to the publisher.
- Authors: list of authors, following the format "LastName1, FirstName1; LastName2, FirstName2; …".
- Title: title of the paper linking to a page with further information.
- TCP: Test Case Prioritization
- TCS: Test Case Selection
- TSR: Test Suite Reduction
- TSA: Test Suite Augmentation

Year	Authors	Title	Techniques	Applicability concerns
2016	Srikanth, Hema Hettiarachchi, Charitha Do, Hyunsook	Requirements Based Test Prioritization Using Risk Factors	TCP	Industry Motivation Industry Evaluation Industry Author
2016	Noor, Tanzeem Bin Hemmati, Hadi	A similarity-based approach for test case prioritization using historical failure data	TCP	Industry Motivation
2016	Schwartz, Amanda Do, Hyunsook	Cost-effective regression testing through adaptive test prioritization strategies	TCP	Industry Motivation
2016	Hirzel, Matthias Klaeren, Herbert	Graph-walk-based selective regression testing of web applications created with Google web toolkit	TCS	Industry Evaluation
	Lu, Yafeng Lou, Yiling Cheng, Shiang Zhang, Lingming	How does regression test		Industry Motivation

Software Testing in Practice Literature Review About Contribute Secondaries

Requirements Based Test Prioritization Using Risk Factors

Published in **Information and Software Technology** on **Sep 26, 2015**

Year	Authors	Publisher	DOI
2016	Srikanth, Hema Hettiarachchi, Charitha Do, Hyunsook	Elsevier	10.1016/j.infsof.2015.09.002

Techniques	Applicability
TCP	Industry Motivation Industry Evaluation Industry Author

Experiment subject(s)	Industrial Partner
IBM analytics application (1700+ TCs)	IBM (USA)

Approach	Metrics	Open Challenges
Requirements-based	APFD	Apply different levels of severity to faults.

Abstract

Context Software testing is an expensive and time-consuming process. Software engineering teams are often forced to terminate their testing efforts due to budgetary and time constraints, which inevitably lead to long term issues with quality and customer satisfaction. Test case prioritization (TCP) has shown to improve test effectiveness. Objective The results of our prior work on requirements-based test prioritization showed improved rate of fault detection on industrial projects; the customer priority (CP) and the fault proneness (FP) were the biggest contributing factors to test effectiveness. The objective of this paper is to further investigate these two factors and apply

FIGURE 7.1: Screenshots from the live repository. From left to right: 1) the main page listing the included papers; and 2) a single paper's page (S1 used as example).

correspond to the data extraction table (Table 3.4). The website itself is open-source and hosted via GitHub Pages.

7.2 Longevity

The main challenge is how to keep this repository alive in the long term. It is unfeasible for us to add a relevant paper to the repository as soon as it is published, so our plan is to update the list in a yearly basis, re-running the query and screening steps detailed in Section 3.3. That way, we can at least assure the most recent paper included is no more than one year old. We are also looking into the possibility of getting automatic notifications when a paper that satisfies certain criteria is published in an online library. For now, this work is done by the original authors of the literature review; according to future necessities, we will appoint other researchers or graduate students to help with the process. In addition, we also encourage authors to submit their own work by filling a form linked on the website[3].

Along with adding more papers to the list, whenever possible we will also add additional information about the ones already included. This can be triggered by a direct contact by one of the authors, some observed update with the state of that piece of research, or by suggestion of an engaged reader.

At some point after a certain number of years, the definitions we selected for including a paper in the repository will likely need be adjusted. Whenever an author submits a paper, we will use the opportunity to consider whether or not the paper itself is a good fit for the

[3]Available at: https://forms.gle/CWGjMrxCe1bnKikk8.

repository, but also if there are new trends that our existing selection process does not account for. There will eventually be a point in the future when the industry/academia landscape has shifted and this study will no longer be needed. When that happens, we will discuss the possibility of freezing the repository and stopping further expansions.

Aside from newer papers, it is always possible that we have missed some relevant papers for a variety of reasons, so the live repository is another way of mitigating that risk. It is impossible to provide a complete and definitive overview of any field, but we believe that a live repository is the closest approximation that can be expected.

Chapter 8

Conclusion

This thesis provides data, insight and discussions centered around regression testing challenges and the overall goal of test suite orchestration. Based on an investigation of the literature, a proof-of-concept implementation of orchestration, and *in loco* interviews with software testing practitioners, the result is a thorough investigation on the potential benefits of orchestration as well as the challenges that are currently faced by researchers and practitioners who wish to implement these techniques on real-world software.

In Chapter 3, we provide a comprehensive literature review of academic papers with proven or apparent potential for real-world software usage. We have shown that applicability is a growing concern in research but, when it comes to actually applying the techniques with an industrial partner, many challenges arise and very few approaches feature lasting longevity.

Chapter 4 shows an example of how test suite orchestration can be used to enhance software testing procedures, incorporating change-based TCS and similarity-based TCP approaches as an initial step towards the goal of full orchestration. Analysis is performed on the approach, which exhibits promise both in terms of effectiveness and of efficiency. The challenge of full orchestration is left open, with a descriptive example of how multiple RT could be combined.

First-hand accounts of the state of software testing in practice are quoted and discussed in Chapter 5. Thanks to the opportunity of collaborating with an industrial partner, we were able to understand the processes and challenges that exist today in software testing.

Reports indicate that testing is far from being a solved problem, showing there is ample opportunity for new techniques to improve workflows and reduce costs in the industry.

The results of Chapter 3, Chapter 4 and Chapter 5 are used to form a list of essential challenges in Chapter 6 that researchers should consider and address if their goal is to provide techniques that are usable by practitioners.

To add longevity to the relevance of this work, Chapter 7 describes the process of converting the literature review from Chapter 3 into a *live* systematic literature review. This repository of studies is available online and provides a starting point and bibliography for researchers to wish to pursue this research direction.

Ultimately, many of the challenges we investigate are as of yet unresolved, remaining a fertile direction for future research. It is clear that software engineering academics and practitioners have distinct goals when it comes to the work they do to improve the quality and efficiency of testing, but there is also plenty of evidence indicating that it is possible to align these motivations and produce even better techniques as a result.

The closing message of this thesis is one of community and collaboration: in order to ensure that software testing research leads to tangible improvements to the lives of developers and users of computer software, it is essential that academia and industry form a tighter bond to explore each other's advantages and cover their weaknesses. It is our sincere belief that such collaboration is possible and will at some point be achieved. We hope that the conclusion and publication of this study will help this happen sooner rather than later.

This chapter contains one additional section, discussing the papers that were submitted and/or published during the development of this PhD thesis.

8.1 Publications

During the development of this thesis, the following papers were accepted for publication at software engineering conferences or journals. We note that [P1] is not related to the topic of this thesis, but was a collaborative effort with other colleagues. [P2] is mostly related to the work seen in Chapter 4, while [P3] is the journal version of our systematic literature review, and relates to Chapters 3, 6 and 7 of this thesis. [P4] is a short paper containing a formal definition of test suite orchestration including examples and ongoing challenges.

[P1] M. T. Rossi, R. Greca, L. Iovino, G. Giacinto, and A. Bertolino. "Defensive Programming for Smart Home Cybersecurity". In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2020, pp. 600–605. DOI: 10.1109/EuroSPW51379.2020.00087.

[P2] R. Greca, B. Miranda, M. Gligoric, and A. Bertolino. "Comparing and combining file-based selection and similarity-based prioritization towards regression test orchestration". In: *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. Pittsburgh, USA: ACM/IEEE, May 2022. DOI: 10.1145/3524481.3527223.

[P3] R. Greca, B. Miranda, and A. Bertolino. "State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review". In: *ACM Computing Surveys* (Jan. 2023). DOI: 10.1145/3579851.

[P4] R. Greca, B. Miranda, and A. Bertolino. "Orchestration Strategies for Regression Test Suites". In: *Proceedings of the 4th ACM/IEEE International Conference on Automation of Software Test*. Melbourne, Australia: ACM/IEEE, May 2023.

We are preparing two additional papers for submission based on different parts of this work.

1. One industry-track paper containing the interviews and observations from Chapter 5 and comparisons with the state of practice in other software companies; and

2. One extension of the work from Chapter 4 including additional components in the orchestration.

Appendix A

Surveys

A.1 E-mail Template Sent to Authors of Surveyed Papers

Dear {{Name}},

I am Renan Greca, a PhD student at the Gran Sasso Science Institute, under the supervision of Professors Antonia Bertolino and Breno Miranda. Currently, we are performing a systematic literature review on the topic of software regression testing, focusing on the real-world relevance and applicability of techniques proposed in academia.

The following paper(s) of your authorship has(have) been selected for our study:

{{Paper 1}}

{{Paper 2}}

{{Paper 3}}

If possible, we would like some information about the outcome of your research after the publication of the paper(s). We have a few questions regarding your work:

1. Is there a functional version of your technique (tool, prototype, source code, etc.) available online? If so, please share with us the URL.

2. Was there an attempt to implement your technique in industrial or large open-source software? Is the technique currently in use with the software?

3. If the technique was put into practice, were the metrics used in the paper relevant for the technique's applicability? If not, were there other metrics that proved to be useful?

Furthermore, please let us know whether we can link your comments directly to your work in our literature review or, if not, if we can mention these comments without referencing which author gave them.

Finally, please inform us if you are interested and available for further contact related to the outcome of this research at a later date.

We kindly ask you to provide answers by September 23, or to let us know by then if more time is needed.

If you have received a similar email before, it is because we are updating our study and we have included another paper from your authorship, so please respond regarding these new paper(s).

Best regards,

Renan Greca (GSSI)

Breno Miranda (UFPE)

Antonia Bertolino (ISTI-CNR)

A.2 Questionnaire Sent to Practitioners During Literature Review

Nº	Question
1	**Survey for software testing practitioners** We are researchers studying the state of collaboration between academic researchers and industrial practitioners, considering the topic of software regression testing. This survey has the goal of better understanding practitioners' perspective on past and ongoing usage of methods, techniques and tools initially developed in an academic context. Fill this form if you are directly or indirectly involved with software testing at the company you work for. If you have colleagues that also work with software testing, please share the form with them. Please submit your responses by deadline. By answering these questions, you provide consent to the authors to handle this data and use it for research purposes. The data will not be used for any other purposes. If you have questions or need clarifications, feel free to contact the authors: Antonia Bertolino (antonia.bertolino@isti.cnr.it) Breno Miranda (bafm@ufpe.br) Renan Greca (renan.greca@gssi.it)
1.1	**Please tell us what are the most common pain points when it comes to running regression testing suites.** *Multiple choices can be selected.* a) Detecting failures b) Flaky (unreliable) tests c) Running time of the test suite d) Creating new tests or updating existing ones e) Cleaning up obsolete tests Other...
1.2	**Do you know of, or have you used, any tool for improving regression testing with roots in academic research?** By regression testing, we mean running and re-running tests that cover previously-existing functionality. Examples of techniques that can help are test case selection, test case prioritization, test suite augmentation or test suite reduction. *A single choice can be selected.* a) I have used one or more tools. (go to Section 2) b) I know about one or more, but never used them. (go to Section 3) c) I don't know about these tools. (go to Section 4)

Table A.1: Questionnaire for practitioners, Section 1.

Nº	Question
2	**About the tools you have used.** In the previous page, you answered that you have used tools to improve regression testing. Please tell us about them.
2.1	**Which tools have you used?** *Full body answer.*
2.2	**Please mark how much you agree with the following statements.** If you have used multiple tools, you may offer general sentiments about them collectively. *Possible values: Strongly disagree, Disagree, Neutral, Agree, Strongly agree.* You had a positive experience using the tool. The tool was easy to learn and use. The tool satisfied what you were looking for. The tool was designed targeting your company. The tool is currently still in use by you or your colleagues.
2.3	**Feel free to share further thoughts about your experience with the tool(s).** *Full body answer.*

Table A.2: Questionnaire for practitioners, Section 2.

Nº	Question
3	**About the tools you know.** In the previous page, you answered that you know about tools to improve regression testing. Please tell us which ones.
3.1	**Which tools do you know about?** *Full body answer.*

Table A.3: Questionnaire for practitioners, Section 3.

Nº	Question
4	**Collaboration with academia.** This page focuses on the possibilities of collaboration between practitioners and academics.
4.1	**Please mark how much you agree with the following statements.** *Possible values: Strongly disagree, Disagree, Neutral, Agree, Strongly agree.* You stay informed about recent research in Software Engineering. A piece of Software Engineering research has directly improved your workflow. You keep contact with a friend or colleague doing research in Software Engineering. There have been attempts of collaboration between members of your company and academia. Your company provide open datasets or open-source software that could be used as research subjects.
4.2	**If you haven't already, what would convince you to try using a technique proposed in academia in your workflow? After trying it, what should it satisfy to become a permanent part of your process?** *Full body answer.*
4.3	**If you have materials to share, such as results of previous collaborations or open-source data provided by your company, please put links below.** *Full body answer.*
4.4	**Feel free to share any other comment you might have regarding industry-academia collaboration for regression testing.** *Full body answer.*

Table A.4: Questionnaire for practitioners, Section 4.

Nº	Question
5	**Demographics** This final section is optional, but would help improve the quality of our results.
5.1	**Company** The company for which you work. *Full body answer.*
5.2	**Country** The country where you work. *A single choice can be selected, from a list of countries.*
5.3	**Role** What title best describes your role at the company? a) Software Engineer b) Software Tester c) Product Manager d) Test Manager Other...
	Thank you for your time!

TABLE A.5: Questionnaire for practitioners, Section 5.

A.3 Questions for Practitioners at the Industrial Partner

Nº	Question
1	**Roles & education**
1.1	What is your role at the company and what are the main ways you interact with the regression testing suite?
1.2	What is your education in testing?
2	**Are these common issues when dealing with the test suite?**
2.1	Correctly detecting failures (avoiding slips)
2.2	Flaky (unreliable tests)
2.3	Running time of the test suite
2.4	Creating or updating tests (increasing test scope)
2.5	Cleaning up obsolete tests (reducing test scope)
3	**Current practices**
3.1	Are you familiar with these terms? Do you use them?
3.1.1	Test Case Selection
3.1.2	Test Case Prioritization
3.1.3	Test Suite Reduction
3.1.4	Test Suite Augmentation
3.2	Does your team separate test execution into shorter, frequent runs and longer overnight runs?
3.3	Are there people in your team solely responsible for testing, or is it a job of the developer?
3.4	Do you believe the current test suite is effective in avoiding fault propagation upon delivery of an update?
3.5	How often are there failing tests from the overnight run in the morning?
3.6	How often is a test failure a sign of actual SUT error vs. poor test quality vs. environment misconfiguration?
3.7	What do you do when the same test fails many days in a row?
3.8	What is the standard procedure for dealing with potentially flaky tests?
3.9	Is there a policy in place for refactoring and/or removing aging tests?
3.10	Do you have an idea of how much money or how much electricity is needed to run a test suite? Is this concern increasing or reducing over the years?
3.11	Do you personally ever review test code? How much trust do you have that it's being done correctly and at high quality?
4	**Interaction with academia**
4.1	Do you stay informed about recent research in Software Engineering?
4.2	Has a piece of Software Engineering research directly improved your workflow?
4.3	Do you keep contact with a friend or colleague doing research in Software Engineering?
4.4	Before me, have there been attempts of collaboration between members of your team and academic researchers?
4.5	What does a tool need to do/have in order to convince you to use it as a permanent part of your workflow?
4.6	What are the risks and benefits of using a new tool for regression testing improvement?
4.7	Have you ever used an open-source regression testing approach as-is, or is it always necessary to re-engineer it for usability/security/compatibility reasons?
4.8	If I came to you saying "I have this algorithm that would help your testing workflow", what would be your reaction?

Table A.6: List of questions asked during interviews.

Bibliography

[1] M. Abdelkarim and R. ElAdawi. "TCP-Net: Test Case Prioritization using End-to-End Deep Neural Networks". en. In: *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Valencia, Spain: IEEE, Apr. 2022, pp. 122–129. ISBN: 978-1-66549-628-5. DOI: 10.1109/ICSTW55395.2022.00034. URL: https://ieeexplore.ieee.org/document/9787970/ (visited on 09/14/2022).

[2] M. S. Abdul Manan, D. N. Abang Jawawi, and J. Ahmad. "A Systematic Literature Review on Test Case Prioritization in Combinatorial Testing". In: *2021 The 5th International Conference on Algorithms, Computing and Systems*. 2021, pp. 55–61.

[3] F. S. Ahmed, A. Majeed, T. A. Khan, and S. N. Bhatti. "Value-based cost-cognizant test case prioritization for regression testing". In: *Plos one* 17.5 (2022), e0264972.

[4] N. bin Ali, E. Engström, M. Taromirad, M. R. Mousavi, N. M. Minhas, D. Helgesson, S. Kunze, and M. Varshosaz. "On the search for industry-relevant regression testing research". In: *Empirical Software Engineering* 24.4 (Aug. 2019). ISBN: 1066401896 Publisher: Springer New York LLC, pp. 2020–2055. ISSN: 15737616. DOI: 10.1007/s10664-018-9670-1.

[5] H. Aman, Y. Tanaka, T. Nakano, H. Ogasawara, and M. Kawahara. "Application of Mahalanobis-Taguchi Method and 0-1 Programming Method to Cost-Effective Regression Testing". In: *Proceedings - 42nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2016*. 2016, pp. 240–244. DOI: 10.1109/SEAA.2016.29.

[6] M. Azizi and H. Do. "ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development". en. In: *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE).* Memphis, TN: IEEE, Oct. 2018, pp. 144–154. ISBN: 978-1-5386-8321-7. DOI: 10.1109/ISSRE.2018.00025. URL: https://ieeexplore.ieee.org/document/8539077/ (visited on 05/11/2021).

[7] T. Bach, A. Andrzejak, and R. Pannemans. "Coverage-Based Reduction of Test Execution Time: Lessons from a Very Large Industrial Project". In: *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2017.* 2017, pp. 3–12. DOI: 10.1109/ICSTW.2017.6.

[8] M. Bagherzadeh, N. Kahani, and L. Briand. "Reinforcement Learning for Test Case Prioritization". en. In: *IEEE Transactions on Software Engineering* 48.8 (Aug. 2022), pp. 2836–2856. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2021.3070549. URL: https://ieeexplore.ieee.org/document/9394799/ (visited on 09/14/2022).

[9] A. Bajaj and O. P. Sangwan. "A Survey on Regression Testing Using Nature-Inspired Approaches". en. In: *2018 4th International Conference on Computing Communication and Automation (ICCCA).* Greater Noida, India: IEEE, Dec. 2018, pp. 1–5. ISBN: 978-1-5386-6947-1. DOI: 10.1109/CCAA.2018.8777692. (Visited on 11/08/2020).

[10] A. Bajaj and O. P. Sangwan. "A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms". en. In: *IEEE Access* 7 (2019), pp. 126355–126375. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2938260. (Visited on 11/08/2020).

[11] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo. "Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration". en. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* Seoul South Korea: ACM, June 2020, pp. 1–12. ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380369. URL: https://dl.acm.org/doi/10.1145/3377811.3380369 (visited on 09/14/2022).

[12] V. Blondeau, A. Etien, N. Anquetil, S. Cresson, P. Croisy, and S. Ducasse. "Test case selection in industry: an analysis of issues related to static approaches". In: *Software Quality Journal* 25.4 (Dec. 2017). Publisher: Springer New York LLC, pp. 1203–1237. ISSN: 15731367. DOI: 10.1007/s11219-016-9328-4.

[13] G. Buchgeher, C. Klammer, W. Heider, M. Schuetz, and H. Huber. “Improving testing in an enterprise SOA with an architecture-based approach”. In: *Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016*. 2016, pp. 231–240. DOI: 10.1109/WICSA.2016.24.

[14] B. Busjaeger and T. Xie. “Learning for test prioritization: An industrial case study”. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Vol. 13-18-November-2016. 2016, pp. 975–980. DOI: 10.1145/2950290.2983954.

[15] I. do Carmo Machado, J. D. McGregor, Y. C. Cavalcanti, and E. S. De Almeida. “On strategies for testing software product lines: A systematic literature review”. In: *Information and Software Technology* 56.10 (2014), pp. 1183–1199.

[16] C. Catal. “On the application of genetic algorithms for test case prioritization: a systematic literature review”. In: *Proceedings of the 2nd international workshop on evidential assessment of software technologies*. 2012, pp. 9–14.

[17] C. Catal and D. Mishra. “Test case prioritization: a systematic mapping study”. In: *Software Quality Journal* 21.3 (2013), pp. 445–478.

[18] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric. “Regression test selection across JVM boundaries”. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Vol. Part F130154. 2017, pp. 809–820. DOI: 10.1145/3106237.3106297.

[19] A. Celik, Y. C. Lee, and M. Gligoric. “Regression Test Selection for TizenRT”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 845–850. ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3275527. URL: https://biblioproxy.cnr.it:2481/10.1145/3236024.3275527.

[20] J. Chen, Y. Lou, L. L. Zhang, J. Zhou, X. Wang, D. Hao, and L. L. Zhang. “Optimizing Test Prioritization via Test Distribution Analysis”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 656–667.

ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3236053. URL: https://biblioproxy.cnr.it:2481/10.1145/3236024.3236053.

[21] Y. Chen, N. Chaudhari, and M.-H. Chen. “Context-Aware Regression Test Selection”. en. In: *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. Taipei, Taiwan: IEEE, Dec. 2021, pp. 431–440. ISBN: 978-1-66543-784-4. DOI: 10.1109/APSEC53868.2021.00050. URL: https://ieeexplore.ieee.org/document/9711972/ (visited on 09/14/2022).

[22] Y. Chen and M.-H. Chen. “Multi-Objective Regression Test Selection”. en. In: 2021, pp. 105–92. DOI: 10.29007/7z5n. URL: https://easychair.org/publications/paper/pgdP (visited on 09/14/2022).

[23] Z. Chi, J. Xuan, Z. Ren, X. Xie, and H. Guo. “Multi-Level Random Walk for Software Test Suite Reduction”. en. In: *IEEE Computational Intelligence Magazine* 12.2 (May 2017), pp. 24–33. ISSN: 1556-603X. DOI: 10.1109/MCI.2017.2670460. URL: http://ieeexplore.ieee.org/document/7895279/ (visited on 11/22/2021).

[24] H. Cibulski and A. Yehudai. “Regression test selection techniques for test-driven development”. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 115–124.

[25] T. Çıngıl and H. Sözer. “Black-box Test Case Selection by Relating Code Changes with Previously Fixed Defects”. en. In: *The International Conference on Evaluation and Assessment in Software Engineering 2022*. Gothenburg Sweden: ACM, June 2022, pp. 30–39. ISBN: 978-1-4503-9613-4. DOI: 10.1145/3530019.3530023. URL: https://dl.acm.org/doi/10.1145/3530019.3530023 (visited on 09/14/2022).

[26] J. Cohen. “A coefficient of agreement for nominal scales”. In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46.

[27] D. Correia, R. Abreu, P. Santos, and J. Nadkarni. “MOTSD: A multi-objective test selection tool using test suite diagnosability”. In: *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 1070–1074. DOI: 10.1145/3338906.3341187.

[28] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino. “Scalable Approaches for Test Suite Reduction”. en. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 419–

429. ISBN: 978-1-72810-869-8. DOI: 10.1109/ICSE.2019.00055. URL: https://ieeexplore.ieee.org/document/8812048/ (visited on 09/06/2019).

[29] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry. “A snowballing literature study on test amplification”. In: *Journal of Systems and Software* 157 (2019), p. 110398.

[30] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche. “Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system”. In: *Software Testing, Verification and Reliability* 25.4 (2015), pp. 371–396.

[31] Dictionary.com. *“Effectiveness” vs. “Efficacy” vs. “Efficiency”: When To Use Each Word For The Best Results.* URL: https://www.dictionary.com/e/effectiveness-vs-efficacy-vs-efficiency-when-to-use-each-word-for-the-best-results/.

[32] H. Do. “Recent Advances in Regression Testing Techniques”. en. In: *Advances in Computers*. Vol. 103. Elsevier, 2016, pp. 53–77. ISBN: 978-0-12-809941-4. DOI: 10.1016/bs.adcom.2016.04.004. (Visited on 11/08/2020).

[33] H. Do, S. Elbaum, and G. Rothermel. “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact”. In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435.

[34] R. Eda and H. Do. “An efficient regression testing approach for PHP Web applications using test selection and reusable constraints”. en. In: *Software Quality Journal* 27.4 (Dec. 2019), pp. 1383–1417. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/s11219-019-09449-2. URL: http://link.springer.com/10.1007/s11219-019-09449-2 (visited on 05/11/2021).

[35] S. Elbaum, A. Mclaughlin, and J. Penix. *The Google Dataset of Testing Results.* https://code.google.com/p/google-shared-dataset-of-test-suite-results. Accessed on 11/10/2022. 2014. (Visited on 10/11/2022).

[36] S. Elbaum, G. Rothermel, and J. Penix. “Techniques for improving regression testing in continuous integration development environments”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 235–245.

[37] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer. “Empirically evaluating readily available information for regression test optimization in continuous integration”. en. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Denmark: ACM, July 2021, pp. 491–504. ISBN: 978-1-4503-8459-9. DOI: 10.1145/3460319.3464834. URL: https://dl.acm.org/doi/10.1145/3460319.3464834 (visited on 09/14/2022).

[38] E. Engström and P. Runeson. “A qualitative survey of regression testing practices”. In: *International Conference on Product Focused Software Process Improvement*. Springer. 2010, pp. 3–16.

[39] E. Engström, P. Runeson, and M. Skoglund. “A systematic review on regression test selection techniques”. In: *Information and Software Technology* 52.1 (2010), pp. 14–30.

[40] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke. “Empirical evaluation of pareto efficient multi-objective regression test case prioritisation”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 2015, pp. 234–245.

[41] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner. “Chapter One - Security Testing: A Survey”. In: ed. by A. Memon. Vol. 101. Advances in Computers. Elsevier, 2016, pp. 1–51. DOI: https://doi.org/10.1016/bs.adcom.2015.11.003. URL: https://www.sciencedirect.com/science/article/pii/S0065245815000649.

[42] M. Felderer and E. Fourneret. “A systematic classification of security regression testing approaches”. In: *International Journal on Software Tools for Technology Transfer* 17.3 (2015), pp. 305–319.

[43] B. Fu, S. Misailovic, and M. Gligoric. “Resurgence of Regression Test Selection for C++”. en. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. Xi’an, China: IEEE, Apr. 2019, pp. 323–334. ISBN: 978-1-72811-736-2. DOI: 10.1109/ICST.2019.00039. URL: https://ieeexplore.ieee.org/document/8730161/ (visited on 05/11/2021).

[44] P. Garner, S. Hopewell, J. Chandler, H. MacLehose, E. A. Akl, J. Beyene, S. Chang, R. Churchill, K. Dearness, G. Guyatt, C. Lefebvre, B. Liles, R. Marshall, L.

Martínez García, C. Mavergames, M. Nasser, A. Qaseem, M. Sampson, K. Soares-Weiser, Y. Takwoingi, L. Thabane, M. Trivella, P. Tugwell, E. Welsh, E. C. Wilson, and H. J. Schünemann. "When and how to update systematic reviews: consensus and checklist". In: *BMJ* 354 (2016). DOI: 10.1136/bmj.i3507. URL: https://www.bmj.com/content/354/bmj.i3507.

[45] V. Garousi, R. Özkan, and A. Betin-Can. "Multi-objective regression test selection in practice: An empirical study in the defense software industry". In: *Information and Software Technology* 103 (2018), pp. 40–54. DOI: 10.1016/j.infsof.2018.06.007.

[46] V. Garousi and M. Felderer. "Worlds apart: industrial and academic focus areas in software testing". In: *IEEE Software* 34.5 (2017), pp. 38–45.

[47] V. Garousi, R. Özkan, and A. Betin-Can. "Multi-objective regression test selection in practice: An empirical study in the defense software industry". In: *Information and Software Technology* 103 (2018), pp. 40–54.

[48] M. Gligoric, L. Eloussi, and D. Marinov. "Practical Regression Test Selection with Dynamic File Dependencies". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis.* ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 211–222. ISBN: 9781450336208. DOI: 10.1145/2771783.2771784. URL: https://doi.org/10.1145/2771783.2771784.

[49] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. "An Empirical Evaluation and Comparison of Manual and Automated Test Selection". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering.* ASE '14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 361–372. ISBN: 9781450330138. DOI: 10.1145/2642937.2643019. URL: https://doi.org/10.1145/2642937.2643019.

[50] A. Gotlieb and D. Marijan. "Using global constraints to automate regression testing". In: *AI Magazine* 38.1 (2017), pp. 73–87. URL: https://doi.org/10.1609/aimag.v38i1.2714.

[51] A. Goyal, R. Shyamasundar, R. Jetley, D. Mohan, and S. Ramaswamy. "Test suite minimization of evolving software systems: A case study". In: *ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies.* 2019, pp. 226–237.

[52] R. Greca, B. Miranda, and A. Bertolino. “State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review”. In: *ACM Computing Surveys* (Jan. 2023). ISSN: 0360-0300. DOI: 10.1145/3579851.

[53] R. Greca, B. Miranda, M. Gligoric, and A. Bertolino. “Comparing and combining file-based selection and similarity-based prioritization towards regression test orchestration”. en. In: *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 115–125. ISBN: 978-1-4503-9286-0. DOI: 10.1145/3524481.3527223.

[54] B. Guo, Y.-W. Kwon, and M. Song. “Decomposing Composite Changes for Code Review and Regression Test Selection in Evolving Software”. In: *Journal of Computer Science and Technology* 34.2 (2019), pp. 416–436. DOI: 10.1007/s11390-019-1917-9.

[55] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja. “Test prioritization in continuous integration environments”. en. In: *Journal of Systems and Software* 146 (Dec. 2018), pp. 80–98. ISSN: 01641212. DOI: 10.1016/j.jss.2018.08.061. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121218301730 (visited on 05/11/2021).

[56] D. Hao, L. Zhang, and H. Mei. “Test-case prioritization: achievements and challenges”. en. In: *Frontiers of Computer Science* 10.5 (Oct. 2016), pp. 769–777. ISSN: 2095-2228, 2095-2236. DOI: 10.1007/s11704-016-6112-3. (Visited on 11/08/2020).

[57] M. Harman. “Making the case for MORTO: Multi objective regression test optimization”. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 111–114.

[58] M. Harman and P. O’Hearn. “From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis”. In: *Proceedings of the 18th International Working Conference on Source Code Analysis and Manipulation (SCAM 18)*. IEEE. 2018, pp. 1–23.

[59] M. J. Harrold and A. Orso. “Retesting software during development and maintenance”. In: *2008 Frontiers of Software Maintenance*. IEEE. 2008, pp. 99–108.

[60] M. Hasnain, I. Ghani, M. F. Pasha, and S. R. Jeong. “A Comprehensive Review on Regression Test Case Prioritization Techniques for Web Services”. en. In:

KSII Transactions on Internet and Information Systems 14.5 (May 2020). ISSN: 19767277. DOI: 10.3837/tiis.2020.05.001. (Visited on 11/08/2020).

[61] M. Hasnain, I. Ghani, M. F. Pasha, and S.-R. Jeong. "Ontology-Based Regression Testing: A Systematic Literature Review". In: *Applied Sciences* 11.20 (2021), p. 9709.

[62] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. "Comparing White-Box and Black-Box Test Prioritization". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).* 2016, pp. 523–534. DOI: 10.1145/2884781.2884791.

[63] K. Herzig. *Let's assume we had to pay for testing.* https://www.slideshare.net/kim.herzig/keynote-ast-2016. Accessed on 06/09/2022. 2016. (Visited on 10/06/2022).

[64] M. Hirzel and H. Klaeren. "Graph-Walk-based Selective Regression Testing of Web Applications Created with Google Web Toolkit". en. In: (2016), p. 15.

[65] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. "Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria". In: *Proceedings of 16th International conference on Software engineering.* IEEE. 1994, pp. 191–200.

[66] "IEEE Standard Classification for Software Anomalies". In: *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (2010), pp. 1–23. DOI: 10.1109/IEEESTD.2010.5399061.

[67] Y. Jia and M. Harman. "An analysis and survey of the development of mutation testing". In: *IEEE transactions on software engineering* 37.5 (2010), pp. 649–678.

[68] R. Just, D. Jalali, and M. D. Ernst. "Defects4J: A database of existing faults to enable controlled testing studies for Java programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* 2014, pp. 437–440.

[69] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani. "Effective Regression Test Case Selection: A Systematic Literature Review". en. In: *ACM Computing Surveys* 50.2 (June 2017), pp. 1–32. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3057269. (Visited on 11/08/2020).

[70] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng. "Test case prioritization approaches in regression testing: A systematic literature review". en. In: *Information and Software Technology* 93 (Jan. 2018), pp. 74–93. ISSN: 09505849. DOI: 10.1016/j.infsof.2017.08.014. (Visited on 12/01/2020).

[71] S. Kirdey, K. Cureton, S. Rick, S. Ramanathan, and M. Shukla. *Lerner — using RL agents for test case scheduling.* https://netflixtechblog.com/lerner-using-rl-agents-for-test-case-scheduling-3e0686211198. Accessed on 06/09/2022. 2019. (Visited on 09/06/2022).

[72] B. Kitchenham. "Procedures for performing systematic reviews". In: *Keele, UK, Keele University* 33.2004 (2004), pp. 1–26.

[73] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, and M. Castell. "Supporting continuous integration by code-churn based test selection". In: *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering.* IEEE. 2015, pp. 19–25.

[74] J.-H. Kwon and I.-Y. Ko. "Cost-effective regression testing using bloom filters in continuous integration development environments". In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC).* IEEE. 2017, pp. 160–168.

[75] K. Land, E.-M. Neumann, S. Ziegltrum, H. Li, and B. Vogel-Heuser. "An Industrial Evaluation of Test Prioritisation Criteria and Metrics". In: *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC).* Bari, Italy: IEEE, Oct. 2019, pp. 1887–1892. ISBN: 978-1-72814-569-3. DOI: 10.1109/SMC.2019.8914505. URL: https://ieeexplore.ieee.org/document/8914505/ (visited on 05/11/2021).

[76] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. "An extensive study of static regression test selection in modern software evolution". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 2016, pp. 583–594.

[77] M. Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[78] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco. "Assessing Transition-Based Test Selection Algorithms at Google". en. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* Montreal, QC, Canada: IEEE, May 2019, pp. 101–110. ISBN: 978-1-72811-760-7. DOI: 10.1109/ICSE-SEIP.2019.00019. URL: https://ieeexplore.ieee.org/document/8804429/ (visited on 11/07/2019).

[79] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets.* New York, NY, USA: Cambridge University Press, 2014. ISBN: 1107077230, 9781107077232.

[80] H. K. Leung and L. White. “Insights into regression testing (software testing)”. In: *Proceedings. Conference on Software Maintenance-1989.* IEEE. 1989, pp. 60–69.

[81] A. Levin. “Latest 737 Max Fault That Alarmed Test Pilots Rooted in Software”. en. In: *Bloomberg.com* (July 2019). URL: https://www.bloomberg.com/news/articles/2019-07-27/latest-737-max-fault-that-alarmed-test-pilots-rooted-in-software (visited on 09/11/2019).

[82] F. Li, J. Zhou, Y. Li, D. Hao, and L. Zhang. “AGA: An Accelerated Greedy Additional Algorithm for Test Case Prioritization”. en. In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2021.3137929. URL: https://ieeexplore.ieee.org/document/9662236/ (visited on 09/14/2022).

[83] J. A. P. Lima and S. R. Vergilio. “A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments”. en. In: *IEEE Transactions on Software Engineering* 48.2 (Feb. 2022), pp. 453–465. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2020.2992428. URL: https://ieeexplore.ieee.org/document/9086053/ (visited on 09/14/2022).

[84] Y. Lou, J. Chen, L. Zhang, and D. Hao. “A Survey on Regression Test-Case Prioritization”. en. In: *Advances in Computers* 113 (2019), pp. 1–46. ISSN: 0065-2458. DOI: 10.1016/bs.adcom.2018.10.001.

[85] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. “How does regression test prioritization perform in real-world software evolution?” en. In: *Proceedings of the 38th International Conference on Software Engineering.* Austin Texas: ACM, May 2016, pp. 535–546. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884874. URL: https://dl.acm.org/doi/10.1145/2884781.2884874 (visited on 05/11/2021).

[86] D. Lübke. “Selecting and Prioritizing Regression Test Suites by Production Usage Risk in Time-Constrained Environments”. In: (2020). ISBN: 9783030355098, pp. 69–83. DOI: 10.1007/978-3-030-35510-4.

[87] Q. Luo, K. Moran, L. Zhang, and D. Poshyvanyk. “How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects”. In: *IEEE Transactions on Software Engineering* 45.11 (2019), pp. 1054–1080. DOI: 10.1109/TSE.2018.2822270.

[88] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. “An empirical analysis of flaky tests”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 643–653.

[89] R. Lynn. *How the Agile Method Transforms Software Testing*. URL: https://www.planview.com/resources/articles/how-agile-method-transforms-software-testing/.

[90] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. “Predictive Test Selection”. en. In: *arXiv:1810.05286 [cs]*. arXiv: 1810.05286. Oct. 2018. URL: http://arxiv.org/abs/1810.05286 (visited on 08/07/2019).

[91] C. Magalhães, J. Andrade, L. Perrusi, A. Mota, F. Barros, and E. Maia. “HSP: A hybrid selection and prioritisation of regression test cases based on information retrieval and code coverage applied on an industrial case study”. en. In: *Journal of Systems and Software* 159 (Jan. 2020), p. 110430. ISSN: 01641212. DOI: 10.1016/j.jss.2019.110430. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121219302043 (visited on 09/14/2022).

[92] C. Magalhães, F. Barros, A. Mota, and E. Maia. “Automatic Selection of Test Cases for Regression Testing”. en. In: *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing - SAST*. Maringa, Parana, Brazil: ACM Press, 2016, pp. 1–8. ISBN: 978-1-4503-4766-2. DOI: 10.1145/2993288.2993299. URL: http://dl.acm.org/citation.cfm?doid=2993288.2993299 (visited on 05/11/2021).

[93] D. Marijan and M. Liaaen. “Effect of Time Window on the Performance of Continuous Regression Testing”. en. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, USA: IEEE, Oct. 2016, pp. 568–571. ISBN: 978-1-5090-3806-0. DOI: 10.1109/ICSME.2016.77. URL: http://ieeexplore.ieee.org/document/7816510/ (visited on 05/11/2021).

[94] S. Mehta, F. Farmahinifarahani, R. Bhagwan, S. Guptha, S. Jafari, R. Kumar, V. Saini, and A. Santhiar. “Data-driven test selection at scale”. en. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM,

Aug. 2021, pp. 1225–1235. ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3473916. URL: https://dl.acm.org/doi/10.1145/3468264.3473916 (visited on 09/14/2022).

[95] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. “Taming Google-scale continuous testing”. en. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. Buenos Aires: IEEE, May 2017, pp. 233–242. ISBN: 978-1-5386-2717-4. DOI: 10.1109/ICSE-SEIP.2017.16. URL: http://ieeexplore.ieee.org/document/7965447/ (visited on 09/09/2019).

[96] E. Mendes, C. Wohlin, K. Felizardo, and M. Kalinowski. “When to update systematic literature reviews in software engineering”. In: *Journal of Systems and Software* 167 (2020), p. 110607. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110607. URL: https://www.sciencedirect.com/science/article/pii/S0164121220300856.

[97] T. Mens. “Introduction and Roadmap: History and Challenges of Software Evolution”. In: *Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–11. ISBN: 978-3-540-76440-3. DOI: 10.1007/978-3-540-76440-3_1. URL: https://doi.org/10.1007/978-3-540-76440-3_1.

[98] N. M. Minhas, K. Petersen, N. B. Ali, and K. Wnuk. “Regression Testing Goals - View of Practitioners and Researchers”. en. In: *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*. Nanjing: IEEE, Dec. 2017, pp. 25–31. ISBN: 978-1-5386-2649-8. DOI: 10.1109/APSECW.2017.23. URL: http://ieeexplore.ieee.org/document/8312521/ (visited on 09/02/2019).

[99] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. “FAST approaches to scalable similarity-based test case prioritization”. en. In: *Proceedings of the 40th International Conference on Software Engineering - ICSE ’18*. Gothenburg, Sweden: ACM Press, 2018, pp. 222–232. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180210. URL: http://dl.acm.org/citation.cfm?doid=3180155.3180210 (visited on 09/06/2019).

[100] M. L. Mohd-Shafie, W. M. N. W. Kadir, H. Lichter, M. Khatibsyarbini, and M. A. Isa. “Model-based test case generation and prioritization: a systematic literature review”. In: *Software and Systems Modeling* (2021), pp. 1–37.

[101] R. Mukherjee and K. S. Patnaik. “A survey on different approaches for software test case prioritization”. en. In: *Journal of King Saud University - Computer and Information Sciences* (Oct. 2018), S1319157818303616. ISSN: 13191578. DOI: 10.1016/j.jksuci.2018.09.005. (Visited on 11/08/2020).

[102] A. Najafi, W. Shang, and P. Rigby. “Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report”. In: *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019.* 2019, pp. 213–222. DOI: 10.1109/ICSE-SEIP.2019.00031.

[103] B. N. Nguyen, T. Henderson, J. Micco, and S. Dhanda. *Google Journal Club.* https://sites.google.com/site/gjournalclub/. Accessed on 18/03/2021. 2016. (Visited on 03/18/2021).

[104] R. Noemmer and R. Haas. “An Evaluation of Test Suite Minimization Techniques”. In: *Lecture Notes in Business Information Processing* 371 LNBIP (2020), pp. 51–66. DOI: 10.1007/978-3-030-35510-4_4.

[105] T. Noor and H. Hemmati. “A similarity-based approach for test case prioritization using historical failure data”. In: *2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015.* 2016, pp. 58–68. DOI: 10.1109/ISSRE.2015.7381799.

[106] S. Omri and C. Sinz. “Learning to Rank for Test Case Prioritization”. en. In: (2022), p. 9.

[107] J. Öqvist, G. Hedin, and B. Magnusson. “Extraction-Based Regression Test Selection”. en. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools.* Lugano Switzerland: ACM, Aug. 2016, pp. 1–10. ISBN: 978-1-4503-4135-6. DOI: 10.1145/2972206.2972224. URL: https://dl.acm.org/doi/10.1145/2972206.2972224 (visited on 09/14/2022).

[108] J. F. S. Ouriques, E. G. Cartaxo, and P. D. Machado. “Test case prioritization techniques for model-based testing: a replicated study”. In: *Software Quality Journal* 26.4 (Dec. 2018). Publisher: Springer New York LLC, pp. 1451–1482. ISSN: 15731367. DOI: 10.1007/s11219-017-9398-y.

[109] C. Pan, Y. Yang, Z. Li, and J. Guo. "Dynamic Time Window based Reward for Reinforcement Learning in Continuous Integration Testing". en. In: *12th Asia-Pacific Symposium on Internetware*. Singapore Singapore: ACM, Nov. 2020, pp. 189–198. ISBN: 978-1-4503-8819-1. DOI: 10.1145/3457913.3457930. URL: https://dl.acm.org/doi/10.1145/3457913.3457930 (visited on 09/14/2022).

[110] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand. "Test case selection and prioritization using machine learning: a systematic literature review". In: *Empirical Software Engineering* 27.2 (2022), pp. 1–43.

[111] Q. Peng, A. Shi, and L. Zhang. "Empirically revisiting and enhancing IR-based test-case prioritization". en. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, July 2020, pp. 324–336. ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397383. URL: https://dl.acm.org/doi/10.1145/3395363.3397383 (visited on 05/11/2021).

[112] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan. "FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services". en. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 408–418. ISBN: 978-1-72810-869-8. DOI: 10.1109/ICSE.2019.00054. URL: https://ieeexplore.ieee.org/document/8812033/ (visited on 12/09/2019).

[113] D. Pradhan, S. Wang, S. Ali, and T. Yue. "Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. New York, NY, USA: Association for Computing Machinery, July 2016, pp. 1085–1092. ISBN: 978-1-4503-4206-3. DOI: 10.1145/2908812.2908850. URL: https://biblioproxy.cnr.it:2481/10.1145/2908812.2908850.

[114] J. A. Prado Lima and S. R. Vergilio. "Test Case Prioritization in Continuous Integration environments: A systematic mapping study". In: *Information and Software Technology* 121 (2020). DOI: 10.1016/j.infsof.2020.106268.

[115] D. Qiu, B. Li, S. Ji, and H. Leung. "Regression testing of web service: a systematic mapping study". In: *ACM Computing Surveys (CSUR)* 47.2 (2014), pp. 1–46.

[116] R. Ramler, C. Salomon, G. Buchgeher, and M. Lusser. "Tool support for change-based regression testing: An industry experience report". In: *Lecture Notes in Business Information Processing* 269 (2017), pp. 133–152. DOI: 10.1007/978-3-319-49421-0_10.

[117] S. U. Rehman Khan, S. P. Lee, N. Javaid, and W. Abdul. "A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines". en. In: *IEEE Access* 6 (2018), pp. 11816–11841. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2809600. (Visited on 11/08/2020).

[118] L. Rosenbauer, A. Stein, and J. Hähner. "An Artificial Immune System for Black Box Test Case Selection". en. In: *Evolutionary Computation in Combinatorial Optimization*. Ed. by C. Zarges and S. Verel. Vol. 12692. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 169–184. ISBN: 978-3-030-72903-5 978-3-030-72904-2. DOI: 10.1007/978-3-030-72904-2_11. URL: http://link.springer.com/10.1007/978-3-030-72904-2_11 (visited on 09/14/2022).

[119] R. H. Rosero, O. S. Gómez, E. R. Villa, R. A. Aguilar, and C. J. Pardo. "Software Regression Testing in Industrial Settings: Preliminary Findings from a Literature Review". In: *The International Conference on Advances in Emerging Trends and Technologies*. Springer. 2021, pp. 227–237.

[120] R. H. Rosero, O. S. Gómez, and G. Rodríguez. "15 Years of Software Regression Testing Techniques - A Survey". In: *International Journal of Software Engineering and Knowledge Engineering* 26.5 (2016), pp. 675–689. DOI: 10.1142/S0218194016300013.

[121] G. Rothermel. "Improving regression testing in continuous integration development environments (keynote)". en. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation - A-TEST 2018*. Lake Buena Vista, FL, USA: ACM Press, 2018, pp. 1–1. ISBN: 978-1-4503-6053-1. DOI: 10.1145/3278186.3281454. URL: http://dl.acm.org/citation.cfm?doid=3278186.3281454 (visited on 08/26/2019).

[122] G. Rothermel and M. J. Harrold. "A Framework for Evaluating Regression Test Selection Techniques". In: *International Conference on Software Engineering*. 1994, pp. 201–210.

[123] Z. Sadri-Moshkenani, J. Bradley, and G. Rothermel. “Survey on test case generation, selection and prioritization for cyber-physical systems”. In: *Software Testing, Verification and Reliability* 32.1 (2022), e1794.

[124] A. Samad, H. Mahdin, R. Kazmi, and R. Ibrahim. “Regression Test Case Prioritization: A Systematic Literature Review”. In: *International Journal of Advanced Computer Science and Applications* 12.2 (2021).

[125] A. Schwartz and H. Do. “Cost-effective regression testing through Adaptive Test Prioritization strategies”. en. In: *Journal of Systems and Software* 115 (May 2016), pp. 61–81. ISSN: 01641212. DOI: 10.1016/j.jss.2016.01.018. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121216000169 (visited on 05/11/2021).

[126] A. Sharif, D. Marijan, and M. Liaaen. “DeepOrder: Deep Learning for Test Case Prioritization in Continuous Integration Testing”. en. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME).* Luxembourg: IEEE, Sept. 2021, pp. 525–534. ISBN: 978-1-66542-882-8. DOI: 10.1109/ICSME52107.2021.00053. URL: https://ieeexplore.ieee.org/document/9609187/ (visited on 09/14/2022).

[127] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov. “Evaluating test-suite reduction in real software evolution”. en. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* Amsterdam Netherlands: ACM, July 2018, pp. 84–94. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213875. URL: https://dl.acm.org/doi/10.1145/3213846.3213875 (visited on 11/22/2021).

[128] A. Shi, T. Yung, A. Gyori, and D. Marinov. “Comparing and combining test-suite reduction and regression test selection”. In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering.* 2015, pp. 237–247.

[129] A. Shi, P. Zhao, and D. Marinov. “Understanding and Improving Regression Test Selection in Continuous Integration”. en. In: *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE).* Berlin, Germany: IEEE, Oct. 2019, pp. 228–238. ISBN: 978-1-72814-982-0. DOI: 10.1109/ISSRE.2019.00031. URL: https://ieeexplore.ieee.org/document/8987498/ (visited on 05/11/2021).

[130] D. Silva, R. Rabelo, M. Campanha, P. S. Neto, P. A. Oliveira, and R. Britto. “A hybrid approach for test case prioritization and selection”. In: *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2016, pp. 4508–4515.

[131] Y. Singh, A. Kaur, B. Suri, and S. Singhal. “Systematic literature review on regression test prioritization techniques”. In: *Informatica* 36.4 (2012).

[132] Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez. “Change-based test selection: an empirical evaluation”. In: *Empirical software engineering* 21.5 (2016), pp. 1990–2032.

[133] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. “Reinforcement learning for automatic test case prioritization and selection in continuous integration”. en. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Santa Barbara CA USA: ACM, July 2017, pp. 12–22. ISBN: 978-1-4503-5076-1. DOI: 10.1145/3092703.3092709. URL: https://dl.acm.org/doi/10.1145/3092703.3092709 (visited on 09/14/2022).

[134] H. Srikanth, M. Cashman, and M. Cohen. “Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study”. In: *Journal of Systems and Software* 119 (2016), pp. 122–135. DOI: 10.1016/j.jss.2016.06.017.

[135] H. Srikanth, C. Hettiarachchi, and H. Do. “Requirements Based Test Prioritization Using Risk Factors”. In: *Inf. Softw. Technol.* 69.C (Jan. 2016). Place: USA Publisher: Butterworth-Heinemann, pp. 71–83. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.09.002. URL: https://biblioproxy.cnr.it:2481/10.1016/j.infsof.2015.09.002.

[136] P. Strandberg, D. Sundmark, W. Afzal, T. Ostrand, and E. Weyuker. “Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors”. In: *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. 2016, pp. 12–23. DOI: 10.1109/ISSRE.2016.23.

[137] S. Tahvili, M. Bohlin, M. Saadatmand, S. Larsson, W. Afzal, and D. Sundmark. “Cost-benefit analysis of using dependency knowledge at integration testing”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10027 LNCS (2016), pp. 268–284. DOI: 10.1007/978-3-319-49094-6_17.

[138] S. Tahvili, M. Saadatmand, S. Larsson, W. Afzal, M. Bohlin, and D. Sundmark. “Dynamic Integration Test Selection Based on Test Case Dependencies”. en. In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Chicago, IL, USA: IEEE, Apr. 2016, pp. 277–286. ISBN: 978-1-5090-3674-5. DOI: 10.1109/ICSTW.2016.14. URL: http://ieeexplore.ieee.org/document/7528974/ (visited on 05/11/2021).

[139] A. Vargha and H. D. Delaney. “A critique and improvement of the CL common language effect size statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[140] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric. “File-level vs. module-level regression test selection for .NET”. en. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, Aug. 2017, pp. 848–853. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3117763. URL: https://dl.acm.org/doi/10.1145/3106237.3117763 (visited on 05/11/2021).

[141] S. Vöst and S. Wagner. “Trace-based test selection to support continuous integration in the automotive industry”. In: *Proceedings - International Workshop on Continuous Software Evolution and Delivery, CSED 2016*. Association for Computing Machinery, Inc, May 2016, pp. 34–40. ISBN: 978-1-4503-4157-8. DOI: 10.1145/2896941.2896951.

[142] S. Wang, S. Ali, T. Yue, O. Bakkeli, and M. Liaaen. “Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search”. In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, May 2016, pp. 182–191. ISBN: 978-1-4503-4161-5. DOI: 10.1145/2889160.2889240.

[143] Z. Wu, Y. Y. Yang, Z. Li, and R. Zhao. “A Time Window Based Reinforcement Learning Reward for Test Case Prioritization in Continuous Integration”. In: *Proceedings of the 11th Asia-Pacific Symposium on Internetware*. Internetware ’19. New York, NY, USA: Association for Computing Machinery, Oct. 2019. ISBN: 978-1-4503-7701-0. DOI: 10.1145/3361242.3361258. URL: https://biblioproxy.cnr.it:2481/10.1145/3361242.3361258.

[144] J. Xu, Q. Du, and X. Li. “A Requirement-based Regression Test Selection Technique in Behavior-Driven Development”. en. In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. Madrid, Spain: IEEE,

July 2021, pp. 1303–1308. ISBN: 978-1-66542-463-9. DOI: 10.1109/COMPSAC51774.2021.00182. URL: https://ieeexplore.ieee.org/document/9529903/ (visited on 09/14/2022).

[145] J. Yackley, M. Kessentini, G. Bavota, V. Alizadeh, and B. Maxim. "Simultaneous refactoring and regression testing". In: *Proceedings - 19th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2019*. 2019, pp. 216–227. DOI: 10.1109/SCAM.2019.00032.

[146] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. Briand. "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts". en. In: *IEEE Transactions on Software Engineering* (2022), pp. 1–24. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2022.3184842. URL: https://ieeexplore.ieee.org/document/9801672/ (visited on 09/14/2022).

[147] U. Yilmaz and A. Tarhan. "A case study to compare regression test selection techniques on open-source software projects". In: *CEUR Workshop Proceedings*. Vol. 2201. 2018.

[148] S. Yoo and M. Harman. "Regression testing minimization, selection and prioritization: a survey". In: *Software testing, verification and reliability* 22.2 (2012), pp. 67–120.

[149] S. Yoo, R. Nilsson, and M. Harman. "Faster fault finding at Google using multi objective regression test optimisation". In: *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary*. 2011.

[150] H. Yoshida, S. Tokumoto, M. Prasad, I. Ghosh, and T. Uehara. "FSX: A tool for fine-grained incremental unit test generation for C/C++ Programs". In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Vol. 13-18-November-2016. 2016, pp. 1052–1056. DOI: 10.1145/2950290.2983937.

[151] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian. "TERMINATOR: better automated UI test case prioritization". en. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn Estonia: ACM,

Aug. 2019, pp. 883–894. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3340448. URL: https://dl.acm.org/doi/10.1145/3338906.3340448 (visited on 05/11/2021).

[152] A. Zarrad. “A Systematic Review on Regression Testing for Web-Based Applications.” In: *J. Softw.* 10.8 (2015), pp. 971–990.

[153] J. Zhang, Y. Liu, M. Gligoric, O. Legunsen, and A. Shi. “Comparing and combining analysis-based and learning-based regression test selection”. en. In: *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 17–28. ISBN: 978-1-4503-9286-0. DOI: 10.1145/3524481.3527230. URL: https://dl.acm.org/doi/10.1145/3524481.3527230 (visited on 09/14/2022).

[154] L. Zhang. “Hybrid regression test selection”. en. In: *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, pp. 199–209. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180198. URL: https://dl.acm.org/doi/10.1145/3180155.3180198 (visited on 05/11/2021).

[155] H. Zhong, L. Zhang, and S. Khurshid. “TestSage: Regression Test Selection for Large-Scale Web Service Testing”. en. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. Xi’an, China: IEEE, Apr. 2019, pp. 430–440. ISBN: 978-1-72811-736-2. DOI: 10.1109/ICST.2019.00052. URL: https://ieeexplore.ieee.org/document/8730207/ (visited on 12/09/2019).

[156] J. Zhou, J. Chen, and D. Hao. “Parallel Test Prioritization”. en. In: *ACM Transactions on Software Engineering and Methodology* 31.1 (Jan. 2022), pp. 1–50. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3471906. URL: https://dl.acm.org/doi/10.1145/3471906 (visited on 09/14/2022).

[157] Z. Q. Zhou, C. Liu, T. Y. Chen, T. H. Tse, and W. Susilo. “Beating Random Test Case Prioritization”. en. In: *IEEE Transactions on Reliability* (2020), pp. 1–22. ISSN: 0018-9529, 1558-1721. DOI: 10.1109/TR.2020.2979815. URL: https://ieeexplore.ieee.org/document/9118977/ (visited on 05/11/2021).

[158] Y. Zhu, E. Shihab, and P. C. Rigby. “Test re-prioritization in continuous testing environments”. In: *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*. IEEE, 2018, pp. 69–79. ISBN: 978-1-5386-7870-1. DOI: 10.1109/ICSME.2018.00016.