Università degli Studi di Napoli Federico II
Ph.D. Program in
Information Technology and Electrical Engineering
XXXV Cycle

Thesis for the Degree of Doctor of Philosophy

Improving End-to-End Testing for Web and Mobile Applications

by

Luigi Libero Lucio Starace

Advisor: Prof. Sergio Di Martino

Co-advisor: Prof. Adriano Peron

Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

To N.

Improving End-to-End Testing for Web and Mobile Applications

Ph.D. Thesis presented
for the fulfillment of the Degree of Doctor of Philosophy
in Information Technology and Electrical Engineering
by

Luigi Libero Lucio Starace

October 2022

Approved as to style and content by

_____

Prof. Sergio Di Martino, Advisor

_____

Prof. Adriano Peron, Co-advisor

Università degli Studi di Napoli Federico II
Ph.D. Program in Information Technology and Electrical Engineering
XXXV cycle - Chairman: Prof. Stefano Russo

http://itee.dieti.unina.it

Candidate's declaration

I hereby declare that this thesis submitted to obtain the academic degree of Philosophiæ Doctor (Ph.D.) in Information Technology and Electrical Engineering is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
Parts of this dissertation have been published in international journals and/or conference articles (see list of the author's publications at the end of the thesis).

Napoli, December 13, 2022

_____

Luigi Libero Lucio Starace

Abstract

End-to-End (E2E) testing is widely-used to improve the quality of web and mobile applications. In this kind of activity, the Application Under Test (AUT) is tested as a whole, in its entirety, simulating real-world usage scenarios. The goal of the research presented in this thesis is to improve the effectiveness of E2E testing processes from multiple perspectives.

In the domain of GUI-level testing of web applications, research presented in this thesis work tackles the problem of near-duplicate web page detection in automatic model inference, which is a prerequisite for the application of many automatic test generation techniques for web apps. Two novel near-duplicate detection techniques are proposed, based on a common underlying framework, and their effectiveness is assessed in an empirical study.

In the domain of performance testing of web applications, we face the problem of workload generation, presenting a novel technique to support their automatic generation from existing E2E GUI-level web tests. The effectiveness of the proposed technique is then evaluated in a preliminary industrial case study, with promising results.

Lastly, in the domain of GUI-level testing of mobile applications, this thesis presents research aimed at supporting Software Project Managers in deciding which techniques to use to test a given mobile application. To this end, two empirical studies are conducted. The first study aims at comparing the testing effectiveness of state-of-the-art automatic testing tools against that of unskilled practitioners using Capture and Replay tools with exploratory testing strategies. The second study investigates the effectiveness of crowdtesting in generating executable test suites for mobile apps.

Keywords: End-to-End Testing, Web Applications, Mobile Applications, Performance Testing, Software Testing.

Sintesi in lingua italiana

Il testing *End-to-End* (E2E) è una pratica largamente usata per migliorare la qualità di applicazioni web e *mobile*. In questo tipo di attività, l'applicazione viene testata nella sua interezza, simulando scenari realistici di utilizzo. Le attività di ricerca presentate in questa tesi mirano al miglioramento dei processi di testing E2E in diversi ambiti.

Nell'ambito del testing di applicazioni web a livello di interfaccia utente, le attività presentate mirano a migliorare l'efficacia delle tecniche di generazione automatica di test allo stato dell'arte. A tal fine, vengono proposte due nuove tecniche per affrontare il problema dell'individuazione di pagine web quasi-duplicate nel processo di inferenza automatica di modelli. L'efficacia delle tecniche proposte e il loro impatto sulla qualità dei test generati automaticamente sono valutati sperimentalmente.

Nell'ambito del testing di *performance*, viene affrontato il problema della generazione di workload, presentando una nuova tecnica che permette di generare automaticamente workload partendo da test E2E a livello di interfaccia utente. L'efficacia dell'approccio proposto viene valutata sperimentalmente in un caso di studio industriale.

Infine, nell'ambito del testing E2E a livello di interfaccia utente per applicazioni mobile, si investiga l'efficacia di diversi approcci, al fine di supportare i *manager* di progetti software nella scelta degli approcci da utilizzare per una data applicazione. A tal fine, vengono descritti due studi empirici. Il primo studio mira a confrontare l'efficacia di strumenti automatici di generazione di test con quella di test generati da sviluppatori inesperti con strumenti di *Capture and Replay* (C&R). Il secondo studio investiga l'efficacia di strumenti di C&R in scenari di *crowdtesting*.

Parole chiave: Testing *End-to-End*, Applicazioni Web, Applicazioni *Mobile*, *Performance* Testing, Testing di Software.

Contents

Acknowledgements

I enjoyed (*almost!*) every minute of the research I carried out during my PhD, part of which has been described in this thesis. I owe that enjoyment largely to the interaction I have had with my advisors, colleagues and people I met and worked with along the way.

I feel very honoured to have worked with my advisors, Sergio Di Martino, Adriano Peron and, even though not officially an advisor, Anna Corazza. I'm very grateful to each of them for the patience, inspiration, support and friendship they granted me throughout my studies. From them, I learned a lot about software engineering, research, and academia.

I'm also grateful to Paolo Tonella for hosting me at the Software Institute of the Università della Svizzera Italiana in Lugano, Switzerland, for my research period abroad, and to Andrea Stocco, for his guidance in my research on near-duplicate detection. In Lugano, I spent countless enjoyable hours with researchers and fellow PhD students at the Testing AUtomated (TAU) lab, chatting about our research in a rich and stimulating environment.

I sincerely thank Ciro D'Addio, Alfredo Troiano and Enrico Landolfi, for the fruitful discussions that allowed me to better understand industrial needs in the field of End-to-End testing of web and mobile applications, as well as *NetCom Group SpA*, which funded the PhD scholarship that supported my work.

I would also like to thank my family, including those who are no longer with us, who have always been understanding and supportive of my studies. Last but not least, I'd like to thank my wonderful partner, Natalia, who has encouraged and supported me so much over the years.

List of Acronyms

The following acronyms are used throughout the thesis.

AIG	Automated Input Generation
AUT	Application Under Test
C&R	Capture & Replay
DOM	Document Object Model
E2E	End-to-End
ET	Exploratory Testing
GUI	Graphical User Interface
HTTP	Hypter-Text Transfer Protocol
IDE	Integrated Development Environment
IET	Informed Exploratory Testing
LOC	Lines of Code
SAF	State Abstraction Function
SUT	System Under Test

SVM	Support Vector Machines
TK	Tree Kernel
UET	Uninformed Exploratory Testing

List of Figures

List of Tables

List of Algorithms

List of Listings

Chapter 1

Introduction

Web and mobile applications have become pervasive and are involved in many aspects of our daily lives. From home banking to public transit trip planning, from e-commerce to social networks, modern society relies on web and mobile applications to an ever-growing extent for a multitude of economic, social, and recreational activities. The impact of failures in such applications may range from simple inconveniences for end-users up to complete business interruption, and can potentially cause significant damages. Hence, ensuring the quality and correctness of web and mobile applications is of undeniable importance [169].

End-to-End (E2E) testing is one of the main approaches to improve the quality of these software systems. In this kind of activity, the Application Under Test (AUT) is tested as a whole, simulating real-world usage scenarios. In most cases, E2E testing aims at exercising the AUT from the perspective of an end-user, typically interacting with its Graphical User Interface (GUI). The goal is typically to verify functional correctness, i.e., that the application behaves as intended in response to user-generated events and interactions with the GUI (e.g., clicks, scrolls, forms filling and submissions, etc.).

Moreover, especially for web applications, which may have to serve a large number of concurrent users, additional testing activities aimed at assessing that the application behaves as expected under different load conditions may be put in place [46]. This kind of testing activities are collectively referred to in the literature as *Performance Testing*.

The research work presented in this thesis is aimed at improving the effectiveness of E2E testing processes for web and mobile applications, from multiple perspectives.

In the domain of GUI-level testing of web applications, the thesis work tackles the problem of near-duplicate web page detection during automatic model inference, which is a prerequisite for applying many automatic test generation techniques for web apps. Indeed, as recent studies highlighted, existing approaches for the detection of near-duplicate web pages are generally not able to produce accurate models, hindering the effectiveness of automatic test generation techniques as well. To address this issue, two novel near-duplicate detection techniques, specifically geared towards model inference for web testing purposes, are proposed, based on a common underlying framework. The effectiveness of the proposed techniques, as well as their practical impact on the quality of automatically generated tests, is assessed in an empirical study involving massive datasets from the literature and open-source web applications. Results show that the proposed techniques outperform state-of-the-art near-duplicate detection solutions, and lead to remarkable practical benefits when employed in automatic test generation for web apps.

In the domain of performance testing of web applications, the thesis tackles the problem of workload generation. Indeed, manually generating workloads is a very time-consuming and error-prone activity, and existing techniques supporting their automatic generation typically leverage real-world system logs, which are not available before the actual release of the software. To overcome these limitations, a novel technique to automatically generate workloads starting from existing E2E functional web tests is proposed. The effectiveness of the proposed technique is then evaluated in a preliminary industrial case study. Results are promising and show that workloads generated automatically using the proposed approach are comparable to those that are generated manually by practitioners.

Lastly, in the domain of GUI-level testing of mobile applications, the thesis work aims at addressing a lack in the literature and at supporting Software Project Managers in deciding which techniques to use to test a given mobile application. To this end, two empirical studies are conducted, involving twenty master's students as subjects. The first study compares the effectiveness of state-of-the-art automatic testing tools against

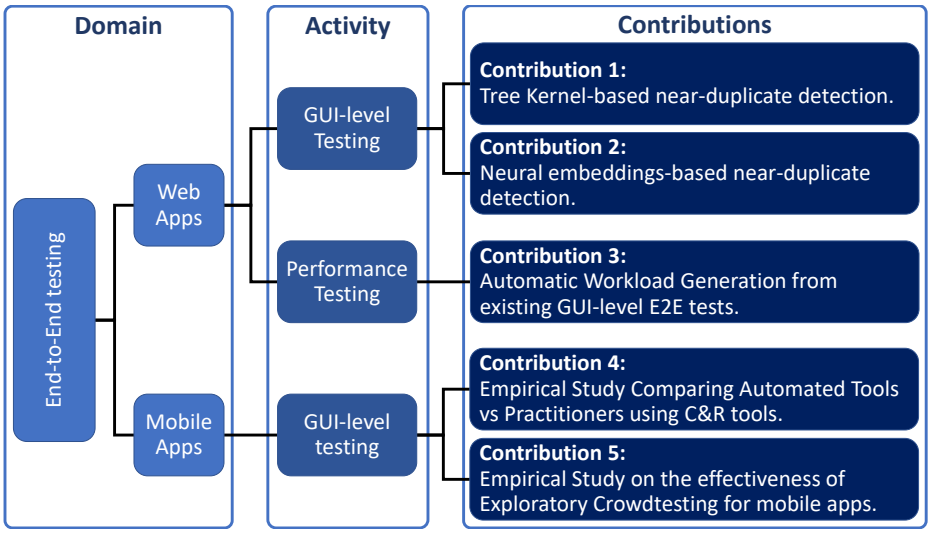


Figure 1.1. Highlights of the contributions of this thesis work.

that of unskilled practitioners using Capture and Replay tools with exploratory testing strategies. The second study investigates the effectiveness of crowdtesting in generating executable test suites for mobile apps, requiring crowdtesters to use exploratory testing strategies and capture and replay tools.

The main contributions of this thesis work are summarized in Figure 1.1.

1.1 Thesis Outline

The remainder of the thesis is structured as follows. Chapter 2 introduces some background notions on E2E testing and on its application in the Web and Mobile Applications domain, hinting at open challenges and limitations of existing approaches. Chapter 3 presents the work we conducted to improve the effectiveness of automatic E2E test generation techniques for web applications, by tackling the problem of near-duplicate states in web application model inference. In this chapter, we propose two different solutions for near-duplicate detection, and assess their ef-

fectiveness with an empirical study. Chapter 4 focuses on performance testing of web applications and presents a novel approach for automating the generation of workloads, starting from existing E2E GUI-level tests. The effectiveness of the proposed approach is evaluated in an industrial case study. In Chapter 5, we investigate the effectiveness of E2E testing techniques for Android applications by conducting two empirical studies. In the first study, we compare the effectiveness of automated test generation tools w.r.t. unskilled practitioners using capture and replay tools and exploratory approaches. Subsequently, we elaborate on the data collected in the first study and conduct a second study, aimed at investigating the effectiveness of Crowdtesting in generating test suites for Android apps. Lastly, in Chapter 6, we give some final remarks and highlight future research directions.

1.2 Origin of Chapters

In Chapter 3, the definition of the framework for near-duplicate detection (Section 3.2) is novel, currently unpublished work. The key ideas behind the Tree Kernel-based near duplicate detection (Section 3.3) have been published in a conference paper by Corazza et al. [61], and presented by the author of this thesis at the 2021 edition of the *ECOOP/ISSTA Doctoral Symposium*[1]. The work on the neural embeddings-based near duplicate detection (Section 3.4) is unpublished and currently under revision. Chapter 4 is based on unpublished work, currently under review. Chapter 5 includes work (Section 5.1) that has been published in the journal paper [66], and contains additional work (Section 5.2) that is currently under peer review.

[1] https://conf.researchr.org/track/issta-2021/ecoop-issta-2021-doctoral-symposium

Chapter 2

End-to-End Testing: Background and Related Work

This chapter provides some background notions on End-to-End (E2E) testing, focusing on the web and mobile apps domains, as well as an overview of the state of the art in this field, hinting at challenges and open issues.

The chapter is structured as follows. Section 2.1 introduces the E2E testing process. Section 2.2 focuses on GUI-level E2E testing, describing the main existing approaches (Section 2.2.1), the state of the art in the web applications domain (Section 2.2.3) and in the mobile apps domain (Section 2.2.4).

Subsequently, Section 2.3 focuses on performance testing, a particular type of non-functional E2E testing that is largely applied in the web applications domain. Section 2.3.1 provides an overview of the different kinds of performance testing activities, while Section 2.3.2 discusses the key common phases of performance testing. Section 2.3.3 presents an overview of the state of the art w.r.t. to performance testing, and Section 2.3.3 describes in detail and compares existing tools supporting performance testing activities.

2.1 End-to-End Testing

E2E testing is one of the main approaches to improve the quality of web and mobile applications [119, 43, 32, 46]. In this kind of activity, testers exercise the Application Under Test (AUT) as a whole, in its entirety, simulating real-world usage scenarios. Typically, E2E testing aims at stressing the AUT from the perspective of an end-user interacting with its Graphical User Interface (GUI), to assess its functional correctness. More in detail, these GUI-level testing activities aim at verifying that the application behaves as intended in response to user-generated events and interactions with the GUI (e.g., clicks, scrolls, forms filling and submissions, etc.).

Moreover, E2E testing approaches can also be used to validate non-functional requirements. For example, performance testing techniques are typically used to ensure that a web application and its current deployment configuration can handle expected workloads while guaranteeing the required Quality of Service.

In the remainder of this chapter, we describe in greater detail GUI-level testing (Section 2.2.3) and performance testing (Section 2.3) in the web and mobile apps domain, providing, for each, background information and an overview of the state of the art.

2.2 GUI-level Testing

In this section, we start by providing, in Section 2.2.1, an overview of the existing approaches for GUI-level testing of web and mobile apps, from a technical point of view. Then, we present Exploratory Testing (ET) (Section 2.2.2), a widely used strategy to design effective GUI-level test cases, and crowdtesting (Section 2.2.5), an emerging testing paradigm that proved its effectiveness in E2E testing for the considered domains. Subsequently, in Section 2.2.3 and Section 2.2.4, we present the state of the art for GUI-level testing for the web and mobile applications domains, respectively.

2.2.1 Overview of Approaches for GUI-level Testing

In the following, we consider existing approaches for GUI-level testing in increasing order of automation level.

Manual Testing

The easiest approach to E2E functional testing consists in requiring testers to manually interact with the AUT, visually assessing that it behaves as specified in its requirements. In this case, practitioners can either be assigned a set of test cases to be manually replicated [191] (also referred to as *scripted* testing [111]), or be left free to follow their own sensibility in interacting with the AUT, in so-called ET approaches [109, 100, 99].

Manual testing is affected by many drawbacks, severely limiting its effectiveness [228]. Firstly, it is remarkably time- and resource-consuming. For complex applications, it may take up to several workdays to manually complete the testing process, which needs to be repeated on each new release of the software [169]. Secondly, it can be error-prone. Practitioners performing repetitive and tedious interactions with the AUT are likely to make mistakes, which may have an impact on software quality [187].

Programmable Tests with Automation Frameworks

To overcome the limitations of manual testing, E2E test automation frameworks, such as Selenium [179] or Espresso [71] have been proposed. These frameworks allow testers to easily write code that interacts with the AUT, generating GUI events such as button clicks, scrolls, filling of input fields, etc. By using these frameworks, testers can manually implement automated test cases that simulate user interactions with the AUT, and automatically verify assertions. An example of Java code implementing an E2E test case using the well-known Selenium automation framework is shown in Listing 2.1.

Automated test suites can be readily re-executed every time the need arises, for example, to check for non-regression on each new release in evolutionary scenarios. Still, developing a thorough E2E test suite using automation frameworks is a time-consuming and error-prone task, and requires testers with adequate programming expertise [111]. Moreover, tests implemented with automation frameworks are also prone to breakage,

Listing 2.1. Example of GUI-level test case written in Java using the Selenium automation framework.

```
@Test
public void testAddReview(){
    driver.get("http://localhost/e-shop/"); //go to homepage
    driver.findElement(By.name("book-a")).click();
    driver.findElement(By.name("add_review")).click();
    driver.findElement(By.name("rev")).sendKeys("review1");
    driver.findElement(By.name("submit")).click();
    String actualTitle = driver.getTitle();
    String expectedTitle = "Review added correctly!";
    Assert.assertEquals(expectedTitle, actualTitle);
}
```

i.e., they can stop working in presence of even minor changes to the GUI of the AUT [123]. Thus, they need appropriate (and costly) maintenance [58].

Capture & Replay

Capture & Replay (C&R) tools allow testers to automatically generate re-executable tests by simply interacting with the AUT and "*capturing*" (recording) these usage scenarios. From an ease-of-use perspective, C&R is a very effective alternative to manually writing test scripts, as it enables even testers with limited testing knowledge to generate tests, reflecting real usage scenarios, in little time. Anyhow, the C&R approach is not exempt from limitations. For example, just as manually-implemented tests, those recorded with C&R tools are fragile w.r.t. app evolution, and may require re-recording when the GUI of the AUT changes.

Automatic Test Generation

Even with C&R approaches, generating recordings and maintaining test cases remains a costly manual practice. Automated test generation techniques aim at completely automating the test generation process, thus relieving developers and testers from such a burden. Indeed, these techniques can automatically exercise the AUT by generating input events according to different exploration strategies, like pseudo-random fashion

[139], by following a model-based approach [38], or other systematic approaches [14].

Automatic test generation tools, however, generally lack domain knowledge, and may generate either unrealistic test cases or fail to find test cases that explore aspects of functionality that matter to users [40, 136], also due to the presence of the so-called 'gate' GUIs, i.e., screens that require a very specific input to proceed with the exploration [193].

2.2.2 Test Design with Exploratory Testing

E2E test design is a creative activity, that can greatly benefit from leveraging testers with different backgrounds and experience. The diversity of testers' performance has been the subject of numerous empirical studies in the past [214, 34, 108, 17], with the aim of comparing the capability of students and/or professionals in fault-finding tasks with those of testing techniques or automatic tools.

In the design of E2E GUI-level test cases, i.e., deciding which sequences of user actions should be used to test the AUT, ET [3, 109] is a widely-adopted strategy in the web and mobile apps domains [100, 98, 99, 210], recognized as fundamental, especially in manual testing activities such as recording test cases using C&R tools [99]. ET emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of their work by considering test design, execution, and result interpretation as mutually supportive activities that run in parallel throughout the project, in a creative, experience-driven approach [99].

2.2.3 GUI-level Testing of Web Applications

In this section, we discuss the state-of-the-art of GUI-level testing of web applications. We start by describing automation frameworks for programmable web testing and C&R tools and then focus on automatic web test generation and its challenges.

Web Test Automation Frameworks

In the web apps domain, most test automation frameworks operate at GUI-level by leveraging the Document Object Model (DOM) representation of web pages to locate and interact with GUI elements. Examples

of frameworks using such an approach include Selenium [179], Cypress [103], and Playwright [150]. These solutions have reached a high level of maturity and popularity [169]. A crucial aspect of developing tests using DOM-based automation frameworks is the specification of adequate locators, i.e., constructs used by automation frameworks to identify the web elements on which the interactions or assertions should be performed [122]. XPath [54] expressions are widely used to this end, due to their flexibility and expressive power. Selecting effective locators can be a challenging task. Indeed, locators that are too specific or too general tend to be more prone to breakage when applied on new versions of the AUT, with changes in the layout of the web page. In the last years, a competing category of tools has also appeared, based on computer vision techniques to identify the elements on which interactions should be performed. Examples of this kind of framework include Sikuli [222] and JAutomate [6], and identify the elements with which to interact using screen captures and image matching techniques. Using a visual approach usually results in more robust tests, when evolutionary changes in the AUT do not affect the looks of the web elements. In [121], Leotta et al. conducted an empirical study comparing layout-based and visual locators in the web context, finding that layout-based locators are generally more robust, and associated with lower test development and test execution times. Their study also highlighted that, in some web applications in which structural changes were more frequent, visual locators performed better.

Capture and Replay

A number of solutions exist to support C&R of web application interactions. For example, Selenium IDE [1] allows users to record interactions and automatically generate executable Selenium tests. Similarly, frameworks such as JAutomate [6] or EyeAutomate [72] allow for the recording of usage scenarios and automatically generate executable tests using visual locators. As for the effectiveness of C&R in the web domain, Leotta et al. [120] conducted an empirical study comparing programmable web testing (using Selenium) and C&R (using Selenium IDE) in the context of web test evolution. Their analysis confirmed that, as expected, the initial development costs of a web test suite are noticeably greater when using the programmable test approach. On the other hand, however, their re-

sults highlighted that programmable tests required significantly less maintenance as the AUT evolved, making the cumulative costs of programmable web testing become lower than those of C&R after a small number of new AUT releases.

Automatic Generation of GUI-level E2E Tests

A number of works have investigated techniques for the automatic generation of E2E GUI-level web tests. Most of these works rely on state-based models of the AUT. Typically, in these models, each state represents a distinct high-level functionality of the AUT, while transitions between states represent navigability relationships.

Andrews et al. [18] propose a test generation approach based on a hierarchical finite state machine model of the AUT, aimed at reaching full transition coverage. Biagiola et al. [36, 39] use Page Objects defined by developers to guide the generation of tests. Marchetto et al. [140] propose a combination of static and dynamic analysis to model the AUT into a finite state machine and generate tests based on multiple coverage criteria. Mesbah et al. [149] propose ATUSA, a tool that leverages the model of the AUT produced by Crawljax to automatically generate test cases to cover all the transitions of the model, with a set of customizable invariants as test oracles. Biagiola et al. [38] propose DANTE, an automated approach to test generation, aimed at producing minimal test suites from web app crawlings. DANTE turns the raw output of a crawler into executable test cases by reusing the same inputs used upon crawling, resolving dependencies among tests, and eliminating redundant tests. Sunman et al. [189] AWET, an approach that leverages existing test cases, possibly obtained via capture-and-replay in an exploratory testing fashion, to guide crawling.

The state-based models used in these approaches can be either manually generated by developers, or automatically inferred, e.g., through crawling. Broadly speaking, crawling-based model inference techniques dynamically and systematically analyze the AUT starting at an initial web page and then explore the application by generating GUI events and checking the responses. When, as a consequence of a fired event, changes in the web page are detected, a new state is added to the model. From a testing viewpoint, these inferred models should contain a minimal set of significantly different states, yet adequately cover all the functionalities of the AUT.

In practice, however, models inferred automatically through crawling, like those of the above-mentioned works, are often affected by *near-duplicate* states [78, 65, 89, 135], i.e., replicas of the same functional web page differing only by minor changes. Near-duplicates have a detrimental impact on the quality of automatically-inferred models, and thus on the effectiveness of the resulting automatically-generated E2E web test suites [218], as explained in the following.

Near-duplicate States in Web App Model Inference

As the detection of near-duplicate states in web app models is a key issue faced in this thesis work, in this section, we provide a deeper insight on the problem and the state of the art of automatically retrieving an accurate web app model for test generation, and on the detecting its near-duplicate states.

Automated Web Model Inference Automated web model inference techniques, such as crawling, operate through state exploration by triggering events (e.g., clicks) and by generating inputs that cause state transitions in the web app. Whenever significant changes in the current web page are detected, a new *state* is added to the model. A state can be viewed as an abstraction of all the dynamic, runtime versions of the same logical web page, often represented by their DOMs. The final model is a set of states, i.e., the set of abstract web pages of the web app, and edges that represent transitions between states.

We use as a running example a simple e-commerce web app showing a product catalogue. A user can view the details of each product, add a review, and buy it (Figure 2.1). From a functional testing viewpoint, a manually-generated web app model for the running example, in terms of logical states and functionalities, is shown in Figure 2.2. The model includes four states, namely *Catalog page*, *Detail page*, *Review Page*, and *Buy page*. From the Catalog page, it is possible to navigate to the Detail page by clicking on a product. From a product Detail page, it is possible to either write a review for the product, which leads back to the Review page or buy the product, which causes a transition to the Buy page. Upon submitting a review from the Review page, the web app returns to the detail page for that product. On the Buy page, after filling out a form,

Figure 2.1. Example of an e-commerce web application.

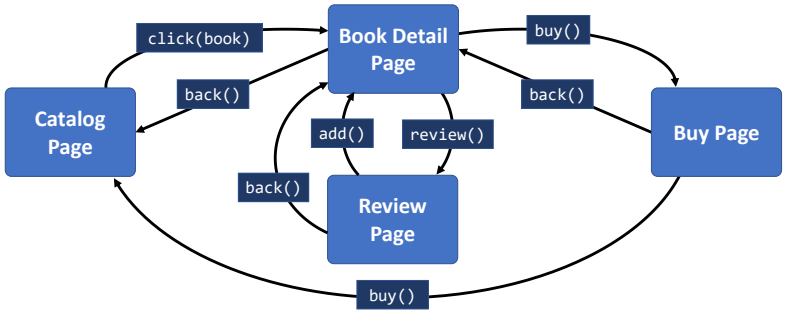


Figure 2.2. Optimal web app model.

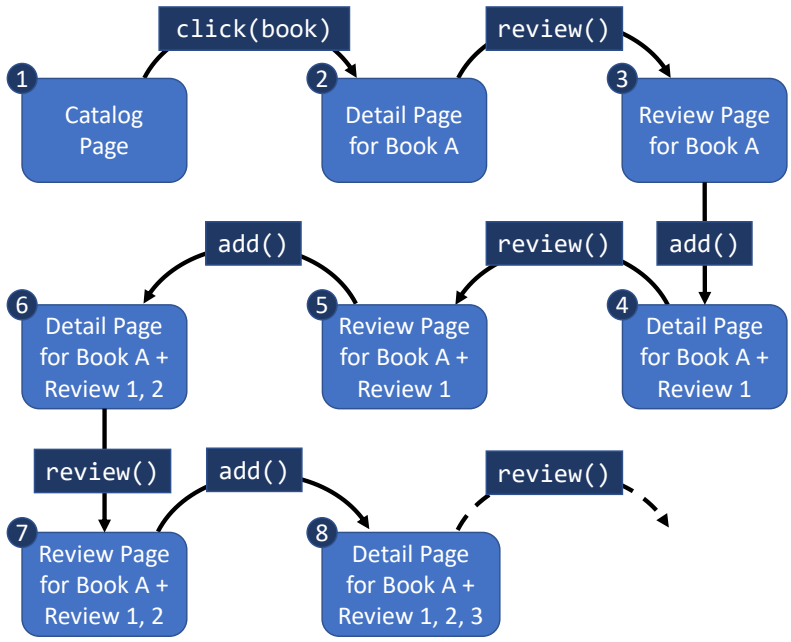


Figure 2.3. Incomplete crawl model w.r.t. the "Buy" functionality due to redundant near-duplicate states for the same "Detail page".

users can place orders and are subsequently redirected to the Catalog page.

Near-duplicate States and their impact on testing Figure 2.3 shows a crawl model for the running example, generated by the state-of-the-art crawler Crawljax [149] with its default configuration, i.e., (1) exact matching is used to determine whether a new web page needs to be added to the model, i.e., all dynamic states corresponding to slightly different web pages are regarded as new states; (2) an ordered GUI events queuing strategy that considers HMTL elements from top to bottom and from left to right; (3) a depth-first exploration strategy.

This model is affected by both redundancy (i.e., contains a number of near-duplicate states for the Detail and Review states) and completeness issues (i.e., it got stuck in a loop adding reviews, and will never get to visit the Buy page). To clarify the reasons behind these issues, in the following,

we describe the crawling process in a step-by-step fashion. Firstly, once the web app is loaded, the crawler saves the initial home page (also called *index* page) as the first state of the crawl model (i.e. State 1). In our running example, the index page is the Catalog page. Then, the crawler clicks on the first displayed product, i.e., *Book A*, which leads to the web page showing details for that book. Such page is saved as a state into the crawl model (*Detail Page for Book A*, i.e. State 2) and marked as *unvisited*. Next, the crawler clicks on the "*Add Review*" button and is redirected to the *Review Page for Book A*, which is added to the model as State 3. Upon filling out the form and submitting a review, the crawler is then redirected back to the *Detail Page for Book A*, which now displays also the newly added review. Since the "*new*" version of the detail page is different from the previously visited version added to the model as State 2, the default state abstraction strategy regards it as a new state, which is added to the model as State 4 and marked as *unvisited*. The crawler then continues its exploration for State 4, adding a new review and resulting in States 5 and 6 being added to the model. This process repeats until the crawler runs out of its allowed time budget.

In the model obtained via crawling, a number of states representing the Detail page for Item A are present. In the literature, such replicas of the same logical page, like the detail pages of our example, are known as clones, or near-duplicate states [217]. The presence of near-duplicate states in web app models has a detrimental effect on the effectiveness of model-based test generation techniques, in terms of *conciseness* and *completeness*.

Concerning the former, the presence of near-duplicates typically leads to test suites containing many redundant tests exercising the same functionality. In our running example, it would be sufficient to cover the Detail page only once with a test case, as covering all potential detail pages with many redundant test cases is unlikely to increase the code coverage achieved by the test suite or to expose new faults [217].

As for completeness, when exploring large web apps, crawlers may waste a considerable part of their time budget visiting near-duplicate states, without exploring other relevant parts of the application. This harms the completeness of the inferred models, and thus the associated test suites.

In the running example, the crawler failed to recognize that the 'new'

updated Detail page for Book A, featuring the reviews, was a *near-duplicate* of the previously-visited Detail page. Therefore, the crawler consumed the entire time budget stuck on these pages, failing to explore other significant parts of the application, such as the "Buy" functionality, leading to an incomplete model.

Notice that configuring the crawler to run with a breadth-first strategy in place of the default depth-first one would solve neither the conciseness issue nor the completeness one. Indeed, in that case, the crawler would add a new Detail page state to the model for each of the (possibly many) books in the catalogue. This would result in a model with significant redundancy, as well as possibly consuming most of the allowed time budget, leading to an incomplete model as well.

State Abstraction Function In practice, web crawlers try to detect whether a certain web page is a near-duplicate of another previously-visited page in the model, and thus should not be added to the model, by means of State Abstraction Functions (SAFs).

The problem of designing a SAF can be framed as an *equivalence problem* [217]: given two web pages p_1 and p_2, the SAF determines whether $p_1 \simeq p_2$. The SAF used by the crawler is the main root cause for the lack of conciseness and completeness of automated crawl models [217]. Yandrapally et al. [217] showed that even state-of-the-art structural and visual SAF implementations lead to large numbers of near-duplicate states in automatically-inferred models. Moreover, their study showed that state-of-the-art SAF implementations are generally not "*able to accurately detect all functional near-duplicates within apps*", and highlights "*the need for further research in devising techniques geared specifically toward the inference of web test models*".

Automated Near-duplicate Detection

Many techniques from different domains have been defined, in different contexts, to design effective SAFs for the near-duplicate detection of web pages, especially *across* different web applications. For instance, the problem of detecting duplicate and near-duplicate web pages arises naturally in the web indexing process of search engines. In this field, the concept of duplication and near-duplication is mainly related to the content of the

web page, and hence Information Retrieval techniques have been found to be quite effective [89]. Since performance is a crucial issue in this domain, due to the amount of involved data, content hashing techniques have been widely adopted thanks to their design simplicity and speed of comparison. Notable examples include the *shingling* algorithm presented by Broder et al. in [41], and the *Simhash* algorithm [45], which is also used by Google in its web page indexing process [135]. In [89], Henzinger carried out a large-scale evaluation of these algorithms on a set of 1.6B distinct web pages, showing that both achieve high precision in detecting near-duplicate web page pairs across different websites, while performing significantly worse in detecting near-duplicated within the same website.

Detecting near-duplicate pages is also a challenge for automatic phishing detection. In this context, malicious websites are often designed to look as similar as possible to the original website they try to impersonate while maintaining an entirely different HTML structure to avoid detection. Hence, techniques from the Computer Vision domain have often been applied to screen captures of web pages with good results [5, 209].

Among those visual-based techniques, the most fine-grained approaches focus on individual pixels composing the image. Examples of such techniques are *colour-histogram* [190] and *Perceptual Diff* (PDiff) [220], which have also been successfully applied in a previous web testing work for detecting cross-browser incompatibilities [133]. Some visual approaches operate at a coarser-grained scale, aiming at quantifying structural similarity or at extracting features from images. Structural similarity-based techniques leverage the intuition that images (and in particular screen captures of web applications) are typically highly structured, and their pixels, especially when they are spatially close, exhibit strong dependencies that convey important information about the structure of the represented objects. Similarity measures such as *Structural Similarity Index* (SSIM) [208], which has been successfully applied in the detection of phishing websites [47], take into account these spatial correlations.

Other visual techniques are based on image hashing, aiming at computing identical or nearly-identical digests for similar images, e.g. the screen captures corresponding to near-duplicate web pages [80]. Examples of image hashing algorithms include block-mean hash [219] and perceptual hash (pHash) [225]. Less work, however, has been directed towards defin-

ing SAFs for the detection of near-duplicate web pages *within* the same web application and with the specific goal of supporting web application testing.

Notable examples of techniques used to detect near-duplicate web pages in the web testing domain include the *Tree Edit Distance*, which is defined as the minimum number of node edit operations that transform one tree into another, and the *Levenshtein Distance* [124], defined as the number of character deletions, insertions, or substitutions required to transform a string into another. Tree Edit Distance, which can be computed efficiently using the Robust Tree Edit Distance (RTED) algorithm [166], can be straightforwardly applied to tree-structured DOM representation of web pages and has been employed in web crawling to detect near-duplicate web pages [74]. The Levenshtein distance, on the other hand, can measure the similarity of two web pages by comparing their HTML contents as strings, as done in [148].

More recently, in [218], Yandrapally, Stocco, and Mesbah presented a comparative study in which 10 different SAFs, based on techniques from different domains (including the aforementioned simhash, PDiff, color-histogram, SSIM, pHash, RTED), are applied and evaluated in the context of model inference. That study showed that none of the considered algorithms borrowed from the domains of information retrieval and computer vision "*is able to accurately detect all functional near-duplicates within apps*", and "*underlined the need for further research in devising techniques geared specifically toward web test models*".

2.2.4 GUI-level Testing of Mobile Applications

In this section, we give an overview of the C&R and Automated Input Generation (AIG) techniques and tools for mobile applications, hinting at their strengths and weaknesses, and at the studies carried out in the literature to evaluate and compare their effectiveness.

Programmable GUI Testing for Mobile Applications

A number of test automation frameworks have been proposed and are used to achieve test automation in GUI-level testing of mobile apps. On the Android platform, solutions such as Robotium [170], UI Automator [198],

Monkeyrunner [153], XCTest [96], or Espresso [71] exist. These solutions provide primitives to programmatically interact with the GUI of the AUT, access the mobile platform functionality, and monitor its behaviour via assertion statements.

Also in the mobile apps domain, manually writing test scripts is recognized as a time-consuming and error-prone activity, and requires testers with adequate programming knowledge. Moreover, functional tests generated using this approach are usually quite fragile [59, 58], i.e. they often fail or need maintenance in presence of GUI modifications, that are part of the natural evolution of the AUT. This fragility issue makes manually-written test scripts quite expensive to maintain as the app evolves, and this can also lead to inertia when there is a need to change the GUI and discourages many developers from using this GUI testing approach in the first place [105].

Capture and Replay Techniques and Tools

Some C&R tools for mobile applications are *coordinate-based*, i.e., they completely disregard the AUT GUI and record events by saving the exact display coordinates at which they occur. Notable examples of tools belonging to this category include Appetizer replaykit [30], RERAN [84], VALERA, proposed by Hu, Azim, and Neamtiu in [95], which besides recording GUI input events is also capable of recording sensor and network input, RandR [174], Mosaic [86], and OBDR [154].

Other approaches to C&R are *layout-based*, i.e., identify the GUI elements involved in the recorded events based on some unambiguous GUI layout property (e.g. unique IDs, query language expressions). Well-known examples of tools belonging to this category are Robotium Recorder [85] and Espresso Test Recorder [62], which is developed by Google as part of the Android Studio IDE and allows testers to generate Espresso test scripts. Both tools generate Android JUnit test cases by recording usage scenarios on a real or emulated Android device. Barista [75] is another tool supporting the capture of user events directly, but only on a real device.

Lastly, visual C&R tools such as Sikuli [222, 44] and EyeStudio [72], previously discussed in Section 2.2.3, can generally be applied also on mobile apps running inside an emulator.

Despite its appeal and ease of use, the C&R approach is not exempt

from limitations. Other than the previoulsy-mentioned fragility issue, C&R tools usually suffer from a trade-off between event-recording accuracy and portability of the recorded scripts: the more accurate the recorded events are (timing, exact position of touch events), the more coupled the generated test is to the device characteristics it was recorded on [127].

Some works compared the performance of different C&R tools. Garousi et al. [81] compared two visual GUI testing tools, namely Sikuli and JAutomate, in a real-world industrial scenario, in terms of features, robustness of the recorded traces, and test development effort. In [116], Lam et al. compared 11 different C&R tools with respect to three different aspects: their ability to reproduce common usage scenarios, the size of the traces created by the tools, and the robustness of traces created by the tools when being replayed on devices with different resolutions. A comparative experiment involving students has been recently presented by Ardito et al. [32], evaluating a layout-based tool (Espresso Test Recorder) vs. a visual one (EyeAutomate) in terms of productivity, i.e. the number of valid test cases recorded by students. Ardito et al. observed no significant productivity difference between the two tools.

Automated Input Generation Techniques and Tools

Even with C&R approaches, generating recordings and maintaining test cases remains a costly manual practice. AIG tools are capable of completely automating the input generation process, thus relieving developers and testers from such a burden. Indeed, AIG tools can automatically exercise an app by generating input events according to different exploration strategies, like pseudo-random fashion, or by following a model-based approach, or other systematic approaches.

The implementation of fully automatic testing tools able to stress the GUI of Android apps is a popular topic since 2011, as surveyed by some systematic mapping studies providing a view of the state of the art of the literature in this field [226, 13, 114, 193].

The first fully automatic testing tool, provided in all the Android releases, is Monkey[1], a simple tool able to generate sequences of random user and system events on the AUT. In order to perform more efficient

[1] https://developer.android.com/studio/test/monkey.html

testing techniques, several different tools supporting the automatic and systematic exploration of the AUT, based on the analysis of its GUI, have been proposed. In particular, Android Ripper [12], Dynodroid [132], A3E [33], SwiftHand [50] are all able to explore a dynamically built GUI model of the AUT with different exploration strategies, including random and systematic ones.

The performance of these tools was compared in [51], revealing that the better ones were barely able to compete with the totally random testing tool Monkey, when executed for the same amount of time. The main reason for the success of Monkey is that it is able to generate random events faster than the other tools since it does not analyze the GUI of the application under test.

A more effective testing tool called Sapienz [139] was proposed in 2016. It is able to generate a set of executable test cases on an Android app, guided by a multi-objective search-based testing strategy aimed at optimising code coverage, failure-finding capability and test sequence length. It combines random and systematic strategies for test case generation and, as shown in [139], its performance overcomes the ones of all the other tools previously presented in the literature.

At last, some experimental cloud-based AIG testing tools have been proposed. A notable example of this is Google Robo[2], an automatic testing tool available on the Firebase platform that is able to generate user events according to a systematic testing strategy.

Comparisons between C&R and AIG testing tools

In the Android context, a recent study by Mohammed et al. [152] focused on the comparison between the traces generated by the random AIG tool Monkey and the executions performed by eight human users on five selected Android applications, without using a C&R tool. The comparison in this study uses an effectiveness metric based on the number of distinct triggered events. No significant difference was observed between the traces generated by Monkey and by the human users.

In a study by Mao et al. [136], the performance of the Sapienz tool has been compared with the ones obtained by a set of 434 testers in the context

[2] https://firebase.google.com/docs/test-lab/android/robo-ux-test

of a crowd-sourcing testing experiment. In this experiment, specific testing tasks were assigned to crowd testers and their effectiveness was measured in terms of the accomplished tasks and the coverage of the Activity classes of nine AUTs. The testers involved in this study have testing skills similar to the ones of graduate students. The study showed that the automatic tool and the crowd testers were able to achieve different results, thus the use of tests generated by crowd testers may represent a valid starting point for search-based approaches such as the one implemented in the Sapienz tool.

2.2.5 Crowdtesting

According to the widely-accepted definition presented by Howe et al. [94], crowdsourcing is the act of an organization outsourcing their work to an undefined, networked labour using an open call for participation. Crowdsourced Software Engineering (CSE) derives from crowdsourcing. Using an open call, CSE aims at recruiting global online labour to carry out various types of software engineering tasks, such as requirements extraction, design, coding, and testing [137]. The validity of the adoption of these approaches in the testing context (also referred to as *crowdtesting*) and the opportunity to use them in conjunction with automatic techniques and tools have been first recognized by Xie et al. [216]. Since then, crowdtesting has been assessed by the scientific community in several different contexts, including GUI-level testing of web [49, 197] and mobile [199, 67, 113, 7, 82] applications. Especially in the mobile applications domain, crowdtesting has proven to be particularly effective [137, 227, 88], and has been adopted by an increasing number of software organizations including Microsoft, Google, PayPal, and Uber [31], to test their mobile apps.

As reported by Steiner [186], a testing task can vary in three dimensions: (I) it can be divisible or unitary; (II) it can be characterized by an objective to maximize or optimize; and (III) the results obtained by different individuals can be additive, conjunctive or disjunctive. Crowdtesting tasks often are not divisible, thus the same task is assigned to a crowd of different testers. Moreover, they are based on effectiveness objectives to maximize, such as the number of found faults, covered scenarios, or covered code. Lastly, they are generally conjunctive because they contribute

to the objective with possible overlapping of the results.

Due to the differences in skills and backgrounds of different testers, selecting more than one tester generally provides better effectiveness, with an increase in costs. However, it is also possible that increases in the number of testers correspond only to slight increases in the overall effectiveness of the tests [204]. More specifically, a linear increase in the number of testers may cause a less-than-linear increase in their overall effectiveness and a reduction in their overall efficiency. The trade-off between the number of testers (and thus costs) and achieved testing effectiveness has been studied in several works, in particular in the context of crowdtesting. The problem of the optimal selection of a set of crowd workers in order to maximize the coverage of test requirements, and minimize the number of testers (and the consequent cost) is addressed in many papers [215, 201, 202] by proposing different search-based approaches. These approaches were tested on a series of historical data on the effectiveness of crowd workers who worked on the Baidu platform. These data show how there is a significant difference in terms of effectiveness between crowd workers and how it can be related to their experience measured from their performance on past tasks solved in the same platform. They also show how the growth trend of test effectiveness is slower and slower as the number of testers independently dealing with the same test task increases. Kamangar et al. [106], instead, try to correlate the test effectiveness of crowd workers to their personality type classified by a psychological approach.

Several other papers in the literature directly studied the relationship between the number of testers and the overall effectiveness of the produced tests. In the context of usability testing, Nielsen et al. [160] evaluated the influence of the number of testers and of their experience (distinguishing by students or professionals) on the overall capability of finding usability issues. They found that sets of five students were able, on average, to find 50% of the issues, while sets of 14 students were able to find 75% of issues. On the other hand, sets of five professionals were able, on average, to find 90% of issues, while the overall set of 15 professionals was able to find all the usability issues. Also in the context of usability testing, Sears et al. [178] found that the percentage of found issues increased from 41% to 61% when the number of testers was increased from 2 to 5.

In the mobile apps domain, a pilot study presented by Wang et al.

[204] on the basis of data from the Baidu crowdtesting platform observed three distinct phenomena: large variation in bug detection rapidity and cost among different applications, decreasing bug detection rates over time and plateau effect of bug finding curve. They proposed iSense [205], an estimation technique to evaluate the optimal trade-off between testing effectiveness (in terms of the number of different bugs detected) and cost. In addition, an analysis of the same dataset reported in [203] found that there is a large number of similar bug reports finding the same bugs, confirming the previously observed phenomena.

In a study by Mao et al. [136], an experiment involving 434 testers in the context of a crowd-testing experiment carried out on the Amazon Mechanical Turk platform showed that crowdtesters were able to achieve greatly varying results, complementary with the ones of automatic tools, thus the use of tests generated by crowd testers may represent a valid starting point for search-based approaches.

2.3 Performance Testing

Performance testing is an umbrella term including any kind of testing task focused on assessing that the System Under Test (SUT) behaves as expected, under different workloads. Broadly speaking, performance testing consists in generating synthetic workloads for the SUT, and in monitoring its behaviour to detect load-related issues. These testing activities play a critical role in providing acceptable quality levels for end users [223].

2.3.1 Types of Performance Testing

Depending on the specific goals at hand, it is possible to identify different types of performance testing. In the following, we discuss some of the most commonly-used ones.

- **Load testing** aims at ensuring that the SUT behaves as expected in production, under anticipated realistic load conditions resulting from interactions by controlled numbers of concurrent users or processes [104, 164].
- **Stress testing** focuses on evaluating the behaviour of a system under extreme load conditions, exceeding the expected field loads. It

can also be used to evaluate the system's ability to handle reduced availability of resources (e.g.: access to memory or servers) [9]. Stress tests can typically be derived from load tests, by extending the load beyond the expected limits, and can help in determining the maximum load a system can handle.

- **Spike testing** aims at evaluating the ability of a system to respond correctly to sudden bursts of peak loads and return afterwards to a steady state [223].

Performance testing is particularly important in the web applications domain [46], as such systems are often business-critical [61], and in many cases they must meet contractually-specified Service Level Agreements. Therefore, load-related issues can often cost companies up to millions of dollars. For example, it has been estimated that increasing the load time of Amazon.com web pages by merely one second may cost the company as much as $1.6 billion in lost sales per year [48, 68].

Workloads for web applications are typically characterized in terms of user sessions [146, 91]. A user session is a sequence of related requests or service invocations issued by the same user when interacting with the application. Each user generally interacts with the web application by alternating the sending of requests (e.g.: requesting a web page) and waiting for a time period (the so-called *think time*) after it has received the server response, before sending the next request.

2.3.2 Phases of Performance Testing

As previously mentioned, different types of performance testing exist, each with different purposes and peculiarities. Nonetheless, all of them share the same core phases: Test Design, Test Execution and Test Analysis. In what follows, we discuss each of these phases in detail.

Test Design Phase

The goal of the Test Design phase is to define suitable synthetic workloads. A workload is characterized by the behaviours of different groups of users. Each group represents concurrent users performing the same kind

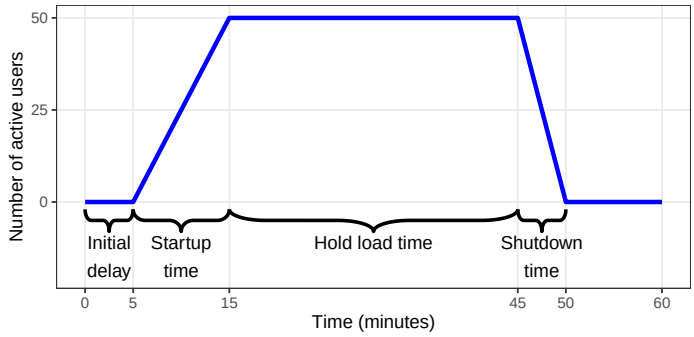


Figure 2.4. Example of a user group behaviour.

of interaction (i.e., the same *user session*). More formally, a user group is characterized by the following variables:

- **User session** The sequence of requests performed by all users in the group;
- **Number of users** The number of concurrent users in the group.
- **Initial delay** Indicates after how much time from the beginning of the performance test the user group becomes active.
- **Startup time** The time required to go from zero to the specified number of active users. Start-up time will be divided among each user as (Start Threads Count / Start-up Time).
- **Hold load time** The time for which all concurrent users will remain active.
- **Shutdown time** The time within which all active users must stop to send requests to the server.

As an example, consider the user group behaviour depicted in Figure 2.4. In the example, after an *initial delay* of 5 minutes, the user group activates. In the subsequent 10 minutes of *startup time*, an increasing number of users start interacting with the web application (5 additional users for each minute), until all the required 50 users are active. All these users will

remain active and continue interacting with the application for an *hold load time* of 30 minutes. Then, in the subsequent 5 minutes of *shutdown time*, users gradually stop interacting with the application, until no active user remains. Notice that a performance test is typically designed as a composition of several different user group behaviours, executed concurrently.

Test Execution Phase

Once a suitable workload has been designed, the performance test can be executed. The Test Execution phase includes (I) a *setup* step, focusing on deploying the SUT and the test execution environment; (II) *workload execution*, consisting in suitably generating synthetic traffic to the web application, as specified by the workload; and (III) *test monitoring and data collection*, which includes logging system behaviour (e.g.: CPU and/or memory usage metrics) during workload execution. The recorded data is then used in the subsequent Test Analysis phase [104].

Two approaches to workload execution exist, namely *Live-user-based* execution and *Driver-based* execution. The former is the most intuitive workload test execution approach and consists of leveraging a group of human testers. This approach reflects realistic user behaviours and allows performance testers to obtain real user feedback. On the other hand, it does not scale well, as it is limited by the number of available testers. *Driver-based execution* overcomes this scalability problem by leveraging suitably-defined scripts to automatically generate user requests, possibly using specialized tools such as Apache JMeter [20].

Test Analysis Phase

During the Test Execution phase, the system behaviour (e.g., logs and metrics) is recorded. These data must be then analyzed to assess whether the SUT meets the test objectives and if any load-related problem is uncovered. Huge amounts of performance counters (e.g., CPU, memory and throughput) and logs may be collected over extended periods of time, making a manual analysis basically unfeasible [48]. Hence, heuristics are typically used to detect potential problems. Some approaches verify that system metrics (such as CPU and memory usage) do not exceed prede-

fined threshold values, while other approaches focus on analyzing system behaviour data in search for known patterns linked to load issues [48, 104, 134].

2.3.3 State of the Art

Most of the approaches proposed in the literature for the automatic design of workloads leverage system log analysis. The idea behind these approaches is to start from real user behaviours, as captured by system logs, to infer realistic workloads. In [147], Menascè et al. proposed a workload characterization methodology to model the interactions between customers and e-commerce websites. Their approach consists of two phases. Firstly, it extracts a number of distinct user session logs from raw HTTP logs. Secondly, starting from the previously generated user session logs, a clustering analysis is conducted to group user sessions characterized by similar navigational patterns. For each cluster, they produce a state-transition graph describing the possible behaviours of that group of users. These models can then be used to generate realistic sequences of requests, and thus workloads, for e-commerce web applications. Other works proved that Markov chain models can be effective in representing the possible behaviours of users and thus in generating realistic workloads [141, 126, 172].

However, approaches based on state-transition models of user behaviours are affected by two key limitations. Firstly, they assume first-order dependencies between requests (i.e.: the next request depends only on the previous one), which is a limitation, given the articulate structure of many web applications.

Secondly, they do not allow for modelling data dependencies between subsequent requests. Indeed, in many scenarios, the parameters used in a given request may depend on values contained in the response to a previous request. For example, in applications that require authentication, the authentication token that is sent in each request is obtained by the client in the response to an initial authentication request. Similarly, the ID of a purchased item in a checkout request in an e-commerce web application must correspond to the ID of the item that was added to the cart in a previous request. Accounting for such data dependencies when modelling the behaviour of classes of users is crucial to guarantee that valid sequences of requests are generated.

Table 2.1. Comparison of Performance Testing tools

Characteristic	Load Runner	JMeter	LoadComplete	Gatling
Licensing	Commercial	Open-Source	Commercial	Open-Source/ Commercial
Capture & Replay	✓	✓	✓	✓
WebSocket support	✓	Via external libraries	✓	✓
WebSocket support (C&R)	✓	✗	✓	✗
Data Correlation detection	Manually specified rules.	Manually specified rules.	Manually specified rules.	✗
SUT monitoring	✓	Via external plugin .	✓	✗

Shams et al. [180] proposed a new application modelling methodology to overcome these limitations. Their approach relies on manually-defined Extended Finite State Machines (EFSM) models of the AUT, that can represent higher-order request dependencies, and allow testers to manually specify data dependencies between requests. Vögele et al. [200] further extend the approach proposed by Shams et al. and presented WESSBAS, a tool that is capable of automatically extracting an EFSM model from system logs. In that work, however, the authors do not address the problem of data correlation between requests.

Still, these approaches require the collection of large amounts of system logs, which limits their applicability in scenarios in which the SUT has not been yet released and alpha/beta testers are not available [48]. Moreover, analyzing large quantities of log data is a challenging and time-consuming task [165], which also lengthens the time to market in evolutionary scenarios in which the SUT changes and updated workloads should be generated as well.

Parrott et. al [165] proposed *Lodestone*, a solution to overcome the problem of obsolete workloads. Lodestone is a real-time data science approach to load testing, leveraging streaming of log data to dynamically generate and update user behaviour patterns, group them into similar behaviour profiles, and instantiate the distributed workload of software

systems. However, this approach is very expensive in terms of architecture and complexity, and its applicability is limited in scenarios in which the SUT has not been yet released.

State of Practice Overview

In this section, we briefly present state-of-practice tools for performance testing, focusing on the strengths and shortcomings of each one. In particular, we consider two largely used open-source tools, namely *Apache JMeter* [20] and *Gatling* [4], and two well-known commercial solutions, namely *LoadRunner* [129] and *LoadComplete* [128]. In Table 2.1, we summarize their characteristics.

Apache JMeter JMeter is an open-source, multi-platform framework for performance testing, supporting both the definition of workloads and their execution. JMeter allows practitioners to design workloads by either manually specifying a sequence of requests, through its Graphical User Interface (GUI), or by leveraging a Capture and Replay (C&R) module. With the latter approach, a tester can directly interact with the SUT as an end-user, and JMeter generates an executable workload by automatically capturing the requests that are sent to the SUT during the interaction. A key limitation of the C&R approach is the inability to record Websocket requests, which must be managed manually using an external module [173]. Furthermore, JMeter does not provide automatic correlation capabilities to address data dependencies, but only a third-party plugin [142] to support correlation detection through the use of manually-defined correlation rules [2, 118, 87]. As for monitoring the behaviour of the SUT, JMeter can be extended with the *PerfMon* plugin, which allows for the collection of load metrics (e.g.: CPU usage, RAM usage) on the SUT.

Gatling Gatling [4] is an open-source performance testing solution, designed to be easily integrated into continuous integration/continuous deployment pipelines. A commercial *enterprise* version is available, including support and additional features. In what follows, we consider the open-source version. Gatling does not allow practitioners to manually specify the requests in a workload but offers the possibility of using a C&R approach. Similarly to JMeter, however, the capture component in Gatling is

incapable of managing WebSocket requests and offers no automatic/partial correlation feature to cope with data dependencies, forcing practitioners to carry out additional time-consuming and tedious manual work. Gatling does not feature built-in monitoring capabilities, and practitioners need to set up appropriate tooling to collect relevant metrics on the SUT.

LoadRunner LoadRunner [129] is a widely-used commercial performance testing solution developed by Micro Focus [87]. Similarly to JMeter, LoadRunner allows users to manually design a workload by specifying a sequence of requests (using the *VUGen* IDE included with the tool) or to use a C&R approach. Unlike JMeter, however, the C&R feature shipped by LoadRunner is capable of recording also WebSocket requests. As for data dependencies, LoadRunner includes a plugin to support the detection of correlations, but the tool requires a time-consuming and tedious configuration of manually-defined rules. LoadRunner includes extensive SUT monitoring capabilities. Licences for using this tool, however, can cost up to several thousand USD per year, and, depending on the selected licensing plan, limitations on the number of virtual users and/or the duration of workloads might apply.

LoadComplete LoadComplete [128] is a commercial performance testing solution developed by SmartBear. It provides a GUI allowing practitioners to design performance tests manually or by using a built-in C&R tool, which is also capable of capturing WebSocket requests. Additionally, LoadComplete includes a plugin to support data dependency detection, but only using manually-specified rules [87, 102]. LoadComplete also features SUT monitoring capabilities and can generate custom reports [102]. Similarly to LoadRunner, LoadComplete is a costly commercial tool. Licences can cost up to several thousand USD per year, and limitations on the number of concurrent users and/or on the duration of workloads might exist depending on the licence.

2.3.4 Challenges in the Definition of Performance Tests

Summarizing, from our analysis of the state of the art, we identified a number of challenges that limit the productivity of practitioners and the effectiveness of the performance testing process. Such challenges are

reported in the following.

Avoiding the necessity of a deployed system to automatically generate performance tests The approaches proposed in the literature require large amounts of user logs. To collect adequate amounts of log data, the system under test needs to be deployed, but deploying an untested application is not a good practice [163].

Automatically managing data dependencies between subsequent requests For a test to run correctly, a request that is sent to a server might need to use a dynamic value that was returned by a previous response in the same session [180]. For example, when a user logs in to a web application, the server might issue a session ID, which is used to identify that user in subsequent requests and is dynamic (i.e.: after each login, the user receives a different session ID). In a similar scenario, performance tests must be parametric, i.e., take into account this data dependency [142].

Existing tools typically offer only partial support for this correlation task, allowing practitioners manually define a set of rules (regular expressions) for identifying correlations. Nonetheless, manually defining rules is still a time-consuming and tedious task, and requires practitioners to be familiar with the data flow between subsequent requests.

Coping with the evolution of workloads Web applications are constantly evolving, with new features being added or existing ones being updated. Moreover, the way users use a web application also can change over time. As a consequence, workloads may become rapidly obsolete, and need to be updated as well. In this evolutionary scenario, workload generation techniques that require long analysis times may become inapplicable [46, 165, 48].

Supporting emerging protocols such as WebSocket WebSocket [77] is an emerging network protocol supporting bidirectional connections where data is simultaneously exchanged in both directions. As shown in Figure 2.5, unlike classic Hypter-Text Transfer Protocol (HTTP), in which requests and responses alternate, with WebSocket, updates are sent imme-

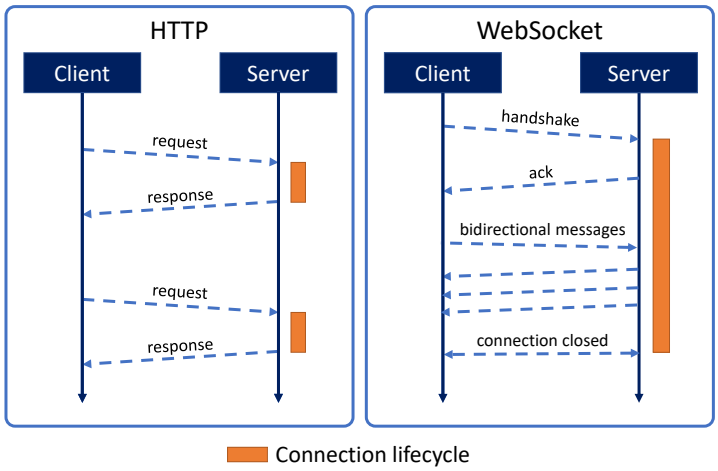


Figure 2.5. Connection lifecycles with HTTP (left) and with WebSocket (right).

diately when they are available, in a bidirectional fashion. These asynchronous, bidirectional communication patterns enabled by WebSocket allow practitioners to build true real-time functionalities, including chat, collaborative document editing, etc [184].

The introduction of WebSocket leads to an increase in Performance Testing complexity. Indeed, WebSocket messages in performance tests are difficult to handle, as they are not constrained by the alternation of requests and responses as in HTTP. With WebSocket, a single connection may correspond to multiple requests and multiple response messages, which significantly complicates the data correlation task.

Chapter 3

Improving Automatic Web Test Generation with Near-duplicate Detection

As discussed in Section 2.2.3, most automatic test generation techniques for web applications proposed in the literature leverage state-based models of the Application Under Test (AUT), in which states represent high-level features of the AUT, whereas transitions represent navigability relations. These models can be either defined manually by testers, which is a time-consuming and costly practice, or automatically inferred via systematic exploration (i.e., crawling) of the AUT. In the latter case, automatically inferred models are typically affected by the so-called near-duplicate states, i.e. states that correspond to slightly different web pages that nonetheless represent the same functionality from a testing viewpoint. Using models containing near-duplicate states has a negative impact on the quality of the resulting test suites, leading to a number of redundant tests and completeness issues.

The goal of the research presented in this chapter is to improve the effectiveness of automatic web test generation by improving the quality of automatically inferred models. To this end, two novel near-duplicate detection techniques are presented and their effectiveness as well as the practical impact of adopting them in automatic test generation are empirically evaluated.

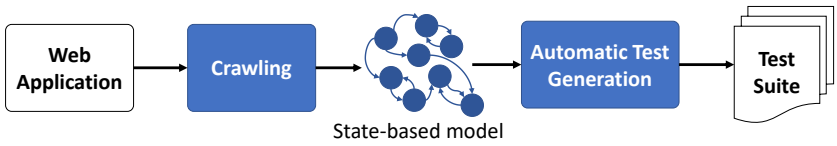


Figure 3.1. Considered reference scenario for automatic web test generation.

The remainder of this chapter is structured as follows. In Section 3.1, the automatic web test generation scenario we consider as a reference is described in greater detail, focusing on the key phases of model inference via crawling. Subsequently, the general framework for near-duplicate detection we propose is sketched in Section 3.2, and two novel techniques for near-duplicate detection are presented in Section 3.3 and in Section 3.4, respectively. Lastly, Section 3.8 presents some closing remarks and future works directions.

3.1 Reference Scenario for Automatic Web Test Generation

In this chapter, we consider a generic reference scenario for automatic web test generation, in which a state-based functional model of the AUT is automatically inferred via crawling. Subsequently, such an automatically-inferred model is used to automatically generate tests according to suitable strategies. Such a scenario, depicted in Figure 3.1, can be considered an abstraction of most automatic web test generation proposed in the literature (see Section 2.2.3).

In the remainder of this section, we discuss in greater detail the key crawling step in the reference scenario.

3.1.1 Crawling

The crawler loads the web pages in a web browser and exercises client-side JavaScript code to simulate user-like interactions with the web app pages. This allows the crawler to support modern, client-side intensive, single-page web applications. The main conceptual steps performed when

exploring a web application are outlined in the Crawl function of Algorithm 1.

Algorithm 1: Web app crawling process

```
1  Function Crawl(initial URL):
2      s_1 ← getState(initial URL)
3      model ← initializeModel(s_1)
4      while ¬ timeout do
5          next ← nextStateToExplore(model)
6          if next = nil then                          ▷ app exhaustively explored
7              break
8          s ← getToState(next)
9          for e ∈ getCandidateEvents(s) do
10             fireEvent(e)
11             s_c ← current state after firing the event e
12             if ¬ IsDuplicate(s_c, model) then
13                 add s_c to model
14     return model
15 Function IsDuplicate(s_c, model):
16     foreach state s' in model do
17         if SAF(s_c, s') = 'clone or near-duplicate' then
18             return True                             ▷ s is a duplicate of s'
19     return False
```

Crawling starts at an initial URL, the homepage is loaded into the browser and the initial DOM state, typically called *index*, is added to the model (Line 3). Subsequently, the main loop (Lines 4–14) is executed until the given time budget expires or there are no more states to visit (i.e., the web app has been exhaustively explored according to the crawler). In each iteration of the main loop, the first unvisited state in the model is selected (line 5), and the crawler puts in place adequate actions to reach said state. If the state cannot be reached directly, it retrieves the path from the *index* page and fires the events corresponding to each transition in the path. Upon reaching the unvisited state, the clickable web elements are collected (i.e., the web elements on which interaction is possible, line 9), and user events such as filling forms or clicking items are generated (line 10). After firing an event, the current DOM state s_c is captured (line 11). The IsDuplicate function supervises the construction of the model and checks whether s_c is a duplicate of an existing state (lines 16–19) by computing

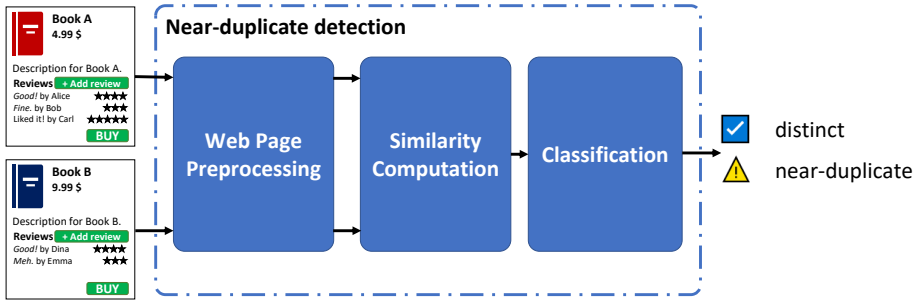


Figure 3.2. Overview of the proposed framework for near-duplicate detection.

pairwise comparisons with all existing states in the model using WebEmbed State Abstraction Function (SAF). The state s_c is added to the model if the SAF regards it as a distinct state, i.e., *a state that is not a duplicate of another existing state in the model* (lines 12–13). Otherwise, it is rejected and the crawler continues its exploration from the next available unvisited state until the timeout is reached.

3.2 Proposed Framework for Near-duplicate Detection

In this section, we describe the generic framework we abstracted to support the definition of novel near-duplicate detection techniques and the investigation of existing ones. The framework, which is depicted in Figure 3.2, identifies three key, abstract, steps in the process of detecting near-duplicates, namely: (1) *Web Page Preprocessing*, (2) *Similarity Computation*, and (3) *Classification*.

The *Web Page Preprocessing* step takes care of manipulating the input web pages, which are HTML documents, to make them suitable for similarity computation. Examples of preprocessing steps include generating a tree-based representation of web pages using the standard Document Object Model (DOM), obtaining a visual representation of the web pages by loading them in a web browser and capturing snapshots, or removing certain parts that are not deemed important in similarity computation.

The *Similarity Computation* step is responsible for computing a rep-

resentation of the degree of similarity between the two web pages. The most simple representation of similarity is a numeric similarity score, but in general, the similarity representation can be complex, structured objects, describing similarity w.r.t. a number of different aspects.

The *Classification* step takes as input a similarity representation and is responsible for classifying the web page pair as distinct or near-duplicate. The most simple classification approaches are threshold-based, i.e., discriminate between distinct and near-duplicates based on whether similarity is below or above given thresholds. In general, classification can be performed by training more sophisticated classification models, such as Decision Trees, Neural Networks, or Support Vector Machines (SVM).

3.3 Tree Kernel-based Near-duplicate Detection

In this section, we start by giving some preliminary notions on Tree Kernel functions in Section 3.3.1, and then we introduce the Tree Kernel-based near-duplicate detection approach we are investigating in Section 3.3.2.

3.3.1 Tree Kernel Functions

Tree Kernel (TK) functions are a particular family of kernel functions which specifically evaluate the similarity between two tree-structured objects. These functions have been extensively studied in Natural Language Processing [157].

To compute the similarity between two trees T_1 and T_2, TK functions consider, for each tree, a set of *tree fragments*. A tree fragment is a subset of nodes and edges of the original tree. Then, the similarity between the tree fragments of the two trees is evaluated, and the overall similarity of the two trees is computed by aggregating, in some meaningful way, the similarities of the single fragments. Depending on how the set of fragments to consider is defined, it is possible to characterize different classes of tree kernel functions. Widely-used classes include [156]:

- *Subtree Kernels*, which consider only proper subtrees of the original trees, i.e., a node and all of its descendants, as fragments.
- *Subset Tree Kernels*, which consider as fragments a more general structure than the one considered by subtree kernels, relaxing the

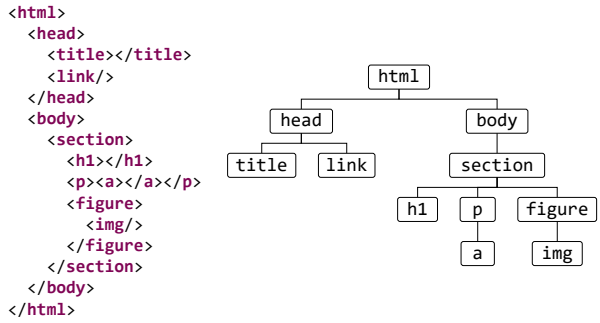


Figure 3.3. An HTML document (left) and its DOM representation (right)

constraint of taking all descendants of a given node and thus allowing for incomplete subtrees, limited at any arbitrary depth.

- *Partial Tree Kernels*, which consider an even more general notion of fragments, in which the constraint of taking either all children of a tree node or none at all is relaxed. In this case, it is possible to include only some of the children of a node in a fragment.

Tree Kernels in Software Engineering

TKs have also been applied with promising results in the Software Engineering domain. In particular, TKs have been applied on Abstract Syntax Tree representations of source code for clone detection [60], and their usage is also being investigated for test case prioritization tasks [11, 10]. More recently, [97, 181] presented an effective approach to fake website detection, which leveraged TK functions.

3.3.2 Proposed Approach

The key intuition behind the proposed TK-based near-duplicate detection approach is to use TK functions to compute the degree of similarity between two web pages which, as shown in Figure 3.3, can be naturally modelled using their tree-structured DOM representation. In the proposed solution we consider three well-known TK functions, namely *subtree kernel*, *subset-tree kernel* and *partial tree kernel*.

Table 3.1. Considered DOM transformation strategies

Strategy	Description
As-is	This transformation strategy leaves the DOM unchanged;
Only body	This transformation strategy considers only the DOM subtree rooted in the `body` element of the web page.
Only body, no scripts	This transformation strategy is the same as the "*Only body*" one, but also removes `script` elements along with their subtrees.

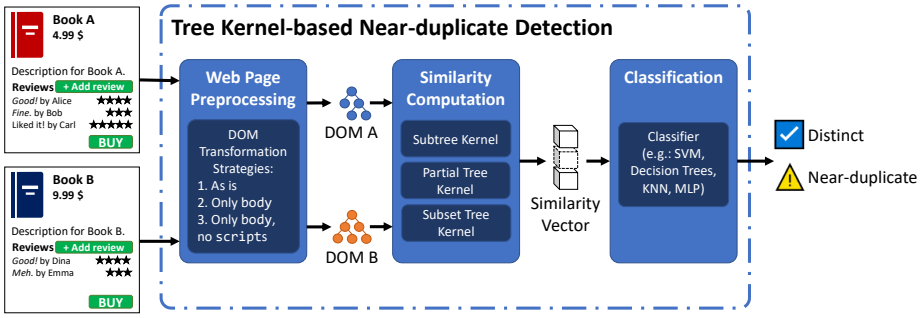


Figure 3.4. Overview of the proposed TK-based approach.

As for web page preprocessing, in addition to representing web pages using their DOM, we devised three different kinds of transformations, designed to investigate how different portions of the DOM tree impact similarity computation and near-duplicate detection, and to make the approach more general and customizable. In particular, we consider three basic DOM transformation strategies, as detailed in Table 3.1.

From the pairwise combination of the three considered TK functions and the three DOM transformation strategies, nine different similarity measures can be computed for a pair of web pages. These different similarity measures (or a subset thereof) are then aggregated in a *similarity vector*, in which each component corresponds to a different similarity measure. Leveraging these similarity vectors and existing open datasets with annotated web page pairs, we use supervised learning approaches to train an *ad-hoc* classifier. An overview of the proposed approach is depicted in Figure 3.4.

3.4 Neural Embedding-based Near-duplicate Detection

3.4.1 Neural Embeddings

Vector Space Models, first proposed by Salton et al. [177], allow objects to be represented as vectors in a multi-dimensional continuous space. In such vectors, the semantics of objects are "distributed" over vector components, so that similar objects correspond to points in the vector space that lie close to each other. Encoding information into a low-dimensional fixed-length vector representation enables easy integration in modern machine learning models, which often require input data to be represented as vectors, and proved to be very effective in the Natural Language Processing domain [194].

In that domain, the continuous vector space is usually referred to as the *semantic space* and the representations of the objects (e.g.: words, sentences, documents, etc.) are called *distributed representations* or *embeddings*. A popular, unsupervised way of learning low-dimensional embeddings for words or documents from large text corpora consists in leveraging neural networks, typically trained with some sort of language modelling objective. For example, Word2Vec [151], one of the most popular approaches for learning word embeddings, is based on a feed-forward neural architecture which is trained with language modelling tasks (e.g.: predict the current word given its context, i.e., the words that surround it). Similarly, Doc2Vec [117], which extends Word2Vec with the capability of learning embeddings for paragraphs and documents, is based on training a neural network on the task of predicting the next word in a document given the current document id, and a number of preceding words.

Doc2Vec aims to find an optimal embedding model such that similar text documents would produce embeddings that lie close in the vector space. Given a document, Doc2Vec creates and projects paragraph embeddings, as well as word embeddings, into the vector space and then uses a trained deep neural network model to predict words of paragraphs or documents in a corpus [117]. Instead of computing an embedding for each word like Word2Vec [151], Doc2Vec creates a different embedding for an entire paragraph or even a document. At inference time, the input para-

graph id vector (a one-hot encoded vector) is unknown, hence it is first derived by gradient descent given the input and output words and it is concatenated with the one-hot encoded vectors of the paragraph words to predict the next word in the paragraph. The internal representation used to make such a prediction is averaged or concatenated across predictions to get the final document embedding [117].

To learn the best vector representation for each document, Doc2Vec can be configured to use two different models: Paragraph Vector Distributed Memory or Distributed Bag Of Words. The former randomly picks a set of consecutive words in the paragraph and tries to predict the word in the middle, using the surrounding words, a.k.a., context words, and the paragraph id. The latter is similar to a `Skip-gram` model, in which, given a paragraph id, the model tries to predict the next word of a randomly picked sequence of words from the chosen paragraph [117].

Embeddings in Software Engineering

Neural embeddings proved to be effective in many code analysis tasks, such as code completion [52], log statement generation [144], code review [196] and other code-related tasks [143, 145].

Alon et al. [8] present Code2Vec, a neural model for learning embeddings for source code, based on its representation as a set of paths in the abstract syntax tree. Hoang et al. [90] propose CC2Vec, a neural network model that learns distributed representations of code changes. The model is applied for log message generation, bug fixing patch identification, and just-in-time defect prediction. Feng et al. [76] use representation learning applied across web apps for phishing detection. Lugeon et al. [130] propose Homepage2Vec, an embedding method for website classification. Namavar et al. [159] performed a large-scale experiment comparing different code representations to aid bug repair tasks. In this work, we propose an embedding method that works at a finer granularity level and that can integrate both structural (HTML tags) and textual (content) information. We study this embedding in the context of automated crawling and testing of web apps.

Among the grey literature, Ma et al. [131] propose GraphCode2Vec, a technique that joins code analysis and graph neural networks to learn lexically and program-dependent features to support method name prediction.

Dakhel et al. [63] propose dev2vec, an approach to embed developers' domain expertise within vectors for the automated assessment of developers' specialization. Jabbar et al. [101] propose to encode the execution traces of test cases for test prioritization.

3.4.2 Proposed Approach

The key idea behind the proposed neural embedding-based approach, which we call WebEmbed, is to leverage specifically-trained neural network embedding models to represent web pages as points in a continuous vector space. Then, the similarity (or lack thereof) of two web pages can be simply computed using standard measures such as *cosine similarity* [83]. With one or more similarity measures in place, obtained possibly using different embedding models and/or different similarity measures, it is possible to compose a similarity vector, as done also in the TK-based near-duplicate detection approach (see Section 3.3). Such similarity vectors can be used with supervised classification approaches to classify web page pairs.

More in detail, WebEmbed uses novel neural embedding models for web pages built on top of Doc2Vec [117] and leveraging different representations of web pages. WebEmbed requires two separate training phases. In the first phase, we train neural embedding models, which can be done using any corpus of web pages and does not require data to be manually labelled as clone or near-duplicate. In the second phase, a classifier is trained using supervised machine learning approaches, and a training set of annotated web page pairs is required. An overview of the proposed neural embedding-based approach is depicted in Figure 3.5

Web Page Embedding Models

Our approach requires computing embeddings for HTML web pages. To this end, we extend the well-known Doc2Vec [117] embedding model, originally conceived for general-purpose textual documents, to support web pages. In particular, we devised three different token extraction strategies to extract a convenient representation of web pages for training a Doc2Vec model, as described in what follows.

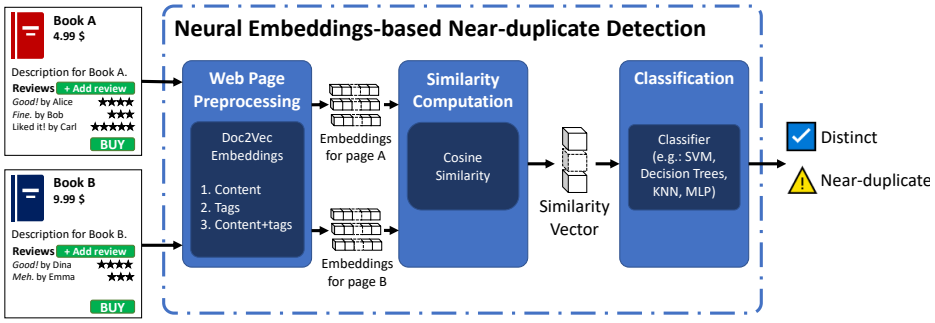


Figure 3.5. Overview of the WebEmbed approach.

Token sequence extraction The general procedure to extract token sequences from a web page is detailed in Algorithm 2. The procedure takes

Algorithm 2: Token Extraction

```
1 Function extractTokens(n, et):                    ▷ et: content, tags, or both
2     let tokens be an empty list
3     tokens.append(getTokens(n, et))
4     foreach children node c of n, from left to right do
5         if c is not a script, style, or comment node then
6             tokens.append(extractTokens(c))
7     return tokens
```

as inputs the root DOM node of the web page and a flag indicating which type of token sequence to extract, and proceeds as follows: (1) the sequence of tokens (either tags, content, or content+tags) for the current node are extracted (line 3); (2) the token extraction procedure is recursively called, in a depth-first fashion, on all children of the current node, from left to right. The result of these calls is then appended to the list of extracted tokens (lines 4–6); (3) the sequence of extracted tokens is returned (line 7).

Throughout this section, let us consider as a running example the HTML code corresponding to the Detail page for Book A (Listing 3.1). In the listing, tag names and attribute names are highlighted in boldface, content is highlighted in blue, and comments in green.

Listing 3.1. HTML document corresponding to the Detail page for Book A shown in Figure 2.1.

```
<html lang="en">
  <head>
    <title>Book A detail page</title>
    <link rel="stylesheet" href="styles.css">
    <script type="text/javascript" src="utils.js"></script>
  </head>
  <body>
    <img src="book_a.jpg" class="item_pic"/>
    <h1>Item A</h1>
    <img src="three-stars.png"/>
    <p class="price">4.99 $</p>
    <p class="descr">Detailed description for Book A.</p>
    <h2>Reviews</h2><!-- Reviews listed here -->
    <a href="/addrev?i=item_a" class="btn">+ Add Review</a>
    <table class="reviews">
      <tr> <!-- First review -->
        <td>Good! by <a href="/u/alice">Alice</a></td>
        <td><img src="three-stars.png"/></td>
      </tr>
      <tr> <!-- Second review -->
        <td>Liked it! by <a href="/u/bob">Bob</a></td>
        <td><img src="five-stars.png"/></td>
      </tr>
    </table>
    <a href="/buy?i=book_a" class="btn">BUY</a>
  </body>
  <script>init_page();</script>
</html>
```

Tags Token Sequence The first extraction function considers only the name of the tags from an HTML page while discarding comments, scripts, and CSS. The intuition is that tags indicate the general layout of an HTML document and may be effective for detecting structurally similar web pages [188]. Consequently, the tag token sequence of the web page in Listing 3.1 is as follows: [`html`, `head`, `title`, `body`, `img`, `h1`, `img`, `p`, `p`, `h2`, `a`, `table`, `tr`, `td`, `a`, `td`, `img`, `tr`, `td`, `a`, `td`, `img`, `a`].

Content Token Sequence The second extraction function only retrieves the textual content of a web page. Intuitively, two web pages sharing similar textual content have some degree of topical relatedness [188]. Consequently, the HTML in Listing 3.1 is converted to the following tokens of DOM content: [*Book, A, detail, page*, *Book, A*, *4.99, $*, *Detailed, description, for, Book, A*, *Reviews*, *+*, *Add, Review*, *Good, by*, *Alice*, *Liked, it, by*, *Bob*, *BUY*].

Content+tags Token Sequence The third extraction function considers both content and tags and combines the output of the two previous extraction functions. This can be effective in cases where using the tags or the content only is not enough to accurately classify two web pages. The HTML in Listing 3.1 is converted to the following content+tags token sequence: [`html`, `head`, `title`, *Book, A, detail, page*, `body`, `img`, `h1`, *Book, A*, `img`, `p`, *4.99, $* `p`, *Detailed, description, for, Book, A*, `h2`, *Reviews*, `a`, *+*, *Add, Review*, `table`, `tr`, `td`, *Good, by*, `a`, *Alice*, `td`, `img`, `tr`, `td`, *Liked, it, by*, `a`, *Bob*, `td`, `img`, `a`, *BUY*].

Model Implementation and Training Once the preprocessing for token sequence extraction is done, three different Doc2Vec models are trained, i.e., one model for each token-sequence type. Hence, we obtain three Doc2Vec models that allow us to compare pairs of web pages and thus compute their similarity based on one token-sequence representation of the pair at a time. For example, the following embeddings are produced

for the HTML of Listing 3.1:

$$doc2vec(tags) = [-0.25, 0.48, ..., 0.03]$$
$$doc2vec(content) = [-0.55, 0.17, ..., 0.90]$$
$$doc2vec(content + tags) = [-0.40, 0.33, ..., 0.44]$$

Training State Abstraction Functions

Once the embedding models are trained, it is possible to train a classifier implementing a SAF. This task requires a *labelled* corpus of web pages, in which each web page pair is manually annotated with a label indicating whether the web pages in the pair are clones/near-duplicates.

For each pair of web pages in such corpus, we use one of the different Doc2Vec models to compute their embeddings. Then, we compute the cosine similarity [183], a widely used metric to assess vector similarity. A combination of the three similarity scores, based on content, tags, or content+tags neural embeddings, is used to train a classifier to discriminate two web pages as being distinct or clones.

Usage of the State Abstraction Function The Classify procedure in Algorithm 3 illustrates our neural-based SAF. Given two web pages p_1,

Algorithm 3: Web App Crawling with WebEmbed

```
1  Function Classify(p1, p2, ET):                    ▷ ET: embedding types
2      let s be an empty list
3      foreach embedding type et in ET do
4          r1 ← extractTokens(p1.getRootNode(), et)
5          r2 ← extractTokens(p2.getRootNode(), et)
6          doc2vec ← getDoc2VecModel(et)
7          e1 ← doc2vec.infer(r1)
8          e2 ← doc2vec.infer(r2)
9          s.append(cosineSimilarity(e1, e2))
10     return classifier.classify(s)
```

p_2 and a list ET of embedding types to consider, we first extract the token-sequence representations from each page based on the selected embedding types (ET can be any non-empty subset of {*content*, *tags*, *content+tags*}), obtaining one list of tokens for each web page (Lines 4–5). Each of the two token sequences r_1 and r_2 is then fed to the appropriate Doc2Vec model

(line 6) to compute an embedding (Lines 7–8). Then, the cosine similarity between the two resulting embeddings e_1 and e_2 is computed, obtaining a similarity score that is appended to the list s of similarities computed so far (Line 9). Next, the classifier marks the two pages as either distinct or clones based on the list s of similarity scores and determines the SAF return value (Line 10), which is '*clone*' in case of near-duplicate detection or '*distinct*' otherwise.

Example Consider the following embeddings produced for our running example, for the embedding type '*tags*' (i.e., $ET = [\text{'}tags\text{'}]$):

$$p_1 = \text{Catalog Page} \qquad e_1 = [-0.45, 0.56, ..., 0.30]$$
$$p_2 = \text{Detail Page A} \qquad e_2 = [-0.55, 0.17, ..., 0.90]$$
$$p_3 = \text{Detail Page A + Review 1} \qquad e_3 = [-0.56, 0.19, ..., 0.95]$$

During crawling, let us assume that a decision tree classifier flags a pair of pages as '*clone*' when the cosine similarity between their embeddings satisfies the root decision node condition ($s > 0.8$). If $sim(e_1, e_2) = 0.56$, p_2 is added to the model, as p_2 is not too similar to p_1. Then, when exploring p_3, we obtain $sim(e_3, e_1) = 0.58$ and $sim(e_2, e_3) = 0.95$. Hence, page p_3 is not added to the model as it is recognized as a near-duplicate ('*clone*') of p_2.

3.5 Empirical Study Design

To assess the effectiveness of the proposed techniques in detecting near-duplicates, as well as the practical impact of their application in the automatic web test generation process, we conducted an empirical study leveraging massive datasets of web pages available in the literature and nine open-source web applications. In this section, we detail the empirical study we conducted in terms of investigated research questions, employed data, baseline techniques, and experimental procedure.

3.5.1 Research Questions

To investigate the effectiveness of the proposed near-duplicate detection techniques and the practical impact of their application in automatic web test generation, we consider the following research questions:

RQ1 (near-duplicate detection): *How effective are the proposed techniques in distinguishing near-duplicate from distinct web pages?*

RQ2 (model accuracy): *How do the web app models generated using the proposed techniques compare to a ground truth model?*

RQ3 (code coverage): *What is the code coverage of the tests generated using web app models inferred with the proposed techniques?*

RQ1 aims at assessing the effectiveness of the proposed techniques in detecting near-duplicate web pages, framing the near-duplicate detection problem as a binary classification task. RQ2 aims at investigating the quality, in terms of completeness and conciseness, of web app models inferred using the proposed techniques. Lastly, RQ3 focuses on the impact of the proposed techniques when used in automatic web testing generation, specifically assessing the test suites generated by crawl models obtained using the proposed techniques in terms of code coverage of the web apps under test.

3.5.2 Datasets

We use three existing datasets available from the study by Yandrapally et al. [217], plus an additional dataset of web pages collected by the *Common Crawl* project [57].

The first dataset is called $\mathcal{DS}$ and contains 33,394 unique web pages derived from automated crawls (using Crawljax [148]) of 1,031 randomly selected websites from the top one million provided by Alexa, a popular website that ranks sites based on their global popularity (dismissed as of May 1, 2022). From these web pages, 493,088 distinct same-website web page pairs can be derived.

The second dataset, referred to as $\mathcal{RS}$, contains 1,000 state-pairs from $\mathcal{DS}$ that Yandrapally et al. [217] manually labelled as either clone, near-duplicate or distinct. Overall, $\mathcal{RS}$ contains 1,826 distinct web pages.

Table 3.2. Web page characteristics across the datasets

		Web page metrics					
		DOM (# nodes)		Source (# chars)		Text content (# chars)	
Dataset	**# pages**	Mean	Std.	Mean	Std.	Mean	Std.
$\mathcal{DS}$	33,394	821	960	107,055	160,897	7,309	10,503
$\mathcal{RS}$	1,826	665	687	91,124	127,116	5,964	8,487
$\mathcal{SS}$	1,313	212	287	16,234	17,320	1,335	1,262
$\mathcal{CC}$	368,927	401	913	51,097	70,541	6,139	14,642

The third dataset, $\mathcal{SS}$, contains ≈97,500 web page pairs from nine subject apps, which were also manually labelled by Yandrapally et al. [217] as clone/near-duplicate or distinct. These nine web apps, which are briefly described in Table 5.2, have been used as subjects in previous research on web testing [188, 187, 39, 37]. Five of them are open-source PHP-based applications, namely Claroline (*v. 1.11.10*) [23], Addressbook (*v. 8.2.5*) [22], PPMA (*v. 0.6.0*) [27], MRBS (*v. 1.4.9*) [28] and MantisBT (*v. 1.1.8*) [29]. Four are JavaScript single-page applications—Dimeshift (*commit 261166d*) [24], Pagekit (*v. 1.0.16*) [25], Phoenix (*v. 1.1.0*) [26] and PetClinic (*commit 6010d5*) [21]—developed using popular JavaScript frameworks such as *Backbone.js*, *Vue.js*, *Phoenix/React* and *AngularJS*.

As for the additional dataset, which we refer to as $\mathcal{CC}$, it contains 368,927 web pages available from the *Common Crawl* project [57], also used in previous research [130]. Similarly to $\mathcal{DS}$, the web pages in $\mathcal{CC}$ are also collected by crawling real-world websites. Table 3.2 reports analytics information about the web pages of the considered datasets in terms of DOM size, length of the HTML source, and amount of text content.

3.5.3 Baselines

Based on the study by Yandrapally et al. [217], we selected two baseline near-duplicate detection techniques for our study, namely RTED (Robust Tree Edit Distance) [166], and Perceptual Diff (PDiff) [221]. RTED is an efficient algorithm for computing the tree edit distance, i.e., the minimum number of node edit operations that can be used to transform one labelled

tree into another. The tree edit distance can be straightforwardly applied to tree-structured DOM representation of web pages, and has been employed in web crawling to detect near-duplicate web pages [74]. The other baseline, PDiff, is a visual-based technique which compares two web pages by analysing their screen captures, based on a human-like concept of similarity that uses spatial, luminance, and colour sensitivity. We chose them as baselines for the following reasons: (1) they were the best DOM-based and visual techniques for near-duplicate detection among those considered in [217], (2) they were used as a SAF for web testing purposes within Crawljax.

3.5.4 Use Cases

In our empirical study, we consider three different practical use cases, namely *Beyond Apps*, *Across Apps*, and *Within Apps*. These use cases differ in what concerns the datasets used to train the classifiers, and the associated labelling cost for developers, as detailed in what follows.

Beyond apps This use case aims at investigating the feasibility of a general-purpose model trained on web pages that are different from the ones it is tested on. Therefore, we train the classifiers on $\mathcal{RS}$ and test them on $\mathcal{SS}$. This use case requires *no labelling costs* to web developers, as the classifier we train on $\mathcal{RS}$ is supposed to be re-used as-is on any new web app.

Across apps This use case investigates the generalizability of the proposed solutions when applied to web apps similar to the ones on which the classifier was trained. Indeed, we train a distinct classifier for each of the nine web apps in $\mathcal{SS}$ in a *leave-one-out* fashion. In other words, for each i-th app in $\mathcal{SS}$, we train the classifier on annotated web page pairs from the remaining eight web apps, using the web page pairs from the current app as a test set. In this use case, developers are supposed to find and manually label all pages of web apps similar to the ones under test. A company may develop a few web apps in a given domain, investing in manual labelling of the near-duplicates of such apps to save the near-duplicate detection effort later, when a new app will be developed in the same domain.

Table 3.3. Considered use cases

	WebEmbed		
	Doc2Vec	Classifiers	
Use case	**Training Set**	**Training Set**	**Test Set**
Beyond apps	$\mathcal{DS} \cup \mathcal{CC}$	$\mathcal{RS}$	$\mathcal{SS}$
Across apps (for each App_i)	$\mathcal{DS} \cup \mathcal{CC}$	$\mathcal{SS} \setminus \text{App}_i$	App_i
Within apps (for each App_i)	$\mathcal{DS} \cup \mathcal{CC}$	80% App_i	20% App_i

Within apps In this use case, we train an app-specific classifier for each of the nine subject web apps. For each app in $\mathcal{SS}$, we use 80% of the web page pairs for training the classifier and the remaining 20% for testing. In this use case, developers are required to label a significant portion of the near-duplicate pages of the web app under test before a classifier can be trained and applied to the other pages of the same web app.

The characteristics of each use case are summarized in Table 3.3. For all use cases, WebEmbed relies on the embeddings computed by a common Doc2Vec model trained on the non-annotated pages of the considered datasets, namely $\mathcal{DS} \cup \mathcal{CC}$.

3.5.5 Procedure and Metrics

RQ1 (near-duplicate detection) For each considered use case, we evaluate different implementations of the TK-based approach and WebEmbed. More in detail, for the TK-based approach, we consider different compositions of the similarity vector: (1) we use a similarity vector containing all the nine different similarity scores arising from the combinations of DOM transformation strategies and tree kernel functions; (2) we use each distinct similarity score alone; (3) we use the tree best components as indicated by a preliminary Recursive Feature Elimination analysis, namely the ones obtained using each tree kernel function with the "*Only body, no scripts*" DOM transformation strategy.

As for the WebEmbed approach, we vary the token sequence used to train Doc2Vec. More in detail, we trained three different Doc2Vec models, one for each representation of the pages in the dataset $\mathcal{DS} \cup \mathcal{CC}$ (tags, content, content+tags). Concerning the training hyperparameters, we used the default parameters of the `gensim` [167] Python library and

Table 3.4. Ground Truth Models for the considered web apps.

	Logical Pages	Concrete Pages
Addressbook	25	131
PetClinic	14	149
Claroline	36	189
Dimeshift	21	153
PageKit	20	140
Phoenix	10	150
PPMA	23	99
MRBS	14	151
MantisBT	53	151

fitted the models for 100 epochs using a vector size of 100.

As for the classifiers, we evaluate a total of eight classifiers. We consider six machine learning classifiers, namely Decision Tree, Nearest Neighbour, SVM, Naïve Bayes, Random Forest, and Multi-layer Perceptron. We also consider their ensemble with majority voting and an additional threshold-based classifier. To measure the effectiveness in near-duplicate detection, we compute the accuracy, precision, recall, and F_1 scores of all the considered variants of the proposed approaches.

RQ2 (model accuracy) We compute the accuracy of the models obtained using WebEmbed w.r.t. the labelled ground-truth crawl models of the nine web apps in $\mathcal{SS}$. The crawl models contain redundant concrete states (web pages) that Yandrapally et al. [217] aggregated into the corresponding logical pages. Logical pages represent clusters of concrete pages that are semantically the same (i.e., that are near-duplicate). Table 3.4 reports details on the ground truth models provided by [217] for each of the nine considered web apps. These figures highlight that models inferred via automatic crawling can contain significant redundancies. For the Phoenix app, for example, only 10 distinct logical pages (i.e., features) exist, but crawling produced a model containing 150 states, with a 1,400% redundancy. To measure WebEmbed's model accuracy w.r.t. the ground truth, we compute the precision, recall, and F_1 scores considering the intra-pairs (IP) in common in the given model and the intra-pairs within each manually identified logical page in a given Ground Truth (GT):

$$p = \frac{|IP_{GT} \cap IP_{\text{WebEmbed}}|}{|IP_{\text{WebEmbed}}|} \qquad r = \frac{|IP_{GT} \cap IP_{\text{WebEmbed}}|}{|IP_{GT}|}$$

We also consider the F_1 score as the harmonic mean of (intra-pair) precision and recall. As an example, let us consider a set of 6 web pages $\{p_1, p_2, p_3, p_4, p_5, p_6\}$ with the following ground truth (GT) model: $\{p_1, p_2\}$, $\{p_3\}$, $\{p_4, p_5, p_6\}$. This means that p_1 and p_2 are instances of the same logical page, and so do p_4, p_5 and p_6, whereas p_3 is the only witness of its logical state. Suppose that a given model inference technique produces the following model: $\{p_1, p_3\}$, $\{p_2\}$, $\{p_4, p_5\}$, $\{p_6\}$. Such inferred model is affected by a completeness issue (the functionality corresponding to p_3 is not represented in the model, as p_3 is assigned the same logical page as p_1) and by conciseness issues (two near-duplicate states exist — the one with p_2 is a near-duplicate of the one containing p_1, and the logical state containing p_6 is a near-duplicate of the one containing p_4 and p_5. The intra-pairs in a state-based model of a web app are defined as the maximal set of web page pairs such that both states in the pair correspond to the same logical page in the model. In our example, the intra-pairs for GT are $\langle p_1, p_2\rangle$, $\langle p_4, p_5\rangle$, $\langle p_4, p_6\rangle$, $\langle p_5, p_6\rangle$, whereas the intra-pairs for the WebEmbed model are $\langle p_1, p_3\rangle$, $\langle p_4, p_5\rangle$. Thus, precision and recall for the model in the example are defined as:

$$p = \frac{|\langle p4, p5\rangle|}{|\langle p1, p3\rangle, \langle p4, p5\rangle|} = 0.5,$$
$$r = \frac{|\langle p4, p5\rangle|}{|\langle p1, p2\rangle, \langle p4, p5\rangle, \langle p4, p6\rangle, \langle p5, p6\rangle|} = 0.25.$$

The F_1 score for the inferred model in the example is $F_1 = 2pr/(p + r) = 0.32$.

RQ3 (code coverage) To assess the effectiveness of WebEmbed when used for web testing, we firstly integrate our neural embeddings-based near-duplicate detection approach within the well-known Crawljax web crawler [149]. Subsequently, we crawl each web application in $\mathcal{SS}$ multiple times, each time varying the SAF. For all tools and all use cases, we set the same

crawling time of 30 minutes. We use the state-of-the-art DANTE web test generator [38] to automatically generate Selenium test cases from the crawl models. Lastly, we execute each test suite, measuring the web app code coverage. For JavaScript-based apps (Dimeshift, Pagekit, Phoenix, PetClinic), we measure *client-side* code coverage using `cdp4j` 3.0.8 and the Java implementation of Chrome DevTools. For PHP-based apps (Claroline, Addressbook, PPMA, MRBS, MantisBT), we measure the *server-side* code coverage using the `xdebug` (v. 2.2.4) PHP extension and the `php-code-coverage` (v. 2.2.3) library.

We assess the statistical significance of the differences between WebEmbed and the baselines using the non-parametric Mann-Whitney U test [211] (with $\alpha = 0.05$) and the magnitude of the differences, if any, using Cohen's d effect size [56].

All our results, the source code of WebEmbed, and all subjects are available in a replication package [168]

3.6 Results

For the sake of conciseness, in this section, we report only the results achieved by the SVM classifier, which consistently performed better than all the other classifiers across all the use cases. Similarly, we report the results only for the best TK-based implementation, namely the one that considered similarity vectors containing three components, namely the ones obtained using each tree kernel function (subtree kernel, subset tree kernel, and partial tree kernel) with the "*Only body, no scripts*" DOM transformation strategy.

3.6.1 RQ1: Near-duplicate detection effectiveness

Results for the *Beyond Apps* use case are reported in Table 3.5, which shows, for each considered technique, accuracy (Acc.), precision (Pr.), recall (Rec.), and F_1 scores. Results highlight that, in the *Beyond Apps* scenario, both the TK-based approach and WebEmbed perform similarly to RTED, whereas they both perform remarkably better than the other baseline, PDiff, with more than 50% increase in accuracy. No significant differences in classification performance can be observed among the differ-

Table 3.5. RQ1 - Near-duplicate Detection (*Beyond Apps* use case). Best averages are boldfaced.

	Beyond Apps			
Technique	Acc.	Pr.	Rec.	F_1
TK-based (three TKs)	0.74	0.91	0.73	0.81
WebEmbed (content)	0.73	0.97	0.67	0.79
WebEmbed (tags)	0.75	0.98	0.70	0.81
WebEmbed (content+tags)	0.75	0.97	0.70	0.81
RTED	0.75	0.86	0.81	0.83
PDiff	0.48	0.81	0.43	0.56

Table 3.6. RQ1 - Near-duplicate Detection (*Across Apps* use case). Best averages are boldfaced.

	Across Apps			
Technique	Acc.	Pr.	Rec.	F_1
TK-based (three TKs)	0.74	0.86	0.80	0.82
WebEmbed (content)	0.82	0.89	0.88	0.87
WebEmbed (tags)	0.79	0.88	0.86	0.84
WebEmbed (content+tags)	0.83	0.89	0.90	0.87
RTED	0.77	0.85	0.86	0.80
PDiff	0.74	0.86	0.80	0.83

ent implementations of WebEmbed. As for the TK-based approach, it achieves a similar accuracy w.r.t. WebEmbed, while exhibiting a lower precision and a greater recall, meaning that its usage leads to a larger number of false positives (i.e., distinct web page pairs that are classified as near-duplicates), but to a lower number of false negatives (i.e.: near-duplicate web page pairs that are classified as distinct).

As for the *Across Apps* use case, results are reported in Table 3.6. In the table, we report the scores averaged over the nine considered apps. Results show that the TK-based approach performs similarly to the baselines, and slightly worse than WebEmbed which, on the other hand, scores higher accuracy with respect to the baselines and TK-based (+8% and +12% than RTED and PDiff/TK-based, respectively). Statistical tests confirmed that the differences in accuracy between WebEmbed and the baselines/TK-based are statistically significant (p-value<0.05), with a *large* effect size.

Table 3.7. RQ1 - Near-duplicate Detection (*Within Apps* use case). Best averages are boldfaced.

	Within Apps			
Technique	Acc.	Pr.	Rec.	F_1
WebEmbed (content)	0.91	0.92	0.95	0.93
WebEmbed (tags)	0.85	0.91	0.87	0.88
WebEmbed (content+tags)	0.93	0.94	0.97	0.95
RTED	0.84	0.92	0.85	0.86
PDiff	0.86	0.87	0.97	0.91

In the remainder of the empirical evaluation, since training classifiers, executing crawls, and automatically generating and running tests to compute code coverage are time-consuming activities, we limit our analyses to the most promising of the proposed techniques, namely WebEmbed.

Results from the *Within Apps* scenario are reported in Table 3.7 and show that, as expected, training the models on web page pairs belonging to the same web app on which they are tested leads to better classification performance. Results highlight that, also in this scenario, WebEmbed scores higher accuracy than the baseline approaches (+11% and +8% increment in accuracy w.r.t. RTED and PDiff, respectively), with the content+tags embedding model achieving the best results. As in the Across App scenario, also in this case statistical tests confirmed that the differences in accuracy between WebEmbed and the baselines are statistically significant, with a *large* effect size.

RQ1: *Among the considered techniques,* WebEmbed *achieves the highest classification accuracy (75–93%, on average) over all the considered use cases, when implemented with content+tags embeddings and SVM classifiers. The differences w.r.t. the baseline approaches are statistically significant in two out of three use cases, with large effect size. As for the TK-based approach, it performs similarly to* WebEmbed *in the Beyond Apps use case, but worse, in a statistically significant way, in the Across Apps use case.*

Table 3.8. RQ2 - Model Coverage (Beyond Apps Scenario). The best average F_1 score is highlighted in bold.

	Beyond Apps								
	WebEmbed			RTED			PDiff		
	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1
addressbook	0.89	0.95	0.92	0.68	0.72	0.70	0.55	0.58	0.57
claroline	0.93	1.00	0.97	0.93	1.00	0.97	0.93	1.00	0.96
dimeshift	0.84	1.00	0.91	0.81	0.96	0.88	0.56	0.67	0.61
mantisbt	0.84	0.97	0.90	0.58	0.67	0.62	0.75	0.86	0.80
mrbs	0.94	0.98	0.96	0.92	0.96	0.94	0.56	0.58	0.57
pagekit	0.56	0.58	0.57	0.21	0.21	0.21	0.21	0.21	0.21
petclinic	0.72	0.79	0.75	0.70	0.77	0.73	0.53	0.58	0.55
phoenix	0.71	0.73	0.72	0.45	0.47	0.46	0.33	0.34	0.34
ppma	0.82	1.00	0.90	0.82	1.00	0.90	0.82	1.00	0.90
Average	0.81	0.89	**0.84**	0.68	0.75	0.71	0.58	0.65	0.61

Table 3.9. RQ2 - Model Coverage (Across Apps Scenario). The best average F_1 score is highlighted in bold.

	Across Apps								
	WebEmbed			RTED			PDiff		
	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1
addressbook	0.89	0.95	0.92	0.69	0.73	0.70	0.86	0.91	0.88
claroline	0.93	1.00	0.97	0.93	1.00	0.97	0.93	1.00	0.96
dimeshift	0.76	0.90	0.82	0.81	0.96	0.88	0.56	0.67	0.61
mantisbt	0.81	0.93	0.87	0.58	0.67	0.62	0.75	0.86	0.80
mrbs	0.92	0.96	0.94	0.92	0.96	0.94	0.50	0.52	0.51
pagekit	0.82	0.84	0.83	0.21	0.21	0.21	0.21	0.21	0.21
petclinic	0.77	0.85	0.81	0.70	0.77	0.73	0.53	0.58	0.55
phoenix	0.71	0.73	0.72	0.45	0.47	0.46	0.33	0.34	0.34
ppma	0.79	0.96	0.87	0.82	1.00	0.90	0.82	1.00	0.90
Average	0.82	0.90	**0.86**	0.68	0.75	0.71	0.61	0.68	0.64

Table 3.10. RQ2 - Model Coverage (Within Apps Scenario). The best average F_1 score is highlighted in bold.

	Within Apps								
	WebEmbed			RTED			PDiff		
	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1
addressbook	0.84	0.89	0.86	0.15	0.16	0.15	0.27	0.28	0.28
claroline	0.93	1.00	0.97	0.93	1.00	0.97	0.93	1.00	0.97
dimeshift	0.84	1.00	0.91	0.83	0.99	0.90	0.62	0.73	0.67
mantisbt	0.87	1.00	0.93	0.87	1.00	0.93	0.87	1.00	0.93
mrbs	0.96	1.00	0.98	0.92	0.96	0.94	0.71	0.73	0.72
pagekit	0.95	0.98	0.96	0.30	0.31	0.30	0.36	0.38	0.37
petclinic	0.91	1.00	0.95	0.82	0.90	0.86	0.91	1.00	0.95
phoenix	0.80	0.83	0.81	0.15	0.16	0.15	0.47	0.48	0.48
ppma	0.82	1.00	0.90	0.82	1.00	0.90	0.82	1.00	0.90
Average	0.88	0.97	**0.92**	0.64	0.72	0.68	0.66	0.73	0.70

3.6.2 RQ2: Accuracy of the inferred models

We report, for each web app, intra-pairs precision (Pr.), intra-pairs recall (Rec.), and intra-pairs F_1 scores for the models automatically inferred by the competing techniques. Model quality results in the *Beyond Apps*, *Across Apps* and *Within Apps* use cases are reported, respectively, in Table 3.8, Table 3.9, and Table 3.10. For WebEmbed, we present the results for the best configuration resulting from RQ1 (content+tags embeddings and SVM classifier).

Overall, results show that WebEmbed produces more accurate models (i.e., models more similar to the ground truth) than the competing techniques across all use cases. Moreover, WebEmbed benefits from app-specific information, and model accuracy is noticeably improved in the *Within apps* use case. as summarized by the intra-pairs F_1 scores.

In the Beyond apps use case, WebEmbed scores +18% and +37% average F_1 w.r.t. RTED and PDiff, respectively. In the Within apps use case, WebEmbed scores +21% and +34% average F_1 w.r.t. RTED and PDiff, respectively. In the Across apps use case, WebEmbed scores an average F_1 of 92%, a +35% and +31% increase w.r.t. RTED and PDiff, respectively.

Statistical tests confirmed that the differences in accuracy are statistically significant (p-value < 0.05) with a *large* effect size in all use cases,

Table 3.11. RQ3 - Code Coverage Percentages. Best average scores are boldfaced.

	Beyond Apps			Across Apps			Within Apps		
	WE	RTED	PDiff	WE	RTED	PDiff	WE	RTED	PDiff
addressbook	14.54	13.76	10.23	14.68	14.78	13.94	15.19	14.06	14.06
claroline	12.89	12.49	3.50	13.27	7.90	4.71	28.81	23.87	5.45
dimeshift	15.27	15.27	13.78	13.46	13.78	13.78	13.46	13.78	13.78
mantisbt	18.13	13.52	15.80	22.65	15.79	18.36	26.60	16.76	18.35
mrbs	16.19	10.50	8.75	17.55	10.50	8.75	18.94	17.49	9.17
pagekit	56.31	56.11	54.13	56.51	56.11	55.57	58.11	56.98	55.67
petclinic	31.63	32.28	31.94	31.94	32.11	32.11	33.44	32.42	32.11
phoenix	28.78	28.78	28.78	29.06	28.78	28.78	45.51	30.74	30.74
ppma	19.04	15.66	18.39	19.18	15.75	17.44	34.97	22.27	23.08
Average	**23.64**	22.04	20.59	**24.26**	21.72	21.49	**30.56**	25.38	22.49

except Across apps, in which the differences between WebEmbed and RTED are statistically significant with a *medium* effect size.

> **RQ2**: WebEmbed *achieves the highest F_1 scores (84–92%, on average) in all use cases: neural embeddings are able to approximate the ground truth model better than structural and visual techniques. The differences with the baseline approaches are statistically significant in all use cases, with a medium to large effect size.*

3.6.3 RQ3: Impact on automatically generated tests

Table 3.11 shows the code coverage results for each app and for each considered near-duplicate detection approach, grouped by use case. Considering the average scores on the nine apps, the scores for WebEmbed (WE) are consistently the best across all use cases. For the Beyond Apps use case, WebEmbed achieves +6-14% code coverage w.r.t. RTED and PDiff. Concerning the Across Apps use case, WebEmbed achieves +12–13% code coverage w.r.t. RTED and PDiff. About the Within Apps use case, WebEmbed achieves +20–36% code coverage w.r.t. RTED and PDiff. The differences in code coverage between WebEmbed and PDiff are statistically significant for all use cases (i.e., p-value < 0.05, with *small*/*negligible*/*medium* effect sizes). The differences in code coverage

between WebEmbed and RTED are significant only for the Within App use case, with a *small* effect size.

> **RQ3**: *The tests generated from* WebEmbed *crawl models achieve the highest code coverage scores in all the considered use cases (up to +36% improvement) thanks to the more accurate and complete web app models generated using neural embeddings.*

3.6.4 Final Remarks

Overall, WebEmbed was more effective than the considered baseline approaches across all use cases. From a practical point of view, looking at the accuracy scores in conjunction with code coverage, we suggest: (1) using WebEmbed (Beyond apps) if no labelling budget is allowed for developers. Indeed, the effectiveness of this configuration is close to WebEmbed (Across apps), which instead requires a non-negligible labelling cost; (2) using WebEmbed (Within apps) in all other cases, especially if the labelling cost is affordable. Indeed, the gain in code coverage is remarkable in this case (+29 w.r.t. the Beyond apps use case, and +26% w.r.t. the Across apps use case), with corresponding positive implications for fault detection.

3.7 Threats to Validity

In this section, we discuss threats to validity that could have affected our empirical study and its results, according to the guidelines proposed in Wohlin et al. [213].

External validity These threats concern the generalizability of the results. In this study, we considered nine open-source web applications, which may not be representative of complex, real-world commercial applications. To mitigate this threat, we selected web applications from different domains, having different sizes, and implemented with different technologies. Still, the limited number of subjects in our evaluation poses a threat in terms of the generalizability of our results to other web apps. Moreover, we considered only the embeddings produced by Doc2Vec [117],

and WebEmbed's effectiveness may change when considering other algorithms.

Internal validity These threats concern uncontrolled factors that may have affected the results. We compared all variants of WebEmbed and baselines under identical experimental settings and on the same evaluation set (Section 3.5.2). The main threat to internal validity concerns our implementation of the testing scripts to evaluate the scores, which we tested thoroughly. An additional threat is represented by the manually created web page pair annotations and ground truth models. This threat is unavoidable, since there exists no automated method to compute the ideal classification of web pages. To minimize this threat, the authors of the original dataset created, in isolation, a ground truth, and then established a discussion to reach an agreement [218].

3.8 Summary and Future Works

The goal of the research presented in this chapter is to improve automatic test generation techniques for web applications by tackling the problem of near-duplicates in automatically-inferred models. To this aim, we designed and implemented two novel techniques for near-duplicate detection, based on machine learning approaches, specifically geared towards model inference for test generation purposes.

The two techniques we proposed can be framed within a general framework for near-duplicate detection we devised. The first technique we proposed leverages TK functions to compute the similarity of two web pages based on their tree-structured DOM representation. The second technique, which we called WebEmbed, leverages a novel neural embedding model specifically designed for web pages to obtain a vector representation of each page, which can then be used to compute a meaningful similarity measure between any two given web pages. Both techniques use their specific similarity measures to train machine learning classifiers for the near-duplicate detection task.

We studied the effectiveness of the proposed techniques in inferring accurate models for functional testing of web apps, while also discussing their cost for developers in three settings, namely *Beyond*, *Across* and

Within web apps. Furthermore, we investigated their practical impact on the quality of automatically-generated GUI-level web tests, by integrating them within a state-of-the-art tool for automatic web test generation.

Results show that the performance of the TK-based approach is in line w.r.t. the best existing near-duplicate detection techniques in the considered scenarios. WebEmbed, on the other hand, manages to consistently outperform all the baselines. Crawl models produced with WebEmbed are more accurate and lead to the generation of test suites achieving higher code coverage.

Future work on the near-duplicate detection problem includes exploring other forms of embeddings to further improve the accuracy of WebEmbed. For example, usage of visual embeddings on the web screenshots, e.g., with autoencoders, will be explored, as well as hybrid solutions. Moreover, embedding models capable of taking into account also the intrinsic structure of web pages, such as the one proposed by Alon et al. [8] for Abstract Syntax Tree representations of source code, could also prove effective in the domain of near-duplicate detection of web pages.

In a broader context, the research on near-duplicate detection carried out in this chapter could be applied in different domains than automatic model inference/test generation. For example, in web test prioritization [185], detecting near-duplicate web pages could prove useful in computing model coverage measures to guide test prioritization so that tests are executed in an order that maximizes the coverage rate of distinct logical pages. This could lead to improvements in fault detection rates. Moreover, in the security domain, our approaches could be used for the task of automatically detecting phishing websites.

Chapter 4

Automating Workload Generation for Web Apps leveraging existing E2E functional tests

The research presented in this chapter aims at addressing some of the challenges practitioners face in the definition of workloads for performance testing of web applications, as highlighted in Section 2.3.4. To this end, we propose a novel solution to automatically generate workloads for performance testing. The remainder of this chapter is structured as follows. In Section 4.1, we describe in detail the novel solution we propose. In Section 4.2, we detail the preliminary industrial case study we conducted to assess the effectiveness of the proposed technique, while in Section 4.3 we present the results, discussing threats to validity in Section 4.4. Lastly, in Section 4.5 we provide closing remarks and future work directions.

4.1 The Proposed Solution: E2E-Loader

4.1.1 Overview of the proposed approach

The key intuition behind the proposed approach is to leverage automated End-to-End (E2E) GUI-level tests, widely used for quality assurance

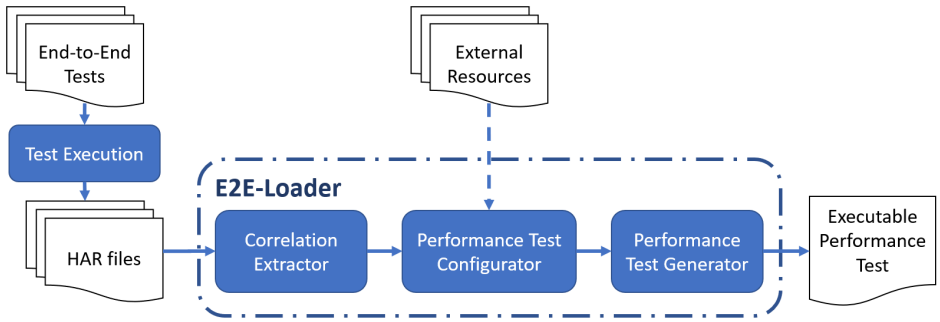


Figure 4.1. Overview of the E2E-Loader approach.

(as discussed in Chapter 3), for the generation of workloads for performance tests. Indeed, automated E2E GUI-level tests aim at testing web apps as a whole, from the point of view of end users, simulating the interactions of a user with the GUI of the system (e.g.: clicking buttons, submitting forms, etc.) to ensure that the applications behave as expected. Thus, E2E test cases can be used as a promising starting point to design realistic user sessions (i.e., sequences of actions performed by a single user).

In order to define a performance testing workload, however, these pseudo-realistic sequences of user actions need to be heavily processed. The overview of the process we proposed is presented in Figure 4.1. We first provide a broad overview of this pipeline, and then in the next Sections will detail each step.

As the first step, E2E tests must be mapped to a user session, i.e., to a sequence of actual user requests. Indeed, a single high-level action, such as clicking a button, submitting a form, or loading a new page, might correspond to a number of different web requests to the SUT. To this end, we exploited the *HTTP ARchive* (HAR) [162] format, a JSON-based standard defined by the *World Wide Web Consortium* for logging of a web browser's interaction with a site. In particular, HAR is supported by any modern web browser [161], to log all network-level interactions occurring, in a given period, between a browser and the SUT, including HTTP and WebSocket messages. Thus, while executing E2E tests, we can collect the corresponding HAR files, which will include all the requests triggered by those tests, that were sent to the web server. Let us note that, thanks to the use of the HAR standard, E2E-Loader is completely independent from

the technologies used to automate E2E tests, and could possibly be used also when E2E tests are not automated at all but are rather performed manually by a tester.

In the next step, E2E-Loader processes the collected HAR files, looking for data correlations between requests. This step is performed by the *Correlation Extractor*, which includes a custom heuristic, considering both values and parameter names. These detected correlations are shown to the performance tester via a GUI component, the *Performance Test Configurator*. This component allows testers to confirm or discard the automatically detected correlations, or to specify new correlations that were not detected by the heuristic. Moreover, the GUI also allows testers to compose the desired final workload, by characterizing the behaviour of each desired group of users (see Section 2.3.2) in terms of the number of concurrent users performing that kind of interaction, initial delay, hold load time, shutdown time, etc.

Lastly, once the tester is satisfied with the current configuration, the *Performance Test Generator* component generates executable performance tests. In our implementation, executable performance tests are generated in JMeter format, but the component could be easily extended to support other executable formats as well. An overview of the E2E-Loader approach is depicted in Figure 4.1. In the remainder of this section, we detail each of these key components of E2E-Loader, namely the *Correlation Extractor*, the *Performance Test Configurator* and the *Performance Test Generator*.

4.1.2 Managing Data Correlations: The Correlation Extractor component

In most web applications, requests that are sent at a given time might need to use dynamic values obtained by previous responses in the same session [180]. Keeping track of these correlations and properly managing them in each user session composing a workload is a crucial task, necessary to properly simulate realistic user behaviours. Consider, for example, the HTTP response and the subsequent request depicted in Figure 4.2, in which some parameters in the request depend on values returned in the previous response. In the example, the `Authorization` header parameter in the request corresponds to the `auth` parameter returned as a field of the JSON body of the response. Similarly, the `sessID` cookie parameter sent

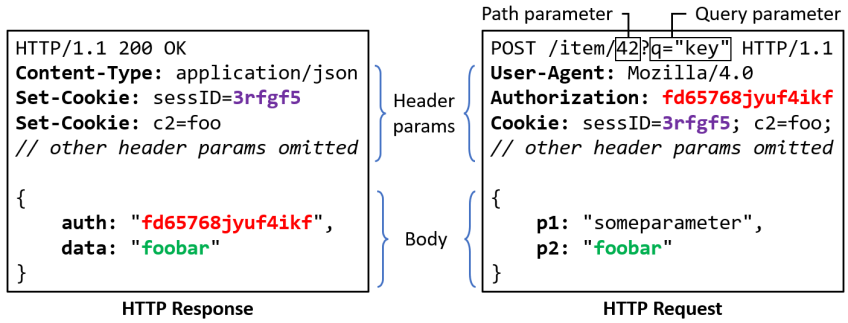


Figure 4.2. Example of HTTP response and subsequent request, with data correlations.

in the request depends on the value set in the `Set-Cookie` header of the response. Lastly, the `p2` field in the body of the request depends on the value returned in the `data` field in the body of the response. Moreover, parameters returned in a response might be used also as path or query string parameters in the URL of subsequent requests (see Figure 4.2). In the literature, to the best of our knowledge, this problem has been addressed only in the work presented by Shams et al. in [180]. That approach, however, relies on manually-defined models of the SUT to automatically identify correlations between requests. Our approach, on the other hand, is fully automated and based only on the analysis of the requests.

To automatically detect correlations between response and subsequent requests, the *Correlations Extractor* component performs a two-step analysis. In the first step, likely correlations are detected by comparing the value of each request parameter with the value of all the parameters in all previous responses. In this phase, the comparison is based only on the value of the parameters and selects as candidate correlations all parameters having the exact same value. This first step may find multiple correlations for the same parameter, possibly with different parameter names. For instance, a request parameter named `paymentMethod` with value `CREDIT_CARD` might match with two different parameters in previous responses, named, respectively, `paymentMethod` and `supportedPayment`. The proper correlation is the one that detects the underlying relation in requests-responses and makes the recorded sequence of requests reproducible, by instantiating a new sequence of requests according to the actual server responses gener-

ated in the new interactions.

In the second step, in cases in which multiple potential correlations are detected for a given request parameter, the set of potential correlations is further refined using a custom heuristic we defined, taking into account also parameter names. The idea is to select the most likely match for the response parameter whose name is most similar to the name of the request parameter. The custom heuristic we defined is based on the *Levenshtein distance* (also known as *edit distance*), a widely-used similarity measure for strings [224]. Given two strings, the Levenshtein distance between them can be defined as the minimum number of character edits (insertions, deletions and substitutions) for transforming one string into another. More in detail, potential matches are ranked in decreasing order of similarity, and only the top three matches are selected as potential candidates. Preliminary experiments suggested that returning the top-three alternatives included the correct correlation in more than 90% of the cases. The above-described matching process is applied to both HTTP and WebSocket messages and is capable of detecting correlations between any response/request pair, regardless of the specific protocol.

4.1.3 Performance Test Configurator

The *Performance Test Configurator* component provides testers with a GUI that can be used to fine-tune the automatically-detected correlations and to better customise the desired workload. In particular, this module allows testers to graphically choose which correlations to adopt, among the candidate ones that were automatically individuated by the *Correlation Extractor* component, as well as to manually specify new correlations using a rule-based approach similar to the one included in most existing tools (see Section 2.3.3), had the correct ones not been automatically detected. This component also enables the automatic generation of parametric tests, leveraging external data sources. For example, in a workload in which a number of concurrent users buy a book from an online store, it is possible to provide a list of books to buy as an external CSV file and to make each distinct synthetic user behaviour parametric w.r.t. the book. This way, each simulated user buys a distinct book, making the workload more realistic. Moreover, the GUI also allows testers to compose the desired final workload, by selecting which user behaviours should be

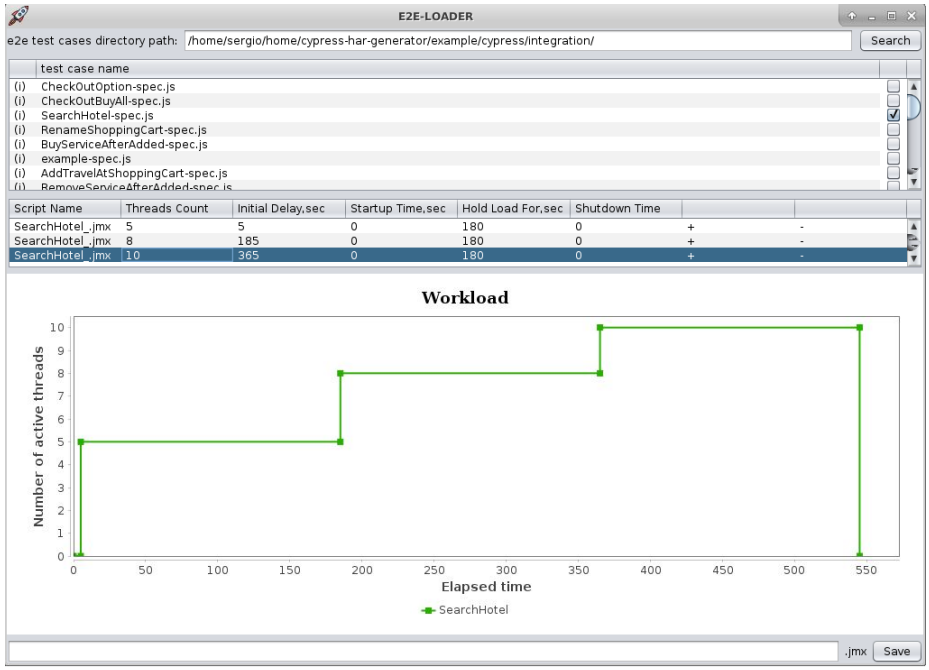


Figure 4.3. Example of workload definition using the E2E-Loader GUI.

Table 4.1. Formalization of the five workloads selected for the case study

Workload	Behavior	Thread Count	Initial Delay (sec)	Hold Load For (sec)	Overall duration (sec)
1	Rename Shopping Cart	10	5	600	605
	Search Hotel	10	5	600	
2	Rename Shopping Cart	10	5	300	605
	Rename Shopping Cart	5	305	300	
	Search Hotel	10	5	300	
	Search Hotel	5	305	300	
3	Rename Shopping Cart	5	5	180	545
	Rename Shopping Cart	10	185	180	
	Rename Shopping Cart	15	365	180	
4	Search Hotel	5	5	180	545
	Search Hotel	8	185	180	
	Search Hotel	10	365	180	
5	Rename Shopping Cart	5	5	900	1 805
	Rename Shopping Cart	10	905	900	
	Search Hotel	5	5	900	
	Search Hotel	10	905	900	

included in the workload, and by characterizing each desired group of users (see Section 2.3.2) in terms of the number of concurrent users performing that kind of interaction, initial delay, hold load time, and shutdown time. The screenshot in Figure 4.3 shows the workload composition feature as provided by the GUI.

4.1.4 Performance Test Generator

Once the tester has finished modelling the workload using the *Performance Test Configurator* GUI, the *Performance Test Generator* component is responsible for "*translating*" the workload in an executable performance test suite. The current implementation of E2E-Loader generates executable performance tests in the format used by *Apache JMeter*, to support practitioners familiar with the tool, and to be adopted in the automatic pipelines based on JMeter.

4.2 Empirical Study Design

In this section, we present the industrial case study we conducted to assess the effectiveness of the proposed approach in generating high-quality workloads for performance testing. In particular, the goal of the case study is to answer the following research question:

RQ *Are workloads automatically generated with E2E-Loader comparable to those manually-generated by practitioners using state-of-the-art tools?*

In this section, we detail the case study we conducted, by describing the adopted experimental procedure in terms of the involved subject system, considered workloads, gold standard implementation of workloads, and the metrics we used to compare workloads.

4.2.1 Subject System

The system used for the evaluation is a modern web application designed to support travel agencies in booking hotels and transportation for customers' holidays, interacting with a number of third-party services. The app features a microservices-based architecture and uses Kubernetes to properly orchestrate containers. As of today, it consists of 51 microservices and makes large use of the WebSocket protocol to provide a swift user experience and reduce load times. The front end is implemented using Angular and communicates with the microservices using REST APIs.

During our empirical evaluation, Booking Engine was deployed on a Kubernetes cluster running in a QA environment, on a server with the following characteristics: AMD Ryzen 9 5900X 12-Core processor 64 GB RAM and 1TB SSD. No other applications or services were running on the server during the execution of the experiments. As for the performance tests, they were executed on a different server, featuring an Intel(R) Core(TM) i9-9940X CPU at 3.30GHz, 64 GB RAM and 1TB SSD.

4.2.2 Workloads

In our study, we consider five different workloads used by our industrial partner to test the app under load. The workloads are detailed in Table 4.1 and consist of different combinations of two user behaviours: *Rename Shopping Cart* and *Search Hotel*. In the first behaviour, users navigate to the shopping carts management page and rename one of the shopping carts. In the second behaviour, users enter the travel planning page, specify a search query in terms of selected destination, check-in and check-out dates, and the number of guests per room), and browse the

returned results. In this second usage scenario, the app makes use of WebSocket to return additional results as soon as they are retrieved from third-party services. In all workloads, both the start-up time and the shut-down time are set to zero seconds. Let us note that these workloads involve 15 of the 51 defined microservices.

4.2.3 Gold Standard Implementation

In our empirical evaluation, we consider, as a *gold standard*, the implementations of the above-described workloads that were manually defined by practitioners working with our industrial partner, using the JMeter tool. It is worth noting that manually implementing these workloads took a practitioner several days of tedious and error-prone work, as data correlations between responses and subsequent requests and WebSocket messages had to be managed entirely manually.

4.2.4 Metrics

To compare the different performance test suites (i.e. the one automatically generated by E2E-Loader vs. the one manually generated by a practitioner), we compare the loads induced by them on the SUT. More in detail, as done in similar works on performance testing [46], we measure the load induced on the SUT over time, by collecting statistics on CPU usage levels (percentage of used CPU) every five seconds. Moreover, since the subject system features a microservices-based, distributed architecture, in our analysis we also collected CPU loads at a much finer-grained level of a single container, using the `kubectl` command line tool provided by Kubernetes, to collect the CPU usage in millicores [115] over time.

4.2.5 Procedure

Firstly, we trained a practitioner working with our industrial partner on how to use E2E-Loader. Let us note that, to avoid biases, this person is not the same one that implemented the gold standard workloads. Subsequently, the practitioner used E2E-Loader to implement the workloads defined in Section 4.2.2, leveraging the available E2E functional tests for the *Rename Shopping Cart* and *Search Hotel* scenarios, implemented

using the *Cypress* framework. Subsequently, we executed both the gold standard implementations of the workloads and the ones obtained using E2E-Loader, measuring, every five seconds, the percentage of CPU used for the entire QA server running the SUT, and the millicore usage of the microservices. Thus, for each of the five workloads and for each of the 15 involved containers, we obtained two sequences of millicore usage levels over time, one for the gold standard implementation and one for the E2E-Loader implementation.

As done in similar works (e.g.: [46]), to determine whether the loads induced by the E2E-Loader test suite are comparable to those induced by the gold standard, we used the Wilcoxon signed-rank test [212] to compare the two sequences of recorded load levels. More in detail, for each workload, we tested the following null hypothesis:

H_0: the load induced by the E2E-Loader implementation has the same distribution as the load induced by the gold standard implementation.

If the test *p-value* is greater than 0.05, the null hypotheses cannot be rejected, and thus we can consider the induced loads to be comparable. If, on the other hand, the test *p-value* is smaller than 0.05, the null hypothesis can be rejected with high confidence, thus accepting the alternative hypothesis that a statistically significant difference exists between the two induced loads. Even when statistically significant, however, the difference in the induced loads might be negligible and have no practical impact [46]. Hence, we measured the magnitude of these differences using the *Cliff's delta* effect size [55], a measure that is largely used in software engineering to determine the degree of difference between two experimental results [107]. Cliff's delta ranges between -1 and 1, and can be interpreted as follows: if $|\delta| < 0.147$, the difference is *negligible*; if $0.147 \leq |\delta| < 0.33$, the difference is *small*; if $0.33 \leq |\delta| < 0.474$, the difference is *medium*; if $|\delta| \geq 0.474$ the difference is considered *large*. In this work, as also done in [46], we consider two loads comparable even if a statistically significant difference exists, but the effect size is small or negligible.

Figure 4.4. CPU loads (%) over time in the considered workloads.

Table 4.2. Results of the Statistical Tests (SUT-level CPU load)

	Workloads				
	1	**2**	**3**	**4**	**5**
p-value	**0.54**	**0.66**	**0.15**	**0.07**	<0.0001
effect size	–	–	–	–	**0.30 (small)**

4.3 Results

The SUT-level CPU loads induced by the performance tests implemented using E2E-Loader and by the gold standard implementations are depicted in Figure 4.4. The figure highlights that, for all the considered workloads, the load induced by the performance tests generated using E2E-Loader are quite similar to those obtained using the manually implemented performance tests.

As for the statistical tests on the loads measured at SUT level, the results are reported in Table 4.2. For each workload, we report the p-value of the statistical test we performed and, in case the p-value is smaller than 0.05, we report the measured effect size. In four workloads out of five (Workload 1 to Workload 4), the statistical tests highlighted no statistically significant difference in the CPU loads. In Workload 5, a statistically significant difference exists (p-value < 0.0001), but the measured effect size is small.

The results of our finer-grained statistical analysis considering CPU loads at container level are reported in Table 4.3. In the table, each row corresponds to a different container, and values report the *p-value* of the statistical test comparing the two loads at millicore level induced on that

Table 4.3. Results of the statistical tests (Container-level CPU loads)

		Workloads				
		Workload 1	**Workload 2**	**Workload 3**	**Workload 4**	**Workload 5**
C1	**p-value**	< 0.0001	**0.74**	**0.42**	**0.28**	< 0.0001
	eff. size	**0.16 (small)**	–	–	–	**0.21 (small)**
C2	**p-value**	**0.45**	0.009	**0.80**	< 0.0001	< 0.0001
	eff. size	–	**0.6 (neglig.)**	–	0.35 (medium)	**0.15 (small)**
C3	**p-value**	0.001	**0.14**	0.001	**0.77**	< 0.0001
	eff. size	**0.23 (small)**	–	**0.12 (neglig.)**	–	0.49 (large)
C4	**p-value**	**0.55**	**0.44**	**0.7**	**0.56**	**0.19**
	eff. size	–	–	–	–	–
C5	**p-value**	< 0.0001	0.0003	**0.12**	**0.26**	< 0.0001
	eff. size	**0.31 (small)**	**0.26 (small)**	–	–	0.55 (large)
C6	**p-value**	< 0.0001	< 0.0001	0.001	**0.05**	< 0.0001
	eff. size	0.45 (medium)	0.47 (medium)	**0.08 (neglig.)**	–	0.7 (large)
C7	**p-value**	**0.91**	**0.95**	**0.43**	**0.68**	**0.69**
	eff. size	–	–	–	–	–
C8	**p-value**	**0.31**	**0.17**	0.001	**0.69**	**0.47**
	eff. size	–	–	**0.15 (small)**	–	–
C9	**p-value**	**0.93**	**0.73**	**0.67**	**0.36**	**0.48**
	eff. size	–	–	–	–	–
C10	**p-value**	**0.09**	0.02	**0.37**	**0.97**	**0.70**
	eff. size	–	**0.02 (neglig.)**	–	–	–
C11	**p-value**	**0.31**	**0.82**	0.04	**0.05**	**0.15**
	eff. size	–	–	**0.12 (neglig.)**	–	–
C12	**p-value**	**0.66**	**0.90**	0.04	**0.87**	**0.60**
	eff. size	–	–	**0.17 (small)**	–	–
C13	**p-value**	**0.14**	**0.31**	**1.0**	**0.51**	**0.24**
	eff. size	–	–	–	–	–
C14	**p-value**	0.001	**0.13**	**0.43**	0.01	< 0.0001
	eff. size	**0.10 (neglig.)**	–	–	**0.05 (neglig.)**	**0.1 (neglig.)**
C15	**p-value**	**0.41**	**0.91**	**0.49**	**0.72**	**0.86**
	eff. size	–	–	–	–	–

container. Also in this case, when a statistically significant difference exists between the two loads ($p\text{-}value < 0.05$, we report also the effect size. Results show that, in 93% of the cases, the loads induced on the containers by E2E-Loader workloads are comparable to those induced by gold standard workloads, with the only exceptions being on Container 5 in Workload 5, in Container 6 for Workloads 1, 2, and 5, and in Container 3 for Workload 5. In these cases, results show a statistically significant difference with a medium or large effect size.

Summarizing, E2E-Loader provides the following advantages, very beneficial for a software-related industry:

- The approach can be used to automatically define workloads **before** the SUT is released and user interactions logs are collected, unlike most log-based solutions presented in the literature.
- The approach provides a significant time speed-up compared to manually defining workloads with generally available tools used in the industry, such as the well-known JMeter. The data correlations suggested by the tool while defining workloads were correct in approximately the 90% of the cases, so our heuristic proved to be valid to effectively assist the testers. Moreover, practitioners are lifted from the burden of manually managing WebSocket messages.
- The workloads defined by our tool are easier to be maintained during the product life-cycle. Indeed, within software evolution, new workloads can be derived automatically from updated E2E test cases.

4.4 Threats to Validity

In this section, we discuss threats to validity that could have affected the results of our empirical study, according to the guidelines proposed in Wohlin et al. [213].

External validity The main threat to external validity that could affect the generalizability of our preliminary results is linked to the fact that we consider a single subject system. The subject system is a modern web application, implemented using state-of-the-art frameworks and

a microservice-based architecture, but nonetheless cannot be considered as representative of all industrial-strength web applications. Additional experiments on a wider set of subject systems need to be put in place to mitigate this threat.

Internal validity We compared the workloads generated automatically using E2E-Loader and those manually implemented by practitioners under the same experimental settings. The workloads were executed at night, when no developer interacted with the test environment. Still, uncontrolled factors such as scheduled tasks at OS level may have affected the measured load levels and thus our results. An additional threat to internal validity concerns our implementation of the scripts to collect load metrics on the server as well as to compare the recorded loads and perform statistical analyses. To mitigate this threat, we thoroughly tested these software components.

4.5 Summary and Future Works

Performance testing is essential to improve the quality of web applications and to ensure a good user experience under different load conditions. A key performance testing activity is the design of synthetic workloads, i.e., deciding which requests should be sent to the SUT to simulate a given load condition.

From the analysis of the literature, we found that existing tools and solutions are affected by some key limitations, hindering the productivity of performance testers and the effectiveness of the entire performance testing process.

This chapter presents a novel solution aimed at automatically generating workloads for performance tests, starting from existing End-to-End functional test cases. We implemented the solution in a tool named E2E-Loader, which we make publicly available for interested researchers and practitioners [35]. We assessed the effectiveness of E2E-Loader in generating realistic workloads in an industrial case study, based on state-of-the-art technical solutions. In particular, we compared the performance tests generated by the tool with a gold standard, intended as tests manually defined by an IT practitioner. Results show that, in most cases, the server loads

induced by the test suites obtained with E2E-Loader are comparable to those obtained with manually defined tests, while requiring a fraction of the time to be generated. Moreover, E2E-Loader tests can also be easily re-generated from E2E functional tests, to keep performance tests updated in scenarios in which the SUT evolves.

In future works, we plan to further validate our proposal by comparing it against log-based workload generation approaches presented in the literature. Furthermore, we also plan to replicate our study on a broader set of subject systems, possibly including also open-source ones, to improve the generalizability of the results. Further work could also be devoted towards extending the tool to introduce support for analysing the results of performance tests and to automatically detect anomalies.

Chapter 5

Investigating Exploratory E2E Functional testing of Android Apps

According to a recent report [64], 4 billion smartphone users have downloaded 230 billion apps in 2021, with an app store spend of $170 billion. In such a competitive market, it is fundamental to adequately test mobile apps, as high-quality apps have a much higher chance of being well-received by users and thus of being profitable [110].

As discussed in Section 2.2.4, leveraging Automated Input Generation (AIG) tools or practitioners using Capture & Replay (C&R) tools and Exploratory Testing (ET) strategies are two widely-used approaches for GUI-level testing of mobile apps. From a practical perspective, a Software Project Manager might be puzzled in deciding whether to use AIG tools or human testers implementing ET approaches, for a specific app under development. Moreover, this decision-making scenario is further complicated by the diffusion, in recent years, of crowdsourced testing [227, 137, 92, 207] (or crowdtesting), that allows companies to conveniently recruit "crowds" of human testers on an on-demand basis and relatively inexpensively [201], thus making ET approaches more appealing.

To the best of our knowledge, little work aimed at supporting these managerial choices has been conducted in the literature. In this chapter, we present two empirical studies we conducted, involving state-of-the-art

AIG tools and 20 masters students. The first study, presented in Section 5.1, aims at comparing the testing effectiveness of AIG tools against those of tests generated by practitioners using ET and C&R tools. The second study, presented in Section 5.2, investigates the effectiveness of crowdtesting in a scenario in which the goal is to generate a test suite using ET and C&R tools, analysing the impact of the number of recruited testers and of the adoption of different (i.e., *Uninformed* and *Informed*) exploratory strategies, on the effectiveness of the resulting test suites.

5.1 Comparing Automated Tools and Practitioners using C&R

In real-life industrial scenarios, where there are budget, temporal and team capability constraints, a Project Manager might need to choose whether to use fully-automated AIG testing techniques, to employ (possibly low-skilled) practitioners with C&R techniques, or to consider both strategies. As for C&R techniques, we can do a further subdivision. Indeed, the simplest scenario we can consider of Exploratory GUI Testing of Android apps with C&R techniques is the one where the tester has no previous knowledge about the AUT nor about its requirements, but is free to define arbitrary sequences of user interactions based on his/her sensibility. In the following, we will refer to this scenario as Uninformed Exploratory Testing (UET). In a more sophisticated scenario, the tester can be driven, in the definition of sequences of user interaction, not only by his/her sensibility but also by some additional information about the code coverage achieved by previous tests. In the following, we will refer to this scenario as Informed Exploratory Testing (IET).

The UET approach can be appealing in mobile application development, where the pressure to reduce time-to-market is very strong [112]. Indeed, with this strategy, even naive practitioners, with no knowledge of the source code of the AUT, can be employed to design and run tests. The IET approach, on the contrary, requires some insight into the source code but can lead to test cases stressing parts of the code that could have been missed by a UET test suite. This clearly requires testers with programming and testing skills.

In the literature, to the best of our knowledge, little work has been di-

rected towards supporting Software Project Managers and Decision Makers in choosing which strategies to use to test a given Android app.

5.1.1 Empirical Study Design

The goal of the empirical study presented in this section is to compare, in terms of testing effectiveness, different strategies for End-to-End (E2E) GUI testing of Android apps. In particular, we consider three testing strategies:

UET An Exploratory Testing strategy in which test suites are developed with a C&R approach by testers with a university-level background of programming and software engineering, basic knowledge of C&R tools and without knowledge of the structure of the AUT. In addition, testers have a strict deadline constraint to complete their testing task;

IET An Exploratory Testing strategy in which test suites are generated with a C&R approach by testers with a university-level background of programming and software engineering, basic knowledge of C&R tools and complete knowledge of the structure of the AUT. In addition, testers can leverage on information about the code coverage obtained with the previous test cases to improve the effectiveness of the produced test suite. Finally, testers have no strict deadlines to complete their work;

AIG A testing strategy relying on fully-automated AIG tools that explore the GUI of the AUT without the need for any manual intervention.

In the remainder of this section, we detail the empirical study we conducted in terms of research questions, experimental subjects and objects, employed metrics and procedures.

Research Questions

To compare these testing approaches, we posed the two following research questions:

RQ1 (UET vs AIG) *How do test cases generated with an Uninformed Exploratory Testing strategy compare, in terms of effectiveness, to those generated by AIG tools?*

RQ2 (IET vs AIG) *How do test cases generated with an Informed Exploratory Testing approach compare, in terms of effectiveness, to those generated AIG tools?*

The answers to the research questions will be evaluated either from a *quantitative* point of view, by comparing coverage percentages obtained by different testing techniques with respect to different code coverage metrics, and by a *qualitative* point of view, aiming at assessing and classifying the code portions of the AUT that remain unexplored by the three different strategies.

Subjects

Our empirical study involved a set of students attending the Advanced Software Engineering course in the M.Sc. in Computer Engineering program at the Università degli Studi di Napoli Federico II, Naples, Italy. Twenty students were enrolled in the experiments, on a voluntary basis. All the involved students had a Computer Engineering Bachelor Degree and a common background in Java programming and software engineering. Since the Advanced Software Engineering course is mainly focused on testing, they received seven lectures on testing techniques (including functional and coverage-based testing techniques), testing automation, JUnit, GUI testing, and C&R techniques. In addition, they attended four lectures on Android programming basics (the final assignment for the course required them to develop an Android app) and were all Android phone users. Finally, all the students were given a dedicated lecture on C&R tools for Android testing.

Objects

C&R Tool The selected C&R tool for this experiment was Robotium Recorder [171]. The main feature of this tool is the possibility to record (capture) the interactions of a user with an Android device in the context of an Android app and to generate a corresponding re-executable JUnit test case, exploiting JUnit and the Robotium library [170]. The tool can be straightforwardly integrated with Integrated Development Environments (IDEs) such as Eclipse and Android Studio.

Moreover, we used the well-known Emma [70] tool to measure the Lines of Code (LOC) coverage percentage achieved by the test cases at source code level (with respect to the Executable Lines of Code metric). According to the operative definition provided by the Emma reference manual [69], an Executable LOC is a line of source code having at least a corresponding statement in the bytecode (e.g.: comments are lines of code that do not yield executable LOCs).

The Automated GUI testing tools Three AIG testing tools were considered in this experiment, i.e., *Android Ripper* [14, 13], *Sapienz* [139] and *Robo* [79]. Android Ripper [14, 13] is one of the first automated GUI testing tools made available to the scientific community. We selected it because it has been often considered as a benchmark in several works reported in the literature [51, 16, 125, 155, 175]. Sapienz [139] is a search-based GUI testing tool, proposed in 2016. We selected it because its performance overcomes the ones of all the other tools previously presented in the literature. Robo test is a GUI testing tool integrated into the Google Firebase Test Lab [79]. We selected this tool because it is the systematic GUI testing tool provided by Google, and includes also a free plan, making it a viable choice for developers. In the following, we briefly describe some technical aspects of these tools.

Android Ripper Android Ripper automatically explores the AUT in a systematic and deterministic way, by analysing its GUI and triggering suitable user/system events. The tool includes a strategy aiming at executing each event handled by the AUT at least once, combined with heuristics to avoid the re-execution of already triggered events. Consequently, its execution usually ends after triggering all the events identified by the strategy, and some empirical evaluations showed that it is not usually required to set time limits for its execution [13]. Android Ripper allows the Tester to customize many options of the execution, like for instance the possibility of selecting a breadth- or a depth-first GUI exploration strategy.

Sapienz Sapienz [139] is a search-based testing tool that executes random-based user/system events on the AUT, guided by a multi-objective optimization strategy. Indeed, it uses a genetic algorithm, starting from a

random set of test cases, composed of a sequence of user/system events. At each iteration, crossover and mutation operators are applied to generate new test cases. Offspring are selected on the basis of a fitness function, aimed at maximizing the code coverage and the failure-finding capability while minimizing the test sequences length. The fact that Sapienz has a dependency on randomness leads to two practical consequences: (I) it can obtain a different coverage for each run, thus multiple executions can lead to more effective test suites; (II) its execution requires setting a timeout or a number of generations of the genetic algorithm. As a reference, the authors of the tools defined a one-hour timeout in the experiments they reported in [139].

Robo Robo is a testing tool integrated into the Google Firebase Test Lab [79], which explores in a systematic and deterministic way the GUI of the AUT. Differently from the two other considered tools, Robo is executed on the Google cloud infrastructure. Running a test in Robo requires uploading the APK of the AUT and selecting a device (either virtual or real) to be used for testing. Optionally, the tester can also define a time-out. Google suggests using a 5 minutes time-out for "moderately complex apps". Robo test is offered with two usage plans: with the free one, it is possible to execute up to 10 tests per day per Google account on a virtual device or 5 on a physical one. With a premium account, it uses a pay-per-use strategy and additional tests can be executed with a cost per hour of 1$ per virtual device or 5$ per physical device. After test execution, Robo generates log files, saves a series of annotated screenshots, and then creates a video from those screenshots, showing the user operations performed on the GUI.

Let us note that all these tools consider only a limited set of user/system events, mostly including the ones of the standard widgets belonging to the Android libraries (e.g. buttons, text fields, spinners, selection boxes and so on). The characteristics of the three considered AIG tools are summarized in Table 5.1.

The Applications Under Test Well-known limitations of C&R tools are related to their difficulty in registering and reproducing preconditions to the test case execution, which can be related to third-party remote

Table 5.1. Main Characteristics of the AIG Tools Considered in the Experiment

Tool Name	Description
Android Ripper	**Exploration approach:** Systematic and Deterministic. **Objective:** Coverage of distinct events. **Termination criteria:** Exhaustion of events to trigger.
Sapienz	**Exploration approach:** Search-Based. **Objective:** Maximize an objective function considering both code coverage, fault detection capability and test sequences length. **Termination criteria:** Time limit reached, or fixed number of generations for the underlying genetic algorithm.
Robo	**Exploration approach:** Systematic and Deterministic. **Objective:** Coverage of distinct events. **Termination criteria:** App exhaustively explored, or time limit reached.

Table 5.2. The Android apps used in our study

Id	Name	Version	Classes	Activ.	Methods	LOCs	Exec. LOCs
A1	MunchLife [158]	1.4.4	10	2	28	486	184
A2	SimplyDo [182]	0.9.2	46	3	246	3566	1281
A3	TippyTipper [192]	1.2	42	6	225	2238	999
A4	Trolly [195]	1.4	19	2	64	1062	364

resources (e.g. remote databases or web services, including also authentication services). In addition, C&R tools may generate unstable test cases in presence of multi-threaded applications and dependencies on race conditions.

We have selected a set of four small-sized Android applications that are single-threaded and that have no dependencies on third-party services. They are all open-source apps (their source codes are also available on the F-Droid [73] repository or on GitHub) and were also used in other empirical studies on Android GUI testing [13, 51, 139]

Table 5.2 reports, for each of the selected apps, its name and version, as well as the total number of classes, activities, methods, LOCs, and Executable LOCs. We also point out that the number of classes reported in Table 5.2 includes both statically defined and anonymous classes, including listener classes related to GUI widgets.

More in detail, MunchLife (A1) is a counter of the score achieved in

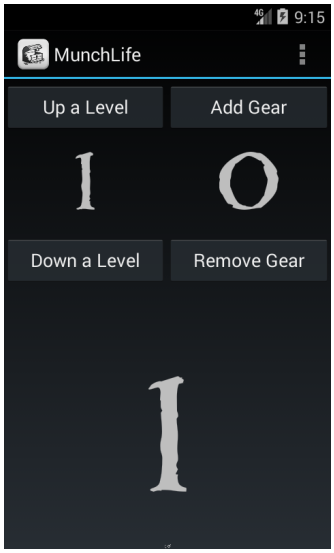
the "Munch" card game. It consists of a main GUI showing a number of different counters and controls to change their values, together with a menu used for changing configuration preferences (e.g. the maximum counter value). Simply Do (A2) is a to-do list management utility. Its main GUI reports the current list of activities organized in a hierarchy, while many different features to organize, sort, delete, and export a backup of to-do items are accessible through menus. TippyTipper (A3) is a utility app for the calculation of tips for restaurant bills. From its three main GUIs, it allows the user to specify the bill amount, the tipping percentage, and the number of people splitting the bill. It calculates the amount of money that anyone has to pay. All the calculation parameters may be modified by operating on menus. Finally, Trolly (A4) is a shopping list manager application, that allows users to add/remove objects from lists, and to maintain a list of the most recently listed objects, as well. It manages three kinds of lists, i.e. items to buy, bought items, and also "off list" items (i.e. items that were inserted in past lists and can be inserted in new lists. This list basically acts as a suggestion list).

The functionalities of these apps are accessible either by app menus or by context menus. Some of the applications are also sensitive to some system events such as device rotation. A screenshot of the main Activity of each of these apps is reported in Figure 5.1. Since the selected apps do not interact with any remote service, not even for authentication purposes, GUI testing was possible at system level without the need of dealing with precondition setting, mocking, and other integration and system-testing issues.

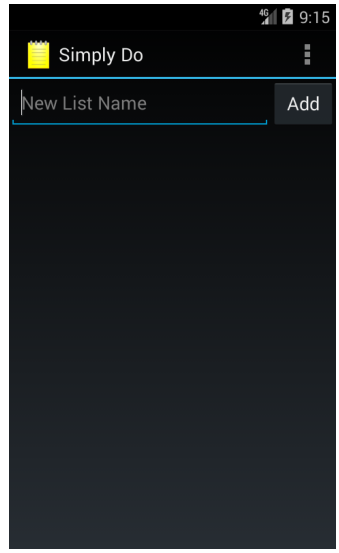
Variables

To compare the effectiveness of the different GUI testing approaches, we used two code coverage metrics, widely employed in similar works (e.g.: [51, 193]): (I) the *lines-of-code* (LOC) coverage, measured at bytecode level by the Emma [70] tool with respect to the Executable Lines of Code, and (II) *branch* coverage, measured at source code level.

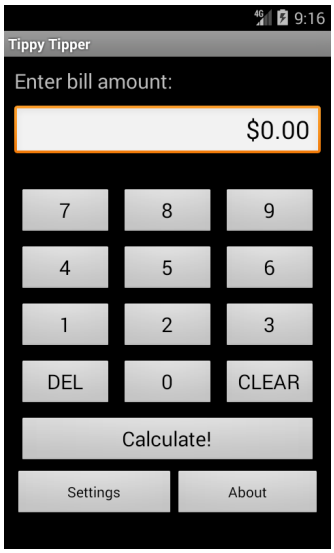
Let us note that, to measure branch coverage, we manually instrumented the AUTs, inserting probes in each branch of their Java source code. Such probes write to the standard log output of Android apps, easily readable using the well-known Logcat tool. The instrumented versions



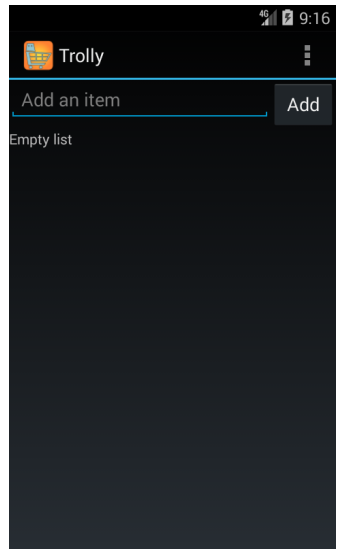
(a) MunchLife (A1)



(b) SimplyDo (A2)



(c) TippyTipper (A3)



(d) Trolly (A4)

Figure 5.1. Screenshots of the considered AUTs.

of the applications are available at http://reverse.dieti.unina.it/index.php/30-support-material for experiment replication purposes.

Due to the lack of advanced customization features with Robo, it was not possible to compute its LOC coverage with Emma. Nevertheless, thanks to the instrumented AUTs, we computed the branch coverage metric with the Robo tool, leveraging the generated log files.

Materials and Procedure

In this subsection, we describe the procedure we adopted for our experiments. Firstly, before the experiments, all the involved students were given several hours of practical lectures on GUI testing techniques and on C&R tools, focusing in particular on Robotium Recorder, to ensure that they all had adequate expertise with the tool.

Subsequently, we assigned the students two testing tasks. The first task consisted in generating a test suite for each of the AUTs using Robotium Recorder and a UET approach. This task was carried out in a controlled environment, in the Software Engineering laboratory, and all the students were supplied with a virtual machine including the Eclipse IDE, the Application Under Tests (AUTs), and an instance of an Android emulator capable of running the AUTs. During this task, all students had no prior knowledge of the AUTs, nor they had access to their source code or to code coverage reports. To prevent the need to set specific preconditions, we suggested the students to produce a single test case for each app, composed of a single, long sequence of interactions. As an alternative, they can produce different test cases by returning each time to the initial state of the virtual machine. We gave the students a time limit of four hours to complete this task. Within this time frame, the students were free to decide the order in which to analyse the AUTs, as well as the amount of time to allocate to each AUT.

The second task was a homework assignment in which the students had to improve their previously-developed test suites. In this task, each student had to first measure the code coverage of the previously developed test cases, as explained during the lectures. Subsequently, based on an analysis of the source code and of the code coverage reports, the students had to generate additional tests using Robotium Recorder and the same testing environment, with the goal of improving code coverage (thus with

an IET approach). Notice that performing the IET task after the UET task on the same applications is not unrealistic. Indeed, even when using an IET approach, testers typically start by recording an initial test suite with an Uninformed approach, to get a basic understanding of the AUT and an initial test suite to be used to compute coverage metrics [19]. The students submitted the output for this task (i.e.: the resulting test suites and the corresponding coverage reports) after the end of the Software Engineering course, when they were ready to take the final exam. Before the experiments, we made clear to the students that the achieved coverage values were not to be considered in the determination of the course grades, in order to discourage plagiarism among students. On the other hand, the obtained result and the adopted methodologies were objects of discussion in the final examination. Upon submitting their work, students were also asked to report how much time it took them to complete the second task. On average, the students reported that they completed the second task in approximately eight hours. All the tests produced by the students were re-executed to ensure that they worked properly and to validate coverage data.

As for the experiments with the AIG tools, we executed both Android Ripper and Sapienz on instances of the same emulator used by students, for testing each of the four apps. Robo was used on emulators provided by the Google Cloud platform having the same characteristics (e.g., the same Android version).

Android Ripper was executed using the configuration named TR_9 in [13], in which the tool automatically explores the GUIs of the AUTs in breadth-first order, adopting an *Active Learning* strategy, and by cutting the exploration branches when the last visited GUI is deemed equivalent to a previously visited one in terms of widget number and typology. The total time spent by Android Ripper for testing the four apps was 4 hours and 50 minutes, varying between 13 minutes (for A1) to 3 hours (for A2), depending on the complexity of the GUI of the AUTs [13].

The Sapienz tests were executed using the tool version presented in [139]. Sapienz was configured as recommended in [139] (crossover and mutation probability set to 0.7 and 0.3 respectively, maximum generations set to 100, population size to 50 and each individual contains 5 test cases) and executed for a fixed testing time of one hour for each application.

Both the testing time with Android Ripper and Sapienz were close to the testing time given to students in the UET scenario. Since the execution of Sapienz depends on randomness, we executed it 20 times for each app, and for each execution, we evaluated the coverage values.

In addition, to get a rough estimation of the possible improvements in coverage that Sapienz may obtain when executed for a longer time, we performed further executions with a time limit of 24 hours per app, considering as total coverage the one obtained as the union of the coverage of all the generated test cases. In order to reduce the dependence on randomness of these results, we repeated each 24-hour run three times, reporting in the following, for each AUT, the average total coverage of these runs.

The Robo tests were executed using the online Firebase platform. With Robo, only the branch coverage was measured since, as previously described, Robo cannot be executed together with the code coverage tool Emma. At the time of the experiments, a constraint of the available version of Robo was that each testing session could last up to 5 minutes, and longer sessions could not be executed. Nevertheless, none of the AUTs required more than 3 minutes to be tested.

Finally, let us note that, for both students and AIG tools, each testing activity started from the state obtained just after app installation.

Analysis of the Uncoverable Code

After the students performed the assigned tasks, we made a preliminary analysis of the AUTs code to assess whether some LOCs and/or branches could not be covered with the selected tools (both C&R and AIG), under the previously discussed experimental configuration. This is fundamental to obtain an upper bound on the results achievable by both C&R and AIG tools. All the considered AUTs presented some uncoverable LOCs/Branches. The maximum achievable coverage measures are reported in Table 5.3.

In the following we briefly explain why some parts of the code were uncoverable:

- **Unreachable code**. All the AUTs presented some code that cannot be exercised by any GUI testing solution. For example, there is dead

Table 5.3. Maximum achievable LOC coverage and branch coverage (BC) percentage for the AUTs.

	A1 - MunchLife		**A2 - SimplyDo**		**A3 - TippyTipper**		**A4 - Trolly**	
	LOC	BC	LOC	BC	LOC	BC	LOC	BC
Max Cov.	96%	92%	85%	79%	91%	86%	88%	67%

code corresponding to deleted menu entries, or also code handling bad data input format exceptions, whose execution is prevented by the behaviour of the input widgets, as in the case of non-numeric data that cannot be inputted from the numeric keyboard.

- **Code related to system events that cannot be triggered at application testing level**. There are some user or system events that are not exercisable by the considered GUI testing tools. For example, some intent calls can be generated only by other applications or by the operating system.

- **Code that can be activated only after multiple starts of the AUT**. In the experiments, all the testing activities started from the same initial state of the AUTs, i.e. the one immediately after the installation. As a consequence, different initial states cannot be covered. For example, SimplyDo can load a previously saved list of things to do, if present. This part of the code cannot be executed under our experimental conditions.

- **Code related to not supported widgets/events**. Both the used C&R tool and the AIG tools might not be able to exercise custom UI controls/events. For example, none of them supported the type of slider widget used in TippyTipper, and thus the code of the corresponding event handlers cannot be covered.

5.1.2 Results

This section presents the experimental results, along with quantitative and qualitative analysis aimed at addressing the research questions.

RQ1: How do test cases generated with an Uninformed Exploratory Testing strategy compare, in terms of effectiveness, to those generated by AIG tools?

This first research question aims at comparing the testing effectiveness, in terms of code coverage, of test suites produced by students using a C&R tool in an Uninformed Exploratory Testing scenario, against AIG tools.

Quantitative analysis The first analysis we have carried out regards the quantitative comparison between the effectiveness of the test cases generated by students and the one of the considered AIG tools, both in terms of LOC and Branch coverage.

Table 5.4 reports the measured coverage values achieved by all the students (named S1 to S20) and by the three AIG tools, for each of the considered apps (A1 to A4). In details, the first eight columns show the LOC coverage and Branch Coverage percentages. The last four columns report the total number of events composing the test cases produced by each student for each app. The number of events provides a measure of the complexity of the test suites produced by students.

Table 5.4 also reports average, median, standard deviation values, as well as the code coverage percentages reached by the union of the coverage sets of all the students. The average, median and standard deviation values of the number of events composing the test cases are also reported. As regards Sapienz, we have reported the average, median and standard deviation values of the coverage percentages obtained by 20 one-hour executions. For the sake of readability, we also included (on the top row of the table) the maximum achievable coverage percentages which were already reported in Table 5.3.

The measured coverage results obtained both by the students with the C&R tool in a UET scenario and by AIG tools are depicted in Figure 5.2 and in Figure 5.3, with boxplots representing, respectively, the LOC and Branch coverage percentages obtained by the students and by Sapienz in its 20 one-hour runs. Moreover, in the figures, we used horizontal lines to represent the LOC/Branch coverage percentage obtained by Android Ripper and Robo, and an additional horizontal line showing the maximum achievable coverage under the experimental conditions and constraints.

When testing A1, the students obtained LOC coverage percentages

Table 5.4. LOC/Branch Coverage (%) and Number of Events of Test Suites Produced with UET Approach

	LOC Coverage				**Branch Coverage**				**Number of Events**			
	A1	A2	A3	A4	A1	A2	A3	A4	A1	A2	A3	A4
Max Cov.	96	85	91	88	92	79	86	67				
Students												
S1	83	77	83	78	62	65	75	62	85	154	207	171
S2	91	84	86	76	80	76	80	60	79	267	240	145
S3	87	80	85	79	69	72	77	63	197	272	219	130
S4	86	61	87	75	70	67	67	62	62	63	122	60
S5	86	81	83	78	70	69	75	62	77	200	208	166
S6	86	82	87	79	64	72	80	64	48	299	159	162
S7	82	82	89	79	62	77	82	63	177	387	351	299
S8	79	82	79	75	56	71	70	58	64	149	124	114
S9	86	79	87	80	69	68	81	64	127	194	220	156
S10	90	77	79	78	74	69	68	62	208	172	204	145
S11	85	78	87	77	66	73	79	58	93	170	288	159
S12	91	79	84	72	87	65	75	55	198	236	270	147
S13	84	82	84	78	64	71	74	62	44	239	212	349
S14	83	82	85	76	59	73	79	58	88	244	135	151
S15	80	70	85	75	57	61	78	60	34	117	125	103
S16	90	74	84	80	82	64	75	62	71	164	150	182
S17	83	82	81	73	61	72	71	54	48	123	67	50
S18	90	82	84	79	75	74	78	63	192	178	154	142
S19	86	82	87	77	69	74	80	60	55	320	174	124
S20	82	78	83	77	70	65	73	62	67	125	114	144
Average	85	79	84	77	68	70	76	61	101	204	187	155
Median	86	81	84	77	69	71	76	62	78	186	189	146
St. Dev.	4	5	3	2	8	4	4	3	59	79	68	67
Union	96	85	91	88	92	79	86	67				
AIG tools												
Android Ripper	77	65	74	64	66	56	64	53				
Sapienz (1 hour)												
Average	85	42	85	64	68	37	76	51				
Median	85	41	85	63	68	36	77	50				
St.Dev.	0	5	2	3	0	4	2	2				
Robo	-	-	-	-	47	27	50	21				

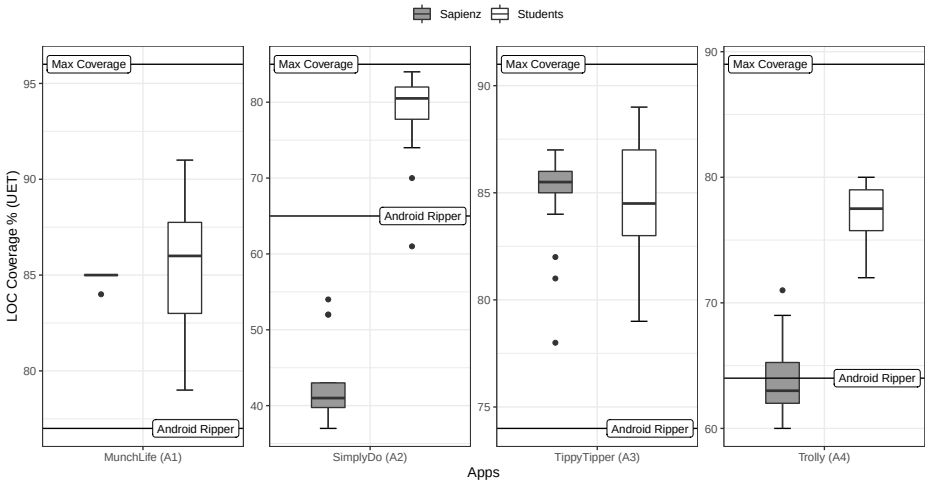


Figure 5.2. Boxplots representing the LOC coverage achieved with the UET approach

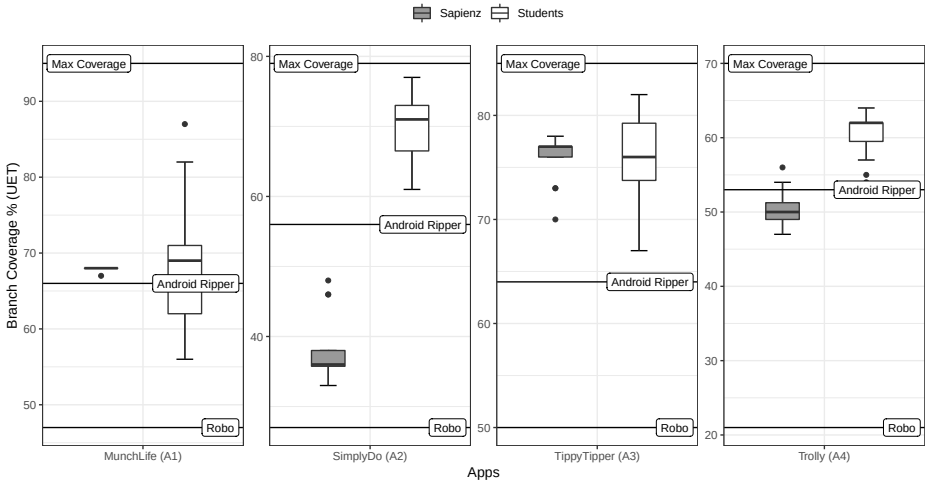


Figure 5.3. Boxplots representing the Branch coverage achieved with the UET approach

ranging between 79% and 91%, averaging at 85% with a standard deviation of 4%. The maximum achievable LOC coverage for A1 is 96%. With A2, the students covered on average 70% of the LOCs, with coverage ranging from 61% to 84% and a standard deviation of 5%, against a maximum achievable LOC coverage of 85%. As for A3, on average 84% of the LOCs were covered by the students, with 91% being the upper bound to the achievable LOC coverage. The students' coverages ranged between 79% and 87%, with a standard deviation of 3%. At last, for A4, in which the 88% of the LOCs were coverable, the students reached LOC coverage percentages in the range 72%-80%, averaging at 77% with a standard deviation of 2%.

Differences in LOC coverage between different students when testing the same app exist, but are quite limited: the standard deviation values are in the range between 2% (for A4) and 5% (for A2).

These data indicate that, when considering LOC coverage, there is no relevant difference among the 20 considered students when they perform Uninformed Exploratory testing in the specified conditions. Moreover, the data show that none of the students was able to reach the maximum achievable coverage in the given testing time, but each of them covered at least more than 60% of the code.

On the other hand, the Branch coverage percentage reached by the students for A1 is in the range between 57% and 80%, with an average of 68% and a standard deviation of 8%. The maximum branch coverage for the same app is 95%. As regards A2, in which 85% of the branches are coverable, the students reached Branch coverage percentages in the range 61%-76%, averaging at 70% with a standard deviation of 4%. With A3, on average 76% of the branches were covered by the students, against a maximum achievable Branch coverage of 85%. The subjects reached Branch coverage values in the range between 67% and 81%, with a standard deviation of 4%. Finally, when testing A4, the students obtained on average a branch coverage of 61%, with a standard deviation of 3% and 70% of the branches being coverable. The best students achieved a branch coverage of 64%, whereas the worst managed to cover 54% of the branches.

As regards the number of events composing the test cases, we observed that, differently from coverage percentages, the number of events presents a large variability among the different students. This datum is witnessed by

high standard deviation values with respect to the corresponding average values. Moreover, the data we gathered highlights that test cases consisting of more events do not generally produce larger coverage percentages.

The collected coverage data indicate that, for each of the considered AUTs, the students obtained much greater values in terms of LOC coverage than of Branch coverage. This fact is essentially due to the large portion of code that is automatically executed at the start of each Activity of each AUT (included in the *onCreate* methods, which were always covered by each student). Such portions of code have a considerable weight in the evaluation of the LOC coverage, whereas, on the contrary, they have a small impact on the Branch coverage, since they include few branches.

Differences in Branch coverage between different students when testing the same app are in a larger range with respect to LOC coverage values, with standard deviation values between 3% (for A4) and 8% (for A1). This is explained by the above considerations on the magnitude of the LOC and Branch coverage achieved by the students: the ability of a particular tester in exploring the AUT has a greater impact on the achieved coverage when considering branch coverage.

Moreover, we can observe that none of the students was able to obtain coverage equal to the overall level obtained by all of the students. In fact, the union of the coverage of the 20 students is always greater than each coverage obtained by a single one.

Furthermore, it is important to note that the maximum achievable coverage, both in terms of LOCs and Branches, coincides with the one obtained by considering the union of the coverage of all the students. In other words, *there is no coverable code under our experimental conditions that has not been covered by at least one of the students*. This is a first useful indicator of the potential capability of students in designing effective test suites by using a C&R tool.

As regards the effectiveness of the AIG tools, the measured coverage percentages are, for the convenience of readers, reported in the lower part of Table 5.4, where the coverage values can be easily compared against the students' results.

The different executions of Sapienz provided different coverage values. For this reason, we have represented them in form of boxplots in Figures Figure 5.2 and Figure 5.3. We have observed that, for application A1,

all the executions reached almost the same values of LOC and Branch coverage percentages, whereas for the other applications we observed larger deviation standard values, between 2% and 5% in LOCs and between 2% and 4% in Branches. The *Sapienz* tool outperformed the other ones, both in terms of LOC and Branch coverage in A1 and A3. As for A4, the performance of one-hour executions of Sapienz are similar to the ones of Android Ripper. As for A2, Android Ripper always overcame Sapienz both in terms of covered LOCs and Branches. The third tool, Robo, proved to be less effective than the others, achieving percentages far lower than those of the other considered tools both in terms of LOC and Branch coverage.

As shown by Figure 5.2, the test suites produced by the students with the UET approach almost always appear to be more effective than those of the AIG tools, for all the considered apps. Indeed, with respect to Android Ripper, there is a remarkable superiority in the coverage achieved by the students. As for Sapienz, the students' LOC coverage values are always better for A2 and A4, while for A1 most of the students cover more LOCs than Sapienz. Finally, for A3, the performance of students and Sapienz are similar.

As for the Branch coverage, the results are more contrasting (see Figure 5.3). Indeed, for A1, Sapienz and Android Ripper obtained notable results, with Sapienz overcoming most of the students. For A2 and A4, instead, the students obtained better coverage than the AIG tools. For A3 the coverage achieved by the students and by Sapienz were pretty close and better than Android Ripper. Finally, the effectiveness of Robo is always by far the lowest.

On the basis of these results, no general conclusions about the LOC and Branch coverage comparisons between students and AIG tools could be made. However, one datum that emerged from our analysis was that neither every single student nor each tool did cover all the coverable code, since none of them reached the Max Coverage of each app.

Qualitative Analysis. In this section, we present the results of the fine-grained qualitative analysis that we carried out to evaluate the capability of both the students using UET and the AIG tools in covering the app code. We restricted our analysis to the subset of coverable code obtained by the analysis reported in Section 5.1.1.

Table 5.5. Partition of Branches with respect to the Number of Students Covering Them in UET Approach

	Branches		Branches Covered in UET by		
	Total	Coverable	every student	the majority of the students	the minority of the students
A1 - MunchLife	61	56	26 (46%)	15 (27%)	15 (27%)
A2 - SimplyDo	419	329	101 (31%)	200 (61%)	28 (8%)
A3 - Tippy Tipper	208	178	91 (51%)	75 (44%)	12 (5%)
A4 - Trolly	120	80	55 (69%)	19 (24%)	6 (7%)

Analysis of the coverage of the students in the UET approach. In order to investigate the capability of the students using UET in covering the app code, we evaluated for each Branch the number of students that were able to cover it. Therefore, we did a qualitative analysis of the Branches that were less frequently covered by the students. Since most of the uncovered LOCs are included in the code underlying the uncovered branches, we decided, for sake of space, of performing a detailed analysis only of the uncovered branches.

To this aim, for each AUT, we classified its branches in three subsets: (I) those covered by all the students (20 out of 20), (II) those covered by the majority of the students ($10 \leq \#students \leq 19$), and (III) those covered only by the minority of them ($1 \leq \#students \leq 9$). We recall that, as discussed in the previous section, each branch of each AUT was covered by at least one student. The resulting numbers are reported in Table 5.5, together with the total number of branches of each AUT, and the number of branches that could be executed with the UET approach in the considered experimental configuration.

To get an insight into the reasons behind the common lack of coverage of some branches with the UET approach, we did a manual analysis of all the branches that the majority of the students failed to cover. For each of these uncovered branches, we also surveyed the students who failed to cover them, asking for the causes of this shortcoming. In the following, for each AUT, we group its uncovered branches in subsets containing branches that were not covered for similar causes. Moreover, we also assess whether the considered AIG tools had been able to cover these branches.

For the four AUTs, we grouped the uncovered branches in nine sets,

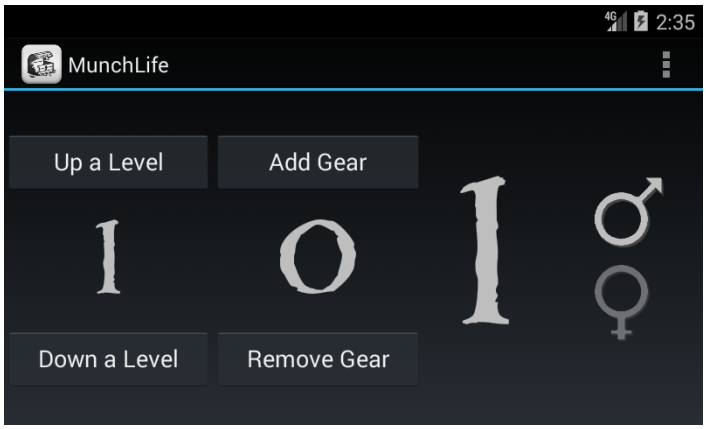


Figure 5.4. A Screenshot of the A1 App, in Landscape Mode

referred to as B1 to B9, described as follows.

B1 This group includes branches of the MunchLife app that could be covered only by triggering orientation change events. The app indeed presents two graphical widgets (representing Male and Female symbols) that are shown in the home activity of MunchLife (see Figure 5.4). They can be selected and deselected and different branches are activated whether the app is rendered in portrait or landscape mode as these symbols have a different layout in the two orientations. Only 4 students took into consideration device rotation events in their test cases, in combination with clicks on these two symbols. The other ones did not imagine the existence of these branches in the source code. Sapienz covered these branches, while Android Ripper systematically explored only the portrait GUI since it considered the behaviour of the portrait and landscape modes to be equivalent. Similarly, also Robo failed to cover these branches.

B2 The home activity of MunchLife shows a counter that can be incremented/decremented until a maximum value (which is set to 10 by default) is reached. This maximum value can be modified from a menu option. If the user sets a maximum value lower than the current counter value, a specific branch is executed to update the counter accordingly. Only 7 students out of 20 have tested this par-

ticular scenario. Analogously, none of the automatic tools triggered the specific sequence of events to activate this branch group.

B3 The maximum counter value in MunchLife can be set by a numeric input field. If a non-numeric symbol is inserted (e.g. / or * symbols), a branch handling a bad data format exception is executed. Only 7 out of 20 students designed this specific validation test case, whereas the others tested only for valid and invalid numeric inputs. Sapienz is the only automatic tool covering this branch, since it automatically generates both valid and invalid input data for numeric input fields.

B4 *Long Click* event handlers are defined in the source code of SimplyDo for some widgets (e.g. the ones used to add or edit items of the list), in addition to *Click* events handlers. Only one student triggered long-click events in its test cases and covered the branches corresponding to these event handlers. None of the considered tools covered these branches.

B5 The source code of SimplyDo includes some branches responsible for storing the managed list of items in the internal memory when the application is stopped and for their restoration when the application is restarted. The application stop and restart can be triggered by a device rotation event. Only 2 students triggered a device rotation after the insertion of items in the list. Sapienz was the only automatic tool able to cover these branches.

B6 The `SimplyDoActivity` class source code includes an event handler corresponding to the triggering of the back event on the device. In SimplyDo, lists of items are managed according to a tree structure. If a list contained in another list is shown, the back button can be used to show the container list. In case the back button is triggered on the root list, the app stops itself. Only 3 students exercised this stopping scenario, as most of the students declared that they didn't consider the need to test the exit from the app with the back button. Sapienz and Android Ripper covered this branch.

B7 Similarly to the branches in B4, there are branches related to *Long Click* events handlers in TippyTipper. Such branches are related

to the buttons used to reset the input number or to delete a digit from it. Only 3 out of 20 students defined test cases including these events. In this case, Sapienz and Robo covered these branches.

B8 In TippyTipper there is a branch related to the opening of a menu in the `SplitBill` Activity. The menu in this Activity has no items (probably the feature is partially unimplemented). Only 6 students tested it. Sapienz and Android Ripper covered this branch.

B9 Trolly is a shopping list app. It can manage three different lists of items: the first one includes those to buy, the second one includes those that have been bought, and the third one (called OffList) includes the items that were added and then removed in the past from the first list. This OffList is used by the app to provide input suggestions, and its visualisation can be activated by clicking on the *Add* widget without having selected an object to add, provided that at least one object was added to the list in the past. This feature is not explained in the app GUI and has not been tested by the majority of the testers (only 4 students out of 20 designed test cases covering this branch). None of the automatic tools was able to cover it.

Summing up, we observed two main reasons for the lack of coverage achieved by the students, as follows:

1. *Lack of Information about the AUT.* Some code was not covered because testers had only limited knowledge of the functionalities offered by the GUI of the AUT. For example, branches B2, B3 and B9 can be covered by a complex sequence of events, not self-explained by the GUI. The majority of the students, in their exploration of the GUI, failed to identify these features. These targets generally were not even covered by automatic tools. Similarly, branches B4, B7 and B8 correspond to uncommon events, such as long clicks on widgets (for which they anyhow explored the standard click events), or opening an empty menu. Most of the AIG tools covered these branches.

2. *Strict testing time limit.* In the cases of branches B1, B5 and B6, students reported that they missed them also for lack of time. These branches, indeed, correspond to interactions between application events

and lifecycle events (e.g. device rotations for B1 and B5, "back" event for B6). Testing all the possible interactions is an onerous activity in terms of the number of test cases to be designed and is often neglected by testers (cfr. [15]), also due to the strict time limit. On the other hand, all these branches have been covered by Sapienz.

In conclusion, we found that the main cause of ineffectiveness for students using a UET approach lies in the lack of information about the AUTs, and in strict time constraints for the execution of the testing tasks.

Analysis of the coverage of the AIG tools. When considering the coverage data of the AIG tools and comparing them to those achieved by the students, we found that each branch covered by at least one tool was covered by at least one student, and a remarkable amount of LOCs (between 9% and 23%) was covered by at least one of the students but none of the AIG tools.

It is worth to note that Sapienz was able to find a crash in A4 (Trolly) that was not found by any of the students. This crash is caused by the input of a single quote ' character in the text input field, leading to an unhandled exception due to the execution of a malformed query on the SQLite database of the app. Since there is no source code for the handling of this exception, no differences in code coverage were observed.

We analysed in details the parts of code of the AUTs that were covered by at least one of the students but not by Sapienz (as this is the most effective of the AIG tools), to understand the causes behind this lack in coverage.

In the following we report some notable examples of lack of coverage of all the AIG tools:

- The first example is represented by the branch B2 above described. This branch can be activated only by a sequence including two or more counter increments, the opening of a specific entry of the menu, and the setting of a maximum counter value, lower than the current counter value. Of course, the probability for a random-based (or search-based) testing tool to explore this sequence of events is very low. A similar case is represented by B9.

- To win a game in MunchLife the counter must reach the winning value (initially set to 10), so a sequence of at least 10 events is needed. Sapienz has not explored this scenario. Android Ripper, similarly, incremented the counter just once because it did not observe any variation in the GUI except for the counter value. On the other hand, the winning scenario has been considered obvious by all of the students (20 out of 20 have covered this branch).
- In TippyTipper a branch is covered when (I) the tax rate has been enabled by selecting a specific entry in the menu, (II) a value is set for the tax rate by selecting another specific entry of the menu (enabled by the previous action), and (III) the tip is calculated by clicking on the corresponding button on the GUI. This scenario has not been found by any AIG tools, but it has been tested by 14 out of 20 students.
- Still in TippyTipper, a branch can be tested by (I) setting the rounding strategy for the tip from a specific menu entry and (II) executing a tip calculation requesting for an explicit rounding of the tip. All the students recognised and covered this branch, but none of the AIG tools.
- Sapienz does not consider events related to the context menu implemented in Trolly, because the latter is implemented using old libraries. A similar problem occurs in SimplyDo, where it fails to acknowledge the existence of events related to a context menu activated when long-clicking on list items. These two problems are the most relevant in terms of lack of coverage.
- In TippyTipper there is a `NumberPicker` widget, being an extension of old widgets, which was not considered by Sapienz.

As for the cause of the above lacks, we can classify them as follows:

- *Technological issues.* AIG tools are not able to deal with unsupported events/widgets. Thus they need to be constantly updated, to keep the pace of the continuous evolution of the Android framework. Similarly, their backward compatibility should also be improved. Furthermore, in presence of custom widgets/events, the effectiveness of these tools is undermined.

- *Methodological issues.* AIG tools are not able to deal with long and/or complex sequences of events needed to activate functionalities. In fact, one of the factors optimised by Sapienz is the length of the executed event sequences, so long event sequences have a lower probability to be tested. Android Ripper follows a heuristic designed to avoid repetitions of events on already visited GUIs: in this way it tries to execute at least once any possible event, but it doesn't try them in all possible orders. These portions of code, instead, are often tested by humans, when they comprehend the existence of a specific execution scenario involving that sequence of events.

RQ2: How do test cases generated with an Informed Exploratory Testing strategy compare, in terms of effectiveness, to those generated by AIG tools?

The second research question aims at the comparison, in terms of LOC/Branch coverage, between the test suites produced with the Informed Exploratory Testing (IET) approach and the AIG tools. In the IET approach, the same students involved in the previous experiment were assigned the task of improving the coverage achieved in the first scenario, by adding further test cases. In this phase of the experiments, the students had the opportunity to inspect the AUTs source code and to consider the actual coverage achieved by the test cases as feedback. Furthermore, they did not have a strict constraint on the testing time. Nevertheless, they could take the exam only after delivering the test suites. For this reason, they still were interested in completing the testing task in the shortest possible time. The students claimed to have taken five to fifteen days to perform this task, during which they also studied for the exam.

As regards the AIG tools, in order to have a fairer comparison with the students, we ran Sapienz multiple times for each app, for a longer testing time (24 hours). Therefore, we evaluated the coverage percentage of the union of all the generated test suites. We compared the coverage percentages achieved by the students with the average ones of Sapienz.

As for Android Ripper and Robo, an increment of the testing time would not lead to any increment in coverage, since, for all the AUTs, these two tools terminated the exploration of the GUIs not due to time constraints. Therefore, in this analysis, we considered the same values

Table 5.6. LOC/Branch Coverage Percentage and Number of Events of Test Suites Produced with IET Approach

	LOC Coverage				Branch Coverage				Number of Events			
	A1	A2	A3	A4	A1	A2	A3	A4	A1	A2	A3	A4
Max Cov.	96%	85%	91%	88%	92%	79%	86%	67%				
Students												
S1	93%	82%	86%	81%	84%	74%	80%	65%	189	199	227	231
S2	95%	85%	89%	82%	89%	77%	82%	66%	87	308	348	282
S3	92%	82%	89%	82%	87%	73%	81%	66%	253	320	277	170
S4	94%	69%	87%	88%	85%	62%	77%	68%	71	97	133	97
S5	92%	82%	86%	83%	89%	73%	80%	67%	115	258	228	200
S6	95%	85%	90%	84%	90%	79%	84%	68%	61	323	194	171
S7	95%	82%	91%	82%	92%	74%	85%	66%	218	387	402	327
S8	91%	83%	86%	83%	80%	76%	78%	67%	109	159	154	136
S9	94%	79%	87%	81%	90%	68%	83%	65%	137	204	242	162
S10	92%	81%	89%	81%	84%	71%	82%	65%	253	205	348	152
S11	95%	80%	89%	81%	92%	72%	81%	67%	318	189	482	177
S12	94%	81%	87%	80%	89%	74%	79%	65%	230	242	538	196
S13	95%	82%	87%	80%	87%	58%	80%	65%	60	255	319	415
S14	90%	82%	85%	81%	84%	64%	79%	64%	99	265	141	170
S15	96%	82%	85%	80%	93%	74%	82%	65%	82	160	168	153
S16	96%	82%	89%	89%	92%	64%	82%	70%	102	232	224	207
S17	94%	84%	85%	81%	89%	75%	78%	66%	93	178	161	100
S18	96%	84%	87%	85%	92%	77%	82%	68%	231	197	177	165
S19	93%	84%	89%	77%	84%	76%	81%	62%	240	344	192	131
S20	96%	85%	88%	84%	95%	78%	81%	66%	76	187	152	155
Average	94%	82%	88%	82%	88%	72%	81%	66%	151	235	255	190
Median	94%	82%	87%	81%	89%	74%	81%	66%	112	219	226	170
St. Dev.	2%	3%	2%	3%	4%	6%	2%	2%	81	73	116	76
AIG tools												
Android Ripper	77%	65%	74%	64%	66%	56%	64%	53%				
Sapienz 24 hrs union avg	88%	54%	87%	72%	71%	48%	81%	57%				
Robo	-	-	-	-	47%	27%	50%	21%				

obtained in the previous experiment.

Quantitative analysis Table 5.6 reports the measured IET coverage values for the students involved in the experiment. In detail, for each student, both the LOC/Branch coverage percentages are reported, for each of the four apps. In addition, the last four columns report the total number of events composing the test cases produced by the students.

In the last rows of the table, the average, median and standard deviation values are reported, too. For the convenience of readers and ease of comparison, we also included the maximum achievable coverage values (which were previously reported also in Table 5.3). In addition, we re-

ported the average values of coverage percentage evaluated by considering the union of the coverage of the test suites produced by Sapienz in 24 hours, measured with respect to three runs. Finally, we reported the coverage reached by Android Ripper and Robo, from Table 5.4.

The boxplots in Figure 5.5 and in Figure 5.6 graphically show the distribution of the LOC/Branch coverage percentages achieved by the students. Moreover, we added horizontal lines corresponding to the coverage percentage obtained by the three considered AIG tools and an additional horizontal line showing the maximum achievable coverage with respect to the experimental conditions and constraints. As highlighted by the plots, the coverage achieved by the students is generally better than the one obtained by each automatic AIG tool, both in terms of LOC and Branch coverage. The only exceptions can be observed for A3, where the longer Sapienz executions achieved better coverage percentages than two of the students. We can conclude that the effectiveness of the exploratory testing highly improved thanks to the source code knowledge and the additional time given to the students.

As regards the number of events composing the test cases, this measure presents a large variability also in this case. Moreover, longer test cases do not generally correspond to test suites achieving greater coverage percentages. In addition, the increase in the number of events of the test suites obtained by the IET approach with respect to the ones obtained by the UET approach is not very large. The average increase per app is indeed less than half the length of the UET test suites.

In order to characterize the code coverage improvements obtained by the IET approach, we performed a further analysis where we compared the coverage sets obtained using the UET (see Table 5.4) and the IET (see Table 5.6) approaches. By these tables, it is possible to observe an increase in coverage for the apps A2, A3 and A4 between 3% and 5% in terms of LOC and between 2% and 5% in terms of covered branches. A larger increase has been observed for app A1, for which the LOC coverage increased by 9% and the branch coverage by 20%. In addition, a general reduction of the standard deviation values with respect to both coverage metrics can be observed.

As regards Sapienz, an increase in coverage can be observed with respect to that of the one-hour runs previously presented. The increase in

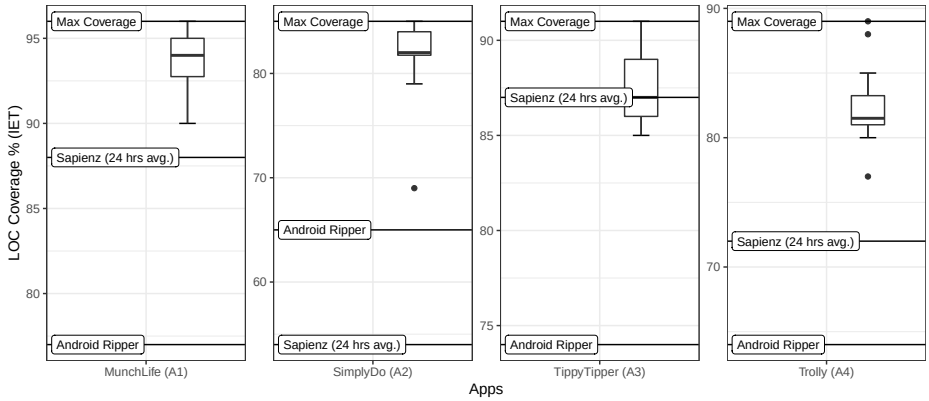


Figure 5.5. Boxplots representing the LOC coverage achieved with an IET approach

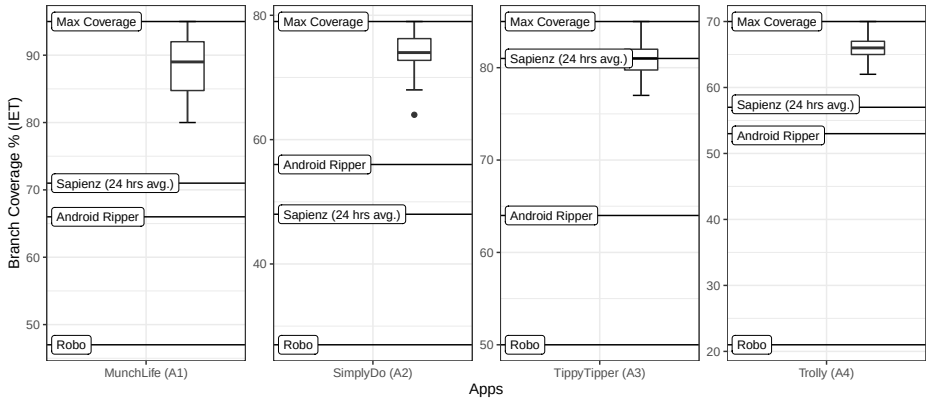


Figure 5.6. Boxplots representing the Branch coverage achieved with an IET approach

LOC coverage is quite small for A1 and A3 (3% and 2%, respectively), whereas it is more relevant for A2 and A4 (12% and 8%, respectively). For all the apps except A3, the LOC coverage provided by Sapienz remains always smaller than the ones obtained by students adopting the IET approach. Only for A3, the average coverage obtained by Sapienz is almost the same as the average of the values obtained by students. Similar results were obtained as regards the Branch coverage.

Qualitative analysis We have measured, as in the previous case, the number of branches covered by all the students, by the majority of the students, and by the minority of the students using the IET approach. In Table 5.7 we reported the number and percentage of branches covered by the test suites produced by (I) all the students (20 out of 20), (II) the majority of the students ($10 \leq \#students \leq 19$), and (III) the minority of them ($1 \leq \#students \leq 9$).

Table 5.7. Partition of Branches with respect to the Number of Students Covering Them in IET Approach

	Branches		**Branches Covered (IET) by**		
	Total	Feasible	every student	the majority of the students	the minority of the students
A1 - MunchLife	61	56	34 (61%)	22 (39%)	0 (0%)
A2 - SimplyDo	419	329	147 (45%)	163 (49%)	19 (6%)
A3 - TippyTipper	208	178	136 (76%)	33 (19%)	9 (5%)
A4 - Trolly	120	80	65 (81%)	15 (19%)	0 (0%)

The amount of branches covered by all the students is strongly increased by the IET w.r.t. UET, as can be seen by comparing Table 5.5 and Table 5.7: for the UET test suites, it varied between 31% and 69% for the four apps, while it is between 45% and 81% for the IET test suites. We can observe that for three out of four applications, the majority of the reachable branches have been covered by all the students, whereas in the other app (A2) the 45% of branches have been covered by all the student's test suites. It can also be noted that the number of branches covered only by a minority of students is sensibly reduced. In particular, in the two smaller applications A1 and A4, all the branches were covered at least by the majority of the students (there were 21 branches covered only by the

minority of the students with the UET test suites).

As in the previous qualitative analysis, we focused our attention on the set of branches B1 to B9, which caused a lack of coverage in the UET approach. During the final examination of each student, we asked about how they reached / why they did not reach these branches. In what follows, we summarised what they reported.

- *Branches covered by the majority of students thanks to the knowledge of the source code.* In three cases (B1, B3 and B7) the knowledge of the source code has allowed the majority of students to design test cases covering these branches. For example, almost all the students (17 out of 20) covered the B1 branches, corresponding to the interactions with the male and female widgets of A1 in landscape mode, too. In fact, in the source code, the branches are explicitly related to landscape mode and male/female widgets, as can be observed in Listing 5.1. Analogously, the majority of the students (11 out of 20) covered the branch B3, since in the source code there is an explicit `NumberFormatException` handler related to the numeric input field. Similarly, the majority of students (17 out of 20) managed to cover branch B7, due to the presence of an explicit *Long Click* event handler in the source code. In these cases, the students highlighted that the uncovered source code was found thanks to the source code level coverage reports provided by Emma.

- *Branches covered by the majority of students thanks to a better comprehension of the behaviour of the application under test.* In branches B2 and B9, the uncovered code corresponded to scenarios that were not identified by students in the UET approach. In both cases, the branches were covered by the majority of students in the IET approach (respectively 13 and 16 out of 20 students). These students reported that they exploited the knowledge of the source code of the AUTs to identify the uncovered branches and to understand the behaviour of the app, also thanks to debugging activities they carried out. The time needed to perform these activities was not compatible with the strict time limit imposed in the UET approach.

For the remaining branches (B4, B5, B6 and B8), the number of students that managed to cover them with their test suites increased, but the ma-

Listing 5.1. Source Code of B1 Branches

```
private OnClickListener genderClickListener = new OnClickListener() {
    @Override
    public void onClick(View v) {
        if(devDisplay.getRotation() == 0) {
            if(gender_female == false) {
                gender.setImageResource(R.drawable.female_portrait);
            } else {
                gender.setImageResource(R.drawable.male_portrait);
            }
        } else {
            if(gender_female == false) {
                gender.setImageResource(R.drawable.female_landscape);
            } else {
                gender.setImageResource(R.drawable.male_landscape);
            }
        }
        gender_female = gender_female == true ? false : true;
    }
};
```

jority of students still failed to cover them. We asked the students who did not cover these branches why they failed in doing so. Interesting observations were brought on by the students, summarised as follows:

- In some cases, such as branches B5 and B6, covered only by 5 and 6 students, respectively, they were not able to design effective test cases. These branches were activated by a combination of user events and activity life cycle events. Some students did not comprehend the importance of testing all the scenarios related to these interactions, whereas some other students did not remember that device rotation events were able to trigger activity lifecycle events, too.

- In other cases, such as branches B4 and B8, students did not cover the branches since they did not consider them as important and they neglected their coverage in order to save testing time.

In conclusion, our results show that the availability of source code and longer testing times lead to a remarkable increase in the obtainable coverage. The remaining lack of coverage can generally be attributed to the

limited testing skills of some students and to their necessity to accomplish the assigned task in a reasonable time and not delay the examination schedule.

5.1.3 Final Remarks

The study we performed provided several useful insights about Exploratory Testing processes based on C&R and AIG tool-based ones.

The experimental results showed that an exploratory testing process executed by a novice tester with a C&R tool and no prior knowledge of the AUT (i.e., an *uninformed* approach) may yield to code coverage comparable to that of the best freely-available AIG tool, at least when both processes are executed in the same amount of time considered in the experiment.

Moreover, the experiment showed that a remarkable increase in coverage w.r.t. the UET strategy can be obtained by providing the testers with more information about the AUTs (i.e., visibility of the source code, details of the achieved coverage) and a suitably longer testing time. In such Informed Exploratory Testing (IET) scenario, the novice testers generally outperformed the AIG tools, even when these were executed for a longer period of time.

On the other hand, we observed that the effectiveness of the considered AIG tools is mainly limited by technological issues. These tools can rapidly become obsolete if they fail to keep the pace of the ever-continuing evolution of the Android framework, and this limits their effectiveness. Therefore, these tools have large margins for improvement, as long as they are kept up to date. As for Sapienz, in particular, we observed that its performance depends on randomness and that longer testing times may result in greater coverage percentages. Anyway, the performance of Sapienz generally remains worse than the ones of students.

Another finding regards the observed diversity between the coverage obtained by different students in ET scenarios. Different students were able to cover different testing targets, and the union of the coverage achieved by multiple students improves significantly w.r.t. the coverage achieved by single students. This finding indicates that the effectiveness of the uninformed exploratory testing approach may be significantly improved by allocating a larger number of testers. Of course, doing so will multiply

the cost of manpower.

Another consideration regards the different typologies of testing targets that were covered by the different testing techniques. Thanks to their capability of deducing the possible behaviour of the AUT, based on the analysis of the app GUI, human testers are generally able to record test cases covering the main app scenarios, even in absence of documentation about app requirements. This observation is coherent with the positive results reported in the literature in the context of exploratory testing. Moreover, human intelligence proved to be by far superior in testing also usage scenarios that are only reachable through complex sequences of input events. Vice-versa, humans often failed in systematically testing all the possible input events for a given widget, or combinations of user and system events. As a consequence, AIG tools appear more useful in these cases, as witnessed by the coverage obtained by the tools in the experiment and by the proliferation in the literature of approaches related to specific testing issues, e.g. concurrency testing, security testing, performance testing, compatibility testing, energy consumption testing and so on [114].

These findings may provide useful indications for a project manager who is in charge of deciding on a testing strategy that combines manual exploratory testing and automated testing, in accordance with the available testing time and resources.

5.2 Investigating Exploratory Crowdtesting

New perspectives for ET in the mobile apps domain raised in the last years with the diffusion of the crowdtesting paradigm [227, 137, 92, 88, 207], discussed in Section 2.2.5.

Crowdtesting activities typically combine well with ET strategies. Indeed, exploratory approaches can greatly benefit from the diverse background and skill sets of the different crowdworkers, resulting in overall more reliable testing results [204]. Moreover, by requiring crowdtesters to record their interactions with the AUT by means of C&R tools, it is also possible to obtain a re-executable test suite to be used in future regression testing activities.

On the other hand, well-known software project management challenges, such as determining the best trade-off between testing costs and

software quality, are particularly critical for crowdtesting, due to the complexity of mobile apps and unpredictability of distributed crowdtesting processes [204]. The decision on how much crowdtesting resources should be allocated is further complicated by the possibility of employing different exploratory testing strategies, i.e., *Informed* and *Uninformed* approaches (see Section 5.1). Indeed, a project manager might be puzzled in deciding which exploratory testing strategy to adopt in a crowdtesting task, with a fixed budget: *is it better to recruit a greater number of crowdtesters without prior knowledge of the AUT, or is it better to pay a smaller number of crowdtesters to carry out more time-consuming Informed exploration activities?*

As discussed in Section 2.2.5, number of works in the literature have proposed decision support systems aimed at assisting practitioners in effectively managing the crowdtesting process by predicting when to stop [204, 205] or by recommending how many and which testers should be recruited [206]. These works, however, consider a scenario in which the goal is to detect faults, and not to also produce a re-executable test suite adequately stressing the AUT, which could be used in subsequent regression testing activities. Moreover, they do not take into account different exploratory strategies. To the best of our knowledge, there exists no work in the literature aimed at supporting the managerial choice of deciding how many crowdtesters should be recruited, and which exploratory strategy should be used, in a scenario in which the goal is to obtain re-executable test suites.

5.2.1 Empirical Study Design

The aim of this study is to investigate the effectiveness of test suites generated by different-sized non-communicating crowds of testers with C&R tools and with different exploratory testing strategies. In this section, we start by presenting the research questions we investigate. Then we detail the evaluation we conducted, by describing the involved subjects (i.e. the testers), the tools, the AUTs, the metrics that we considered when comparing the effectiveness of test suites, and the experimental procedure.

Research Questions

To investigate the impact of crowd size on the effectiveness of the GUI test suites generated by independent crowdtesters with C&R techniques and different exploratory testing strategies, we consider the following research questions.

RQ1 (Impact of increasing crowd size) *What is the effect on test effectiveness of adding additional workers to non-communicating crowds of testers using a UET (resp., IET) strategy?*

RQ2 (Comparing UET and IET) *How do test suites generated by different-sized crowds of testers with a UET strategy compare, in terms of effectiveness, to those generated by different-sized crowds of testers with an IET strategy?*

RQ3 (Cost-effectiveness of UET, IET and mixes strategies) *Under a fixed budget, how do test suites generated by crowdtesters using a UET, IET, and mixed approach compare, in terms of effectiveness?*

Procedure

The empirical study described in this section elaborates on the results of the study described in Section 5.1. Indeed, under the assumption that the master's students involved in that study can be considered as representative of practitioners taking part in crowdtesting activities, we base our crowdtesting analyses on the test suites each of the students generated in isolation using UET and IET strategies, as described in Section 5.1.1. Starting from tests produced by each subject in isolation, we combinatorially computed, for each of the AUTs, for each ET strategy, and for each of the 1 048 575 distinct subsets arising from our subject set of 20 testers, the aggregate coverage that particular crowd would have achieved. In Table 5.8, we report more details on the investigated subsets of testers and show, for each considered cardinality in $[1, \ldots, 20]$, the corresponding number of subsets. The number of considered subsets ranges from a single subset of size 20 to 184 756 subsets of size 10. For the sake of compactness, since the number of subsets of cardinality i and $20 - i$ is the same for all i in $1, \ldots, 20$, we report only one row for both cardinalities. For example,

Table 5.8. Details on the investigated crowds of testers.

Subset cardinality	Number of distinct subsets
20	1
1 or 19	20
2 or 18	190
3 or 17	1 140
4 or 16	4 845
5 or 15	15 504
6 or 14	38 760
7 or 13	77 520
8 or 12	125 970
9 or 11	167 960
10	184 756
Total	**1 048 575**

there exist 20 different subsets of a single tester, as well as 20 distinct subsets of 19 testers.

To answer RQ3 and ensure a fair comparison between different-sized crowds using different strategies, we took into account the overall effort (measured in man-hours) required to perform each testing task. Such effort is directly proportional to the monetary rewards crowdtesters would receive upon completing the tasks. In particular, in our analysis, we assigned a one-hour effort to Uninformed testing tasks, and a three-hour effort to Informed ones (e.g.: three testers using an Uninformed approach cost the same as one tester performing an Informed approach). We assigned a three-hour effort to IET tasks because we consider both the initial Uninformed exploration (which lasted one hour per app) and the subsequent Informed exploration (which lasted on average two hours per app) as part of the IET process.

Furthermore, to answer RQ3, we also computed the average LOC coverage achieved by crowds of testers using heterogeneous exploratory strategies (i.e.: some testers in the crowd use a UET approach, whereas some others use an IET approach). We refer to this heterogeneous approach as *Mixed*. Mixed crowds can be a very interesting solution for crowdtesting, in scenarios in which it is not possible to share the source code of the AUT with external crowdtesters. Indeed, a project manager might recruit a number of external crowdtesters using an *Uninformed* approach, and have some other workers, internal to the company, record test suites with

an Informed strategy. In this case, to face the combinatorial explosion in the number of possible subsets to consider, we limited our analysis to the configurations in which the overall effort in man-hours does not exceed 20 hours. We selected this threshold so that the results of *mixed* crowds can be compared with both those of UET and IET crowds, under the same man-hours budgets. Moreover, for each valid combination of IET and UET testers not exceeding the 20-hour effort threshold, we randomly sampled 1 000 distinct crowds, guaranteeing that the subsets of testers using a UET and an IET strategy are always disjoint.

The data and code to replicate our findings are publicly available in a replication package [35]. In particular, the package contains:

- the source code of the four Android applications we considered in our study;
- the test suites developed by the students using UET and IET strategies, including code coverage reports;
- Python scripts we developed to compute the aggregate coverage of all the considered subsets of students' test suites and of a sample of the subsets including mixes of test suites produced by UET and IET tester;
- R scripts we developed to carry out statistical analyses and draw plots.

5.2.2 Results

RQ1: Impact of Increasing Crowd Size

In Table 5.9, we report, for each AUT, for each exploratory strategy, and for each considered number of testers, the average LOC coverage percentage achieved by crowds of that size. Figure 5.7 shows the average and median LOC coverage achieved by different-sized crowds of testers, as well as the measured standard deviation. These plots show that, regardless of the considered exploratory strategy, the benefit on effectiveness of adding an additional tester to the crowd is greater the smaller the original crowd, and becomes negligible as the size of the original crowd increases. To better visualize the differences in terms of achieved LOC coverage between

Table 5.9. Average LOC coverage percentage achieved by different-sized crowds of testers.

	MunchLife		SimplyDo		TippyTipper		Trolly	
Num. of testers	**UET**	**IET**	**UET**	**IET**	**UET**	**IET**	**UET**	**IET**
1	84,1	93,4	78,5	82,8	83,9	87,3	77,5	83,5
2	89,7	95,9	82,3	84,0	86,9	89,2	79,0	84,3
3	91,7	96,3	82,9	84,6	87,6	89,7	80,0	85,4
4	92,8	96,5	83,1	85,0	88,0	90,1	80,5	86,1
5	93,5	96,6	83,2	85,2	88,3	90,4	80,8	86,7
6	93,9	96,7	83,3	85,4	88,5	90,6	81,0	87,1
7	94,3	96,8	83,5	85,6	88,8	90,7	81,1	87,4
8	94,6	96,8	83,6	85,7	89,0	90,9	81,2	87,6
9	94,8	96,9	83,7	85,8	89,2	91,0	81,3	87,8
10	95,0	97,0	83,8	85,8	89,4	91,2	81,4	87,9
11	95,2	97,1	83,9	85,9	89,6	91,3	81,4	88,1
12	95,4	97,2	84,0	86,0	89,8	91,4	81,4	88,1
13	95,5	97,3	84,1	86,0	90,0	91,5	81,4	88,2
14	95,7	97,3	84,3	86,0	90,2	91,6	81,4	88,2
15	95,8	97,4	84,4	86,1	90,4	91,6	81,4	88,3
16	95,9	97,5	84,5	86,1	90,5	91,7	81,4	88,3
17	96,0	97,6	84,6	86,2	90,7	91,8	81,4	88,3
18	96,1	97,7	84,7	86,2	90,9	91,8	81,4	88,3
19	96,1	97,7	84,8	86,2	91,1	91,9	81,4	88,3
20	96,2	97,8	84.9	86,3	91,2	91,9	82,1	88,9

different-sized crowds of testers using the same exploratory strategy, in Figure 5.8 we depict such differences using tile diagrams. In that figure, darker tiles are associated with higher differences in coverage. We can observe that the area with white tiles (and, in general, lighter tiles) is more extended in the diagrams corresponding to crowds of IET testers, independently from the AUT. In other words, the benefit on effectiveness of adding an additional tester to the crowd is generally greater for UET testers with respect to IET testers.

To determine whether the differences in LOC coverage percentage among different-sized crowds are statistically significant, we performed statistical tests. In particular, for each AUT and exploratory strategy s, and for each pair of cardinalities (k, k'), with $k, k' \in [1, \ldots, 19]$, $k > k'$, we tested the null hypothesis:

$\mathbf{H_0^{AUT,s,k,k'}}$: The LOC coverage achieved using the exploratory strategy

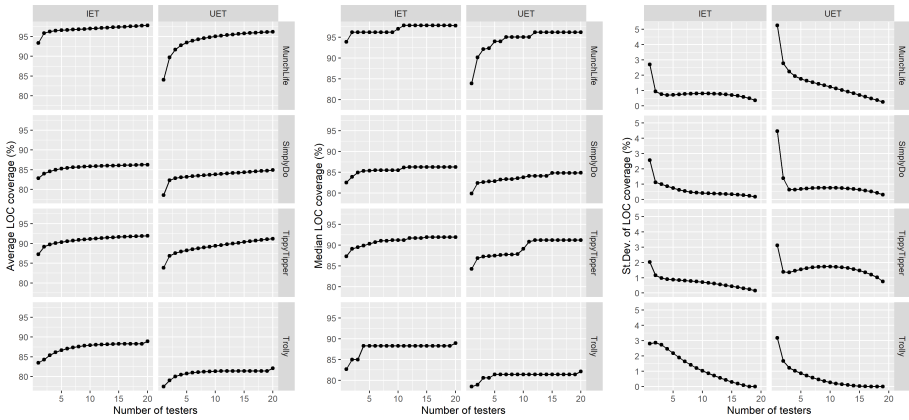


Figure 5.7. Average, median, and St. Dev. of the LOC coverage achieved by different-sized crowds of testers.

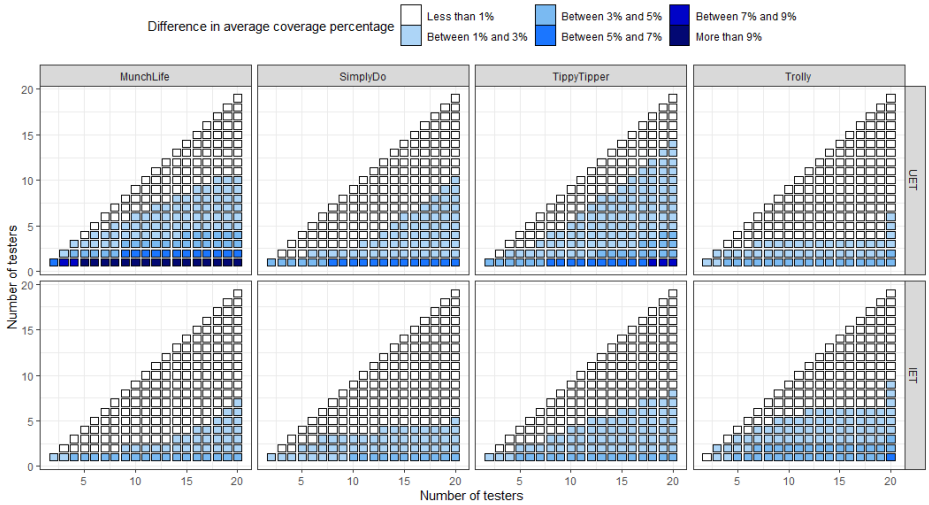


Figure 5.8. Difference in LOC coverage percentage achieved by different-sized crowds of testers using the same exploratory testing strategy.

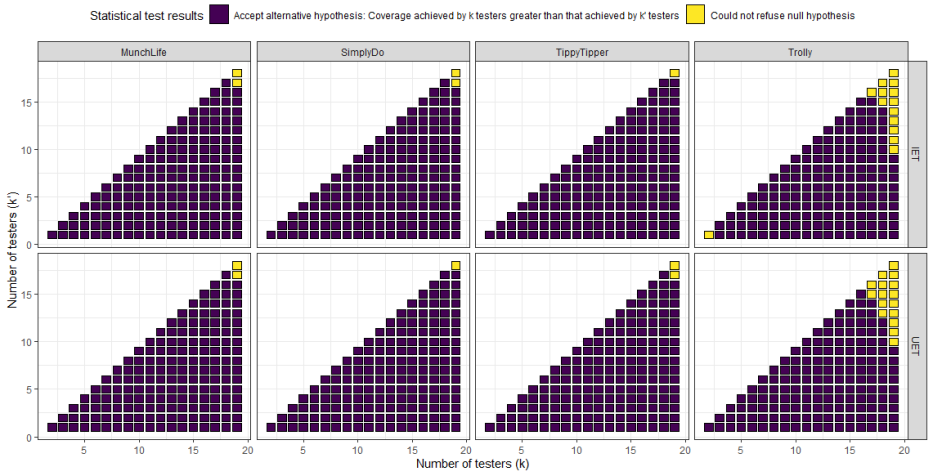


Figure 5.9. Results of hypotheses testing: comparing the effectiveness of test suites generated by different-sized crowds using the same exploratory strategy.

s on AUT by k testers is smaller than or equal to the one achieved by k' testers.

Notice that we do not include in our statistical analysis crowds of cardinality 20 because, as shown in Table 5.8, only one subset of that cardinality exists in our setting, and thus the reduced sample size hinders the applicability of statistical tests. To test the null hypotheses, we first tested the LOC coverage distributions for normality using the Shapiro-Wilk test. Then, when both distributions of LOC coverage were normal, we used the Student's T test to evaluate the null hypotheses. In the other cases, we used the Mann–Whitney–Wilcoxon test [212], which does not assume normal distributions. When the test $p\text{-}value < 0.05$, we reject the null hypothesis with high confidence, accepting the alternative hypothesis that the LOC coverage achieved by k testers is greater than that achieved by k' testers. Statistical test results are reported with tile plots in Figure 5.9. In the figure, light tiles correspond to null hypotheses that we could not reject, whereas dark tiles correspond to scenarios in which the tests confirmed statistically significant differences in coverage. More in detail, the dark tiles are associated with scenarios in which the number of testers on

the x-axis achieves a greater coverage than the corresponding number of testers on the y-axis. As the plot highlights, in almost every scenario, we detected statistically significant improvements in LOC coverage when increasing the number of testers, with the only exceptions being situations in which the original crowd had already a large number of testers (light area in the upper-right corner of each tile plot in Figure 5.9).

Even when statistically significant, however, the difference in LOC coverage achieved by different-sized crowds of testers might be negligible and have no practical impact. Hence, we measured the magnitude of these differences using the *Cliff's delta* effect size [55], a metric that is largely used in software engineering studies to compare the degree of difference between the two experimental results [107]. Cliff's delta ranges between -1 and 1, and can be interpreted as follows: if $|\delta| < 0.147$, the difference is *negligible*; if $0.147 \leq |\delta| < 0.33$, the difference is *small*; if $0.33 \leq |\delta| < 0.474$, the difference is *medium*; if $|\delta| \geq 0.474$ the difference is considered *large*. Results of this analysis are shown in Figure Figure 5.10.

RQ1: *Results highlight that, regardless of the considered exploratory strategy, increasing the size of the crowd of testers by small amounts (e.g.: 1 to 3 additional testers) generally leads to small or negligible improvements in coverage, especially when starting with an already consistent crowd (e.g.: more than 10 testers).*

Figure 5.7 and Figure 5.8 show that the greater sensitivity to the increase in the number of UET testers is observable for MunchLife, for which there are steeper slopes of the curves related to the average and to the median of the LOC coverage (cfr. Figure 5.7). Similarly, the darker area is larger for MunchLife (cfr. Figure 5.8). We analyzed the source code and the coverage sets produced by the UET testers for this application and we noticed the presence of relatively large areas of code that were executable only by test cases including device orientation changes events. The presence of device orientation-dependent code is unusual in small-sized Android apps and most UET testers did not consider these events in their test cases because their systematic inclusion certainly has remarkably increased the time needed to capture the test cases. In addition, other chunks of code that have not been covered by many UET testers are related to complex sequences of events, not self-explained into the GUI screens, as described

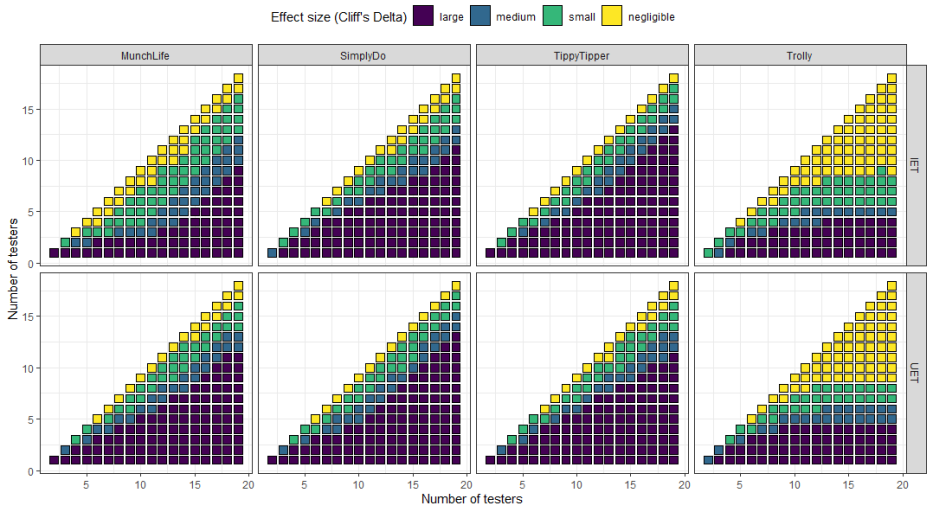


Figure 5.10. Measured effect size (Cliff's Delta): comparing the effectiveness of test suites generated by different-sized crowds using the same exploratory strategy.

in greater detail in Section 5.1.2.

In conclusion, the lack of knowledge about the application under test and the strict time limit represents the main causes for the variability of the coverage results achieved by different UET testers.

On the other hand, residual differences in terms of coverage achieved can be observed for IET testers, too. For example, some complex interactions in SimplyDo and TippyTipper have been found by a few IET testers. In addition, a specific undocumented and unintuitive feature of Trolly (i.e. the "offlist" list) represented an unsolvable coverage problem for about half of the IET testers and it is responsible for the few darker tiles observable in Figure 5.8 for Trolly and IET testers.

In general, it appears that the residual differences in coverage between different IET testers depend on their ability in producing effective test cases: although they received a common background, we observed some significant differences between their results can be observed.

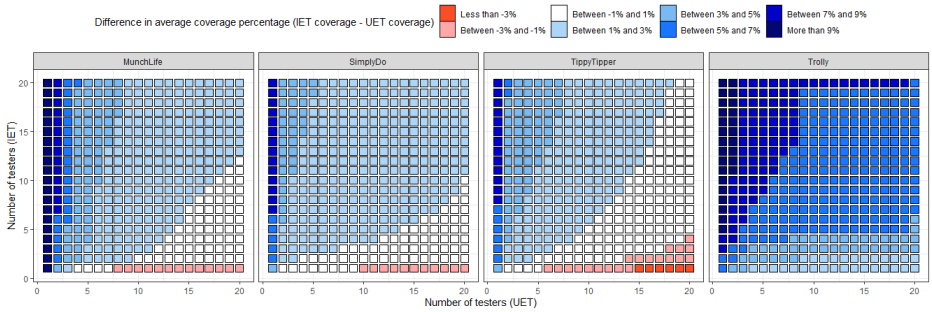


Figure 5.11. Difference in LOC coverage achieved by different numbers of testers using different testing strategies.

RQ2: Comparing UET and IET

In order to answer this research question, we investigate the differences in coverage achieved by different-sized crowds of testers using different exploratory strategies. In particular, for each of the AUTs and for each combination of the number of testers (k, k'), $k, k' \in [1, \ldots, 20]$, we computed the difference between the LOC coverage achieved on average by a crowd of k non-communicating testers using an IET strategy and the one achieved by k' testers using a UET strategy. Negative values correspond to cases in which UET testers achieved a higher coverage, whereas positive values, on the other hand, correspond to scenarios in which IET testers achieved a higher coverage. In Figure 5.11, we report the results of this analysis. The white areas in the figure correspond to cases in which the coverage achieved using UET and IET strategies is comparable (less than 1% difference in absolute value), red areas correspond to situations in which the coverage achieved by UET testers is greater than that achieved by IET testers, while blue areas correspond to situations in which IET testers achieved higher coverage. As Figure 5.11 highlights, in three of the four considered AUTs, namely MunchLife, SimplyDo, and TippyTipper, a noticeable equivalence area exists, and in some cases, the LOC coverage achieved by larger crowds of UET testers (i.e., generally more than 10 testers) is higher than that achieved by smaller crowds of IET testers (i.e., 1 to 3 testers). In Trolly, on the other hand, the LOC coverage achieved even by a single IET tester exceeded that of a crowd of 20 UET testers by

more than 1%, and therefore no equivalence area exists.

To determine whether these differences in LOC coverage are statistically significant, we performed statistical tests and measured the effect size following the same procedure described in the previous subsection. More in detail, for each AUT and for each pair of cardinalities (k, k'), with $k, k' \in [1, \ldots, 19]$, we tested the null hypotheses:

$\mathbf{H_{0-smaller}^{AUT,k,k'}}$: The LOC coverage achieved using a UET strategy on AUT by k testers is smaller than or equal to the one achieved by k' testers using an IET strategy.

$\mathbf{H_{0-greater}^{AUT,k,k'}}$: The LOC coverage achieved using a UET strategy on AUT by k testers is greater than or equal to the one achieved by k' testers using an IET strategy.

To test the null hypotheses, we used the Student's T test when both the sample distributions are normal, and the Mann–Whitney–Wilcoxon test [212] otherwise. When the test $p\text{-}value < 0.05$, we reject the null hypothesis with high confidence, accepting the alternative hypothesis that the LOC coverage achieved by k testers using UET is greater (resp., smaller) than that achieved by k' testers using IET. Statistical test results are reported with a tile plot in Figure 5.9. In the figure, blue tiles correspond to cases in which we could reject $\mathbf{H_{0-greater}^{AUT,k,k'}}$, thus accepting the alternative hypothesis that the LOC coverage achieved by k testers using UET is smaller than that achieved by k' testers using IET. Red tiles, on the other hand, correspond to cases in which we could reject $\mathbf{H_{0-smaller}^{AUT,k,k'}}$, thus accepting the alternative hypothesis that the LOC coverage achieved by k testers using UET is greater than that achieved by k' testers using IET. Lastly, white tiles correspond to cases in which we could not reject any null hypotheses, and thus we can draw no statistical conclusion. Results show that, in a preponderant number of cases, the coverage achieved by crowds of testers using UET is smaller, in a statistically significant way, than that achieved by crowds of IET testers.

In MunchLife and SimplyDo, larger crowds of UET testers manage to achieve a statistically higher LOC coverage than crowds of IET testers of sizes 1 and 2. In particular, in MunchLife, sets of 7 or more UET testers achieve a higher coverage than sets with one IET tester, and it takes 16 or

more UET testers to achieve a higher coverage than a pair of IET testers. In SimplyDo, it takes 4 or more UET testers to cover more LOCs than a single IET tester, while 14 or more UET testers are needed to achieve better coverage than a pair of IET testers. In both MunchLife and SimplyDo, even 20 UET testers cannot achieve a higher coverage than 3 or more IET testers. In TippyTipper, there is a bigger number of cases in which UET crowds achieve a higher coverage than IET crowds, as witnessed by the red area in the bottom-right corner of the tile diagram. Nonetheless, 9 or more IET testers achieve a higher or comparable coverage than any number of UET testers. Lastly, in Trolly, even a single IET tester achieves a higher coverage than any number of UET testers.

The reason behind the better results obtained by IET testers compared to large sets of UET testers for Trolly is related to the existence of the "hidden" functionality related to the management of "offlist" items, which can only be activated with long click events in a particular execution scenario: it was discovered almost exclusively by the IET testers who could observe the source code related to its implementation.

On the contrary, the better results obtained by large sets of UET testers compared to small sets composed of one or more IET testers observed for TippyTipper depend on the absence of code that was difficult to be observed and tested in this app. In these cases, UET testers were limited mainly by the strict time limit and their diversity allows them to obtain excellent results by considering the union of the coverage obtained by a large group of them.

Even when statistically significant, however, the differences in LOC coverage achieved by different-sized crowds of testers using different exploratory strategies might be negligible. Hence, we measured the magnitude of these differences using the *Cliff's delta* effect size, as discussed in Section 5.2.2. Results of this analysis are shown in Figure 5.13, and highlight that in three of the considered AUTs, namely MunchLife, SimplyDo, and TippyTipper, there exists an "*equivalence*" zone in which different numbers of UET and IET testers achieve comparable coverage (i.e., the effect size is small or negligible). In Trolly, on the other hand, the coverage achieved by IET testers is always greater than that achieved by any number of UET testers, with a large or medium effect size.

The performance of both IET and UET crowds and the extent of such

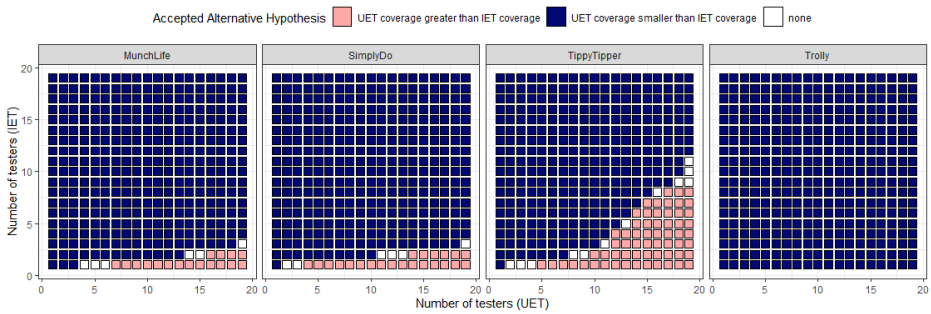


Figure 5.12. Results of alternative hypotheses testing: comparing the effectiveness of test suites generated by different-sized crowds using different exploratory strategies.

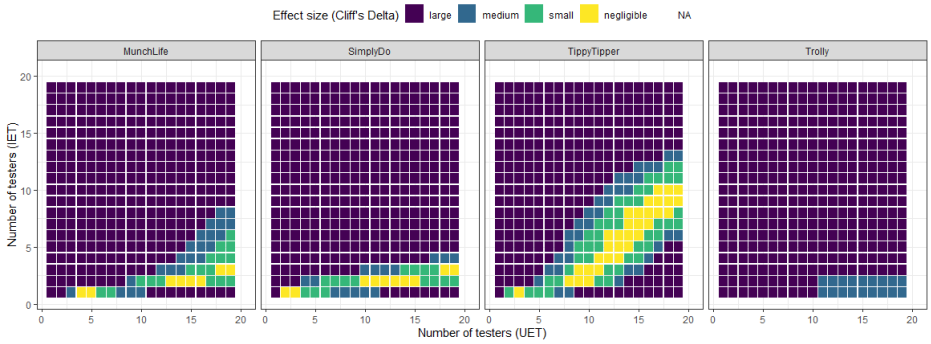


Figure 5.13. Measured effect size (Cliff's Delta): comparing the effectiveness of test suites generated by different-sized crowds using different exploratory strategies.

equivalence areas are significantly influenced by the nature of the AUT. In AUTs in which some features are unintuitive, *Informed* approaches prove to be significantly more effective, as testers using an *Uninformed* strategy often fail to stress these '*hidden*' features. In AUTs featuring a larger number of functionalities, on the other hand, the coverage achieved by each tester appears to be mainly limited by the given time budget. In these cases, leveraging larger numbers of testers using an *Uninformed* approach leads to overall better coverage.

> **RQ2:** *Results highlight that small crowds of IET testers generally achieve similar results w.r.t. larger crowds of UET testers. In apps with unintuitive features, even a single IET tester can achieve better results than a large crowd of UET testers.*

RQ3: Cost-effectiveness of UET, IET, and mixed strategies

Generating tests with an *Informed* exploratory strategy requires computing and analysing code coverage reports, and is, therefore, more time-consuming than using a simpler *Uninformed* exploratory approach. The goal of this research question is to investigate which exploratory strategy (or mix thereof) is the most cost-effective. As discussed in Section 5.2.1, we assigned a one-hour effort to *Uninformed* testing tasks and a three-hour effort to *Informed* ones.

In Table 5.10, we report the average LOC coverage percentage achieved by different-sized crowds of testers using different strategies, with the same overall man-hours effort. In particular, we consider: (I) crowds using an *Uninformed* approach; (II) crowds using an *Informed* approach; and (III) crowds in which some testers use an IET strategy, while some others use a UET approach (which we refer to as *Mixed*). For each AUT and for each overall man-hours budget, we highlight in bold the maximum LOC coverage achieved by any of the considered strategies. For the sake of compactness, we report in Table 5.10 only a subset of the overall man-hours budgets for which an IET crowd exists. Complete data are available in the replication package [35].

Results show that, under the same effort budget, the coverage achieved by IET crowds is always greater than or comparable to the one achieved by UET crowds. In some cases, the difference in coverage between IET

Table 5.10. LOC coverage percentage achieved using different strategies, with the same overall man-hours effort.

Effort	Strat.	Num. IET	Num. UET	MunchLife	SimplyDo	TippyTipper	Trolly
3	UET	0	3	91,7	**82,9**	**87,6**	80,0
	IET	1	0	**93,4**	82,8	87,3	**83,5**
6	UET	0	6	93,9	83,3	88,5	81,0
	IET	2	0	**95,9**	**84,0**	**89,2**	**84,3**
	Mixed	1	3	95,6	83,7	88,9	83,0
9	UET	0	9	94,8	83,7	89,2	81,3
	IET	3	0	**96,3**	**84,6**	89,7	**85,4**
	Mixed	1	6	95,9	84,0	**89,8**	83,1
	Mixed	2	3	96,2	84,3	89,6	85,6
12	UET	0	12	95,4	84,0	89,8	81,4
	IET	4	0	**96,5**	**85,0**	**90,1**	**86,1**
	Mixed	1	9	96,1	84,3	89,9	83,2
	Mixed	2	6	96,3	84,5	90,4	85,1
	Mixed	3	3	96,2	84,1	90,0	84,7
15	UET	0	15	95,8	84,4	90,4	81,4
	IET	5	0	96,6	**85,2**	90,4	**86,7**
	Mixed	1	12	96,2	84,6	90,4	83,2
	Mixed	2	9	96,1	84,5	90,4	84,4
	Mixed	3	6	**96,8**	85,1	**90,6**	86,1
	Mixed	4	3	96,3	**85,2**	90,4	85,4
18	UET	0	18	96,1	84,7	**90,9**	81,4
	IET	6	0	96,7	**85,4**	90,6	**87,1**
	Mixed	1	15	96,2	84,8	90,8	83,2
	Mixed	2	12	96,4	85,0	90,7	84,0
	Mixed	3	9	96,4	85,1	90,6	85,4
	Mixed	4	6	96,6	85,3	90,7	85,2
	Mixed	5	3	**96,8**	**85,4**	90,6	86,4

and UET crowds is remarkable. In MunchLife, for example, with a 6 hours overall effort, 2 IET testers achieve on average 2% more coverage than 6 UET testers. Similarly, in Trolly, with an 18 hours budget, 6 IET testers achieve 5,7% more coverage than 18 UET testers. In most cases, however, the difference is less noticeable (1% or less). As for mixed crowds, we can observe that, for small budgets (e.g. 6, 9 or 12 man-hours), the coverage percentage achieved with mixed approaches has intermediate values w.r.t. UET and IET testers. In these cases, the effectiveness of the resulting mixed test suites appears to be generally greater than that of UET crowds, and comparable (less than 0.5% difference in LOC coverage) to that of IET crowds. On the contrary, as the budget increases to 15 or 18 man-hours, there are frequent cases in which combinations including both UET testers and IET testers provide the best coverage results. For example, with 15 hours of overall effort, a mixed crowd of 3 IET testers and 6 UET testers achieves higher LOC coverage than an IET crowd of 5 testers on both MunchLife and TippyTipper. In both cases, however, the difference in LOC coverage is rather small (0.2%).

These results support the insight of the existence of a trade-off between the advantages of the IET approach (which provides effective test suites even with a few testers) and those of the UET approach (which improves its effectiveness as the number of tester increase, due to the greater diversity among the results of individual testers). This trade-off could be optimized by a mixed approach involving both UET and IET testers.

RQ3: *Results highlight that, in the considered setting, an Informed exploratory strategy is generally more cost-effective than an Uninformed approach or combinations of both approaches. Mixed crowds appear to be generally more cost-effective than UET ones and can represent a valid trade-off solution when larger budgets are available.*

5.3 Threats to Validity

In this section, we discuss some threats that could have affected the results of our empirical study and their generalizability, according to the guidelines proposed in Wohlin et al. [213].

Threats to Internal Validity.

To ensure a fair comparison between the effectiveness of Uninformed Exploratory Testing (UET) tasks and that of Informed Exploratory Testing tasks, we enforced the same experimental conditions for all the experiments. To this aim, each student act on freshly-installed apps, without any data from previous executions, in the context of an instance of the same Android emulator with the same Java Virtual Machine. In addition, we provided students with a pre-installed instance of the Eclipse IDE, integrated with Robotium Recorder and Android support, and with projects related to the applications under test. In this way, we mitigated the threat related to the waste of testing time due to environment setup activities that could differ between different students. All the students had a basic knowledge of that environment, which was presented in previous lectures of the same course. Furthermore, we selected apps with no dependencies on external resources such as remote data sources or services, so that their behaviour was not influenced by external factors. We did not control the order of analysis of the apps, having required that the students freely organized and accomplished their tasks. Of course, it is not possible to exclude the threat of a learning effect, in case all the students analysed the apps exactly in the same order. In addition, we were not able to reconstruct the time allocated by any student to the testing of each specific application because we asked only to provide the total time spent on the testing of all the apps, without tracing the ending time for each application testing.

Threats to External Validity.

These threats limit the generalizability of the results and are often posed by the way experimental objects and subjects are selected. In this study, we considered four simple open-source Android apps. We are aware that the selected apps cannot be considered representative of complex commercial applications. This represents a threat to the possibility of generalising our findings to every Android app. Additional experiments involving industrial-strength apps should be carried out in order to mitigate this threat.

As for the C&R tool, in our experiments, we used the Robotium Recorder tool. Currently, Android Espresso Test Recorder, a free tool

included in the Android Studio suite, has surpassed Robotium Recorder in popularity. Anyhow, the features of the two tools are quite similar. We performed other experiments with different graduate students using Android Espresso Test Recorder in a UET scenario on the same AUTs, and the results showed that there are no significant differences in the effectiveness of the recorded test cases with respect to the ones recorded with Robotium.

Concerning the selected subjects, the involvement of students as participants may affect the generalization of results [42, 53]. Anyway, different recent studies state that it is reasonable to recruit crowdworkers from the researchers' organization [138, 82]. The 20 students involved in this experiment had basic experience in Android development and GUI testing, and several studies [93, 49, 176] observed that graduate students and practitioners perform similarly when applying a new technology during experimentation. Thus, we suggest that our results could be generalizable to crowdtesting scenarios in which C&R testers with limited testing skills are recruited [136].

5.4 Summary and Future Works

E2E testing of mobile apps is a challenging activity, due to the time-to-market pressure of a very competitive scenario and to the fragmentation of mobile systems and devices, requiring repeated tests on multiple different devices. Many solutions have been proposed in the literature to support practitioners, like C&R with ET or AIG tools. From a practical perspective, a Software Project Manager might be puzzled in deciding whether to use AIG tools or ET approaches, for a specific app under development. Moreover, this decision-making scenario is further complicated by the diffusion, in recent years, of crowdtesting, which allows companies to conveniently recruit human testers on an on-demand basis and relatively inexpensively, thus making ET approaches more appealing.

Nevertheless, to the best of our knowledge, little work aimed at supporting these managerial choices has been conducted in the literature, even though the choice can be of paramount importance for a Software Project Manager.

In this chapter, we conducted two empirical studies aimed at addressing

this lack in the literature and providing useful insights to software project managers, based on which they could make more informed decisions.

In the first study, we measured, using well-known metrics such as LOC and Branch coverage, the testing effectiveness of unskilled practitioners using a C&R tool, on four apps on which they had no information at all, under strict temporal constraints. In this scenario, which we named Uninformed Exploratory Testing (UET), human testers obtained testing performances just slightly better than AIG tools. In a second experiment, we gave the same testers more time and information about the AUTs, i.e., source code and coverage reports, and asked them to record more tests to improve the previously-achieved coverage. In this scenario, which we called Informed Exploratory Testing (IET), we found that human testers generally outperformed the considered AIG tools, even when AIG tools were executed for longer periods of time.

Moreover, we conducted a detailed qualitative analysis of the lack of coverage obtained by the testers and the AIG tools. This analysis allowed us to investigate the limitations of both approaches. In particular, we found that the main factors hindering the effectiveness of the human testers are the lack of time and the lack of information on the AUT. As for the AIG tools, their effectiveness was limited by both technological and methodological issues. The technological issues are related to unsupported GUI events/widgets and could be easily solved by technical improvements of the tools, which should be adapted to the continuous evolution of the Android framework. The methodological issues, on the other hand, are related to the ineffectiveness in testing functionalities that are reachable only through complex interactions and represent a well-known intrinsic limitation of AIG testing techniques [13].

In the second study, we elaborated on the results of the first study to investigate the effectiveness of exploratory strategies and capture and replay in crowdtesting scenarios, based on the assumption that the 20 involved master's students can be considered representatives of practitioners taking part in crowdtesting activities. To this end, we computed, for every possible distinct subset of subjects, the aggregate code coverage that a given crowd would have achieved in a crowdtesting scenario.

The analysis of the experimental results provided some interesting conclusions and insights. First of all, we confirmed the effectiveness of the

crowdtesting paradigm even in scenarios in which the goal is to produce re-executable test suites with C&R, and not only to identify bugs. Furthermore, we have observed that increasing the size of the crowd of testers by small amounts generally leads to small or negligible improvements in coverage, especially for larger crowds. The comparisons between the coverage results achieved by crowds of UET and IET testers showed that small crowds of IET testers generally achieve similar results with respect to larger crowds of UET testers, but the test suites produces by IET testers are superior with respect to the coverage of complex or unpredictable functionalities. The comparisons between the results of different crowds having the same overall testing effort (in terms of man-hours) and composed of UET testers, IET testers or a mix of them, provided some insights about the optimal composition of crowds. In the context of our experimental setting, IET crowds were generally more cost-effective than UET crowds or mixed crowds, whereas mixed crowds generally outperform UET crowds, and appear as a viable alternative when larger budgets are available.

In future work, we plan to replicate our experiments in an industrial context, involving industrial-strength apps, to evaluate the effect of the testers' skill level on the effectiveness of the generated test suites. Moreover, we also plan to replicate the study on crowdtesting involving larger crowds of real crowdtesters in place of students, to improve the generalizability of the results, and include real crowdtesting costs in our analyses.

Furthermore, on a broader horizon, similar studies could also be carried out in the context of web applications, to investigate whether the same findings apply to that domain as well.

Chapter 6

Conclusions

End-to-End (E2E) testing is a widely-used approach to improve the quality of web and mobile applications. In this kind of testing activity, the Application Under Test (AUT) is tested as a whole, in its entirety, simulating real-world usage scenarios. The research presented in this thesis aims at improving the effectiveness of E2E testing processes from multiple perspectives.

In the domain of GUI-level testing of web applications, research presented in Chapter 3 of this thesis work aims at improving the effectiveness of automatic GUI-level test generation techniques by tackling the problem of near-duplicate web pages in automatic model inference, which is a prerequisite for the application of many automatic test generation techniques for web apps. Indeed, most automatic GUI-level test generation techniques for web applications rely on state-based models of the AUT, which are typically automatically inferred via crawling. In such models, states represent high-level functionalities of the AUT, while transitions represent navigability relationships between these functionalities. Automatically-inferred models, however, are affected by near-duplicate states, i.e., states corresponding to slightly different web pages that nonetheless represent the same functionality from a testing point of view. Near-duplicate states have a negative impact on the quality of automatically-inferred models, and thus on the quality of the test suites which are automatically generated from them [217]. To this end, two novel near-duplicate detection techniques are proposed, based on a common underlying framework. The

effectiveness of the proposed techniques and their impact on the quality of automatically-generated test suites is assessed in an empirical study against state-of-the-art baselines. Results show that the proposed techniques outperform state-of-the-art approaches for near-duplicate detection of web pages, and their usage can lead to remarkable improvements in the quality of the resulting test suites. Future work in this domain could be aimed at further improving the effectiveness of the proposed techniques, for example by using visual embeddings on web screenshots, or by using more advanced embedding models, capable of taking into account also the intrinsic structure of web pages, such as the one proposed by Alon et al. [8]. In a broader context, the research on near-duplicate detection carried out in this thesis could be applied in different domains than automatic model inference/test generation. For example, in web test prioritization, selection, and/or minimization [185], similar techniques could be used to define more advanced model-based coverage measures.

In the domain of performance testing of web applications, the research described in Chapter 4 faces the problem of workload generation, i.e., specifying which sequence of web requests should be sent to the AUT to simulate a given load level. In a collaboration with an industrial partner, we found that existing solutions and tools for the automatic generation of workloads for web applications are affected by limitations hindering the productivity of practitioners and the overall effectiveness of the performance testing process. To overcome these limitations, we presented a novel technique to support their automatic generation of workloads leveraging existing E2E GUI-level web tests. The effectiveness of the proposed technique is then evaluated in a preliminary industrial case study, with promising results. In future works, we plan to further validate our proposal by conducting an extensive empirical evaluation and comparing it against state-of-the-art workload generation approaches presented in the literature and on a broader set of subject systems, possibly including also open-source ones, to improve the generalizability of the results.

Lastly, in the domain of GUI-level testing of mobile applications, the research activities described in Chapter 5 aim at investigating the effectiveness of different GUI-level testing approaches for mobile apps, to support Software Project Managers in deciding which techniques to use to test a given mobile application. To this end, two empirical studies are con-

ducted, comparing among themselves Automated Input Generation (AIG) tools, practitioners using Capture & Replay (C&R) tools and different exploratory testing strategies, and crowdtesting scenarios involving practitioners using C&R tools. Experimental results provided some interesting insights that could prove useful to both software project managers deciding which testing strategies to adopt for their mobile app projects, and to researchers and practitioners working on novel AIG tools. Future works in this domain will aim at replicating the experiments in an industrial context, involving industrial-strength apps, to also evaluate the effect of the testers' skill level on the effectiveness of the generated test suites. Moreover, we also plan to replicate the study on crowdtesting involving larger crowds of real crowdtesters in place of students, to improve the generalizability of the results, and include real crowdtesting costs in our analyses. Furthermore, on a broader horizon, similar studies could also be carried out in the context of web applications, to investigate whether the same findings apply to that domain as well.

Bibliography

[1] Software Freedom Conservancy (SFC). *Selenium IDE - Open source record and playback test automation for the web*. URL: https://www.selenium.dev/selenium-ide/.

[2] Rabiya Abbas, Zainab Sultan, and Shahid Nazir Bhatti. "Comparative analysis of automated load testing tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege". In: *2017 International Conference on Communication Technologies (ComTech)*. 2017, pp. 39–44. DOI: 10.1109/COMTECH.2017.8065747.

[3] Alain Abran, James W Moore, Pierre Bourque, Robert Dupuis, and L Tripp. "Software engineering body of knowledge". In: *IEEE Computer Society, Angela Burgess* (2004), p. 25.

[4] Admin. *Open Source Load Testing*. Oct. 2022. URL: https://gatling.io/open-source/.

[5] Sadia Afroz and Rachel Greenstadt. "Phishzoo: Detecting phishing websites by looking at them". In: *2011 IEEE fifth international conference on semantic computing*. IEEE. 2011, pp. 368–375.

[6] Emil Alegroth, Michel Nass, and Helena H Olsson. "JAutomate: A tool for system-and acceptance-test automation". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE. 2013, pp. 439–446.

[7] Mario Almeida, Muhammad Bilal, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Matteo Varvello, and Jeremy Blackburn. "CHIMP: Crowdsourcing Human Inputs for Mobile Phones". In:

Proceedings of the 2018 World Wide Web Conference. WWW '18. WWW. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 45–54. ISBN: 9781450356398. DOI: 10.1145/3178876.3186035. URL: https://doi.org/10.1145/3178876.3186035.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "code2vec: Learning distributed representations of code". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[9] Karen M Alsante, Linda Martin, and Steven W Baertschi. "A stress testing benchmarking study". In: *Pharmaceutical technology* 27.2 (2003), pp. 60–73.

[10] Francesco Altiero, Giovanni Colella, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Change-Aware Regression Test Prioritization using Genetic Algorithm". In: *Proceedings of the 48th Euromicro Conference on Software Engineering and Advanced Applications*. To appear in the proceedings. IEEE. 2022.

[12] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato. "A toolset for GUI testing of Android applications". In: cited By 16. 2012, pp. 650–653. DOI: 10.1109/ICSM.2012.6405345.

[13] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana, and Anna Rita Fasolino. "A general framework for comparing automatic testing techniques of Android mobile apps". In: *Journal of Systems and Software* 125 (2017), pp. 322–343. DOI: 10.1016/j.jss.2016.12.017. URL: http://dx.doi.org/10.1016/j.jss.2016.12.017.

[14] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. "Using GUI ripping for automated testing of Android applications". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2012, pp. 258–261.

[15] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. "Why does the orientation change mess up my Android application? From GUI failures to code faults". In: *Software Testing, Verification and Reliability* 28.1 (2018). e1654 stvr.1654, e1654. DOI: 10.1002/stvr.1654. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1654. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1654.

[16] R. Anbunathan and A. Basu. "A recursive crawler algorithm to detect crash in Android application". In: cited By 0. 2015, pp. 256–267. DOI: 10.1109/ICCIC.2014.7238518.

[17] C. Andersson, T. Thelin, P. Runeson, and N. Dzamashvili. "An experimental evaluation of inspection and testing for detection of design faults". In: *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.* 2003, pp. 174–184.

[18] Anneliese A Andrews, Jeff Offutt, and Roger T Alexander. "Testing web applications by modeling with FSMs". In: *Software & Systems Modeling* 4.3 (2005), pp. 326–345.

[19] Mauricio Aniche, Christoph Treude, and Andy Zaidman. "How Developers Engineer Test Cases: An Observational Study". In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. DOI: 10.1109/TSE.2021.3129889.

[20] *Apache JMeter*. URL: https://jmeter.apache.org/.

[21] *Angular version of the Spring PetClinic web application*. https://github.com/spring-petclinic/spring-petclinic-angular. 2018.

[22] *Simple, web-based address & phone book*. http://sourceforge.net/projects/php-addressbook. Accessed: 2018-10-01. 2015.

[23] *Claroline. Open Source Learning Management System*. https://sourceforge.net/projects/claroline/. 2015.

[24] *DimeShift: easiest way to track your expenses*. https://github.com/jeka-kiselyov/dimeshift. 2018.

[25] *Pagekit: modular and lightweight CMS*. https://github.com/pagekit/pagekit. 2018.

[26] *Phoenix: Trello tribute done in Elixir, Phoenix Framework, React and Redux.* https://github.com/bigardone/phoenix-trello. 2018.

[27] *PHP Password Manager.* https://github.com/pklink/ppma. 2018.

[28] *Meeting Room Booking System.* https://mrbs.sourceforge.io/. 2018.

[29] *Mantis Bug Tracker.* https://github.com/mantisbt/mantisbt. 2018.

[30] *Appetizer replaykit – GitHub repository.* URL: https://github.com/appetizerio/replaykit (visited on 05/09/2019).

[31] Applause. *Customer Stories.* https://www.applause.com/customers. Seen on Oct. 22, 2022. URL: https://www.applause.com/customers.

[32] Luca Ardito, Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. "Espresso vs. EyeAutomate: An Experiment for the Comparison of Two Generations of Android GUI Testing". In: *Proceedings of the Evaluation and Assessment on Software Engineering.* ACM. 2019, pp. 13–22.

[33] Tanzirul Azim and Iulian Neamtiu. "Targeted and Depth-first Exploration for Systematic Testing of Android Apps". In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 641–660. ISSN: 0362-1340. DOI: 10.1145/2544173.2509549.

[34] V. R. Basili and R. W. Selby. "Comparing the Effectiveness of Software Testing Strategies". In: *IEEE Transactions on Software Engineering* SE-13.12 (1987), pp. 1278–1296.

[35] Ermanno Battista, Sergio Di Martino, Sergio Di Meglio, Fabio Scippacercola, and Luigi Libero Lucio Starace. *E2E-Loader: A Framework to Support Performance Testing of Web Applications.* Version 1.0.0. Nov. 2022. DOI: 10.5281/zenodo.7286734. URL: https://doi.org/10.5281/zenodo.7286734.

[36] Matteo Biagiola, Filippo Ricca, and Paolo Tonella. "Search based path and input data generation for web application testing". In: *International Symposium on Search Based Software Engineering*. Springer. 2017, pp. 18–32.

[37] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. "Web Test Dependency Detection". In: *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. ACM, 2019, 12 pages.

[38] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. "Dependency-aware web test generation". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 175–185.

[39] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. "Diversity-based Web Test Generation". In: *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. ACM, 2019.

[40] Mustafa Bozkurt and Mark Harman. "Automatically generating realistic test input from web services". In: *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*. IEEE. 2011, pp. 13–24.

[41] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. "Syntactic clustering of the web". In: *Computer networks and ISDN systems* 29.8-13 (1997), pp. 1157–1166.

[42] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. "Issues in using students in empirical studies in software engineering education". In: *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*. IEEE. Sept. 2003, pp. 239–249. DOI: 10.1109/METRIC.2003.1232471.

[43] Hari Sankar Chaini and Sateesh Kumar Pradhan. "Test script execution and effective result analysis in hybrid test automation framework". In: *2015 International Conference on Advances in Computer Engineering and Applications*. IEEE. 2015, pp. 214–217.

[44] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. "GUI testing using computer vision". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2010, pp. 1535–1544.

[45] Moses S Charikar. "Similarity estimation techniques from rounding algorithms". In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 2002, pp. 380–388.

[46] Jinfu Chen, Weiyi Shang, Ahmed E Hassan, Yong Wang, and Jiangbin Lin. "An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 669–681.

[47] Teh-Chung Chen, Scott Dick, and James Miller. "Detecting visually similar web pages: Application to phishing detection". In: *ACM Transactions on Internet Technology (TOIT)* 10.2 (2010), pp. 1–38.

[48] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. "Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 2017, pp. 243–252. DOI: 10.1109/ICSE-SEIP.2017.26.

[49] Zhenyu Chen and Bin Luo. "Quasi-Crowdsourcing Testing for Educational Projects". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. IEEE/ACM. Hyderabad, India: Association for Computing Machinery, 2014, pp. 272–275. ISBN: 9781450327688. DOI: 10.1145/2591062.2591153. URL: https://doi.org/10.1145/2591062.2591153.

[50] Wontae Choi, George Necula, and Koushik Sen. "Guided GUI testing of android apps with minimal restart and approximate learning". In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. ACM. 2013, pp. 623–640.

[51] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. "Automated test input generation for android: Are we there yet?(e)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 429–440.

[52] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. "An Empirical Study on the Usage of Transformer Models for Code Completion". In: *CoRR* abs/2108.01585 (2021). arXiv: 2108.01585.

[53] M. Ciolkowski, D. Muthig, and J. Rech. "Using academic courses for empirical validation of software development processes". In: *Proceedings. 30th Euromicro Conference, 2004.* Euromicro. Sept. 2004, pp. 354–361. DOI: 10.1109/EURMIC.2004.1333390.

[54] James Clark, Steve DeRose, et al. *XML path language (XPath)*. 1999.

[55] Norman Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[56] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Hillsdale, N.J: L. Erlbaum Associates, 1988. ISBN: 978-1-134-74270-7.

[57] *commoncrawl.org*. https://commoncrawl.org/the-data/get-started/.

[58] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. "Mobile GUI Testing Fragility: A Study on Open-Source Android Applications". In: *IEEE Transactions on Reliability* 68.1 (2019), pp. 67–90.

[59] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. "Scripted gui testing of android apps: A study on diffusion, evolution and fragility". In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM. 2017, pp. 22–32.

[60] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. "A tree kernel based approach for clone detection". In: *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–5.

[61] Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Web Application Testing: Using Tree Kernels to Detect Near-duplicate States in Automated Model Inference". In: *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2021, pp. 1–6.

[62] *Create UI tests with Espresso Test Recorder*. https://developer.android.com/studio/test/espresso-test-recorder. Seen on Oct. 22, 2022. URL: https://developer.android.com/studio/test/espresso-test-recorder.

[63] Arghavan Moradi Dakhel, Michel C. Desmarais, and Foutse Khomh. *Dev2vec: Representing Domain Expertise of Developers in an Embedding Space*. 2022. DOI: 10.48550/ARXIV.2207.05132.

[64] data.ai. *State of Mobile 2022*. https://www.data.ai/en/go/state-of-mobile-2022/. Last accessed: Oct. 25, 2022. URL: https://www.data.ai/en/go/state-of-mobile-2022/.

[65] Giuseppe Antonio Di Lucca, Massimiliano Di Penta, Anna Rita Fasolino, and Pasquale Granato. "Clone analysis in the web era: An approach to identify cloned web pages". In: *Seventh Workshop on Empirical Studies of Software Maintenance*. 2001, p. 107.

[66] Sergio Di Martino, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. "Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing". In: *Software Testing, Verification and Reliability* 31.3 (2021), e1754.

[67] Eelco Dolstra, Raynor Vliegendhart, and Johan Pouwelse. "Crowdsourcing GUI Tests". In: *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. ICST '13. IEEE. USA: IEEE Computer Society, 2013, pp. 332–341. ISBN: 9780769549682. DOI: 10.1109/ICST.2013.44. URL: https://doi.org/10.1109/ICST.2013.44.

[68] Kit Eaton. *How One Second Could Cost Amazon $1.6 Billion In Sales*. Last accessed: Oct. 24, 2022. 2012. URL: https://web.archive.org/web/20221006004855/https://www.fastcompany.com/1

825005/how-one-second-could-cost-amazon-16-billion-sales.

[69] *Emma Reference Manual*. https://emma.sourceforge.net/userguide_single/userguide.html. Seen on Oct. 22, 2022. URL: https://emma.sourceforge.net/userguide_single/userguide.html.

[70] *EMMA: a free Java code coverage tool*. Seen on Oct. 22, 2022. URL: http://emma.sourceforge.net/.

[71] *Espresso*. Seen on Oct. 22, 2022. URL: https://developer.android.com/training/testing/espresso.

[72] *Eyestudio*. URL: http://www.eyeautomate.com/eyestudio.html (visited on 05/09/2019).

[73] *F-Droid - Free and Open Source Android App Repository*. https://f-droid.org. Seen on Oct. 22, 2022. URL: https://f-droid.org.

[74] Amin Milani Fard and Ali Mesbah. "Feedback-directed exploration of web applications to derive test models." In: *ISSRE*. Vol. 13. 2013, pp. 278–287.

[75] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso. "Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2017, pp. 149–160. DOI: 10.1109/ICST.2017.21.

[76] Jian Feng, Lianyang Zou, Ou Ye, and Jingzhou Han. "Web2Vec: Phishing Webpage Detection Method Based on Multidimensional Features Driven by Deep Learning". In: *IEEE Access* 8 (2020), pp. 221214–221224. DOI: 10.1109/ACCESS.2020.3043188.

[77] Ian Fette and Alexey Melnikov. "The WebSocket Protocol". In: *RFC* 6455 (2011), pp. 1–71. URL: https://www.rfc-editor.org/rfc/rfc6455.

[78] Dennis Fetterly, Mark Manasse, and Marc Najork. "On the evolution of clusters of near-duplicate web pages". In: *Proceedings of the IEEE/LEOS 3rd International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices (IEEE Cat. No. 03EX726)*. IEEE. 2003, pp. 37–45.

[79] *Firebase Test Lab Robo Test*. Seen on Oct. 22, 2022. URL: https://firebase.google.com/docs/test-lab/android/robo-ux-test.

[80] Abhishek Gangwar, Eduardo Fidalgo, Enrique Alegre, and Vıctor González-Castro. "PhishFingerprint: A Practical Approach for Phishing Web Page Identity Retrieval Based on Visual Cues". In: *International Conference of Applications of Intelligent Systems*. 2018.

[81] Vahid Garousi, Wasif Afzal, Adem Cauglar, Ihsan Berk Icsik, Berker Baydan, Seckin Caylak, Ahmet Zeki Boyraz, Burak Yolacan, and Kadir Herkiloglu. "Comparing Automated Visual GUI Testing Tools: An Industrial Case Study". In: *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*. A-TEST 2017. Paderborn, Germany: ACM, 2017, pp. 21–28. ISBN: 978-1-4503-5155-3. DOI: 10.1145/3121245.3121250. URL: http://doi.acm.org/10.1145/3121245.3121250.

[82] Xiuting Ge, Shengcheng Yu, Chunrong Fang, Qi Zhu, and Zhihong Zhao. "Leveraging Android Automated Testing to Assist Crowdsourced Testing". In: *IEEE Transactions on Software Engineering* (2022), pp. 1–18. DOI: 10.1109/TSE.2022.3216879.

[83] Wael H Gomaa, Aly A Fahmy, et al. "A survey of text similarity approaches". In: *international journal of Computer Applications* 68.13 (2013), pp. 13–18.

[84] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. "Reran: Timing-and touch-sensitive record and replay for android". In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 72–81.

[85] S Gunasekaran and V Bargavi. "Survey on automation testing tools for mobile applications". In: *International Journal of Advanced Engineering Research and Science* 2.11 (2015), pp. 2349–6495.

[86] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2015, pp. 215–224.

[87] Maher Hayek, Peter Farhat, Youssef Yamout, Charbel Ghorra, and Ramzi A. Haraty. "Web 2.0 Testing Tools: A Compendium". In: *2019 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*. 2019, pp. 1–6. DOI: 10.1109/3ICT.2019.8910274.

[88] Gilber van der Heiden and Susan Matson. *Market Guide for Application Crowdtesting Services*. Ed. by Gartner Research. https://www.gartner.com/en/documents/3890079. URL: https://www.gartner.com/en/documents/3890079.

[89] Monika Henzinger. "Finding near-duplicate web pages: a large-scale evaluation of algorithms". In: *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. 2006, pp. 284–291.

[90] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. "CC2Vec: Distributed Representations of Code Changes". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: ACM, 2020, pp. 518–529. ISBN: 9781450371216. DOI: 10.1145/3377811.3380361.

[91] André van Hoorn, Matthias Rohr, and Wilhelm Hasselbring. "Generating Probabilistic and Intensity-Varying Workload for Web-Based Software Systems". In: *Performance Evaluation: Metrics, Models and Benchmarks*. Ed. by Samuel Kounev, Ian Gorton, and Kai Sachs. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 124–143. ISBN: 978-3-540-69814-2.

[92] Mahmood Hosseini, Keith Phalp, Jacqui Taylor, and Raian Ali. "The four pillars of crowdsourcing: A reference model". In: *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)*. IEEE. 2014, pp. 1–12.

[93] Martin Host, Bjorn Regnell, and Claes Wohlin. "Using students as subjects a comparative study of students and professionals in lead-time impact assessment". In: *Empirical Software Engineering* 5.3 (2000), pp. 201–214.

[94] J. Howe. "The rise of crowdsourcing". In: *Wired Magazine* 14.6 (2006), pp. 1–4.

[95] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. "Versatile yet lightweight record-and-replay for android". In: *ACM SIGPLAN Notices* 50.10 (2015), pp. 349–366.

[96] Apple Inc. *XCTest Framework - Create and run unit tests, performance tests, and UI tests for your Xcode project.* URL: https://developer.apple.com/documentation/xctest.

[97] Taichi Ishikawa, Yu-Lu Liu, David Lawrence Shepard, and Kilho Shin. "Machine learning for tree structures in fake site detection". In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 2020, pp. 1–10.

[98] J. Itkonen, M. V. Mäntylä, and C. Lassenius. "The Role of the Tester's Knowledge in Exploratory Software Testing". In: *IEEE Transactions on Software Engineering* 39.5 (May 2013), pp. 707–724. ISSN: 2326-3881. DOI: 10.1109/TSE.2012.55.

[99] J. Itkonen and M.V. Mäntylä. "Are test cases needed? Replicated comparison between exploratory and test-case-based software testing". In: *Empirical Software Engineering* 19.2 (2014). cited By 25, pp. 303–342. DOI: 10.1007/s10664-013-9266-8.

[100] Juha Itkonen and Kristian Rautiainen. "Exploratory testing: a multiple case study". In: *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE. 2005, 10–pp.

[101] Emad Jabbar, Soheila Zangeneh, Hadi Hemmati, and Robert Feldt. *Test2Vec: An Execution Trace Embedding for Test Case Prioritization.* 2022. DOI: 10.48550/ARXIV.2206.15428.

[102] Manisha Jailia, Manisha Agarwal, and Ashok Kumar. "Comparative Study of N-Tier and Cloud-Based Web Application Using Automated Load Testing Tool". In: *Information and Communication Technology*. Springer, 2018, pp. 239–250.

[103] *JavaScript End to End Testing Framework | cypress.io testing tools.* URL: https://www.cypress.io/.

[104] Zhen Ming Jiang and Ahmed E. Hassan. "A Survey on Load Testing of Large-Scale Software Systems". In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1091–1118. DOI: 10.1109/TSE.2015.2445340.

[105] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. "Real challenges in mobile app development". In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2013, pp. 15–24.

[106] Z.U. Kamangar, U.A. Kamangar, Q. Ali, I. Farah, S. Nizamani, and T.H. Ali. "To enhance effectiveness of crowdsource software testing by applying personality types". In: ACM. Association for Computing Machinery, 2019, pp. 15–19. DOI: 10.1145/3328833.3328838.

[107] Vigdis By Kampenes, Tore Dybå, Jo E Hannay, and Dag IK Sjøberg. "A systematic review of effect size in software engineering experiments". In: *Information and Software Technology* 49.11-12 (2007), pp. 1073–1086.

[108] Erik Kamsties and Christopher M. Lott. "An Empirical Evaluation of Three Defect-Detection Techniques". In: *Proceedings of the 5th European Software Engineering Conference*. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 362–383. ISBN: 3540604065.

[109] Cem Kaner. *A Tutorial in Exploratory Testing*. https://www.kaner.com/pdfs/QAIExploring.pdf. 2008. URL: https://www.kaner.com/pdfs/QAIExploring.pdf.

[110] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. "What do mobile app users complain about?" In: *IEEE Software* 32.3 (2015), pp. 70–77.

[111] Hiroyuki Kirinuki and Haruto Tanno. "Automating End-to-End Web Testing via Manual Testing". In: *Journal of Information Processing* 30 (2022), pp. 294–306.

[112] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. "Understanding the test automation culture of app developers". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10.

[113] Steven Komarov, Katharina Reinecke, and Krzysztof Z. Gajos. "Crowdsourcing Performance Evaluations of User Interfaces". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '13. ACM. Paris, France: Association for Computing Machinery, 2013, pp. 207–216. ISBN: 9781450318990. DOI: 10.1145/2470654.2470684. URL: https://doi.org/10.1145/2470654.2470684.

[114] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein. "Automated Testing of Android Apps: A Systematic Literature Review". In: *IEEE Transactions on Reliability* 68.1 (Mar. 2019), pp. 45–66. ISSN: 1558-1721. DOI: 10.1109/TR.2018.2865733.

[115] Kubernetes. *Resource Management for Pods and Containers*. Last seen: Nov. 11, 2022. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[116] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. "Record and replay for Android: are we there yet in industrial cases?" In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 854–859.

[117] Quoc Le and Tomas Mikolov. "Distributed representations of sentences and documents". In: *International conference on machine learning*. PMLR. 2014, pp. 1188–1196.

[118] Rakesh Kumar Lenka, Sunakshi Mamgain, Srikant Kumar, and Rabindra Kumar Barik. "Performance Analysis of Automated Testing Tools: JMeter and TestComplete". In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. 2018, pp. 399–407. DOI: 10.1109/ICACCCN.2018.8748521.

[119] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. "Approaches and Tools for Automated End-to-End Web Testing". In: *Advances in Computers* 101 (Jan. 2016), pp. 193–237.

[120] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE. 2013, pp. 272–281.

[121] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. "Visual vs. DOM-based web locators: An empirical study". In: *International Conference on Web Engineering*. Springer. 2014, pp. 322–340.

[122] Maurizio Leotta, Filippo Ricca, and Paolo Tonella. "SIDEREAL: Statistical adaptive generation of robust locators for Web testing". In: *Software Testing, Verification and Reliability* 31.3 (2021), e1767.

[123] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. "ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing". In: *Journal of Software: Evolution and Process* (2016), 28:177–204.

[124] VI Levenshtein. "Binary Codes Capable of Correcting Deletions, Insertions and Reversals". In: *Soviet Physics Doklady* 10 (1966), p. 707.

[125] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. "DroidBot: A Lightweight UI-guided Test Input Generator for Android". In: *Proceedings of the 39th International Conference on Software Engineering Companion*. ICSE-C '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 23–26. ISBN: 978-1-5386-1589-8. DOI: 10.1109/ICSE-C.2017.8.

[126] Zhao Li and J. Tian. "Testing the suitability of Markov chains as Web usage models". In: *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*. 2003, pp. 356–361. DOI: 10.1109/CMPSAC.2003.1245365.

[127] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 399–410.

[128] *LoadComplete | TestComplete Documentation*. URL: https://support.smartbear.com/testcomplete/docs/other-tools/loadcomplete.html.

[129] *Loadrunner: Strumenti per il test di carico delle applicazioni: Micro focus*. URL: https://www.microfocus.com/it-it/products/loadrunner-professional/overview.

[130] Sylvain Lugeon, Tiziano Piccardi, and Robert West. "Homepage2Vec: Language-Agnostic Website Embedding and Classification". In: *Proceedings of the International AAAI Conference on Web and Social Media*. Vol. 16. 2022, pp. 1285–1291.

[131] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. *GraphCode2Vec: Generic Code Embedding via Lexical and Program Dependence Analyses*. 2021. DOI: 10.48550/ARXIV.2112.01218.

[132] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: An input generation system for android apps". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 224–234.

[133] Sonal Mahajan and William GJ Halfond. "Finding HTML presentation failures using image comparison techniques". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 91–96.

[134] Haroon Malik, Bram Adams, and Ahmed E. Hassan. "Pinpointing the Subsystems Responsible for the Performance Deviations in a Load Test". In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*. 2010, pp. 201–210. DOI: 10.1109/ISSRE.2010.43.

[135] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. "Detecting near-duplicates for web crawling". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 141–150.

[136] K. Mao, M. Harman, and Y. Jia. "Crowd intelligence enhances automated mobile testing". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM. Oct. 2017, pp. 16–26. DOI: 10.1109/ASE.2017.8115614.

[137] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. "A survey of the use of crowdsourcing in software engineering". In: *Journal of Systems and Software* 126 (2017), pp. 57–84. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.09.015. URL: http://www.sciencedirect.com/science/article/pii/S0164121216301832.

[138] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. "A survey of the use of crowdsourcing in software engineering". In: *Journal of Systems and Software* 126 (2017), pp. 57–84. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.09.015. URL: https://www.sciencedirect.com/science/article/pii/S0164121216301832.

[139] Ke Mao, Mark Harman, and Yue Jia. "Sapienz: Multi-objective automated testing for Android applications". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 94–105.

[140] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. "State-based testing of Ajax web applications". In: *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE. 2008, pp. 121–130.

[141] Kaszo Mark and Legany Csaba. "Analyzing Customer Behavior Model Graph (CBMG) using Markov Chains". In: *2007 11th International Conference on Intelligent Engineering Systems*. 2007, pp. 71–76. DOI: 10.1109/INES.2007.4283675.

[142] Vincenzo Marrazzo. *How to handle correlation in jmeter: Blazemeter by perforce*. URL: https://www.blazemeter.com/blog/correlation-in-jmeter#what.

[143] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. *Using Transfer Learning for Code-Related Tasks*. 2022. DOI: 10.48550/ARXIV.2206.08574.

[144] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. "Using Deep Learning to Generate Complete Log Statements". In: *Proceedings of the 44th International Conference on Software Engineering*.

ICSE '22. Pittsburgh, Pennsylvania: ACM, 2022, pp. 2279–2290. ISBN: 9781450392211. DOI: 10.1145/3510003.3511561.

[145] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. "Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks". In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE '21. Madrid, Spain: IEEE, 2021, pp. 336–347. ISBN: 9781450390859. DOI: 10.1109/ICSE43902.2021.00041.

[146] Daniel A Menascé. "Load testing of web sites". In: *IEEE internet computing* 6.4 (2002), pp. 70–74.

[147] Daniel A Menascé, Virgilio AF Almeida, Rodrigo Fonseca, and Marco A Mendes. "A methodology for workload characterization of e-commerce sites". In: *Proceedings of the 1st ACM conference on Electronic commerce*. 1999, pp. 119–128.

[148] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes". In: *ACM Transactions on the Web*. TWEB 6.1 (2012), 3:1–3:30.

[149] Ali Mesbah, Arie van Deursen, and Danny Roest. "Invariant-based Automatic Testing of Modern Web Applications". In: *IEEE Transactions on Software Engineering (TSE)* 38.1 (2012), pp. 35–53.

[150] Microsoft. *Fast and reliable end-to-end testing for modern web apps | Playwright*. URL: https://playwright.dev/.

[151] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems* 26 (2013).

[152] Mostafa Mohammed, Haipeng Cai, and Na Meng. "An Empirical Comparison Between Monkey Testing and Human Testing (WIP Paper)". In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2019. Phoenix, AZ, USA: ACM, 2019,

pp. 188–192. ISBN: 978-1-4503-6724-0. DOI: 10.1145/3316482.3326342. URL: http://doi.acm.org/10.1145/3316482.3326342.

[153] *Monkeyrunner*. URL: https://developer.android.com/studio/test/monkeyrunner/index.html (visited on 05/08/2019).

[154] K. Moran, R. Bonett, C. Bernal-Cárdenas, B. Otten, D. Park, and D. Poshyvanyk. "On-Device Bug Reporting for Android Applications". In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. May 2017, pp. 215–216. DOI: 10.1109/MOBILESoft.2017.36.

[155] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. "Automatically discovering, reporting and reproducing android application crashes". In: *2016 IEEE international conference on software testing, verification and validation (icst)*. IEEE. 2016, pp. 33–44.

[156] Alessandro Moschitti. "Efficient convolution kernels for dependency and constituent syntactic trees". In: *European Conference on Machine Learning*. Springer. 2006, pp. 318–329.

[157] Alessandro Moschitti. "Making tree kernels practical for natural language learning". In: *11th conference of the European Chapter of the Association for Computational Linguistics*. 2006.

[158] *MunchLife: A Munchkin level counter for Android - GitHub repository*. Seen on Oct. 22, 2022. URL: https://github.com/pacebl/MunchLife.

[159] Marjane Namavar, Noor Nashid, and Ali Mesbah. "A Controlled Experiment of Different Code Representations for Learning-Based Bug Repair". In: *Empirical Software Engineering* (2022).

[160] Jakob Nielsen. "Finding Usability Problems through Heuristic Evaluation". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '92. ACM. Monterey, California, USA: Association for Computing Machinery, 1992, pp. 373–380. ISBN: 0897915135. DOI: 10.1145/142750.142834. URL: https://doi.org/10.1145/142750.142834.

[161] Jan Odvarko. *Har Adopters*. Last seen: Nov. 11, 2022. URL: http://www.softwareishard.com/blog/har-adopters/.

[162] Jan Odvarko, Arvind Jain, and Andy Davies. *HTTP Archive (HAR) format*. URL: https://w3c.github.io/web-performance/specs/HAR/Overview.html.

[163] Alessandro Orso. "Monitoring, analysis, and testing of deployed software". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 2010, pp. 263–268.

[164] Ramakanth P., Kalpan Bhargav, and M Tech. "A Survey on Performance Testing Approaches of Web Application and Importance of WAN Simulation in Performance Testing". In: *International Journal on Computer Science and Engineering* 4 (May 2012).

[165] Chester Parrott and Doris Carver. "Lodestone: A Streaming Approach to Behavior Modeling and Load Testing". In: *2020 3rd International Conference on Data Intelligence and Security (ICDIS)*. 2020, pp. 109–116. DOI: 10.1109/ICDIS50059.2020.00021.

[166] Mateusz Pawlik and Nikolaus Augsten. "Efficient computation of the tree edit distance". In: *ACM Transactions on Database Systems (TODS)* 40.1 (2015), pp. 1–40.

[167] Radim Řehůřek and Petr Sojka. "Software Framework for Topic Modelling with Large Corpora". In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Malta: ELRA, May 2010, pp. 45–50.

[168] *Replication Package*. https://github.com/anonymousconferencesubmitter/icse2023. 2022.

[169] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. "Three open problems in the context of E2E web testing and a vision: NEONATE". In: *Advances in Computers*. Vol. 113. Elsevier, 2019, pp. 89–133.

[170] *Robotium: User Scenario Testing for Android – GitHub repository*. Seen on Oct. 22, 2022. URL: https://github.com/RobotiumTech/robotium.

[171] *Robotium: User Scenario Testing for Android – GitHub repository*. Seen on Oct. 22, 2022. URL: https://marketplace.eclipse.org/content/robotium-recorder.

[172] Giancarlo Ruffo, Rossano Schifanella, Matteo Sereno, and Roberto Politi. "Walty: a user behavior tailored tool for evaluating web application performance". In: *Third IEEE International Symposium on Network Computing and Applications, 2004.(NCA 2004). Proceedings.* IEEE. 2004, pp. 77–86.

[173] Divya Saharan, Yogesh Kumar, and Rahul Rishi. "Analytical Study and Implementation of Web Performance Testing Tools". In: *2018 International Conference on Recent Innovations in Electrical, Electronics & Communication Engineering (ICRIEECE)*. 2018, pp. 2370–2377. DOI: 10.1109/ICRIEECE44171.2018.9008408.

[174] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse Coskun, and Manuel Egele. "Towards Practical Record and Replay for Mobile Applications". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. Las Vegas, NV, USA: ACM, 2019, 230:1–230:2. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3322476. URL: http://doi.acm.org/10.1145/3316781.3322476.

[175] Ibrahim Anka Salihu and Rosziati Ibrahim. "Systematic Exploration of Android Apps' Events for Automated Testing". In: *Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media*. MoMM '16. Singapore, Singapore: ACM, 2016, pp. 50–54. ISBN: 978-1-4503-4806-5. DOI: 10.1145/3007120.3011072.

[176] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. "Are students representatives of professionals in software engineering experiments?" In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 666–676.

[177] Gerard Salton, Anita Wong, and Chung-Shu Yang. "A vector space model for automatic indexing". In: *Communications of the ACM* 18.11 (1975), pp. 613–620.

[178] Andrew Sears. "Heuristic Walkthroughs: Finding the Problems Without the Noise". In: *International Journal of Human–Computer Interaction* 9.3 (1997), pp. 213–234.

[179] *SeleniumHQ Web Browser Automation*. http://www.seleniumhq.org/. Accessed: 2017-08-01. 2018.

[180] Mahnaz Shams, Diwakar Krishnamurthy, and Behrouz Far. "A Model-Based Approach for Testing the Performance of Web Applications". In: *Proceedings of the 3rd International Workshop on Software Quality Assurance*. SOQUA '06. Portland, Oregon: Association for Computing Machinery, 2006, pp. 54–61. ISBN: 1595935843. DOI: 10.1145/1188895.1188909. URL: https://doi.org/10.1145/1188895.1188909.

[181] Kilho Shin, Taichi Ishikawa, Yu-Lu Liu, and David Lawrence Shepard. "Learning DOM Trees of Web Pages by Subpath Kernel and Detecting Fake e-Commerce Sites". In: *Machine Learning and Knowledge Extraction* 3.1 (2021), pp. 95–122.

[182] *Simply Do – F-Droid*. Seen on Oct. 22, 2022. URL: https://f-droid.org/en/packages/kdk.android.simplydo/.

[183] Amit Singhal. "Modern Information Retrieval: A Brief Overview." In: *IEEE Data Eng. Bull.* 24.4 (2001), pp. 35–43.

[184] D. Skvorc, M. Horvat, and S. Srbljic. "Performance evaluation of Websocket protocol for implementation of full-duplex web streams". In: *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2014, pp. 1003–1008. DOI: 10.1109/MIPRO.2014.6859715.

[185] Óscar Soto-Sánchez, Michel Maes-Bermejo, Micael Gallego, and Francisco Gortázar. "A dataset of regressions in web applications detected by end-to-end tests". In: *Software Quality Journal* 30.2 (2022), pp. 425–454.

[186] I.D. Steiner. *Group Process and Productivity*. New York, NY, USA: Academic Press, 1972.

[187] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. "APOGEN: automatic page object generator for web testing". In: *Software Quality Journal* 25.3 (2017), pp. 1007–1039.

[188] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. "Clustering-Aided Page Object Generation for Web Testing". In: *Proceedings of 16th International Conference on Web Engineering*. ICWE 2016. Springer, 2016, pp. 132–151.

[189] Nezih Sunman, Yiğit Soydan, and Hasan Sözer. "Automated web application testing driven by pre-recorded test cases". In: *Journal of Systems and Software* (2022), p. 111441. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2022.111441.

[190] Michael J Swain and Dana H Ballard. "Indexing via color histograms". In: *Active perception and robot vision*. Springer, 1992, pp. 261–273.

[191] Suresh Thummalapenta, Pranavadatta Devaki, Saurabh Sinha, Satish Chandra, Sivagami Gnanasundaram, Deepa D Nagaraj, Sampath Kumar, and Sathish Kumar. "Efficient and change-resilient test automation: An industrial case study". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 1002–1011.

[192] *Tippy Tipper (Tip Calculator) – Github Repository*. Seen on Oct. 22, 2022. URL: https://github.com/mandlar/tippytipper.

[193] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. "Automated functional testing of mobile applications: a systematic mapping study". In: *Software Quality Journal* 27.1 (Mar. 2019), pp. 149–201. ISSN: 1573-1367. DOI: 10.1007/s11219-018-9418-6. URL: https://doi.org/10.1007/s11219-018-9418-6.

[194] Jatin Karthik Tripathy, Sibi Chakkaravarthy Sethuraman, Meenalosini Vimal Cruz, Anupama Namburu, P Mangalraj, Vaidehi Vijayakumar, et al. "Comprehensive analysis of embeddings and pre-training in NLP". In: *Computer Science Review* 42 (2021), p. 100433.

[195] *Trolly – F-Droid*. Seen on Oct. 22, 2022. URL: https://f-droid.org/en/packages/caldwell.ben.trolly/.

[196] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. "Towards Automating Code Review Activities". In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE '21. Madrid, Spain: IEEE, 2021, pp. 163–174. ISBN: 9781450390859. DOI: 10.1109/ICSE43902.2021.00027.

[197] Yuan-Hsin Tung and Shian-Shyong Tseng. "A novel approach to collaborative testing in a crowdsourcing environment". In: *Journal of Systems and Software* 86.8 (2013), pp. 2143–2153. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.03.079. URL: http://www.sciencedirect.com/science/article/pii/S0164121213000782.

[198] *UI Automator*. Seen on Oct. 22, 2022. URL: https://developer.android.com/training/testing/ui-automator.

[199] Raynor Vliegendhart, Eelco Dolstra, and Johan Pouwelse. "Crowdsourced User Interface Testing for Multimedia Applications". In: *Proceedings of the ACM Multimedia 2012 Workshop on Crowdsourcing for Multimedia*. CrowdMM '12. ACM. Nara, Japan: Association for Computing Machinery, 2012, pp. 21–22. ISBN: 9781450315890. DOI: 10.1145/2390803.2390813. URL: https://doi.org/10.1145/2390803.2390813.

[200] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. "WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems". In: *Software & Systems Modeling* 17.2 (2018), pp. 443–477.

[201] J. Wang, S. Wang, J. Chen, T. Menzies, Q. Cui, M. Xie, and Q. Wang. "Characterizing Crowds to Better Optimize Worker Recommendation in Crowdsourced Testing". In: *IEEE Transactions on Software Engineering* (2019). DOI: 10.1109/TSE.2019.2918520.

[202] J. Wang, Y. Yang, S. Wang, C. Chen, D. Wang, and Q. Wang. "Context-Aware Personalized Crowdtesting Task Recommendation". In: *IEEE Transactions on Software Engineering* 48.8 (2022). cited By 2, pp. 3131–3144. DOI: 10.1109/TSE.2021.3081171. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85106729716&doi=10.1109%5C%2fTSE.2021.3081171&partnerID=40&md5=67f37fe295058a12ed25afe41c5f8b51.

[203] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. "Images don't lie: Duplicate crowdtesting reports detection with screenshot information". In: *Information and Software Technology* 110 (2019), pp. 139–155. ISSN: 0950-5849. DOI: https://do

i.org/10.1016/j.infsof.2019.03.003. URL: https://www.sciencedirect.com/science/article/pii/S0950584919300503.

[204] Junjie Wang, Ye Yang, Rahul Krishna, Tim Menzies, and Qing Wang. "iSENSE: Completion-aware crowdtesting management". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 912–923.

[205] Junjie Wang, Ye Yang, Tim Menzies, and Qing Wang. "isense2. 0: Improving completion-aware crowdtesting management with duplicate tagger and sanity checker". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29.4 (2020), pp. 1–27.

[206] Junjie Wang, Ye Yang, Song Wang, Yuanzhe Hu, Dandan Wang, and Qing Wang. "Context-aware in-process crowdworker recommendation". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM/IEEE. 2020, pp. 1535–1546.

[207] Yihong Wang, Konstantinos Papangelis, Ioanna Lykourentzou, Vassilis-Javed Khan, Michael Saker, Yong Yue, and Jonathan Grudin. "The Dawn of Crowdfarms". In: *Commun. ACM* 65.8 (July 2022), pp. 64–70. ISSN: 0001-0782. DOI: 10.1145/3490698. URL: https://doi.org/10.1145/3490698.

[208] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.

[209] Liu Wenyin, Guanglin Huang, Liu Xiaoyue, Zhang Min, and Xiaotie Deng. "Detection of phishing webpages based on visual similarity". In: *Special interest tracks and posters of the 14th international conference on World Wide Web*. 2005, pp. 1060–1061.

[210] James A Whittaker. *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*. Pearson Education, 2009.

[211] Frank Wilcoxon. "Individual Comparisons by Ranking Methods". In: *Biometrics Bulletin* 1.6 (Dec. 1945), p. 80. DOI: 10.2307/3001968.

[212] Frank Wilcoxon. "Individual comparisons by ranking methods". In: *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[213] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering*. Springer, 2012.

[214] Murray Wood, Marc Roper, Andrew Brooks, and James Miller. "Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study". In: *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC '97/FSE-5. ACM. Zurich, Switzerland: Springer-Verlag, 1997, pp. 262–277. ISBN: 3540635319. DOI: 10.1145/267895.267915. URL: https://doi.org/10.1145/267895.267915.

[215] M. Xie, Q. Wang, Q. Cui, G. Yang, and M. Li. "CQM: Coverage-constrained quality maximization in crowdsourcing test". In: IEEE/ACM. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 192–194. DOI: 10.1109/ICSE-C.2017.112.

[216] T. Xie. "Cooperative testing and analysis: Human-tool, tool-tool and human-human cooperations to get work done". In: cited By 10. IEEE. 2012, pp. 1–3. DOI: 10.1109/SCAM.2012.31. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84872327610&doi=10.1109%2fSCAM.2012.31&partnerID=40&md5=b3ffc3cbef1d6f00b28e3b8a09e8b5b3.

[217] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. "Near-Duplicate Detection in Web App Model Inference". In: *Proceedings of 42nd International Conference on Software Engineering*. ICSE '20. ACM, 2020, 12 pages.

[218] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. "Near-duplicate detection in web app model inference". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 186–197.

[219] Bian Yang, Fan Gu, and Xiamu Niu. "Block mean value based image perceptual hashing". In: *2006 International Conference on Intelligent Information Hiding and Multimedia*. IEEE. 2006, pp. 167–172.

[220] Hector Yee, Sumanita Pattanaik, and Donald P Greenberg. "Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments". In: *ACM Transactions on Graphics (TOG)* 20.1 (2001), pp. 39–65.

[221] Hector Yee, Sumanita Pattanaik, and Donald P. Greenberg. "Spatiotemporal Sensitivity and Visual Attention for Efficient Rendering of Dynamic Environments". In: *ACM Trans. Graph.* 20.1 (Jan. 2001), pp. 39–65. ISSN: 0730-0301.

[222] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. "Sikuli: using GUI screenshots for search and automation". In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM. 2009, pp. 183–192.

[223] Keith Yorkston. "Performance Testing Tasks". In: *Performance Testing: An ISTQB Certified Tester Foundation Level Specialist Certification Review*. Berkeley, CA: Apress, 2021, pp. 195–354. ISBN: 978-1-4842-7255-8. DOI: 10.1007/978-1-4842-7255-8_4. URL: https://doi.org/10.1007/978-1-4842-7255-8%5C_4.

[224] Li Yujian and Liu Bo. "A normalized Levenshtein distance metric". In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1091–1095.

[225] Christoph Zauner. "Implementation and benchmarking of perceptual image hash functions". In: (2010).

[226] Samer Zein, Norsaremah Salleh, and John Grundy. "A Systematic Mapping Study of Mobile Application Testing Techniques". In: *J. Syst. Softw.* 117.C (July 2016), pp. 334–356. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.03.065. URL: https://doi.org/10.1016/j.jss.2016.03.065.

[227] Xiaofang Zhang, Yang Feng, Di Liu, Zhenyu Chen, and Baowen Xu. "Research progress of crowdsourced software testing". In: *Journal of Software* 29.1 (2018), pp. 69–88.

[228] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. "Automatic web testing using curiosity-driven reinforcement learning". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 423–435.

Author's Publications

Journal Papers

Sergio Di Martino and Luigi Libero Lucio Starace. "Towards Uniform Urban Map Coverage in Vehicular Crowd-Sensing: a Decentralized Incentivization Solution". In: *IEEE Open Journal of Intelligent Transportation Systems* (2022).

Dario Asprone, Sergio Di Martino, Paola Festa, and Luigi Libero Lucio Starace. "Vehicular crowd-sensing: a parametric routing algorithm to increase spatio-temporal road network coverage". In: *International Journal of Geographical Information Science* 35.9 (2021), pp. 1876–1904.

Sergio Di Martino, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. "Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing". In: *Software Testing, Verification and Reliability* 31.3 (2021), e1754.

Valentina Casola, Alessandra De Benedictis, Sergio Di Martino, Nicola Mazzocca, and Luigi Libero Lucio Starace. "Security-Aware Deployment Optimization of Cloud–Edge Systems in Industrial IoT". In: *IEEE Internet of Things Journal* 8.16 (2020), pp. 12724–12733.

Conference Papers

Francesco Altiero, Giovanni Colella, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Change-Aware Regression Test Prioritization using Genetic Algorithm". In: *Proceedings of the 48th Euromicro Conference on Software Engineering and Advanced Applications*. To appear in the proceedings. IEEE. 2022.

Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "ReCover: a curated dataset for regression testing research". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 196–200.

Sergio Di Martino and Luigi Libero Lucio Starace. "Vehicular Crowd-Sensing on Complex Urban Road Networks: A Case Study in the City of Porto". In: vol. 62. Elsevier, 2022, pp. 350–357.

Luigi Libero Lucio Starace, Andrea Romdhana, and Sergio Di Martino. "GenRL at the SBST 2022 Tool Competition". In: *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. IEEE. 2022, pp. 49–50.

Massimo Benerecetti, Fabio Mogavero, Adriano Peron, and Luigi Libero Lucio Starace. "Expressing Structural Temporal Properties of Safety Critical Hierarchical Systems". In: *International Conference on the Quality of Information and Communications Technology*. Springer. 2021, pp. 356–369.

Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Web Application Testing: Using Tree Kernels to Detect Near-duplicate States in Automated Model Inference". In: *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2021, pp. 1–6.

Francesco Altiero, Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. "Inspecting code churns to prioritize test cases". In: *IFIP International Conference on Testing Software and Systems*. Springer. 2020, pp. 272–285.

Massimo Benerecetti, Ugo Gentile, Stefano Marrone, Roberto Nardone, Adriano Peron, Luigi Libero Lucio Starace, and Valeria Vittorini. "From dynamic state machines to promela". In: *International Symposium on Model Checking Software*. Springer. 2019, pp. 56–73.