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

Automatic Systematic GUI Testing for Web Applications

Thiago Santos de Moura

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

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

Prof. Dr. Everton Leandro Galdino Alves
Prof. Cláudio de Souza Baptista, Ph.D.
(Supervisors)

Campina Grande, Paraíba, Brasil

Resumo

A testagem automatizada é crucial para o desenvolvimento de software, proporcionando eficiência, redução de custos e repetibilidade. No nível da Graphical User Interface (GUI), ela valida funcionalidades e detecta falhas em aplicações web. Ferramentas que utilizam processos de geração e execução frequentemente visam identificar falhas visíveis, como travamentos, mensagens de erro e comportamentos inesperados. Nesse contexto, propomos o Cytestion, uma abordagem e ferramenta automatizada e sistemática de teste de GUI para aplicações web, que aplica uma abordagem progressiva e sem scripts. Começando com um caso de teste inicial, ele explora progressivamente os elementos usando erros do console do navegador, status de solicitações HTTP e mensagens de falha da GUI para validação. Para ser eficaz, o Cytestion teve que enfrentar três desafios principais: descoberta automática e única de elementos acionáveis, sincronização robusta com a aplicação em teste e gerenciamento de tempo de execução prolongado em contextos web industriais. Para enfrentar esses desafios, introduzimos e avaliamos a abordagem Unique Actionable Elements Search (UAES), o mecanismo Network Wait e o algoritmo Iterative Deepening URL-Based Search (IDUBS). Essas soluções avançaram o campo da testagem automatizada. Nossos estudos empíricos utilizando quatro aplicações web de código aberto e vinte industriais demonstram o desempenho superior do Cytestion na detecção de falhas e eficiência de tempo de execução em comparação com uma ferramenta de teste GUI do estado da arte.

Palavras-chave: ferramenta de testagem automatizada, aplicações web, detecção faltas visíveis, exploração sistemática

Abstract

Automated testing is crucial for software development, providing efficiency, cost reduction, and repeatability. At the Graphical User Interface (GUI) level, it validates functionalities and detects faults in web applications. Tools that utilize generation and execution processes often aim to identify visible failures, such as crashes, error messages, and unexpected behaviors. In this context, we propose Cytestion, an automated and systematic GUI testing approach and tool for web applications, which applies a scriptless and progressive approach. Starting with an initial test case, it progressively explores elements using browser console errors, HTTP request status, and GUI failure messages for validation. In order to be effective, Cytestion had to face three primary challenges: automatic and unique discovery of actionable elements, robust synchronization with the application under test, and managing extended runtime in industrial web contexts. To address these, we introduced and evaluated the Unique Actionable Elements Search (UAES) approach, the Network Wait mechanism, and the Iterative Deepening URL-Based Search (IDUBS) algorithm. These solutions advanced the field of automated testing. Our empirical studies using four open-source and twenty industrial web applications demonstrate Cytestion superior performance in fault detection and runtime efficiency compared to a state-of-the-art GUI testing tool.

Keywords: automated testing tool, web applications, visible fault detection, systematic exploration

Acknowledgments

Here, I express my sincere gratitude to all who helped me throughout the long journey of this master's research.

To God, truthful friend, that is always with me through the good and bad moments. Without Him, I would never have the strength to overcome challenges or achieve my goals.

To my family for their love and support. My parents, João Ronaldo and Lucineide, for never sparing any effort to provide me with a quality education throughout my schooling and for understanding my absence while I dedicated myself to this master's research. I am who I am today because of what you taught me. To you, my unconditional love.

To my wife, Ingrid, the love of my life, for all the support, patience, and care, especially during the most difficult moments. I dedicate all my achievements to you, today and always.

To professors Everton Alves and Cláudio Baptista, your guidance, trust, supervision, and direction greatly helped me throughout the research and writing of this dissertation. Your examples were essential in sparking a passion for scientific research that I will carry with me throughout my life. I could not have imagined having better mentors for my master's.

To professor Hugo Figueirêdo, my informal supervisor, for your corrections and teachings that contributed immensely to this master's journey. Without your help, I would not have made it here, and you know it.

To my brother Márcio Torres, for all the support, encouragement, and for always believing that I was capable of completing this research. Your belief in me has been invaluable.

To my friends Mateus, Francisco, Regina, Igor, Ismael, Gregório, Danilo, Lucian, and the other members of the Information Systems Laboratory (LSI), for all the support, learning, and partnership. I will always cherish the friends I made here and the knowledge I gained.

To the Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for partial financial support for my education.

Last but not least, I thank the Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande and its team for their administrative support.

Contents

List of Symbols

GUI - *Graphical User Interface*

AUT - *Application Under Test*

DOM - *Document Object Model*

SPAs - *Single-Page Applications*

RIAs - *Rich Internet Applications*

DFS - *Depth-First Search*

BFS - *Breadth-First Search*

IDS - *Iterative Deepening Search*

UAES - *Unique Actionable Elements Search*

IDUBS - *Iterative Deepening URL-Based Search*

List of Figures

List of Tables

Chapter 1

Introduction

Web development is a fast-paced field often shaped by the ever-changing demands of clients who seek high-quality software releases in short timeframes. This context highlights the critical need to ensure stability and reliability in web applications [51]. Although manual testing is essential, it is often considered a costly and error-prone activity. Therefore, testers have shifted towards automated strategies for testing web applications, especially in industrial settings [34]. Automated testing strategies offer an effective and repeatable way to test software [22, 41].

Most web applications incorporate a Graphical User Interface (GUI) for user interaction. Therefore, GUI testing has become a significant testing strategy, as various behaviors are triggered by sequences of user events (e.g., clicks, text inputs, menu choices) resulting from interactions with GUI elements (e.g., buttons, text boxes, dropdown menus) [18, 62]. In GUI testing, events are used to explore different states of the Application Under Test (AUT), validate specific functionalities, and/or detect faults based on visible failures generated by the system [96].

GUI testing frameworks are typically classified into three categories [7, 61, 106]: coordinate-based, image recognition, and Document Object Model (DOM)-based. The first relies on screen coordinates to simulate user interactions. However, such tests are known to be fragile due to their dependence on specific screen resolutions and window configurations, characteristics that often change. The second uses visual localization tools, employing screenshots and image recognition algorithms. This approach is sensitive to visual changes and computationally intensive, which may result in longer executions and less reliable re-

sults. The third employs a DOM-based approach, which identifies web elements through their properties. In this work, we focus on the last category, DOM-based, which is widely embraced by testers due to its simplicity and reliability. Selenium[1] and Cypress[2] are examples of two well-known DOM-based testing frameworks [76].

GUI test suites can be created manually (scripted testing) or automatically (scriptless testing) [21]. In scripted testing, testers visually interpret web pages, find desired elements, and write tests based on contextual understanding of each element's role in the GUI to meet specific requirements. This approach is commonly used to verify functionalities and address specification errors [21]. Capture-and-replay can also be employed to record test sequences and generate precise locators for found elements during this process [57, 56]. However, maintaining web test scripts poses a well-known challenge in GUI testing. The need for frequent updates in test scripts relates to developmental changes in the AUT. Therefore, updating a test suite after significant changes in GUI design is often impractical [8, 83, 93].

Scriptless testing, on the other hand, involves the automatic generation of test cases by exploring web pages and simulating user actions. This method provides the advantage of generating new tests with each execution [23, 46, 49]. However, it may not offer the same level of control and customization as scripted approaches. Therefore, scriptless testing is more suitable for detecting "faults that cause visible failures", such as failing text messages presented through the GUI (e.g., errors, exceptions), browser console error messages, or unsuccessful results from requests made to the application's server (e.g., an HTTP status code of the 400 or 500 families) [21, 112].

The effectiveness of scriptless testing depends on the ability to automatically discover actionable elements within the pages [3, 36]. This allows for proper exploration and generation of meaningful test event sequences by navigating the GUI. Additionally, synchronization issues may arise when a test needs to wait for the result of some action, either for identifying visible failures or initiating the process of discovering new elements in a newly found state. This issue is so common in GUI testing that it is often referred to as the *synchronization challenge*[*] [58, 97, 105].

Monkey GUI testing, a type of scriptless test, generates test cases randomly based on

[1] `https://selenium.dev`

[2] `https://www.cypress.io`

[*] Also known as the *asynchronous challenge* or *race condition challenge*.

discovered GUI elements [30, 117]. However, in AUTs with many – potentially deeply nested – dialogues and actions, it is unlikely that a random algorithm sufficiently exercises most parts of the GUI within a reasonable time frame. Certain actions are easier to access and execute more often, while others might not be executed at all [113]. Heuristics can be applied to improve the discovery of new states and elements. However, due to its random nature, Monkey testing may still fail to trigger some actionable GUI elements while also leading to unnecessary repetitive interactions.

Alternative scriptless testing strategies include systematic GUI testing [16, 122] and model-based GUI testing [72, 108]. The first involves an exhaustive exploration of all actionable GUI elements, while the latter generates test cases based on application models. Both strategies present important challenges. For model-based GUI testing, specification or behavioral models are often not available [17]. On the other hand, systematic exploration often presents practical problems related to execution time and state explosion, where the number of potential test cases increases exponentially depending on the complexity of the system [14].

A combined approach can be used to incrementally build a GUI model through systematic exploration and iteratively generate tests [11, 124]. However, achieving a finite process depends on discovering a non-redundant set of GUI elements, posing challenges that necessitate advancements in exploration techniques [96]. An alternative is to rely on the tester to manually inspect the source code of the AUT and identify the regions that will become actionable in the GUI during testing, marking them uniquely to be discovered by automated tools [36]. However, this manual approach can be costly, error-prone, and not well-suited for black-box testing.

This work focuses on addressing critical challenges in scriptless GUI testing, particularly the detection of faults that lead to visible failures within dynamic web contexts. An effective approach to detect these faults requires the unique* and automated discovery of actionable elements, as well as algorithms to interpret a wide range of visible failures [45, 89]. Achieving high coverage of actionable elements can be accomplished through systematic exploration, but raises concerns about runtime efficiency, especially in complex industrial settings. An-

*We use the term "unique" to refer to the process of discovering each component of the GUI system that performs a specific function or has a distinct effect in all states of the system, exactly once.

other challenge that adds more complexity is the *synchronization challenge*. Therefore, there is a need for innovative solutions to improve automation and efficiency in web GUI testing [42, 58].

1.1 Motivational Examples

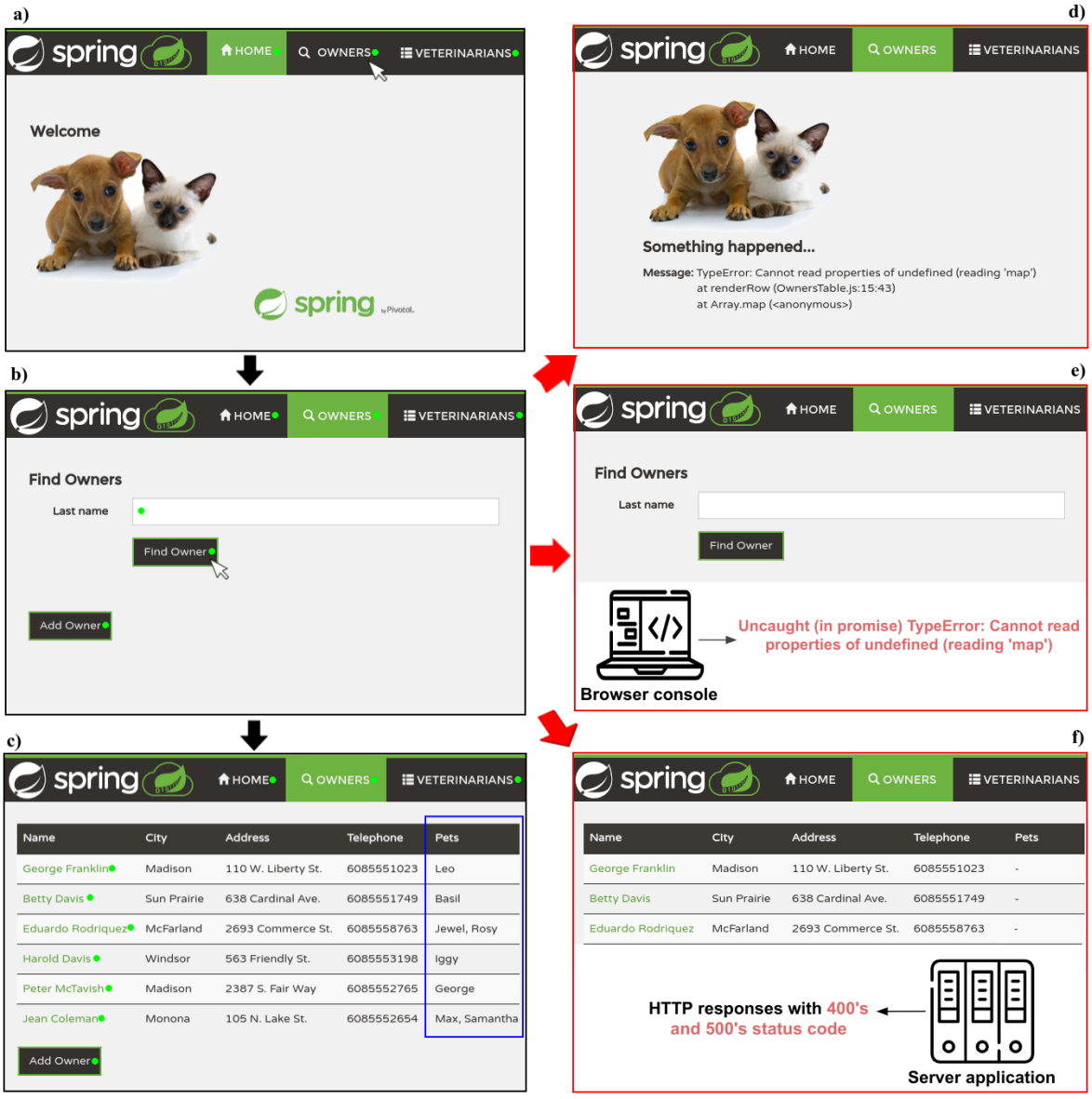

Figure 1.1: Exploring the listing owner feature and identifying three visible failures.

To illustrate faults that can cause visible failures and their impact, we will use the Pet-

Clinic[5] application. This system was implemented with ReactJS[6] and Spring Boot[7] and allows users to manage information about owners and their pets, as well as make appointments with veterinarians. Suppose that Thomas was the developer who implemented the feature for listing owners, as illustrated in Figure 1.1 (c). This state shows details related to each owner entity in the system, such as name, address, city, telephone number, and pets' names.

In this system, the process of inserting owners and their respective pets occurs in two steps, therefore, owners can be first registered without pets. However, Thomas was unaware of this possibility. The code snippet in Listing 1.1 shows a part of the code written by Thomas to render the rows of the owner table. When the function `renderRow` attempts to concatenate pet names in the column, if the owner has no pets, a `TypeError` is thrown (line 8) since `owner.pets` is undefined, and thus the `map` function cannot be called. This failure prevents the table from being properly rendered.

```
1  const renderOwners = (owners) => (
2      ...
3        {owners.map(renderRow)}
4      ...
5  );
6  const renderRow = (owner) => (
7      ...
8      <td>{owner.pets.map(pet => pet.name).join(', ')}</td>
9      // The line above causes a TypeError if owner.pets is undefined.
10 );
11 try {
12   renderOwners(owners); // Call to render the owners' table.
13 } catch (error) {
14   displayError(error); // Function to display the error in the GUI.
15 }
```

Listing 1.1: Snippet of code to render the rows of the owner table.

However, Thomas cautiously added a generic try-catch block to handle potential exceptions and display them in the GUI (lines 11-15). As a result, when a fail occurs, an unhelpful message is shown in the GUI (Figure 1.1 (d)). It is crucial to note that this message provides no details regarding the issue, offering limited insight. Without this error-handling mecha-

[5] https://github.com/spring-petclinic/spring-petclinic-reactjs

[6] https://react.dev

[7] https://spring.io/projects/spring-boot

nism, a worse case scenario can occur where the failure would only be visible in the user's browser console, as illustrated in Figure 1.1 (e). In such cases, the user receives no response after attempting to view a list of owners. This highlights two types of visible failures that could negatively impact how users perceive the system.

Suppose that Thomas handled the problems found in the frontend code to address these exposed failures. Now, a third scenario could occur when viewing the owner list. The application's client side sends a request to the server to fetch the owners' pets, but due to a fault in the server-side code handling pet information retrieval, it fails to process the request properly. This results in an unexpected "HTTP 500 - Internal Server Error" status code in the request made. In this scenario, the list of owners is successfully retrieved, but the client-side rendering of pets' names fails due to missing data. As a result, only the owners' details are displayed without listing their associated pets' names (Figure 1.1 (f)).

In this final example, the visible failure is evident in the request-response status code. Compared to the other two failures, this fault is particularly challenging to detect, as it may only result in incomplete information being displayed to the user. For instance, despite the absence of their pets' names, the owners are still listed as expected, leading the user to assume that everything is working properly. However, in cases where users notice the problem, they may perceive the application as unreliable or incomplete, resulting in frustration and confusion. By detecting and addressing faults that trigger a HTTP status code of the 400 or 500 families after server requests, developers can ensure that the system operates correctly, providing users with the necessary information without disruptions or unexpected errors in the GUI.

1.2 Challenges in Scriptless GUI Testing

To automatically identify visible failures such as the ones presented in Section 1.1, one can use a scriptless GUI testing tool. These tools can explore through the AUT trying to uncover the faults, which must involve the detection of faults from different perspectives, including the GUI, browser console, and server requests status. Therefore, implementing this approach requires using technologies that provide capabilities such as accessing the DOM, monitoring the browser console, and intercepting requests to ensure comprehensive fault detection and

optimal results.

An effective GUI testing approach/tool must navigate an unknown search space, considering the possibility of non-existence of predefined models. Moreover, this approach should explore the AUT to uncover potential faults that may appear in various GUI states. To address this challenge, a systematic exploration can be particularly effective. This technique ensures high coverage of actionable elements by systematically exploring the AUT, starting from a central point that provides access to as many functionalities as possible, often the home page.

By starting from this central point, the testing tool can identify multiple potential pathways through the application's GUI. Each path can represent a distinct branch of functionality, enabling independent exploration and thorough testing of different areas of the AUT. This approach enhances the likelihood of detecting faults that trigger visible failures across various states of the GUI, thereby achieving comprehensive fault coverage. However, developing a scriptless GUI approach and applying a systematic technique presents inherent challenges, as we will describe in Sections 1.2.1, 1.2.2, and 1.2.3.

1.2.1 Challenge 1: Unique Discovery of Actionable Elements

One of the main challenges in scriptless GUI testing is the ability to automatically discover available actionable elements in the reached states [2, 73]. As depicted in Figure 1.1 (a-c) and highlighted with green dots, user-interactable elements can appear as menu items, buttons, or input fields. The difficulty lies in enabling a tool to interpret this information as easily as a human can. This can be achieved by searching the AUT's DOM or by using image recognition algorithms to detect elements that match patterns of actionable components.

Given the capability to discover these elements, applying a systematic exploration intensifies the problem. For instance, from the state depicted in Figure 1.1 (a), we can generate three new test cases to explore actions on each of the three discovered elements. This means that each element found in the state must be interacted with separately to ensure a comprehensive coverage of actionable elements and systematic exploration.

The test case that clicks on the "Owners" item could take us to the state depicted in Figure 1.1 (b), which includes six actionable elements highlighted by green dots. This state also includes the three menu elements found in Figure 1.1 (a). To ensure a finite and non-

redundant exploration, it is necessary to disregard these elements, as they have already been covered by other test cases. This issue is characterized by the need to uniquely discover actionable elements in the AUT while ensuring the exploration reaches a conclusion.

1.2.2 Challenge 2: Synchronization

Synchronization is a critical challenge in automated GUI testing [58, 83]. During the execution of test cases, both scripted and scriptless testing can encounter flaky tests and intermittent failures. For instance, consider the states in Figure 1.1 (a-b). When a test case clicks on the "Owners" item and tries to click on the "Add Owner" button, if the page has not fully loaded, it might try to interact with the element before it is available. This premature execution results in flaky tests and breakages, falsely indicating faults [38, 37, 107]. In scripted testing, human testers can insert appropriate wait mechanisms to ensure the AUT is ready for the next action. However, in scriptless testing, the tool must automatically manage these waits.

Furthermore, in scriptless testing, automated test generation must try to discover actionable elements at the appropriate moment. After executing a test case that transitions to a new state (e.g., Figure 1.1 (b)), the tool must determine the correct waiting time for the state to be fully displayed. Unlike humans, machines may prematurely search for new actionable elements before the page is ready, leading to either incomplete or incorrect test cases. This synchronization issue is critical for accurately generating new test cases and ensuring thorough exploration of the AUT.

Additionally, another challenge is the timing of evaluating newly discovered states for visible failures. In cases of faulty code, the test case might lead to an unexpected state (e.g., Figure 1.1 (d)). Accurately timing the evaluation of these states is essential to identify failures that may appear after a slight delay. For instance, a failure might become visible one second after an action is performed. Failing to account for this delay can result in missing fault detection, thus undermining the reliability of the testing process.

1.2.3 Challenge 3: Efficient Systematic Exploration

Another key challenge arises from the potential excessive runtime during systematic exploration, making the testing process lengthy. For instance, consider a system with an initial state containing ten actionable elements. Exploring each element may lead to the discovery of ten new states. If each of these new states introduces ten additional actionable elements, a test suite that systematically explores the application could potentially include a minimum of 100 test cases. This exponential growth in test cases can significantly impact runtime execution.

Real-world scenarios significantly exacerbate this challenge, particularly in industrial web applications. These applications frequently feature numerous interactive elements and complex workflow branches per page. As a result, the runtime execution of systematic exploration grows exponentially, which undermines the feasibility of the testing process. Running every generated test case sequentially could potentially result in minutes or even hours of execution time. Therefore, optimizations are essential for practical integration into the development process.

1.3 Goals

This dissertation focuses primarily on automating the detection of faults that result in visible failures in web applications through systematic GUI testing. To achieve this objective, specific challenges within scriptless GUI testing (Section 1.2) need to be addressed. Our aim is to propose a solution for Challenge 1 by developing an automatic approach to discover a non-redundant set of actionable elements for systematic exploration, ensuring each element is interacted with only once. In this way, we hope to achieve finite exploration, higher coverage of actionable elements, and uncover more faults in web applications.

To address Challenge 2, we aim to automatically reduce the *synchronization challenge* in order to achieve a scriptless functional approach. For that, we propose a new waiting mechanism tailored to address all the synchronization issues identified by our automated approach. Finally, we aim to work with acceptable execution times (Challenge 3). To that end, we present an optimized algorithm that explores the AUT using the information presented in the web context. Additionally, we explore parallel executions to further improve execution time.

1.4 Relevance

A bibliometric study on GUI testing over the past 30 years concluded that automated exploration techniques, such as the one used in this work, will become a prominent trend in the coming years and require advances in research, especially within the web context [96]. Furthermore, the challenges highlighted in Section 1.2 were identified by a systematic literature review as accidental challenges that significantly impact GUI test automation and should be addressed through new research initiatives [83]. While existing tools for automated GUI testing exist, they have yet to fully address these persistent challenges. This dissertation aims to directly tackle these obstacles with the goal of advancing scriptless GUI testing and driving forward automation and reliability within web application development.

In our work, we propose a solution that can hold significant relevance for the industrial context by addressing key challenges in GUI testing automation. We propose an automated approach to explore the GUI and detect faults leading to visible failures, enabling industrial projects to enhance their testing processes, efficiently detect faults, and improve software reliability. The effectiveness, demonstrated by empirical studies on both open-source and industrial projects, highlights the practical relevance of the proposed approach and its potential value.

The findings of this dissertation are also relevant to toolmakers involved in the development of automated testing tools for web applications. The proposed approach and solutions to the proposed challenges can inform the design and enhancement of other testing tools, leading to more robust and efficient results that cater to the evolving needs of software development, quality assurance practices, and overall industry standards.

1.5 Contributions

This work presents several key contributions. Firstly, we introduce Cytestion, an approach and tool designed to automate systematic GUI testing on web applications to detect faults resulting in visible failures. Cytestion employs a scriptless and progressive approach, starting with an initial test case to discover actionable elements in the system's initial state. It then progressively generates new tests by exploring each actionable element. Our oracle uses

browser messages, HTTP request statuses, and both default and customized GUI failure messages for validation. Cytestion not only generates and executes tests during state exploration but also provides outputs such as a generated test suite for regression, a summary of visible failures found, and videos demonstrating the encountered failures. Through the development of Cytestion, we were able to empirically demonstrate the challenges involved in creating a scriptless GUI testing tool [79].

Secondly, we address the challenges discussed in Section 1.2 by proposing innovative solutions. We introduce the Unique Actionable Elements Search (UAES) [81], an automated approach that uniquely discovers actionable elements for systematic GUI testing in web applications, and we integrate it into Cytestion. Additionally, we introduce the Network Wait [78], a waiting mechanism designed to handle synchronization issues effectively in the nondeterministic context of scriptless testing, and incorporate it into Cytestion. Furthermore, we present the Iterative Deepening URL-Based Search (IDUBS) [80], an optimized exploration algorithm for web GUI trees that considers URL changes to shorten paths. To further improve runtime, we incorporate parallelization of test case executions. Combining all these approaches, we designed the Cytestion approach and tool (presented in Section 4).

Thirdly, we present a comprehensive set of empirical studies and reproduction kits. These studies use four open-source web applications and twenty industrial applications for comparison. We evaluate the effectiveness and costs of Cytestion compared to a state-of-the-art GUI testing tool. The studies offer detailed insights into the practical applications of Cytestion, demonstrating its advantages and limitations in real-world scenarios. Additionally, we conduct separate empirical evaluations for each proposed approach that addresses the referred challenges. By providing reproduction kits, we ensure that our empirical studies can be independently validated and extended by other researchers, contributing to the robustness and reliability of our findings.

We documented and published individual academic papers that better described the prototype version of Cytestion [79], the UAES approach [81], the Network Wait mechanism [78], and the IDUBS algorithm [80]. This set of papers contributes to the wider body of knowledge in automated GUI testing methodologies and evidences the scientific contributions of this work.

1.6 Cytestion and Our Previous Work

The first idea and prototype of Cytestion comes from my undergraduate thesis [77]. In this initial phase, we developed the prototype tool, which was later registered [26]. During my dissertation, we greatly evolved the previous work by turning Cytestion to an approach, not only a tool, adapting its procedure, and running empirical studies that showed its value [79].

In this work, we present the newer version of Cytestion, which refers to the updated version of the approach and addresses key challenges related to GUI testing [81, 78, 80].

1.7 Structure

The remainder of the document is organized as follows: Chapter 2 offers a background on GUI testing, systematic exploration, and automated GUI frameworks and tools. Chapter 3 presents our solutions to the challenges of scriptless GUI testing. Chapter 4 introduces Cytestion, detailing its approach and tool features. Chapter 5 presents the evaluation studies comparing Cytestion to a state-of-the-art tool. Chapter 6 discusses related work, focusing on challenges and tools with similar goals. Finally, Chapter 7 provides concluding remarks and outlines potential future work.

Chapter 2

Background

This chapter introduces important concepts to this document. We begin by exploring GUI testing (Section 2.1), addressing the specific challenges posed by web applications (Section 2.1.1), and examining the distinctions between static and dynamic elements (Section 2.1.2). Subsequently, we delve into model-based GUI testing and systematic exploration techniques (Section 2.2), emphasizing the Iterative Deepening Search algorithm (Section 2.2.1). Additionally, we discuss key aspects of prominent GUI testing frameworks (Section 2.3). Finally, we provide an overview of scriptless GUI testing tools, with a focus on the TESTAR tool (Section 2.4 and 2.4.1).

2.1 GUI Testing

GUI testing is a crucial component of software quality assurance, which involves evaluating a system through its GUI elements and properties [74]. It includes performing sequences of user-like interactions such as clicks, scrolls, and keystrokes on available GUI elements in different states of the AUT [18, 62]. Testers can perform GUI testing either manually or using automated frameworks. Manual testing requires human interaction to validate the functionality, usability, and compliance with specified requirements of the GUI. While this approach can be thorough and contextually aware, it is often time-consuming, error-prone, and lacks the repeatability necessary for comprehensive testing across multiple iterations of software development [104].

Automated GUI testing utilizes frameworks to automatically execute predefined test se-

quences, enhancing testing efficiency and reducing human error. It enables consistent test execution across different software versions and can perform repetitive tasks quickly and accurately, providing rapid feedback during development cycles [5]. Automated GUI testing is classified into scripted and scriptless approaches, each with unique advantages and challenges.

In scripted testing, testers manually write scripts for each test sequence, either from scratch or by using capture-replay tools to record user interactions [56]. Scripted tests provide precise control and customization but require substantial maintenance as the application's GUI evolves. Updating test scripts to accommodate changes in the GUI can be labor-intensive and time-consuming, making this approach difficult for rapidly changing applications [21]. Despite these challenges, scripted testing enables detailed and specific testing scenarios tailored to particular functionalities and workflows, offering profound insights into application behavior.

In contrast, scriptless testing dynamically generates test sequences without manual scripting. It often begins with an initial visit to a state of the AUT and then explores and generates actions by interacting with the discovered elements [82]. This method minimizes the need for extensive scripting knowledge, adapts more easily to GUI changes, and reduces maintenance overhead. Scriptless testing is useful for both exploratory and regression testing, allowing new test cases to be generated on-the-fly [21]. However, it may lack the precision and context-specific understanding of scripted tests, potentially resulting in less targeted testing outcomes.

2.1.1 Web Application GUI Testing

Web applications introduce specific challenges to GUI testing, requiring testers to address the dynamic nature of web interfaces and the complexities of modern web technologies [68]. GUI testing in web applications involves validating the visual elements, interactive features, and user experience across different browsers, devices, and screen sizes. Testers need to consider factors such as responsive design, cross-browser compatibility, and the impact of client-side scripts on GUI behavior [100].

One of the key considerations in GUI testing for web applications is the handling of asynchronous behavior and AJAX requests [68]. As web interfaces often rely on asyn-

chronous communication between the client and server, testers must ensure that the GUI responds correctly to data updates and user interactions in real-time. This aspect of GUI testing requires specialized techniques to capture and validate the dynamic behavior of web elements that change asynchronously based on server responses. Moreover, the proliferation of Single-Page Applications (SPAs) and Rich Internet Applications (RIAs) has further complicated GUI testing in web environments [10]. SPAs, which load content dynamically without refreshing the entire page, present challenges in verifying the GUI state transitions and interactions. RIAs, on the other hand, leverage advanced client-side technologies to deliver interactive and engaging user interfaces, necessitating comprehensive testing to validate the behavior of complex GUI components and data flows.

For web applications, DOM-based GUI testing is widely adopted in the industry [24]. This approach relies on the DOM to interact with web elements. Test scripts use locators to find elements, perform actions such as filling input fields, clicking buttons, and verifying outputs by examining the elements that display results [57]. However, in scripted testing, locators must be regularly checked for accuracy and may need updates with each software release. Even a slight modification of the AUT can have a significant impact on locators. Solutions like Robula+, Sidereal, and Similo aim to preserve the descriptive nature of these scripts while improving the robustness of the locators [60, 59, 86].

Overall, GUI testing in web applications requires a holistic approach that considers the unique challenges posed by web technologies, asynchronous behavior, and the evolving landscape of GUI. By combining manual testing expertise with automated testing frameworks and specialized tools for web GUI testing, testers can effectively validate the functionality, usability, and performance of web interfaces to deliver high-quality web applications to end-users [68].

2.1.2 Static vs. Dynamic Elements

Modern web systems often differentiate between static and dynamic interactive components. Understanding this concept is essential to comprehend how different approaches behave with respect to the unique discovery of actionable elements. Static elements remain fixed and do not change based on a website user interaction, yet they are crucial for facilitating navigation throughout the web application. In Figure 2.1, we illustrate such elements (highlighted in

green). Examples of static elements can be found within static content such as headers, footers, sidebars, and navigation menus.

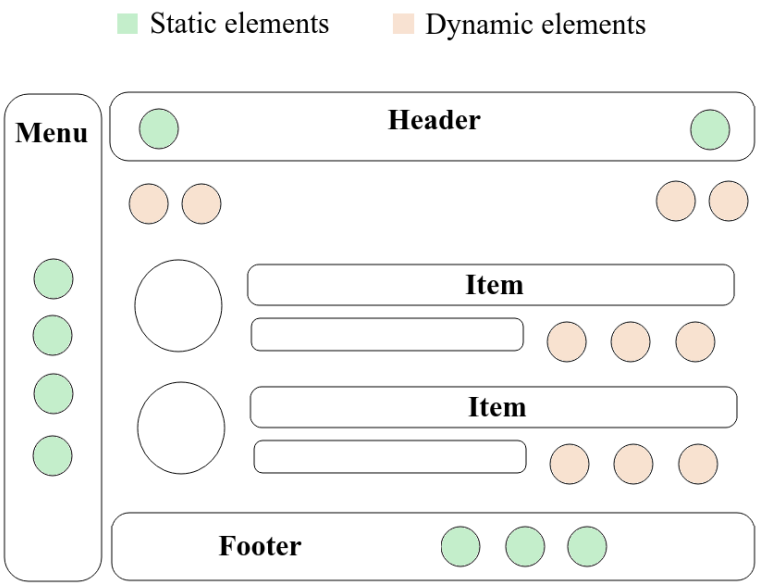

Figure 2.1: Example of static and dynamic elements.

On the other hand, dynamic elements (e.g., buttons, links, filtering, sorting options, and forms), provide personalized experiences by adapting in response to user input or system data. They facilitate navigation through the web application but are also associated with actions that may have collateral effects. As depicted in Figure 2.1, dynamic elements (highlighted in red) are often linked to an item list or functionalities of the respective state of the AUT, such as creating, updating, or deleting data items.

For tools that automate the generation of GUI tests in web applications, differentiating between static and dynamic elements is crucial. Static elements refer to those previously explored when first discovered, whereas dynamic elements refer to unexplored elements that should be covered by new tests. This distinction ensures comprehensive and systematic testing of web applications, addressing both the fixed navigational components and the interactive, adaptive elements.

2.2 Model-based GUI Testing and Systematic Exploration

Model-based GUI testing involves using formalisms such as Finite State Machines (FSM), Event Flow Graphs (EFG), and Unified Modeling Language (UML) diagrams to represent

the behavior of a GUI [50, 52]. Test cases are directly derived from these models, allowing for a precise verification process based on the defined behavior captured in the models. This approach leverages the formalism of the models to guide the generation of test cases and ensure comprehensive coverage of the GUI functionalities [18].

In contrast, systematic exploration requires a comprehensive and structured approach to test all actionable GUI elements within the AUT. This method methodically traverses the application's GUI, identifying possible paths and interactions for testing [16, 122]. By systematically exploring the GUI's functionalities through these test cases, testers can uncover faults and ensure the robustness of the software system [118].

While these strategies can be used independently, they are also highly complementary. Model-based GUI testing can guide systematic exploration by providing the necessary information to know the GUI tree and explore it more efficiently. With a pre-existing model, the systematic exploration process can follow a well-defined structure, ensuring thorough and directed coverage of the GUI. Conversely, systematic exploration can be used to build a model incrementally when one is not initially available. This involves conducting systematic interactions with GUI elements to discover new GUI states iteratively and finitely [124]. This process results in the creation of a GUI tree that represents the discovered AUT states accessed through the GUI, where nodes represent states and directed edges with actions represent the transitions between states and what triggers them [43].

Both strategies face challenges: model-based GUI testing struggles when behavioral models are unavailable, while systematic exploration encounters issues such as generation and execution time and state explosion in complex systems [16, 17]. Moreover, for systematic exploration without pre-existing models, it is crucial to uniquely recognize actionable elements within the GUI. Each element must be uniquely identified and interacted with to avoid redundant exploration and ensure the process terminates. Without this, the exploration could potentially be infinite, repeatedly visiting the same states and transitions without making progress. Techniques such as assigning unique identifiers to GUI elements or using algorithms to detect previously visited states can help manage this challenge and ensure the exploration is comprehensive yet finite [52, 82].

The GUI tree enables the use of graph traversal algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS) to systematically explore the search space for creating test

cases [16, 39]. Previous work has shown the practical advantages of such algorithms for GUI testing [47]. However, both algorithms impose practical limitations. While DFS may get trapped in deep branches, BFS may demand excessive memory due to its expansive exploration [99].

2.2.1 Iterative Deepening Search

Considering the incremental discovery aspect and the unknown-depth search space, the Iterative Deepening Search (IDS), also known as Iterative Deepening Depth-First Search (IDDFS), can be used to systematically explore the GUI tree of states [114]. IDS efficiently traverses the graph-based search spaces by gradually increasing the depth limit with each iteration, combining the strengths of DFS and BFS while mitigating their limitations [101]. This iterative approach starts with a depth limit of zero and increases it with each iteration until a goal is reached or the search space is exhausted. In this context, a goal could be the discovery of a state with visible failure or exhaustive exploration. Each iteration starts from the root node, ensuring a comprehensive and systematic traversal of the search space.

IDS is preferred for uninformed searches when the search space is large and the depth of the solution is unknown [99], aligning with the challenges of systematic GUI testing [16, 47]. However, IDS may generate redundant and costly test suites by potentially revisiting nodes at each iteration. In large graphs with deep paths to the solution, the time taken to revisit nodes can become significant, affecting the overall performance of the algorithm [63, 64]. Moreover, a single version of an AUT could have multiple faults that manifest as visible failures in different states (nodes). This requires multiple executions of IDS or integrating a multi-goal strategy into IDS, enabling it to efficiently navigate towards multiple objectives within the search space [40, 28].

Listing 1 presents the IDS algorithm with the multi-goals strategy [99]. It begins by initializing an empty list called `goalNodes` to store the goal nodes found during the search (line 1). The main function `IDS` takes the root node of the graph and the goal as inputs (line 2). It iterates over increasing depths from zero to infinity (line 3). At each depth, it calls the `DFS` function to explore nodes up to that depth and receives a boolean value indicating whether at least one new node was found, which potentially allows further exploration (line 4).

Algorithm 1 The IDS with Multi-Goals Algorithm

```
1:  goalNodes ← []
2:  function IDS(root, goal)
3:      for depth from 0 to ∞ do
4:          remaining ← DFS(root, goal, depth)
5:          if not remaining then
6:              return goalNodes
7:          end if
8:      end for
9:  end function
10:
11: function DFS(node, goal, depth)
12:     if depth = 0 then
13:         if node is a goal then
14:             goalNodes.add(node)
15:         end if
16:         return TRUE
17:     else if depth > 0 then
18:         anyRemaining ← FALSE
19:         for all child of node.children do
20:             anyRemaining ← DFS(child, goal, depth − 1)
21:         end for
22:         return anyRemaining
23:     end if
24: end function
```

The `DFS` function recursively explores nodes in the graph up to a specified depth. If the depth is zero, it checks if the current node is a goal node. If so, the node is added to the `goalNodes` list and returns `true` to evaluate possible children in the next iteration (lines 13-16). If the depth is greater than zero, the function explores all child nodes of the current node recursively, each time decreasing the depth by one (lines 20). The variable `anyRemaining` serves as a flag indicating whether any new nodes were found during the loop of child nodes at this level of depth (lines 18). When it remains `false`, it indicates that no new nodes were found in any branch, signaling an end to the search (lines 5-6).

Running Example

To illustrate the execution of the IDS algorithm with multi-goals, consider a GUI exploration of the open-source project PetClinic[1] application. It is represented as a tree-like structure that will be progressively constructed (Figure 2.2). Each node in the tree corresponds to a unique GUI state, and edges represent transitions between triggered states. The goal is to identify all GUI states where failures occur (nodes C and H). However, at the beginning of the execution, we have no prior knowledge of which states lead to failures.

We begin with a depth limit of zero, enabling the finding of the root node A. As A is not a goal node, i.e., it does not include a visible failure, we progress to the next depth level by indicating the `true` boolean value (line 16). At depth one, we start again with the root node A and recursively explore its children B and C (line 20). We come across node C, which is identified as a goal node due to a visible failure. We add node C to the list of goal nodes and continue exploring the graph.

At depth two, we pass through nodes A, B, C again and then make a recursive DFS call for the children of B. Since no failure is found but a child was found, another remaining value is returned. Moving to a depth three, we encounter nodes A, B, C, D, E once more and proceed to call the DFS on their respective children E and D. As F and G are not goal nodes, we continue to depth four where node H is discovered as a goal node and included in `goalNodes`. Finally, one more deep iteration is done, and all DFS calls return `false` as the deeper nodes do not have children therefore concluding IDS's execution and returning the goal nodes C and H.

Based on the presented execution, the test suite generated by IDS has the following test sequences: (1) A; (2) $A \rightarrow B$; (3) $A \rightarrow C$; (4) $A \rightarrow B \rightarrow D$; (5) $A \rightarrow B \rightarrow E$; (6) $A \rightarrow B \rightarrow D \rightarrow F$; (7) $A \rightarrow B \rightarrow E \rightarrow G$; (8) $A \rightarrow B \rightarrow E \rightarrow G \rightarrow H$; (9) $A \rightarrow B \rightarrow E \rightarrow G \rightarrow I$. There is a clear redundancy in the number of accessed states, especially the initial state A, which is visited in nine test cases.

[1] `https://github.com/spring-projects/spring-petclinic`

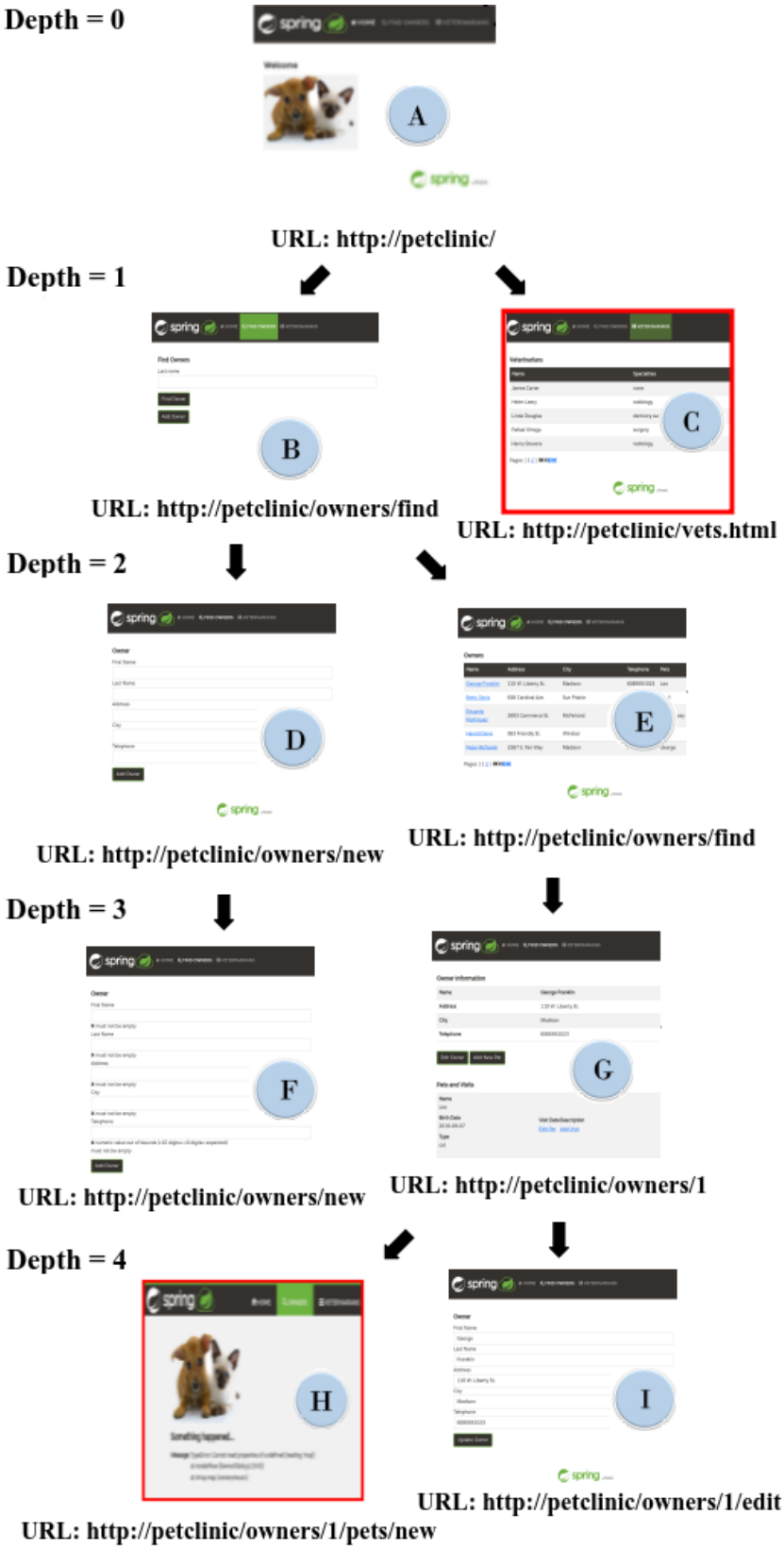

Figure 2.2: Example of the GUI tree of an AUT.

2.3 GUI Testing Frameworks

Selenium[2] is a testing framework that started in 2004. Since then, it has become a cornerstone in automated testing for web applications due to its robust functionality and versatility [33]. Supporting multiple programming languages (e.g, Java, Python, and Ruby), Selenium enables developers to write test scripts that interact seamlessly with web browsers. Its API allows integration with other automated tools, extending its capabilities for more complex test scenarios and enhancing adaptability across various testing environments. Despite its extensive capabilities, Selenium requires integration with additional testing frameworks like JUnit for comprehensive test management and reporting. Testers have reported challenges related to infrastructure setup, result reporting, and handling dynamic and complex web systems [58]. These aspects highlight Selenium's foundational role in web application testing while indicating areas where improvements or alternatives might be beneficial.

Emerging in 2017, Cypress[3] represents a modern approach to GUI testing frameworks, tailored specifically for modern web applications [54, 55, 76]. Unlike Selenium, Cypress integrates an all-inclusive architecture that combines browser engines, frameworks, and assertion libraries into a single downloadable package. This integration simplifies the setup process significantly, eliminating the need to manage disparate components. Cypress distinguishes itself with an API that directly interacts with the DOM. This approach accelerates testing cycles by leveraging real-time reloads and automatic waiting features, enhancing test efficiency and tester productivity. Moreover, Cypress supports headless browser testing and includes a built-in test runner that generates detailed reports and screenshots, facilitating comprehensive test result analysis.

Selenium and Cypress adopt distinct approaches to GUI testing automation. Selenium excels in cross-browser compatibility and extensive language support, making it a versatile choice for functional and regression testing across diverse web environments. Its API allows integration with other automated tools, expanding its functionality and adaptability. However, Selenium's reliance on external frameworks for test management and its slower execution speeds in certain scenarios pose operational challenges. Conversely, Cypress prioritizes simplicity and speed in GUI testing, offering an integrated testing environment that

[2] `https://selenium.dev`

[3] `https://www.cypress.io`

minimizes setup complexities and accelerates test execution. By leveraging direct DOM manipulation and innovative testing methodologies, Cypress enhances test reliability and developer experience, particularly in agile development environments where rapid feedback loops are crucial. Cypress's automatic waiting, real-time reloads, and built-in test runner streamline the testing process, making it a more efficient framework for modern web development practices.

Cypress's growing popularity is highlighted by its substantial increase in downloads, surpassing 5.7 million as of June 2024, compared to Selenium WebDriver's stable download figures around 2 million during the same period[4]. This surge reflects a preference shift among testers towards Cypress, driven by its user-friendly features and robust testing capabilities tailored for modern web development practices. In conclusion, while Selenium remains a foundational framework in web application testing, Cypress emerges as a compelling alternative for modern software development. Its streamlined architecture, real-time testing capabilities, and growing community support signify Cypress's evolution as a preferred choice for GUI testing automation.

2.4 Scriptless GUI Testing Tools

Scriptless GUI testing tools present an innovative approach to streamline GUI testing by eliminating the need for manual scripting [46]. These tools expedite the testing process by automating test case generation and execution through graphical interactions with the AUT. Techniques such as model-based testing and systematic exploration enable these tools to interact with GUI elements and verify application behavior automatically, thereby reducing the manual effort required for script development [49].

A notable feature of scriptless GUI testing tools is their ability to automatically discover and interact with GUI elements without manual intervention [112]. By utilizing element recognition algorithms using image or strings, these tools dynamically adapt to changes in the GUI layout and structure, ensuring robust test case execution across different application versions. Additionally, they need to manage application response times by optimizing interaction sequences, and incorporating wait mechanisms, thus maintaining test accuracy and

[4] https://npmtrends.com/cypress-vs-selenium-webdriver

efficiency despite delays.

To further enhance testing efficiency, scriptless GUI testing tools can prioritize critical test scenarios, optimize test case sequences, and parallelize test execution where feasible [110]. By focusing on essential test objectives and streamlining test flows, these tools aim to achieve high coverage of actionable elements within a reasonable timeframe. By leveraging innovative approaches, these tools improve the effectiveness and efficiency of GUI testing processes while adapting to the dynamic nature of modern software applications.

2.4.1 TESTAR

TESTAR [113] is an open-source, scriptless tool that performs automated GUI testing. Originally designed for desktop GUIs, it was later extended to cover web applications [9] and has undergone significant advancements [21, 95, 112]. TESTAR explores different paths of the AUT through the GUI, utilizing the Selenium WebDriver as an accessibility API to interact with DOM elements. Unlike traditional testing approaches, TESTAR does not require knowledge of the GUI's source code to discover and interact with the GUI elements.

Supporting various testing strategies, TESTAR can simulate user interactions and seamlessly integrate with popular testing frameworks like JUnit and TestNG. Its scriptless approach eliminates the need for predefined test cases, randomly selecting actions available at a given GUI state to detect faults causing visible failures. This generation strategy treats the GUI as a long test case, terminating upon fault detection and providing an HTML report with the execution results. Empirical studies conducted with TESTAR in industrial settings have demonstrated its efficiency and effectiveness, solidifying its position as a valuable complement to manual testing and traditional scripted GUI test automation methods [6, 94, 98].

2.5 Concluding Remarks

In this chapter, we cover the basic concepts related to our research. We will systematically explore web applications through automated GUI testing, using the Cypress framework to execute the generated tests. Understanding these concepts will enable the reader to fully comprehend the subsequent chapters of this document. In the following Chapter 3, we present the solutions we developed for each challenge outlined in Section 1.2. These so-

lutions will later be integrated into our approach for a scriptless GUI testing tool (in Chapter 4), which will be empirically evaluated in comparison with TESTAR (in Chapter 5).

Chapter 3

Addressing Challenges in Scriptless GUI Testing

Automated GUI testing faces numerous challenges, particularly in scriptless environments where synchronization and element discovery are critical. In this chapter, we introduce innovative solutions aimed at enhancing the efficiency of GUI testing tools. These solutions not only address existing challenges but also pave the way for more robust and reliable testing methodologies. By integrating these advancements into Cytestion (in Chapter 4), we aim to streamline the testing process, improve fault detection accuracy, and reduce redundancy in state exploration.

3.1 Unique Actionable Elements Search

A key challenge in scriptless GUI testing that employs a systematic exploration is the automatic and unique discovery of actionable elements, such as buttons, menus, and inputs. Accurately finding these elements is crucial for achieving high coverage of actionable elements and requires techniques like DOM analysis or image recognition. As the testing tool explores the application, it must generate test cases for each discovered element, while avoiding redundancy by ignoring previously interacted elements, ensuring efficient and finite exploration. This challenge is discussed in more detail in Section 1.2.1.

To address this challenge, we present the Unique Actionable Elements Search (UAES)*, a

*This approach along with its evaluation studies were published in a conference paper [81].

novel automatic approach for uniquely discovering actionable elements within web applications. It aims to offer a robust and adaptable way for enabling systematic GUI testing, where completeness and non-redundancy are key issues. This approach operates independently of API and offers a way to differentiate elements that can be interacted with using a GUI testing tool. It achieves this by treating the DOM of web pages as strings, enabling efficient extraction of relevant elements through predefined snippets and string search algorithms.

UAES discovers actionable elements based on a set of common code elements. Table 3.1 presents those code elements which includes common HTML element snippets and locator keys. This set was defined after we ran a comprehensive preliminary study on 24 web projects, where we analyzed nearly 10,000 actionable elements. The properties found in these elements were scrutinized to derive the predefined snippets.

HTML Element Snippets		**Possible Locator Keys**
Tag	**Attributes**	`inner-text`
`<a`	`onclick=`	`id`
`<button`	`href=`	`name`
`<input`	`type="button"`	`title`
`<select`	`type="submit"`	`description`
`<textarea`	`type="reset"`	`placeholder`
`<datalist`	`class="*value*"`	`tooltip`
`<details`		`alt`
`<summary`		`aria-label`
`<menu`		`value`
`<menuitem`		`class`

Table 3.1: HTML element snippets divided by tag and attributes, with possible locator keys.

The HTML element snippets include tag snippets, attributes, and class values provided by testers who recognize the value in actionable elements of the system's web pages. These snippets serve as keys passed to a string search algorithm, such as Aho-Corasick [1], to discover elements during the element searching process. With the positions of the found elements acquired, we can accurately extract the tag from the DOM string and proceed to verify their uniqueness.

The *inner text*, also known as the text content enclosed within HTML tags, plays a significant role in conveying semantic information and helps users understand the purpose of the associated elements on the GUI [53]. In Table 3.1, we use possible locator keys to define a unique locator for the found elements. The locator validates and distinguishes candidates among discovered elements. These keys are chosen based on their order in the list, with *inner text* being the first option. Elements that do not include an *inner text*, such as buttons with icons, are selected using unique attributes such as *id*, *name*, *tooltip*, *alt*, *aria-label*, *title*, *description*, and *placeholder*, which provide meaning to indicate their role. Attributes such as *value* and *class* can also be used because they play a minimum semantic role.

The locator (key + value found) is crucial for uniquely discovering elements in UAES. To avoid redundancies, it takes into account locators from the previously explored state when defining a locator for a new element. This comparison is executed within the elements associated with the current URL and the previous one. This strategy is particularly useful to avoid the discovery of static elements that do not change after executing an action that causes an URL change. Web applications can change entire pages without requiring another URL. Therefore, a comparison within the same URL allows us to discover only dynamic elements that appear after executing an action.

Listing 2 presents the algorithm of the UAES approach, responsible for dynamically updating the set of Actionable Elements (`setAE`). We identify an Actionable Element by the triple $< tag, locator, urls >$, where the `tag` is the string of the tag found. The `locator` is a unique identifier attributed to the tag, and `urls` is a list of URLs where that element was found. The algorithm starts by initializing an empty set (line 1), and setting up the predefined HTML snippets and possible locator keys (lines 2-3). The procedure receives as arguments the content of the GUI state (the page's HTML code) as string, along with its current and previous URL as inputs (line 4).

First, the `DiscoverTags` function is called for discovering potential interactive tags within the provided content (line 5). For that, `DiscoverTags` applies the Aho-Corasick string search algorithm [1] to locate all instances of interactive tags using the `HTMLsnippets`, and then extracting the corresponding tags (line 15). These tags are subsequently filtered to ensure that they are visible and available in the GUI, meaning that they appear on screen and are not disabled.

Algorithm 2 The UAES Algorithm

1: $setAE \leftarrow \{\}$ \\ **set of AE triples** $< tag, locator, urls >$
2: $HTMLsnippets \leftarrow$ **list of predefined HTML snippets.**
3: $locatorKeys \leftarrow$ **list of predefined possible locator keys.**
4: **procedure** UAES($content$, url, $previousUrl$)
5: $tags \leftarrow$ DISCOVERTAGS($content$)
6: **for all** $tag \in tags$ **do**
7: $locator \leftarrow$ DEFINELOCATOR(tag, url, $previousUrl$)
8: **if** $locator$ is not null **then**
9: Add AE triple $(tag, locator, url)$ to $setAE$
10: **end if**
11: **end for**
12: **end procedure**
13: **function** DISCOVERTAGS($content$)
14: $setTags \leftarrow \{\}$
15: $discoveredTags \leftarrow$ **Aho-Corasick**($content$, $snippetsHTML$)
16: **for all** $tag \in discoveredTags$ **do**
17: **if** tag is visible and available **then**
18: Add tag to $setTags$
19: **end if**
20: **end for**
21: **return** $tags$
22: **end function**
23: **function** DEFINELOCATOR(tag, url, $previousUrl$)
24: $locators \leftarrow$ **getPossibleLocators**(tag, $locatorKeys$)
25: $locator \leftarrow$ the first locator in $locators$, or $null$ if no next exists.
26: $previousAE \leftarrow$ **getPreviousAE**($setAE$, url, $previousUrl$)
27: **for all** $AE \in previousAE$ **do**
28: $locatorsAE \leftarrow$ **getPossibleLocators**($AE.tag$, $locatorKeys$)
29: **if** $locators \equiv locatorsAE$ **then**
30: **updateAEURLs**($setAE$, AE, url)
31: $locator \leftarrow$ null
32: **break**
33: **else if** $locator \equiv AE.locator$ **then**
34: $locator$ next locator in $locators$, or $null$ if no next exists.
35: **end if**
36: **end for**
37: **return** $locator$
38: **end function**

After obtaining the candidate tags, the UAES procedure iterates through each tag and starts defining a locator for each one (line 7) by calling the `DefineLocator` function. The `DefineLocator` function starts by retrieving all potential locators using the `getPossibleLocators` function and assigns them to the `locators` list (line 24). This function looks for the valid locator keys linked with the given tag and provides a complete list of locators (key + value) if any are found. Then, it generates a list of `previousAE`, encompassing all AE triples within the `setAE` that contain either the current URL or the previous URL within their corresponding triple's URLs.

Subsequently, a loop is executed to ensure the uniqueness of locators for each tag (lines 27-36). This loop iterates through all `previousAE`, and during each iteration, all possible locators for the AE at that time are stored in locatorsAE. If the lists `locators` and `locatorsAE` are identical, it means that the elements are identical as well. Therefore, we need to update this AE triple by associating it with the current URL, assigning *null* to the current locator, and terminating the execution of the loop (lines 30-32). Additionally, we check whether or not the current locator is the same as the current AE locator. If so, we select either the next available locator from locators or we assign *null* if it does not exist (line 34).

Upon the successful identification of an unique locator for the candidate element, the corresponding AE triple (tag, locator, and `[URL]` as list), is added to the `setAE` (line 9). By working this way, UAES maintains in `setAE` a comprehensive catalog of AE that are unique to the AUT.

3.1.1 Running Example

To demonstrate the use of UAES, we present an execution flow from the open-source PetClinic[2]. The initial state, shown in Figure 3.1 (a), begins with an empty `setAE` (line 1). The initial content and URL are provided to UAES, with the previous URL being *null* since it is the initial state (line 4). The `DiscoverTags` function is then invoked to process the page content (line 5), extracting three navigation menu tags: *Home*, *Find Owners*, and *Veterinarians*. Each tag is processed through the `DefineLocator` function (line 7).

For the first tag, locators are identified, starting with `inner-text="Home"` (lines 24-25). Since `setAE` is empty, no previous AE triples are returned (line 26), allowing the first locator

[2] `https://github.com/spring-projects/spring-petclinic`

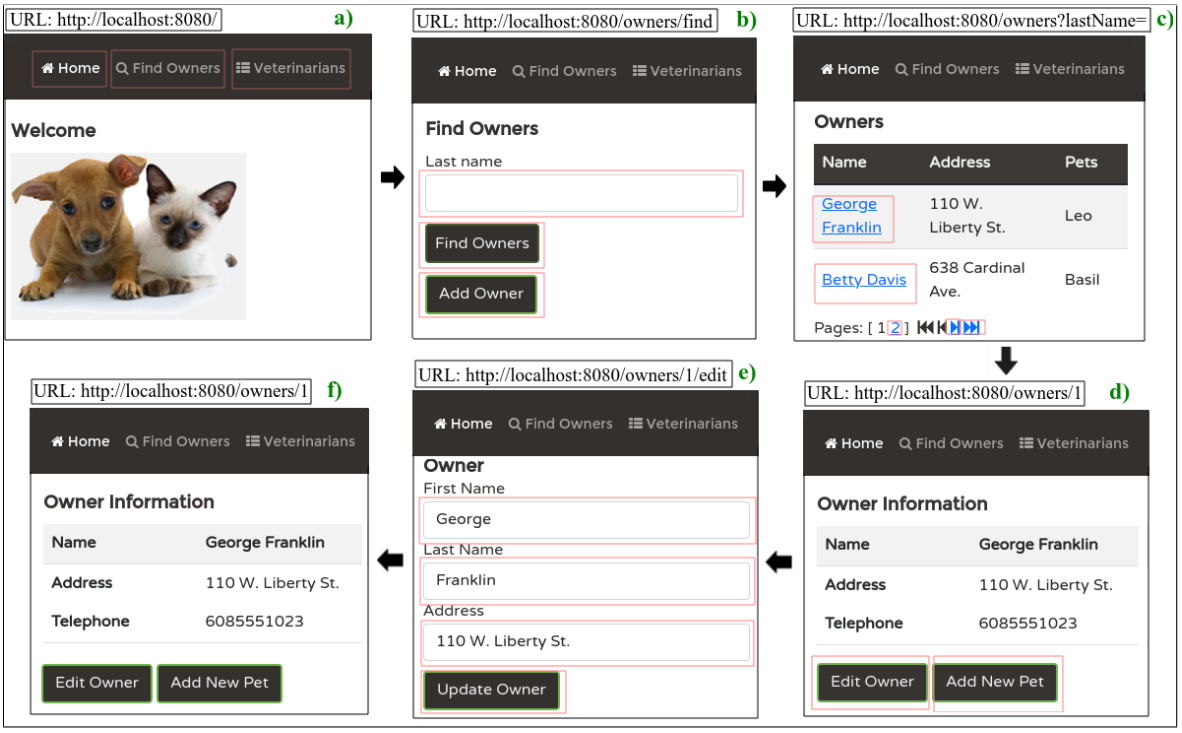

Figure 3.1: Running example to illustrate UAES executing through test generations.

to be successfully obtained and added to `setAE`. This process is repeated for the other two tags. With `setAE` updated, the testing tool can interact with the application. For this example, suppose the `inner-text="Find Owners"` item is clicked.

Upon clicking, the discovery process restarts, invoking UAES again. This time, `DiscoverTags` identifies six elements: three new ones (highlighted in Figure 3.1 (b)) and the three existing menu items. Applying `DefineLocator` to each tag finds three AE triples in `setAE` associated with the previous URL, updating their URLs list (line 30) and skipping the loop for identical elements (line 32). Thus, these three menu items are not added to `setAE`.

One of the newly discovered elements in this state is a button with the *inner text* identical to a static element in the menu, "Find Owners" (see Figure 3.1 (b)). However, it has another available locator key, with `id="findOwners"` being the second locator encountered on line 34 and resulting in its successful addition to `setAE`. Additionally, no *inner text* was found for the input element, but a locator key `id` was identified resulting in a locator such as: `id="lastName"`. Thus, the three new elements have their respective triples added to `setAE`.

In this specific execution, clicking the button with `id="findOwners"` navigates to a new

page with additional elements and resulting in a changed URL (Figure 3.1 (c)). On this page, eight elements are discovered, including the three menu items. These menu items are not added again to `setAE`, but their AE triples are updated to reflect the new state.

Following the flow, by clicking on `inner-text="George Franklin"` brings up the item details state, where two new buttons are added to `setAE`. Subsequently, by clicking on `inner-text="Edit Owner"` transitions to the form state (Figure 3.1 (e)). Here, all inputs and the update button are classified as unique and added to `setAE`. Finally, clicking on `inner-text="Update Owner"` leads to a previously found state (Figure 3.1 (f)). Since all elements are in `previousAE`, no new elements are added to `setAE`.

It is important to note that when the URL does not change between actions, the behavior of UAES remains consistent. The current and previous URLs remain the same, and only new elements found in the new state are added to `setAE`. This allows UAES to efficiently handle interactions within a single page without unnecessary duplication of elements.

This example demonstrates how UAES discovers and interacts with web elements through an execution flow, showcasing its ability to manage state transitions and maintain element locators effectively.

3.1.2 Overview of the UAES Empirical Evaluation

We evaluated the potential of the UAES approach with two empirical studies that compared UAES with the Markup approach. The goal was to assess the effectiveness of unique discovery of actionable elements. The Markup approach requires manual identification and marking of actionable elements in web systems by testers to facilitate GUI testing (more details in Appendix A). The studies utilized a version of Cytestion that integrates both approaches to detect elements across four open-source and twenty industrial web applications, cataloging the discovered elements and categorizing them as new, the same, or missed.

The first study examined four open-source web applications built using SpringBoot, HTML, JavaScript, and CSS. Two testers manually marked the actionable elements, which were then discovered by Cytestion using both the Markup approach and UAES. UAES identified 79.81% of the elements found by the Markup approach, with 8.1% being unique to UAES, revealing overlooked functionalities such as table pagination. However, UAES missed 20.19% of the elements due to inconsistencies in development standards. The study

concluded that UAES offers comparable performance, avoiding manual marking errors and potentially enhancing test suites.

The second study evaluated UAES using twenty industrial web applications developed with React and the Ant Design component library. Testers of the company had previously manually marked elements to enable Cytestion's use. UAES identified 95.30% of the manually marked elements, with 48.4% being unique to UAES, addressing the challenges of manual marking under tight schedules and third-party components. Only 4.7% of elements found by the Markup approach were missed by UAES due to strict adherence to best practices. The study concluded that UAES outperformed the Markup approach in an industrial context, effectively managing the error-proneness and time constraints of manual markup.

Overall, UAES not only improved fault detection by identifying production code faults related to unmarked modal elements, but also streamlined GUI testing processes through automated element discovery, eliminating the need for intrusive source code modifications. These findings evidence UAES's potential to support GUI testing practices, offering a systematic, efficient, and more effective approach to ensuring the quality and reliability of web applications. For more details on these studies, readers are referred to Appendix B.

3.2 Network Wait Mechanism

The *synchronization challenge* refers to scenarios where the tester or automated testing tool does not properly consider response time when creating test cases. This may result in flaky tests and intermittent failures during the execution of the test case. Such issues often arise when test cases attempt to perform actions on a web page not yet ready to handle them, leading to breakage in the test execution. For instance, when a test case logs into a web application and tries to interact with elements on the home page, if the server response is not received quickly enough, it may try to execute instructions before the application's home page is fully loaded. This premature execution can lead to test breakages that mistakenly indicate fault detection [37, 38, 107]. This challenge is discussed in more detail in Section 1.2.2.

The Cypress framework[3] tries to attenuate this issue by executing scripts on the browser

[3] `https://www.cypress.io`

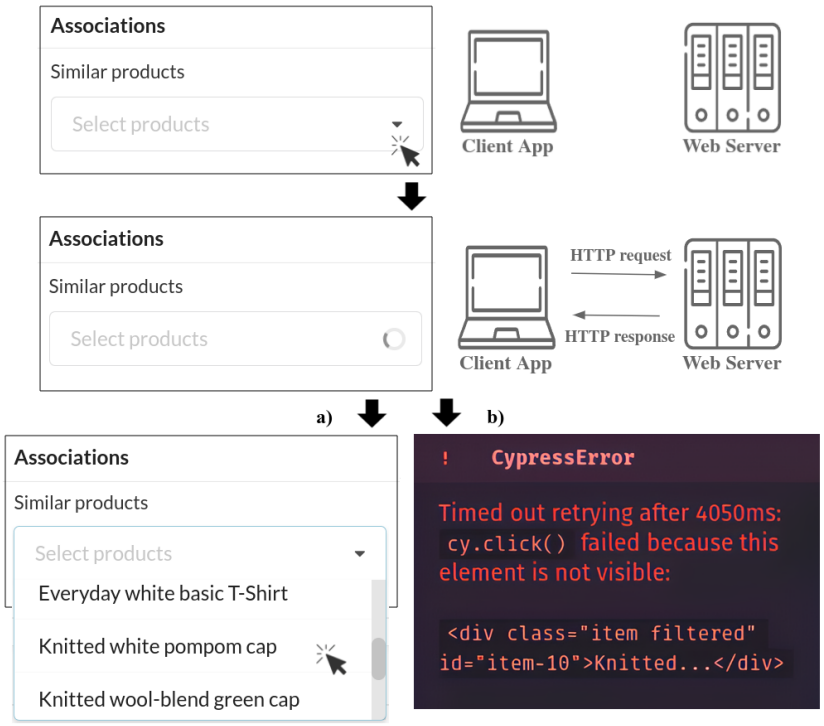

Figure 3.2: Example of a testing action that might fail due to synchronization issues.

side. It discerns the application's transitional state after an action and waits for the complete page to load. However, the *synchronization challenge* remains when the test case interacts with elements that trigger asynchronous calls [32, 88, 112]. Such calls occur whenever a page interaction requires data retrieval from a web server.

Figure 3.2 exemplifies such a scenario. Suppose a Cypress test case tries to interact with the tenth product that appears after accessing the *Select products* drop-down input. The *click* action triggers an asynchronous call that returns the list of registered products. The *select* action becomes viable only after the request is fulfilled and the application lists the found product (Figure 3.2-a). However, if the test does not properly handle synchronization, the test case might try to click on a product before the list is loaded, leading to a test breakage (Figure 3.2-b). Reasons for an application delay that may lead to synchronization issues are numerous (e.g., network latency, slow server response, complex database queries) and often can be neither controlled nor properly anticipated by the tester.

In this context, waiting mechanisms can act as traffic lights for testing, controlling when to stop, when to go, and when to wait. This orchestration may help avoid synchronization issues.

We conducted a thorough literature review to address the *synchronization challenge* in automated GUI tests. This review utilized common keyword searches combined with snowballing techniques, ensuring a comprehensive identification of relevant studies. Our findings led to the creation of a catalog of waiting mechanisms, which includes *Implicit Wait*, *Static Wait*, *Explicit Wait*, *Fluent Wait*, and a Cypress-specific mechanism identified from grey literature: *Stable DOM Wait*. Additionally, we introduce a novel mechanism exclusive to Cypress: the *Network Wait*. For a detailed description of these mechanisms and their implementation in the Cypress framework, please refer to Appendix C. The comprehensive study and results using the mechanisms described in the catalog are provided in Appendix D.

The *Network Wait*[*] is a novel mechanism that closely observes requests made during test execution. Its purpose is to ensure that a test proceeds only when all these requests are completed. The *Network Wait* operates in a simple way: as new requests are initiated, a counter is incremented; upon request completion, the same counter is decremented. The test execution resumes its course when this counter reaches zero, remaining in this state for a predetermined period of time.

```
1  let pendingCount = 0;
2  function routeHandler(request) {
3    pendingAPICount++;
4    request.on('response', () => pendingCount--);
5  }
6
7  cy.intercept('*', '*', routeHandler);
8
9  Cypress.Commands.add('waitNetworkFinished', () => {
10   while (pendingCount > 0) {
11     cy.log('Waiting for pending requests.');
12     cy.wait(500);
13   }
14   cy.log('All requests completed!');
15 });
16
17 cy.get('[id="product-select"]').click();
18 cy.waitNetworkFinished();
19 cy.get('[id="item-10"]').click();
```

Listing 3.1: Simplified implementation of Network Wait and its use in Cypress.

[*]The catalog, the new mechanism, and the studies were published in a conference paper [78].

Listing 3.1 presents an implementation of the *Network Wait* in Cypress. In this implementation, the *Network Wait* is integrated into Cypress using its native `intercept` command (line 6). Each network request increments the `pendingCount`, and upon completion, it decrements. The `waitNetworkFinished` command (lines 9-15) halts test execution until all pending network requests are resolved. This approach ensures tests proceed only when the network state stabilizes, enhancing reliability and reducing the likelihood of synchronization failures.

The *Network Wait* mechanism is particularly beneficial in scenarios where subsequent test actions depend on network responses, common in scriptless GUI testing environments [21]. It dynamically adapts to network conditions, reducing waiting times on faster networks and extending them on slower networks. This adaptability may lead to an effective balance between precision and efficiency during test execution.

In scenarios where network requests are delayed or never answered, such as due to server failures or network disruptions, the test could wait indefinitely, blocking execution. To prevent this, a timeout mechanism could be added to the loop condition (line 10), freeing the test if requests are not completed in a set time. While not included in the simplified version here, this logic exists in the final *Network Wait* mechanism. By incorporating a timeout, tests avoid getting stuck, improving resilience in handling unresponsive requests.

We have also developed and published a dedicated NPM package for the *Network Wait* mechanism, available for the Cypress community[5]. Since its release, the package has garnered significant interest, being downloaded almost 2000[*] times within days of publication, without any advertising, highlighting its practical relevance and adoption in automated testing scenarios.

3.2.1 Overview of Empirical Studies using the Network Wait

We conducted two empirical studies to evaluate the potential of the Network Wait mechanism. These studies focused on assessing the impact of synchronization issues on test suites and the effectiveness of various waiting mechanisms in addressing these challenges in the Cypress context. In the first study, we utilized a test suite generated by students and exe-

[5] `https://www.npmjs.com/package/cypress-network-wait`

[*] Data collected in June 2024.

cuted it multiple times, gradually introducing delays. We found a test suite breakage rate of 16%. After creating versions of the test suite using different waiting mechanisms — *Static Wait*, *Explicit Wait*, *Network Wait*, and *Stable DOM Wait* — both *Network Wait* and *Explicit Wait* demonstrated zero breakages across all executions and delays.

In the second study, we used a real test suite from a partner company and created multiple versions of this test suite, each employing a different waiting mechanism — *Static Wait*, *Explicit Wait*, *Network Wait*, and *Stable DOM Wait*. The original version of the test suite exhibited a breakage rate of 32%. Both *Network Wait* and *Explicit Wait* mechanisms reduced the breakage rate to 2%, uncovering new waiting points not previously detected by the QA team. *Network Wait* and *Explicit Wait* emerged as promising strategies with equal breakage rates for test cases and suites. *Explicit Wait* showed better execution times, indicating its efficiency in handling synchronization issues. However, it inherently adds complexity to the test design process, often requiring human involvement to accurately define the appropriate wait conditions. For more details on these studies, readers are referred to Appendix D.

3.3 Iterative Deepening URL-Based Search

Systematic exploration poses a significant challenge due to its inherently lengthy process, leading to inefficiencies in GUI testing. Tools employing this technique must navigate through the AUT, interacting with numerous elements, which results in an exponential growth of test cases and runtime. In the development of Cytestion, we approached this challenge by conceptualizing exploration as a graph problem. Initially, we adopted the IDS algorithm to incrementally explore unknown-depth search spaces (presented in Section 2.2.1). However, the complexity of real-world systems highlighted the temporal challenges inherent in applying such techniques to GUI testing contexts. This challenge is discussed in more detail in Section 1.2.3.

To address extended runtime issues, we introduce the Iterative Deepening URL-Based Search (IDUBS)* algorithm, which differs from IDS by retaining minimal information from prior nodes to establish a fresh starting point in the depth search. This eliminates redundancies upon returning to the initial node during subsequent iterations. IDUBS is agnostic and

*This algorithm along with its evaluation studies were published in a conference paper [80].

can be used in different web contexts, including testing, crawling, and data mining applications. Its effectiveness, simplicity, and scalability make it a versatile solution.

Our goal is to optimize test case execution in GUI tests by using IDUBS, which can help avoid redundancy of accessed GUI states and ultimately reduce execution time. The proposed algorithm integrates each graph node that represents a GUI state with an associated URL. When a new URL is discovered, its node becomes a new root. During exploration, the algorithm can access the previously obtained state from the previous iteration by directly accessing this new root and continuing the search in that branch. This approach is feasible in the context of web systems, as direct visits to URLs provide efficient access to specific states of the AUT and are aligned with contemporary web practices [111, 116].

Listing 3 presents the IDUBS algorithm with support for multiple goal states. The algorithm initializes two empty lists: `roots`, which tracks root nodes, and `goalNodes`, which stores discovered goal nodes (lines 1-2). In the main function, `IDUBS`, the root node and the goal state are received as arguments (line 3). It begins by setting the initial depth to zero and assigning the level value to the root node, marking its position. After that, the root is added to the `roots` list (lines 4-6). During each iteration of depth, it utilizes all nodes in this list and calls the DFS function while passing the parameters `node`, `goal`, and $|depth - node.level|$ (line 9). This subtraction ensures that the search will adhere to the depth limit even when a deeper root node is used.

The `DFS` function recursively explores nodes in the tree up to a specified depth. If the depth is zero, indicating the end of exploration for this branch, it checks if the current node is a *goal node* (line 21). If so, the node is added to the `goalNodes` list (line 22). Subsequently, it returns `true` to evaluate possible children in the next iteration (line 24). If the depth is greater than zero, the function explores all child nodes of the current node to check if it is present in the `roots` list (lines 27-28). This presence indicates that this child has been found and considered a starting point in previous iterations, being used as a new root and having its own separated flow. It occurs due to a difference in the node URL and the child URL being found (lines 30-32). This change indicates that the node can be directly accessed in the next iteration.

It is important to note that new roots are found in deeper levels of the tree. To handle this and ensure that all flows respect the depth limit, it is crucial to save the level of each child

Algorithm 3 The IDUBS with Multi-Goals Algorithm

```
1:  goalNodes ← []
2:  roots ← []
3:  function IDUBS(root, goal)
4:      depth ← 0
5:      root.level ← 0
6:      roots.add(root)
7:      while roots is not empty do
8:          for all node in roots do
9:              remaining ← DFS(root, goal, depth − node.level)
10:             if not remaining then
11:                 roots.remove(node)
12:             end if
13:         end for
14:         depth ← depth + 1
15:     end while
16:     return goalNodes
17: end function
18:
19: function DFS(node, goal, depth)
20:     if depth = 0 then
21:         if node is a goal then
22:             goalNodes.add(node)
23:         end if
24:         return TRUE
25:     else if depth > 0 then
26:         anyRemaining ← FALSE
27:         for all child of node.children do
28:             if child not in roots then
29:                 child.level ← node.level + 1
30:                 if child.url ≠ node.url then
31:                     roots.add(child)
32:                 end if
33:                 anyRemaining ← DFS(child, goal, depth − 1)
34:             end if
35:         end for
36:         return anyRemaining
37:     end if
38: end function
```

node by adding its parent node's level plus one (line 29). Subsequently, a DFS call is made for the child, which returns a boolean value to `anyRemaining` (line 33). When it remains `false`, this indicates that no new nodes were found in this branch, signaling that the node can be removed from the root list (lines 10-12). Eventually, when no new nodes are found in any branch, this list will become empty. This culminates with the end of the search and results in returning `goalNodes`.

To properly explore the GUI and reveal faults, it is important to adhere to the test case execution order proposed by IDUBS. Faults can manifest in two scenarios: when initially accessing a faulty state or when directly accessing a previously visited state. The latter can be achieved through direct URL access, which may cover different parts of the code. This is particularly evident in systems developed with modern web frameworks which enable server-side rendering and efficient data binding [44]. Code parts may only be accessed through direct URL accesses, due to the way such frameworks handle routing, state management, and data binding. Direct URL access may involve more server-side processing and additional code execution to render the desired GUI state.

3.3.1 Running Example

To demonstrate the execution of the IDUBS algorithm with multiple goals, we reuse the example presented in Section 2.2.1 and Figure 2.2. Our goal remains to identify all GUI states where failures occur. We start with a depth limit of zero, including only the `root` in the `roots` list. Since A is not a goal node, we only move on to the next depth level by returning the boolean value `true` (line 24). At depth one, we again perform a DFS passing through the root node A. We visit its children B and C since these are not included in `roots` and have different URLs compared to A, both are added to `roots`. Furthermore, node C is identified as a goal node due to a visible failure, so it is added to the list of goal nodes.

At depth two, there are nodes A, B, and C as roots (line 8), so three different DFS calls are made. Since both B and C are found at level one, their passed depth is decremented to one. The DFS for node A finds all children inside the `roots`, while the C's DFS finds no child. Both have `false` in $anyRemaining$ and are removed from the `roots` list. The DFS of B finds the nodes D and E. None of them are included in the `roots` list. However, D presents a different URL (Figure 4.1-Depth 2) which leads to its inclusion in the roots list.

Both of these nodes are not goal nodes, prompting us to move on to the next iteration.

At depth three, nodes B and D serve as roots, leading to two different DFS calls. Node D was found at level two, so we use a depth of one (line 9). The DFS for node B identifies its child node E, which was not included in the list of `roots`. Another DFS call for node E results in finding its child node G. Since G is not included in the `roots` list and has a different URL, it is added to `roots` list. Simultaneously, during the DFS of D, a child node F is found that is not included in the root nodes. None of these discovered nodes are goal nodes, then we just move to the next iteration.

At depth four, we have the nodes B, D, and G as roots, so three different DFS calls are made. The DFS of B goes to E and does not find any child nodes not included in the `roots`, therefore being removed. The DFS of D goes to F and does not find any child nodes at all, also being removed. The DFS of G finds child nodes H and I with different URLs. They are pointed as roots and H is found as a goal node. One more iteration is performed with roots G, H, and I which are then all removed from the root, finalizing the search result by returning the goal nodes C and H.

Based on the given execution, the test suite generated by IDUBS has the following test sequences: (1) A; (2) $A \rightarrow B$; (3) $A \rightarrow C$; (4) $B \rightarrow D$; (5) $B \rightarrow E$; (6) $B \rightarrow E \rightarrow G$; (7) $D \rightarrow F$; (8) $G \rightarrow H$; (9) $G \rightarrow I$. Comparing to the one presented in Section 2.2.1, we can see that the new suite is composed of smaller test cases with fewer state repetitions that uses direct accesses to nodes.

3.3.2 Overview of the IDUBS Empirical Evaluation

The evaluation studies conducted to assess the effectiveness of the IDUBS algorithm in GUI testing involved two distinct empirical studies: one focusing on twenty industrial web applications and the other on four open-source web applications. The primary objectives of these studies were to compare IDUBS with the baseline strategy IDS across several metrics, including test case execution time, number of revisited states, test suite coverage, and the number of detected faults.

In the first study, which examined the industrial applications, IDUBS achieved a significant reduction in test execution time by 43.60% and decreased test case redundancy by 50%. The study revealed that IDUBS maintained comparable code coverage levels to IDS while

detecting a total of 317 faulty states, including critical issues that had previously been overlooked by the quality assurance process. This demonstrated IDUBS's capability to enhance fault detection in real-world scenarios.

The second study focused on four open-source web applications, where IDUBS reduced execution time by 39.03% and minimized test case redundancy by 36.01%. Although the open-source study did not yield as many detected faults as the industrial study, it still confirmed IDUBS's effectiveness in reducing testing costs and maintaining performance metrics. Overall, the results from both studies underscore the effectiveness of IDUBS in optimizing testing processes. By reducing testing costs, improving fault detection, and enhancing overall performance in GUI testing scenarios, IDUBS presents a valuable advancement in automated testing methodologies. For more details on these studies, please refer to Appendix E.

3.4 Concluding Remarks

In this chapter, we have explored innovative solutions aimed at overcoming the intricate challenges inherent in scriptless GUI testing, particularly the need for a systematic approach. The UAES algorithm efficiently discovers unique actionable elements, while the Network Wait mechanism optimizes synchronization by dynamically managing network requests. Each of these solutions enhances the effectiveness and reliability of scriptless testing tools like Cytestion. Additionally, the IDUBS algorithm offers a scalable method for exploring diverse GUI states, minimizing redundancy and improving test case execution. We have leveraged this algorithm to develop our approach, providing a practical and simplified implementation. These advancements not only address critical pain points in GUI testing but also lay a foundation for more robust and reliable testing methodologies, ultimately advancing the field towards greater automation and reliability.

Furthermore, the proposed solutions have broader applications beyond just GUI testing. The UAES algorithm's principles, which focus on systematically discovering actionable elements, can be adapted for automated web scraping and Robotic Process Automation (RPA), where thorough and precise interaction with web elements is crucial. The *Network Wait* mechanism, with its dynamic approach to managing network requests, offers improvements in synchronization for various network-dependent applications. Additionally, the scalable

state exploration method of the IDUBS algorithm is valuable in contexts that require exhaustive exploration of state spaces and where accessing any point within that space is necessary through node information.

Chapter 4

Cytestion

This chapter introduces Cytestion, an approach and tool designed to automatically detect faults that cause visible failures in web applications. It utilizes the UAES for comprehensive element discovery, the Network Wait mechanism for robust synchronization, and a practical adaptation of the IDUBS algorithm for efficient state exploration. Cytestion systematically explores web-based GUIs and dynamically builds test suites to streamline the testing process.

4.1 The Cytestion Approach

We propose Cytestion[*], an approach that systematically explores a web-based GUI and dynamically builds a test suite for detecting faults that cause visible failures. We use a practical and simplified adaptation of the IDUBS algorithm to achieve this. From an initial URL, Cytestion analyzes all actionable elements of the page and, while exploring them, generates and executes new test cases.

Listing 4 presents the pseudo-code of the Cytestion algorithm. The algorithm first initializes three sets (line 1): T_f, the set that will compose the final test suite; T, the set of generated but not yet executed test cases; and AE, the set of actionable elements identified so far. The `cytestion` function creates an initial test case, $test_0$, that accesses the URL provided as input and waits for the state to be fully loaded using the Network Wait mechanism (Section 3.2). To execute this test, it uses the `executeTest` function. Then, the function enters a loop until no tests are yet to be executed ($T \neq \varnothing$). In each iteration, a $test$ case is

[*]The initial version of Cytestion along with its evaluation studies were published in a conference paper [79].

Algorithm 4 Cytestion pseudo-code

```
1: T_f ← {}, T ← {}, AE ← {}
2: function CYTESTION(url)
3:     test_0 ← {access(url), waitNetwork()}
4:     EXECUTETEST(test_0)
5:     while T ≠ ∅ do
6:         test ← getElement(T)
7:         EXECUTETEST(test)
8:         remove(test, T)
9:     end while
10:    return T_f
11: end function
12: procedure EXECUTETEST(test)
13:    state ← execute(test)
14:    if oracle(state) = PASS then
15:        GENERATETEST(test, state)
16:    else
17:        T_f ← T_f ∪ {test}
18:    end if
19: end procedure
20: procedure GENERATETEST(test, state)
21:    AE' ← UAES(state, state.url, state.previousUrl)
22:    if AE' = ∅ then
23:        T_f ← T_f ∪ {test}
24:    else
25:        while AE' ≠ ∅ do
26:            ae ← getElement(AE')
27:            if state.url ≠ state.previousUrl then
28:                test' ← {access(state.url), waitNetwork(), interact(ae), waitNetwork()}
29:            else
30:                test' ← test ∪ {interact(ae), waitNetwork()}
31:            end if
32:            T ← T ∪ {test'}
33:        end while
34:    end if
35: end procedure
```

selected from T (`getElement` - (line 6)), the selected test is executed (line 7), and removed from the T set (line 8).

The `executeTest` function executes a given $test$ case in the AUT and gets to the resulting $state$ (line 13). The `oracle` function evaluates the $state$ to verify whether the test case passed or failed based on three sources of information. A test case is decided as failing if it generated: (i) a browser console failure message; (ii) a HTTP status code of the 400 or 500 families after server requests; or (iii) a default or customized failure message in the GUI, such as "Error" and "Exception".

For the third criterion, the `oracle` searches for the presence of predefined failure strings in the application state. While "Error" and "Exception" are suggested defaults, the list of failure strings is customizable to better suit the application's context. Testers can add or remove strings based on their specific needs to avoid false positives or capture relevant failures. Instructions on configuring these strings in our tool are detailed in Section 4.3, ensuring that the oracle's behavior can be adapted to the unique characteristics of the AUT. By using this oracle strategy, Cytestion is able to detect a wide range of faults that can manifest in different ways.

If the test case passes, the `generateTest` function is called to create new test cases (line 15). Otherwise, the failed test case is added to the T_f set (line 17). The `generateTest` function discovers actionable elements in the current $state$ that are not present in the AE set using the `UAES` function (line 21), which was detailed in Section 3.1. The unique actionable elements discovered are then placed into the set AE'.

If no new actionable elements are found, the current test case is added to the T_f set (line 23). However, if new actionable elements are discovered, the function generates new test cases for each element obtained using the `getElement` function (lines 25-26). Then, we apply a simplification of IDUBS (detailed in Section 3.3), which is possible because we are storing test case actions in memory, allowing us to reset the test and start a visit to any place of the AUT using the URL. In lines 27 to 31, when the actual state has a different URL compared to previous states, we create a test case that accesses a new URL, waits for a state update, performs a new interaction, and again waits for a state update. If it is detected that URLs remain unchanged between states, subsequent test cases execute both previous actions of $test$ alongside any upcoming interactions utilizing the `interact` and

`waitNetwork` functions.

These newly generated test cases are added to the T set (line 32). The algorithm continues iterating through the T set until it is empty, at which point the `cytestion` function returns the final set of test cases, T_f, which represents the test cases that explore all reachable states of the AUT. It is important to clarify that Cytestion's exploration is limited to pages within the initial URL and does not extend to external websites.

4.2 Running Example

To illustrate Cytestion's test generation process, consider the example presented in Figure 4.1. For that, we reuse the system presented in Section 1.1. The initial URL of the system is provided to the `cytestion` function, and an initial test case ($test_0$) is created (line 3) and executed. In Figure 4.1 (a), the state ($state_0$) found at line 13 displays a home page with three menu items, each being a valid and unexplored actionable element. After evaluating $state_0$ (line 14), no faults were found, resulting in the invocation of the `generateTest` function at line 15. The `UAES` function is executed, which produces the set AE' and AE = $\{ae_1, ae_2, ae_3\}$, leading to three new test cases and the update of T = $\{test_1, test_2, test_3\}$.

With the T set populated, we enter the test case generation loop (line 5). The first test case, $test_1$, is selected from T, and the `executeTest` function is called (line 7). This test case involves four steps: accessing the URL, waiting for an updated state, clicking on the *Home* menu item to reach state $state_1$ (Figure 4.1 (b)), which is then evaluated by the oracle (line 14) and no faults were found. After that, the `generateTest` function is called. Following `UAES` execution, no new elements are found as all three menu items already exist in the previous state. Therefore, this test case is included in the final test case set (T_f) at line 23.

Next, test case $test_2$ is selected from T and executed. Two actionable elements are discovered: the *Last name* input and the Find Owners button. This results in AE' = $\{ae_4, ae_5\}$ and AE = $\{ae_1, ae_2, ae_3, ae_4, ae_5\}$. Test cases $test_4$ and $test_5$ are then created and added to T. Following this, $test_3$ is selected from T for execution. During evaluation a fault is detected in the server response (Figure 4.1 (d)). Thus, this test case is included in the final set of test cases (T_f).

Subsequently, t_4 is selected from T and executed. During the evaluation, a fault is de-

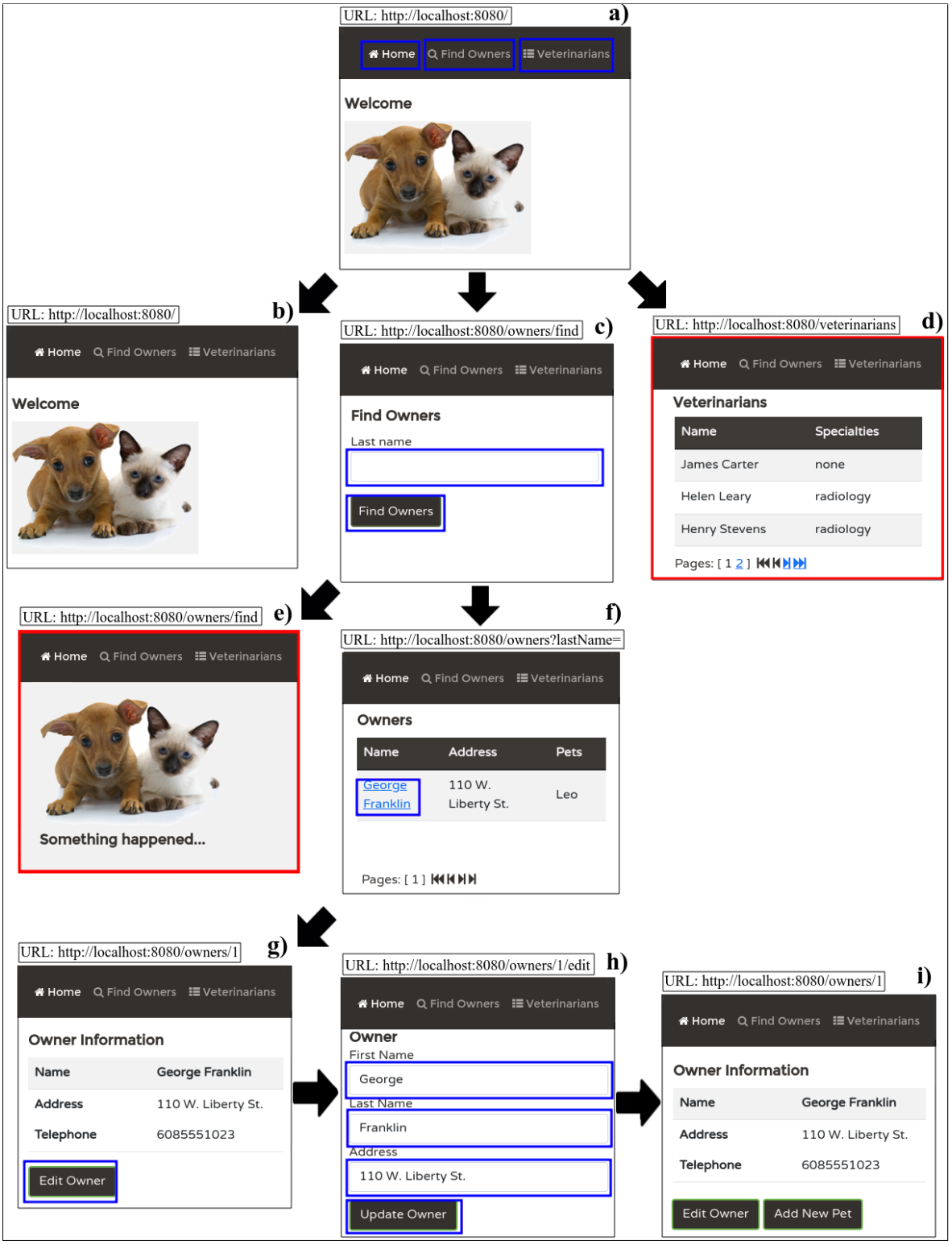

Figure 4.1: Running example to illustrate the test generation and execution process of Cytestion.

tected from the error message presented in the GUI (Figure 4.1 (e)). This test case becomes part of the final test case set (T_f). Next, test case $test_5$ is selected from T and executed. It passes the oracle evaluation and leads to a call to the `generateTest` function. In $state_5$, presented in Figure 4.1 (f), one new actionable element (*George Franklin* anchor) is discovered, updating AE' to $\{ae_6\}$ and A to $\{ae_1, ae_2, ae_3, ae_4, ae_5, ae_6\}$ while generating $test_6$ as a new test case.

Still in the loop, $test_6$ is selected from T, executed successfully, and a new state $state_6$ is found with an additional actionable element. The sets AE' and AE are updated to $\{ae_7\}$ and $\{ae_1, ae_2, ae_3, ae_4, ae_5, ae_6, ae_7\}$. Subsequently, the test case $test_7$ is generated and added to T. Upon execution of this test case, it produces a state featuring four actionable elements as shown in Figure 4.1 (h). However, the three inputs are grouped into the same test case. This results in creating the cases $test_8$ and $test_9$, which are added to T.

In the following executions, $test_8$ found the same state with no new actionable elements, which were added to T_f. Additionally, $test_9$ found the state in Figure 4.1 (i) with the same URL and elements as those found in the state in Figure 4.1 (g). `UAES` ensures uniqueness and does not provide any new actionable element. Therefore, the test case is added to T_f (line 23). When the loop exits, Cytestion returns the test suite $T_f = \{test_1, test_3, test_4, test_8, test_9\}$, which represents the entire exploration of the web application.

4.3 The Cytestion Tool

We implemented the Cytestion approach as a Node.js open-source tool[2]. This tool can be executed through a command line interface and utilizes the Cypress framework to execute the generated test cases. By doing so, we achieve the ability to record the test executions and also create legible and re-executable test scripts. It also utilizes the dependencies for employing the Network Wait mechanism[3] and enables parallel execution through the *cypress-parallel* dependency[4].

In terms of actionable elements, the tool uses a default UAES implementation that covers buttons, links, menu items, and form inputs, as found by the HTML element snippets

[2] `https://gitlab.com/lsi-ufcg/cytestion/cytestion`

[3] `https://www.npmjs.com/package/cypress-network-wait`

[4] `https://www.npmjs.com/package/cypress-parallel`

presented in Table 3.1. For the first three types of elements, the tool attempts to perform a click action on the web element. This process involves several validations to ensure that the element is ready for interaction: checking if the element exists in the HTML, if it is visible on the GUI, if it is enabled, and if it is not covered by another web element. These steps are essential to avoid false negatives and prevent test execution failures caused by attempts to interact with elements that are not fully actionable in the current state of the GUI.

Beyond these common HTML elements, the UAES implementation is designed to handle all possible elements in modern web applications, including those created with non-default tags such as *div* and *span*, which are often used to represent clickable components. The tool achieves this flexibility by combining native Cypress functions with customized implementations developed to handle more complex web elements. This approach ensures that the tool can adapt to a wide variety of representations for buttons, links, menu items, and form inputs, allowing for robust test case generation across different web environments.

When interacting with form inputs, elements from the same state are grouped together. For example, a state might include a text field for *Name*, a date picker for *Date of Birth*, and a dropdown for *Country*. The tool generates data for each input type based on the *type* attribute specified in standard HTML[5]. For custom or non-standard inputs, a configuration mapping feature allows testers to define unique classes or attributes used in their system. Data generation is handled using the faker-br library[6], which creates random values for inputs. However, since the data is randomly generated, it may not always meet form validation requirements, which can limit the ability to fully test form functionality without prior knowledge of the system.

Figure 4.2 illustrates the architecture of Cytestion. The tool starts by receiving input from a .env file (Figure 4.2 (a)), which includes crucial information such as the URL of the web application, the URLs of the server(s) that the application communicates with, login credentials if needed, customized failure messages that might appear in the GUI, specific HTTP codes to ignore when deciding if a test case fails, and the path to a directory for storing output artifacts. Utilizing this input, Cytestion starts its exploration by generating an initial test case as a Cypress script (Figure 4.2 (b)). This script comprises functions to

[5] `https://www.w3schools.com/html/html_form_input_types.asp`

[6] `https://www.npmjs.com/package/faker-br`

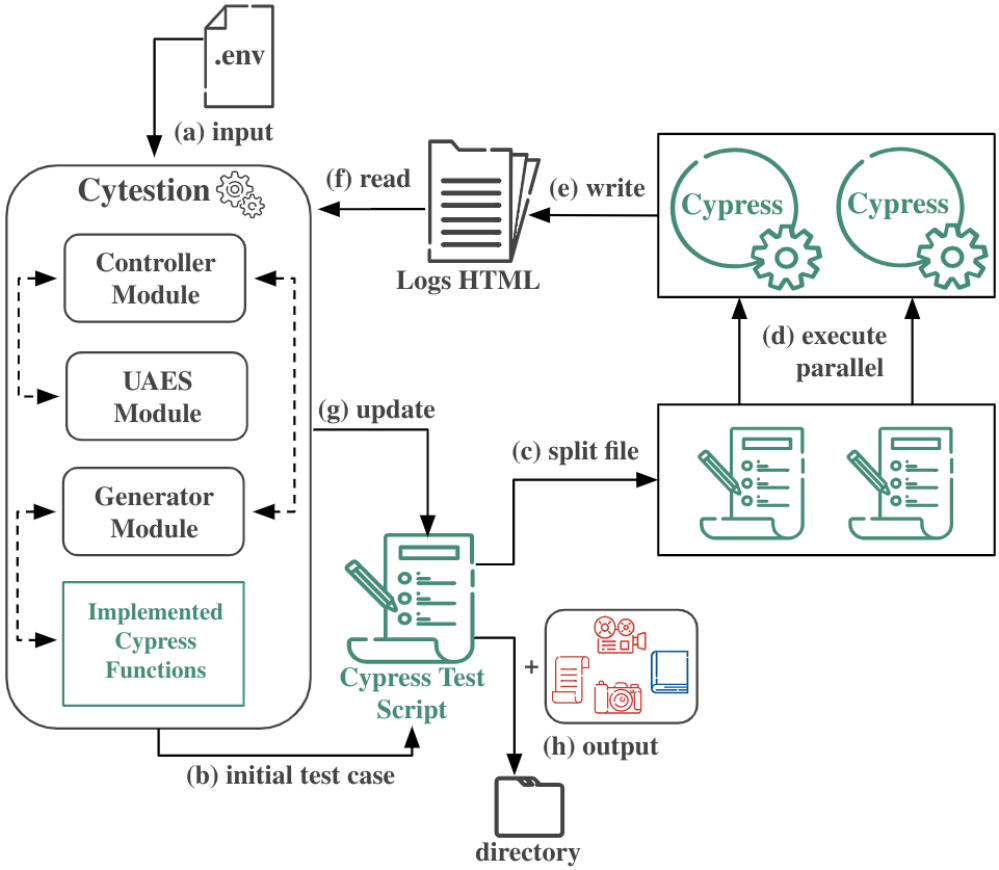

Figure 4.2: High-level view of the tool architecture.

visit the URL of the web application, perform login if required, wait for network activities to complete, and execute the oracle validation.

Once the initial Cypress test script is generated, it is split for parallel execution (Figure 4.2 (c)) and executed (Figure 4.2 (d)). The creators of the *cypress-parallel* dependency claim it can reduce execution time by up to 40% on the same machine, and during our tests, we observed similar reductions. Our tool leverages this dependency by splitting the ongoing Cypress test file into multiple segments, allowing parallel execution. The dependency scans the project directory, identifies all Cypress files, and distributes them across multiple threads to balance the load and optimize execution time. Each thread runs its assigned test cases concurrently, and the results are then consolidated into a single comprehensive report stored in the same location.

The execution results in HTML log files, which are then written to a directory (Figure 4.2 (e)). Cytestion reads and analyzes these log files to discover actionable elements on the web page (Figure 4.2 (f)). For the next generations, Cytestion explores the actionable elements

and generates new test cases, which are copies of their parent test added to the new element action. These new test cases are added to the same Cypress test script created initially (Figure 4.2 (g)).

Cytestion's *UAES Module* is responsible for uniquely discovery the actionable elements, which are then passed on to the *Controller Module*, which controls the explorations, and then to the *Generator Module*. The *Generator Module* uses implemented Cypress functions to create new actions.

As output (Figure 4.2 (h)), Cytestion provides a Cypress test script to execute the generated suite. Additionally, it offers a catalog file containing all actionable elements of the system (highlighted in blue), structured as a map with URLs as keys and lists of elements as values. Each element in the list includes a locator and the corresponding HTML tag. In case that faults are detected during the generation process, Cytestion also provides a set of artifacts to assist testers in reproducing failing scenarios and gaining better insight into identified faults: a separate Cypress test file for running only the failed cases; screenshots of failures; and a video illustrating the steps leading to each failure. All these artifacts are saved in the folder given as input.

4.4 Concluding Remarks

This chapter presents Cytestion, an approach and tool specialized in systematic and automated GUI testing. It integrates the UAES approach for unique discovery of actionable elements, ensuring comprehensive coverage of actionable elements and finite executions. To manage synchronization, we incorporate the Network Wait mechanism, strategically placed after each action to ensure element discovery and fault detection in the right moment. Additionally, we adapt the IDUBS algorithm to systematically generate test cases, enabling direct state access when URLs differ. With Cytestion established, the next chapter will evaluate its efficacy and costs compared to TESTAR, assessing its practical impact in open-source and industrial applications.

Chapter 5

Evaluation Studies

In this chapter, we present a set of empirical studies designed with the goal of evaluating Cytestion's capacity and costs for detecting faults that cause visible failures. Although the Cytestion tool offers a feature that can restrict the exploration to a single URL, which reduces the risks of state explosion, we chose not to use it in our studies to access Cytestion's fault detection capabilities entirely. To guide this investigation, we established two research questions:

- RQ_1: Is Cytestion effective for detecting faults that cause visible failures?
- RQ_2: How costly is it to use Cytestion?

RQ_1 refers to the detection capacity of our approach/tool, while RQ_2 refers to its execution time, which can be a key factor when examining the costs of adopting an automatic and systematic tool.

To answer these questions, we ran two empirical studies. The first deals with injected faults on open-source projects, while the second presents an empirical investigation of Cytestion's use in a set of industrial projects.

In both studies, we selected the TESTAR tool for web applications [112] as a baseline. TESTAR is a state-of-the-art tool for automated GUI testing and has goals similar to Cytestion. Its continued relevance is highlighted by its adoption in various studies [20, 21, 23, 70, 95]. We chose TESTAR over other tools such as Crawljax [75], WEBMATE [27], and Murphy [4] due to their limitations, including lack of active maintenance, license

constraints, or insufficient support for web applications. For our studies, we used version 2.5.3 of TESTAR[1][2].

Our empirical studies executed on a desktop with an Intel Core i7 10700KF processor, 32GB of RAM DDR4 3200MHz, an Nvidia GTX 1060 6GB GDDR5 video card and a SATA SSD 1TB 500Mpbs/s.

For Cytestion, we used its default configuration, which involves running two test cases simultaneously to optimize the exploration of the GUI elements and reduce execution time.

5.1 A Study with Injected Faults

In our first study, we worked with open-source projects and injected faults. Four open source web applications were selected as objects: i) *petclinic*, as introduced in Section 1.1, a Spring Boot application that manages pet owner registrations and schedules veterinary visits; ii) *bistro restaurant*, a website made with HTML, CSS and JavaScript that works as a portfolio for restaurants; iii) *learn educational*, a responsive website that can be used as a portfolio for educational courses; and iv) *school educational*, a website in HTML5 that employs a set of common features for school websites. These applications are web systems used for academic purposes[3]. Table 5.1 provides information about the projects, including their size (KLOC), the number of actionable elements, the number of faults we injected in our study, and the number of test cases generated and executed by Cytestion. Despite their simplicity, those systems offer navigation functionalities, expose relevant information, and support registration operations, which often lead to faults that cause visible failures.

Prior to this study, we ran a side activity with a group of 52 students with web-development experience where we asked them to identify in the projects opportunities to inject faults that cause visible failures. They were asked to propose mutants in the code that emulate faults that they frequently make while developing web applications. From this study, we collected 165 faults to be injected into the systems (see Table 5.1). Those faults cover a variety of scenarios, such as accessing non-existent or inaccessible variables and functions, erroneous file paths/routes, and the omission of annotations that map entities in a database.

[1] `https://github.com/TESTARtool/TESTAR_dev/releases/tag/v2.5.3`

[2] `https://hub.docker.com/r/testartool/testar-chromedriver`

[3] `https://gitlab.com/lsi-ufcg/cytestion/new-cytestion-study/applications`

Project	KLOC	# of Actionable Elements	# of Injected Faults	# of Generated Tests
petclinic	25.7	183	45	130
bistro restaurant	33.4	220	40	218
learn educational	19	253	45	246
school educational	30.2	240	35	238

Table 5.1: Objects of the study with injected faults.

All injected faults were manually validated as causing visible failures, meaning that they are faults and could be detected by GUI testing. Furthermore, they closely resembled the real faults found in our study with industrial applications (Section 5.2). A description of the injected faults and our script for injecting faults are available on our website[4].

Figure 5.1 exemplifies two injected faults in the *bistro restaurant* and *petclinic* systems, respectively. Figure 5.1 (a), shows the use of an erroneous file path that was exposed during navigation through a link, while Figure 5.1 (b) presents a scenario in which the `GenerateValue` annotation for the `id` attribute of an entity was omitted, which leads to the failure to insert a record and to the loss of the information submitted by the user.

a)
```
<li><a href="pricing.html">Pricing</a></li>
<li><a href="contact.html">Contact</a></li> -
<li><a href="contact.hml">Contact</a></li> +
</ul>
```

b)
```
public class BaseEntity implements Serializable {

@Id
@GeneratedValue (strategy = GenerationType.IDENTITY) -
private Integer id;
```

Figure 5.1: Examples of injected faults.

To work with a more controlled scenario, we implemented a script that injected one fault at a time in each application, and then ran both the Cytestion and TESTAR tools. To prevent interference from previous faults, we reset the application source code after each execution

[4] `https://gitlab.com/lsi-ufcg/cytestion/new-cytestion-study/execute-study`

before introducing the next fault. We collected the output from each execution, including whether the tool detected the inserted fault and its execution time. It is important to note that both TESTAR and Cytestion use an oracle configured to detect suspicious texts in the GUI and classify the execution as a failure if any are found. Therefore, we used the same configuration for suspicious texts in the GUI for both tools. However, Cytestion's oracle has the additional ability to identify failures through request status and console error messages.

The projects used in our study exhibit a limited number of actions, with an average of 224 actionable elements. Cytestion's find and interact with each one, while TESTAR works with a predefined number of possible actions. Therefore, for this study, we executed TESTAR six times, using 100, 200, 500, 1,000, 2,000, and 4,000 actions. By doing so, we aimed to maximize the likelihood of TESTAR finding the injected faults and acquire a fair comparison with Cytestion. The results presented consider all these TESTAR executions, compared to the single execution of Cytestion.

5.1.1 Results and Discussion

Figure 5.2 presents the number of detected faults per system and tool across four systems. Each subplot (a, b, c, d) shows the number of faults detected by different configurations of TESTAR and by Cytestion, alongside the total number of injected faults for each system.

In the *petclinic* system, TESTAR detected between 23 and 33 faults, while Cytestion detected 33 out of 45 injected faults. For the *bistro restaurant* system, TESTAR detected 14 to 26 faults, and Cytestion detected all 40 injected faults. In the *learn educational* system, TESTAR detected 16 to 26 faults, and Cytestion detected 28 out of 45 injected faults. In the *school educational* system, TESTAR detected 11 to 26 faults, and Cytestion detected 35 out of 35 injected faults. The results show that TESTAR's fault detection stabilized between 2,000 and 4,000 actions in three out of four systems, suggesting that increasing the number of actions beyond 2,000 did not uncover additional faults. This highlights a potential limitation in TESTAR's efficiency.

Overall, Cytestion detected 83% of all injected faults, whereas TESTAR's best configuration (4,000 actions) detected 67.2% of these faults. Cytestion outperformed TESTAR in any configuration across three of the four systems. Notably, Cytestion detected 94.45% of the faults that TESTAR detected, while TESTAR detected 77.83% of the faults found by

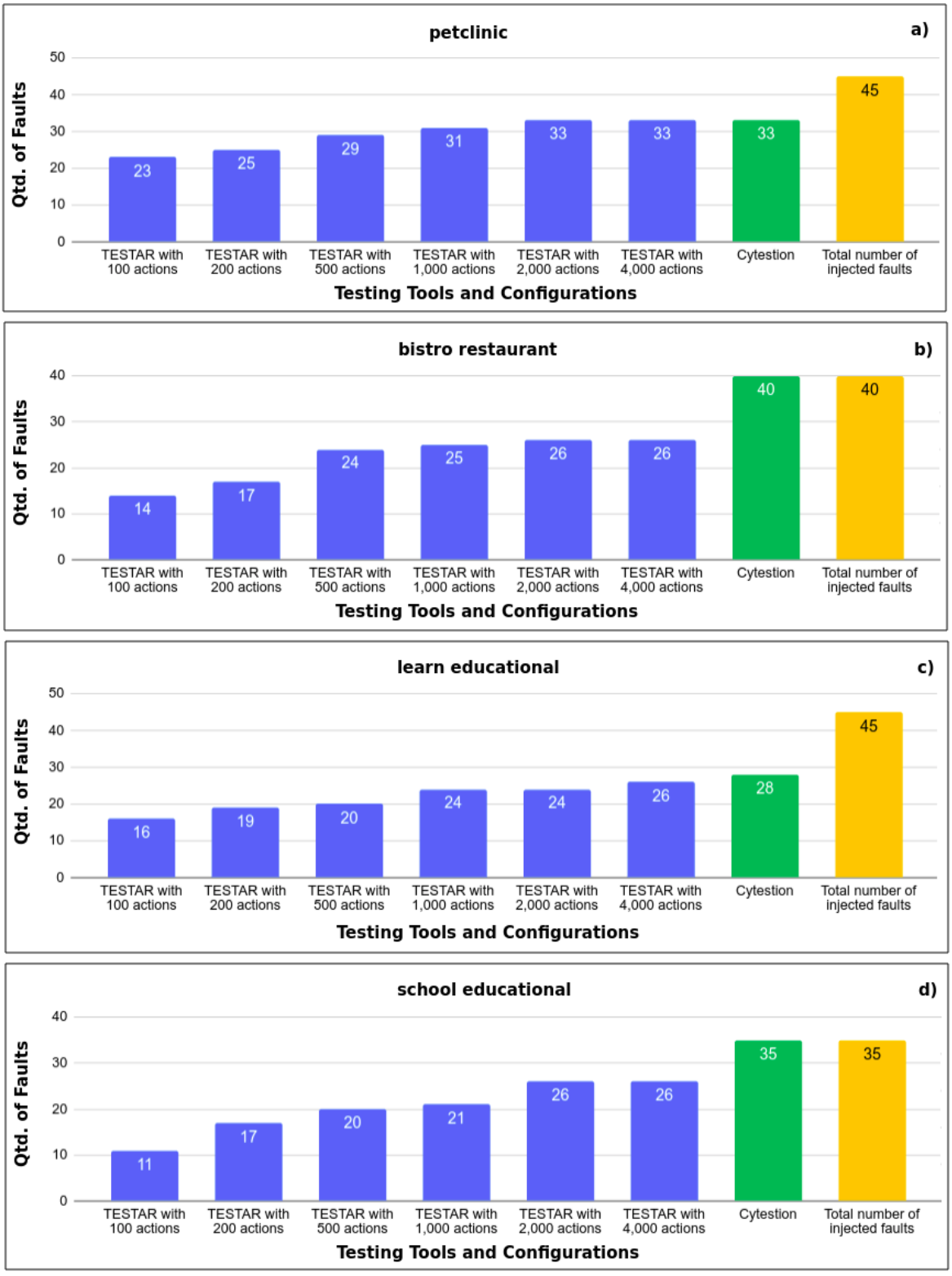

Figure 5.2: Injected faults detected by the tools.

Cytestion. To check the statistical significance of this result, we ran a proportion test comparing Cytestion's and TESTAR's detection rates per system [13]. For TESTAR, we used

the results obtained with 4,000 actions. The p-values obtained were 1.0000 for *petclinic*, 0.0001 for *bistro restaurant*, 0.8296 for *learn educational*, and 0.0043 for *school educational*. The results indicate that Cytestion significantly outperformed TESTAR in detecting faults in the *bistro restaurant* and *school educational* systems, with p-values less than 0.05. In the *petclinic* and *learn educational* systems, Cytestion's performance was equivalent to that of TESTAR. This dual achievement — surpassing TESTAR in two systems and matching its performance in two others — demonstrates Cytestion's robustness and effectiveness in fault detection across diverse testing scenarios. Consequently, Cytestion stands out as a compelling choice for GUI testing methodologies, capable of delivering reliable results that are comparable to or better than established tools like TESTAR. Therefore, we can confidently conclude, with 95% confidence, that Cytestion was effective in detecting the injected faults leading to visible failures.

When we analyze Cytestion's results per system, we find that its best performance was for the *school educational* (100%) and *bistro restaurant* (100%) systems, while its worst performance was for the *learn educational* (62.2%) system. A manual investigation of the *learn educational* scenario revealed that Cytestion was unable to detect certain faults due to the absence of valid and unique locators for some actionable elements, which prevented the creation of test cases capable of interacting with them. This occurs because of a limitation of UAES and more details about that is evidenced in Appendix B.

Similarly, TESTAR's best and worst results were for *school educational* (78.8%) and *learn educational* (57.8%), respectively. The main reason for this lower performance was the random selection of actions during execution, which often resulted in executions that did not interact with the actionable elements that could expose the fault. Another reason was the discovery of all actionable elements. Despite the correct configuration, some elements were not detected by the tool.

Finally, 29 injected faults were not detected by either tools. Part of those faults remained undetected because they required a specific sequence of actions to be triggered, such as properly completing and submitting a form.

For addressing RQ_2, we measured the average execution time in seconds for both Cytestion and TESTAR across four web applications. The execution times are illustrated in Figure 5.3. The results indicate that, on average, Cytestion took 344.84 seconds to run. TES-

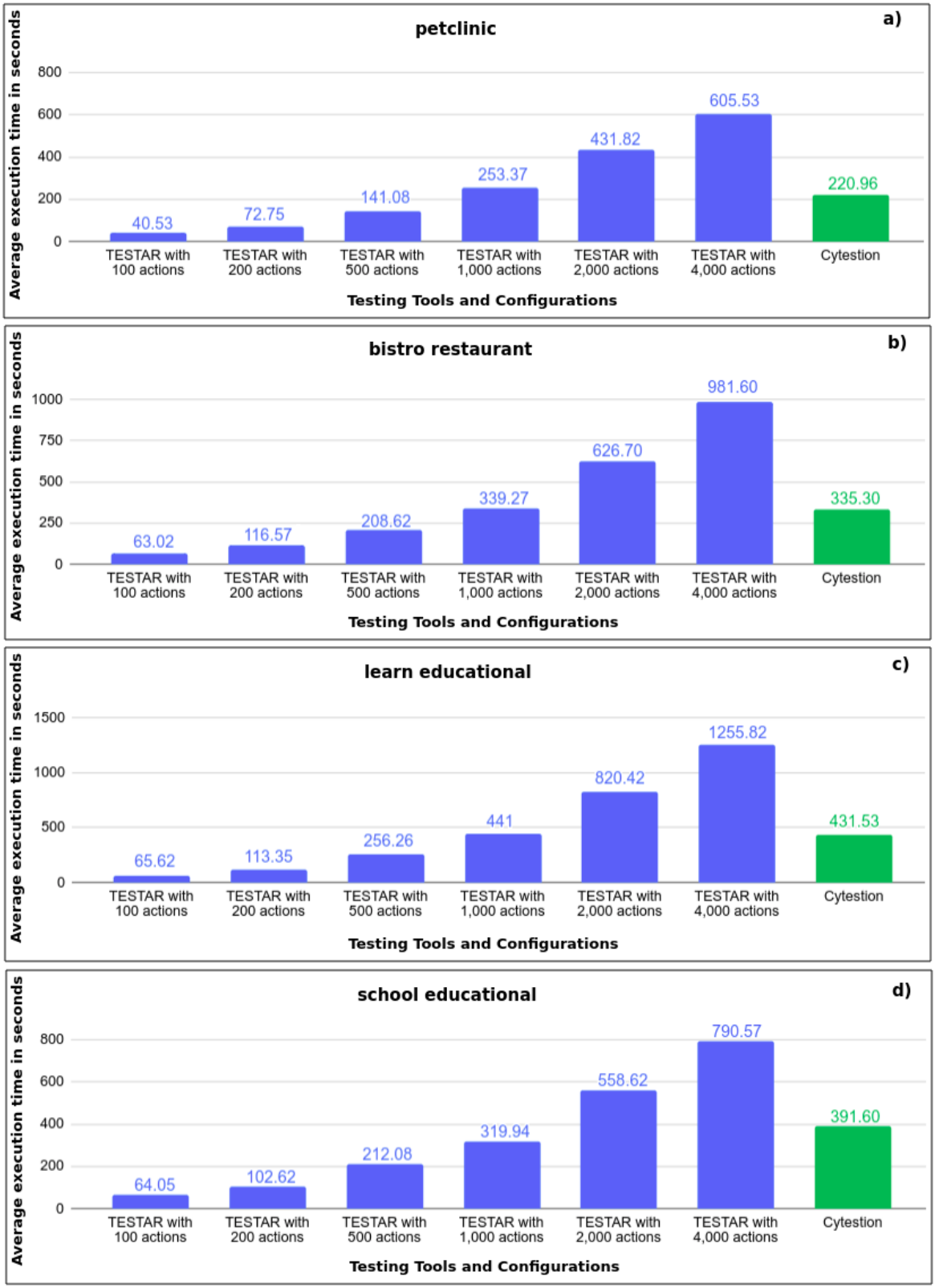

Figure 5.3: Average time in seconds for the execution of TESTAR and Cytestion.

TAR's execution time, however, varied significantly based on the number of actions, with a minimum of 40.53 seconds for 100 actions and a maximum of 1,255.82 seconds for 4,000 actions.

Considering Cytestion's average execution time of 344.84 seconds, we analyzed the number of actions TESTAR could perform within the same timeframe. Given the variability in TESTAR's execution time, it is estimated that TESTAR could execute approximately 1,000 actions within this period. This comparison is crucial as it underscores Cytestion's comprehensive exploration of all actionable elements, compared to the pre-defined action limits in TESTAR.

For instance, in the tested web applications, TESTAR found fewer faults when limited to 1,000 actions compared to the 4,000 actions configuration. Specifically, TESTAR missed two faults in the *petclinic* application, one in the *bistro restaurant* application, two in the *learn educational* application, and four more in the *school educational* application. TESTAR's limited action scope could potentially miss significant faults that a more exhaustive approach like Cytestion would detect, highlighting the importance of a thorough testing strategy for comprehensive fault detection.

The minimum execution time of Cytestion was 3.6 minutes, in contrast to TESTAR's shortest recorded execution time of only 40 seconds. The primary reason for this divergence is Cytestion's systematic and exhaustive approach, which does not impose a limit on the number of actions. Unlike TESTAR, which stops its execution upon finding the first fault, Cytestion continues its exploration, ensuring a more thorough testing process. This difference in methodology is a critical factor that justifies the increased execution time for Cytestion, as it leads to the detection of more faults.

Furthermore, while Cytestion's longest running time was recorded at 7.2 minutes, this duration is still relatively low, especially considering the benefits of its fault detection capabilities. On the other hand, TESTAR's execution time can vary widely based on the tester's configuration, and in realistic scenarios, the need to rerun TESTAR multiple times to uncover all faults could potentially increase its overall execution time beyond that of Cytestion.

To complement this cost analysis, we examined the execution times where neither Cytestion nor TESTAR found any faults during their runs across the open applications. That way, we could measured the average time per action for each tool. TESTAR demonstrated faster

performance, with times ranging between 0.60 and 0.78 seconds per action across all applications. In contrast, Cytestion's times were longer, ranging from 1.76 to 2.15 seconds per action. This disparity in time is due to Cytestion's waiting mechanism, which ensures that all network requests complete before the tool proceeds to the next action.

5.2 A Study with Industrial Applications

To evaluate the use of Cytestion in practice, we conducted an empirical study with twenty industrial applications. These projects consisted of React-based applications developed by different teams within a partner software development company. Each application is tailored to address distinct tasks related to fiscal and cost management for businesses. Table 5.2 refers to the size of the projects and the number of test cases generated and executed by Cytestion. For confidentiality reasons, we named them A1 - A20. It is important to note that all these projects are already in production, having undergone testing by both their development teams and the company's QA team prior to release.

In our second study, we ran both Cytestion and TESTAR on all twenty applications, collecting the reported faults and execution times. For TESTAR, each application was executed 15 times, with each execution consisting of a sequence of 1,000 actions, totaling 15,000 actions. By dividing this execution into sequences of a thousand actions, we mitigate the possibility of TESTAR getting stuck in a specific state and returning to the initial page in the next sequence, which is a possibility for a more complex system.

Any found fault was later submitted and revised by a member of the project's team, and if found as a fault (true positive), it was registered as a bug to fix.

5.2.1 Results and Discussion

As presented in Figure 5.4, Cytestion detected a total of 28 real faults across the twenty applications, all of which were accepted by the developers and registered as bugs. These faults were categorized by severity, with 5 high-severity faults, 19 medium-severity faults, and 7 low-severity faults. High-severity faults made certain functionalities unusable, medium-severity faults caused visible error messages or user confusion, and low-severity faults were minor but unexpected behaviors that did not significantly impact usability.

Application	KLOC	# of Generated Tests
A1	68.5	594
A2	82	801
A3	52.8	630
A4	77.6	1059
A5	306.8	3282
A6	178.5	1139
A7	65.9	222
A8	75	738
A9	37	412
A10	228.9	1975
A11	78.1	1123
A12	32.4	69
A13	109.1	829
A14	43.7	626
A15	62	454
A16	73	624
A17	58.5	338
A18	41.4	503
A19	42.1	408
A20	397.9	882

Table 5.2: Objects of the industrial applications study: KLOC and test counts.

Out of the 28 faults detected by Cytestion, 21 were identified through requests with failure statuses, which often resulted in failure messages in the GUI. This highlights the importance of intercepting and analyzing server requests, as it is not possible to provide all possible visible error messages to an automated tool used to detect them. Some messages may fall outside the standard, but can still reveal critical faults. This situation is closely related to one of the main challenges of creating an oracle that can identify failures like a human [19, 45].

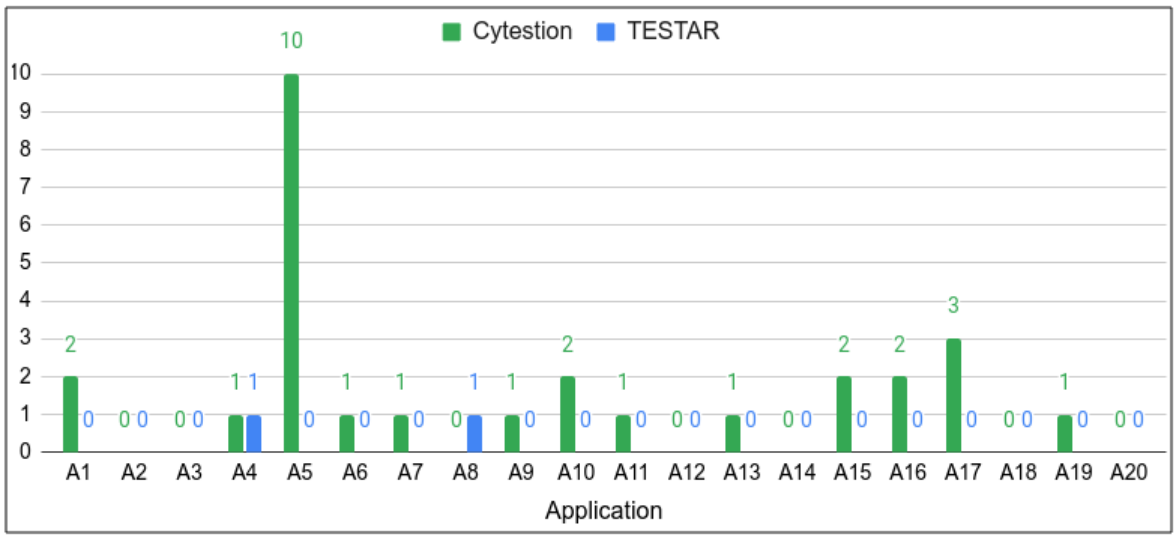

Figure 5.4: Comparison of real faults detected by Cytestion and TESTAR across twenty industrial applications.

The high-severity faults were deemed critical by the development teams and were quickly fixed, while the remaining 23 faults are registered for future fixes. Examples of detected faults include a malfunctioning button-triggered process that displayed the error message "An unexpected error occurred" and an issue with a form save button that failed to provide expected feedback, returning an HTTP 500 code instead.

However, presenting these faults to the company required careful manual analysis, as not all visible failures indicated real faults. Cytestion initially identified 494 states with visible failures. Upon closer examination, it was found that a recurring fault in a horizontal component was responsible for 348 of these failures. This fault was exposed only when the page was reloaded or accessed directly via the URL, an insight made possible by the IDUBS algorithm (Section 3.3). After classifying this as a real fault, the remaining 146 states with visible failures were analyzed, leading to the detection of 27 additional faults.

Despite the high number of false positives related to failure status requests, Cytestion's ability to identify subtle and impactful faults was evident. Many false positives were due to intentionally returned failure statuses for specific functionalities, which affected the effectiveness of analyzing visible failures. It is noteworthy that the 28 faults detected by Cytestion had gone unnoticed after several rounds of testing by the development and QA teams. These subtle but visible bugs could significantly impact user perception of the systems. By using

Cytestion, the teams were able to anticipate and address these issues effectively.

In contrast, TESTAR detected only 2 faults, which were not identified by Cytestion. These faults were exposed only after multiple repeated actions, a scenario facilitated by TESTAR's random heuristic. This type of use is uncommon and intrigued the developers. TESTAR's analysis involved 49 states with visible failures, 47 of which were later classified as false positives. These false positives often involved suspicious GUI messages that developers could not validate, leading them to conclude that such faults did not exist. Additionally, some sequences of actions reported by TESTAR could not be reproduced, such as inserting an email into a field intended for numerical input.

None of the 28 faults detected by Cytestion were identified by TESTAR, highlighting the latter's ineffectiveness in this context. TESTAR's random heuristic limits its ability to perform in-depth exploration of industrial systems and its focus solely on GUI messages, ignoring server requests, contributed to this result. Cytestion's systematic exploration of all actionable elements in a given state enabled it to detect more subtle faults effectively.

As depicted in Figure 5.5, Cytestion demonstrated a lower execution time in all applications, with an average of 1 hour and 42 minutes. The median execution time was even shorter (1 hour and 22 minutes). The maximum observed execution time for Cytestion was 6 hours and 7 minutes, while the minimum was 9 minutes and 38 seconds. This consistency suggests that Cytestion can maintain a relatively stable execution duration regardless of the application's complexity, making it predictable and manageable in industrial contexts.

In contrast, TESTAR exhibited significantly higher and more variable execution times. The average execution time for TESTAR was approximately 6 hours and 3 minutes, with a median time of 5 hours and 48 minutes. The execution times ranged from a minimum of 4 hours and 22 minutes to a maximum of 8 hours and 27 minutes. This variability indicates that TESTAR's performance can be highly influenced by the specific configuration and characteristics of the AUT.

Cytestion consistently outperformed TESTAR in terms of execution time across nearly all applications. The only exception was application A5, where TESTAR had a shorter execution time. However, it is important to note that A5 is the application where Cytestion identified 10 out of the 28 real faults, emphasizing its effectiveness despite the longer execution time in this particular case.

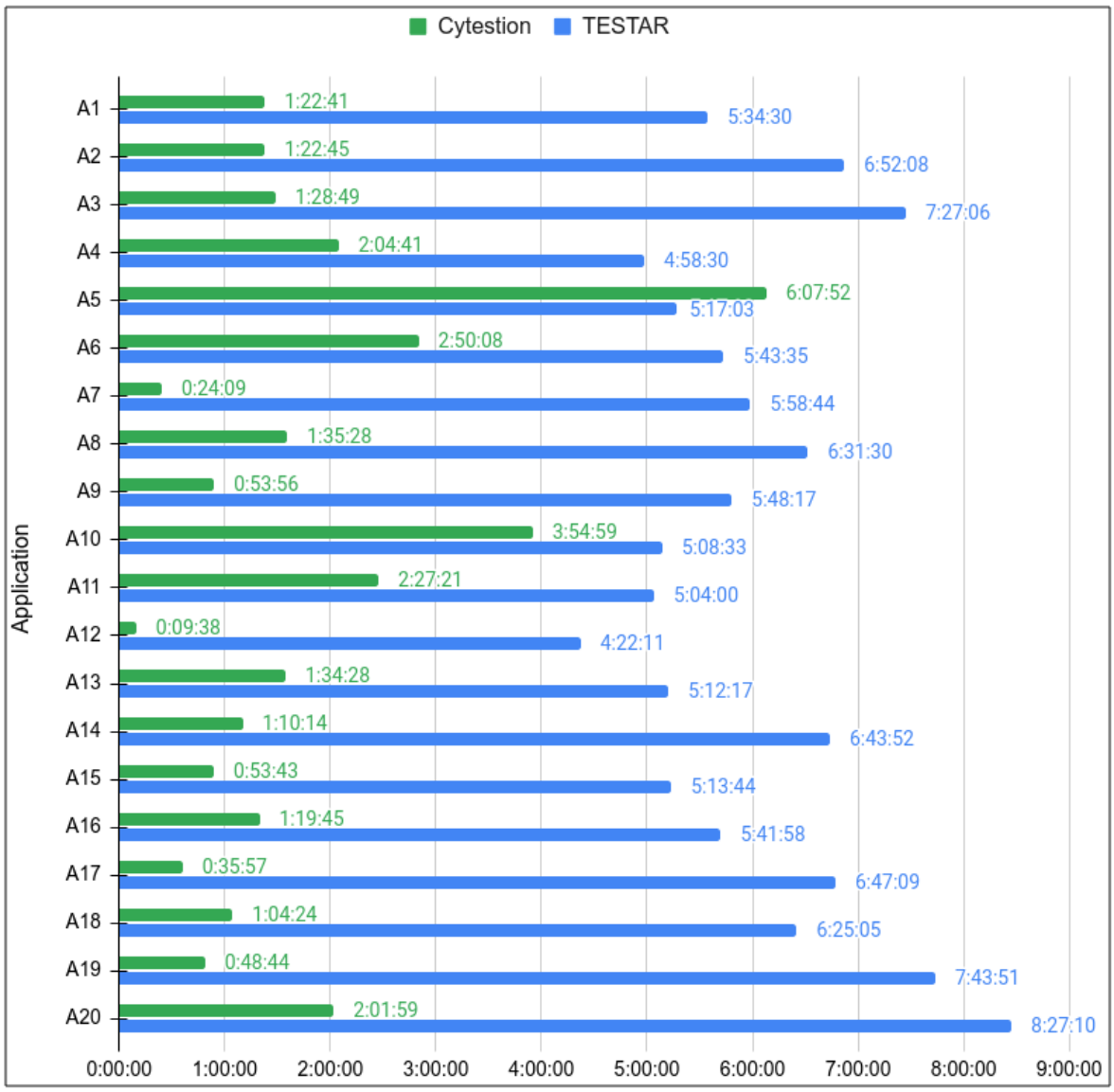

Figure 5.5: Comparison of execution times between Cytestion and TESTAR across twenty industrial applications.

The systematic nature of Cytestion allows it to thoroughly explore and test applications, resulting in more efficient fault detection. TESTAR, with its random heuristic approach, can vary significantly in execution time based on the configuration used, making it less predictable and potentially less efficient in industrial settings. Given Cytestion's superior fault detection capabilities and more consistent execution times, the costs associated with its use are justified. Additionally, Cytestion's execution time can be further optimized by configuring it to run on specific parts of the system, although this scenario was not explored in the current study as our goal was to test the system as a whole.

The results of Cytestion were later presented to the company QA team and to the manager of the projects. In general, the feedback was very positive. They reported that the type of faults detected by Cytestion was commonly neglected by both the developers and even the QA team. They also reported that, though costly (time-wise) the gains of using such automated tools would help to improve the project's code quality and avoid harms on the user experience. Finally, since then, the Cytestion tool has been integrated into the release pipeline of the companies' projects.

Based on the results discussed in Sections 5.2 and 5.1, we can answer RQ_1 and RQ_2 by stating that Cytestion is effective for detecting faults that cause visible failures, since it detected both injected and real faults in our empirical studies, outperforming a state-of-the-art tool. Moreover, though sometimes it takes a considerable time to run, its results were found important by practitioners.

5.3 Threats to Validity

The selection of study objects poses a potential threat, as the chosen projects used for testing may not fully represent the broader range of web applications. This could limit the applicability of the results to different application types. However, we used a diverse set of both open-source and industrial projects, spanning various domains and sizes, which we believe to be good representatives of the overall universe of web applications.

Another potential threat to the validity of our results in Section 5.1.1 is the use of injected faults. However, the practice of seeding faults is well-established in testing research [20, 21, 113]. Previous studies have demonstrated that mutants can be good representatives of real faults in testing experiments [12, 48]. While mutation GUI operators could serve as an alternative for fault injection, some proposed operators may not result in visible failures. Therefore, we chose to conduct our study with students, who injected specific mutations. Additionally, all injected faults were manually validated by the author. Furthermore, the injected faults closely resemble those found in our study with industrial applications, as discussed in Section 5.2, which does not rely on injected faults.

A potential threat to the validity of the second study is that faults might be obscured if they occur in states deeper than those already presenting a visible failure. Cytestion halts

exploration of the current branch, while TESTAR stops the entire execution upon identify a visible failure. As a result, faults that manifest visible failures in deeper states might remain undetected if they are overshadowed by earlier faults. However, our sample analysis indicates that this is not a frequent issue, as no such cases were observed. Future research could explore methods to continue exploration beyond the initial fault detection.

Relying solely on TESTAR as the comparison tool could bias the analysis of Cytestion's performance, risking the validity of the studies. Using multiple tools or additional metrics would strengthen the evaluation and reduce bias. Despite the limitation of a single tool, the consistent use of TESTAR enables a direct comparison to a state-of-the-art benchmark, facilitating a focused assessment of Cytestion's strengths and weaknesses relative to accepted standards in GUI testing.

Additionally, the configurable nature of TESTAR as a tool introduces a potential threat to internal validity in our studies. We utilized the official Docker image provided by the tool's website, employing the generic web protocol. Following the tool's recommendations, we also customized some functions to accommodate the application under test and address limitations related to click-ability.

The manual validation and verification of detected faults performed in the study with industrial applications introduces potential researcher bias, which could undermine the accuracy of assessing Cytestion's fault detection capabilities. To address this concern, rigorous validation procedures were implemented, including the involvement of multiple reviewers and consensus-based decision-making, to enhance the reliability and objectivity of fault identification and classification.

Finally, while there are other aspects beyond fault detection and time that could have been considered in evaluating our work (such as generation depth and test case similarity), we chose to focus on these two factors because they address crucial and practical scenarios in testing: detecting faults and managing costs effectively. These criteria were selected to provide a comprehensive assessment of Cytestion's performance in real-world testing environments.

5.4 Concluding Remarks

This chapter presents two empirical studies comparing the Cytestion and TESTAR tools, focusing on their visual fault detection capabilities and execution times. Our studies provided evidence that Cytestion systematically and effectively discovered elements in the GUI of the applications used. In open-source applications, Cytestion outperformed TESTAR in detecting injected faults, completing its tasks within minutes. Additionally, Cytestion managed the dynamism and complexity of web systems, particularly industrial ones, achieving excellent results.

In our study with industrial applications, 30 real faults that resulted in visible failures, of which Cytestion detected 28, while TESTAR detected only 2. Despite Cytestion's robust fault detection, the cost of systematic exploration was highlighted. Even with the IDUBS algorithm to prune paths between states and the parallel execution of two test cases at a time, Cytestion required an average of one hour and 42 minutes for industrial applications. This extended execution time makes it impractical to run Cytestion several times a day. However, it proves valuable for weekly releases. Consequently, the partner company included Cytestion in their weekly approval pipeline. A recent analysis shows that it found an average of two real faults per week.

In the next chapter, we explore the related work, considering the challenges of scriptless GUI testing, and reviewing a comprehensive set of tools designed for this type of testing.

Chapter 6

Related Work

In this section, we discuss research results related to our work, focusing on studies primarily investigating the discovery and localization of GUI elements (Section 6.1); studies addressing synchronization issues and solutions (Section 6.2); studies exploring efficient GUI exploration using model-based or search algorithms to explore a GUI tree (Section 6.3); and existing automated GUI testing tools with goals similar to Cytestion (Section 6.4).

6.1 Related to Challenge 1: Unique Discovery of Actionable Elements

Chen et al. [25] investigate the discovery of GUI elements in GUI images using a combination of conventional methods and deep learning models. They conduct a large-scale empirical study on over 50,000 GUI screenshots to compare different discovery methods and address unresolved research questions in GUI element discovery. By combining effective designs of existing methods with a novel GUI-specific region discovery method, their study advances the state-of-the-art in GUI element discovery.

Xue et al. [121] propose a method for visually discovering mobile app GUI elements using object detection technology, aiming to enable vision-driven robotic testing for mobile apps. Their approach focuses on accurately capturing GUI element information from a black-box perspective, emphasizing effective classification based on external image features. They utilize the YOLOv3 model to implement the discovery of GUI elements, showcasing the

potential of this approach for automated robotic testing of mobile apps.

Degott et al. [29] present a method for learning GUI element interactions using reinforcement learning. Their approach models the problem as a multi-armed bandit, aiming to improve test generation by understanding interactions such as clicking and typing. This method enhances test coverage, demonstrating up to a 20% improvement in statement coverage when employing learned models.

White et al. [119] propose a novel approach using machine learning for image-based detection of GUI widgets. They generate synthetic Java Swing GUIs to train a model to predict widget types and positions, which guides the test generator. This improves testing without relying on external GUI APIs, addressing challenges posed by visual variations across GUI libraries and operating systems.

Unlike the UAES approach, these four studies fall under the image recognition category of GUI testing frameworks and do not specifically focus on differentiating the GUI elements discovered.

A3E [16] is an open-source tool that enables systematic exploration of Android applications without requiring access to the source code. It generates test cases by exploring the application's GUI, complementing the existing test suite. However, it allows for repetitive interaction with Android elements during automated exploration. In contrast, UAES offers a systematic exploration approach for web applications and ensures the uniqueness of the found elements.

AMOGA [102] is a tool designed for systematic exploration in Android applications. It extracts a Window Transition Graph (WTG) from the AUT, enabling the derivation of available actions in the application and subsequent stress testing of these actionable elements. While it focuses on uniquely discovering elements within mobile applications, it requires analysis of the app's code rather than employing a black-box approach.

Zimmermann et al. [125] integrate GPT-4 and Selenium WebDriver to enhance testing coverage and accuracy through AI model integration. The GPT-4 model interprets GUI element states based on Selenium WebDriver's updated DOM state, enabling automated testing without human interaction. However, their method lacks systematic exploration and does not prioritize the unique discovery of actionable elements. Additionally, UAES delivers semantically meaningful results for tested applications without requiring industrial GPT-4 usage

rights.

The locator definition process after the discovery stage conducted by UAES addresses a well-known issue in GUI testing literature, particularly in scripted approaches. Filippo et al. [93] identified three key problems in GUI testing: test case fragility, strong coupling and low cohesion with the AUT, and the incompleteness problem. This fragility is closely linked to the locators used to interact with web page elements. Several studies have attempted to address this problem [59, 60, 84, 85, 86]. However, in our scriptless context, this issue is mitigated, as new locators are generated during each execution.

Finally, Kirinuki et al. [53] use Natural Language Processing (NLP) to locate actionable elements on a web page by interpreting test cases in a domain-specific language. This approach emphasizes selecting readable texts to identify actionable elements, prioritizing visible texts over attribute values for semantic representation. UAES adopts a similar idea of semantic representation to define unique and expressive locators for the found elements. Unlike UAES, their technique requires testers to specify the existence of elements, so automatic discovery is not included.

6.2 Related to Challenge 2: Synchronization

Nass et al. [83] conducted a systematic review identifying key challenges in GUI-based test automation, classifying them as essential or accidental. They noted the *synchronization challenge* as accidental, meaning it can be solved or mitigated through further research and development. They explained that this challenge is not inherent to GUI-based test automation, as human testers do not face this issue when testing manually. Instead, it is a technical challenge that arises due to the complexity of synchronizing test execution with the AUT.

Leotta et al. [58] conducted a survey with 78 industry experts and found that synchronization issues (referred to as asynchronous) are widely recognized as a challenge for GUI testing with Selenium. Testers frequently apply waiting strategies to address this issue. However, these strategies, while sometimes effective, can increase execution time and introduce testing flakiness. The authors emphasize the importance of choosing appropriate waiting strategies and advocate for further research into developing more effective tools.

Sousa et al. [105] investigated the main causes and correction strategies for flaky tests in

automated GUI tests across 24 open-source projects. Their analysis of 123 flaky test commits showed that *synchronization challenges* were the primary cause, accounting for 60% of the cases. The most common correction strategy was the inclusion of wait mechanisms, present in 53% of the related commits. This approach involved adding delays between actions to mitigate the impact of *synchronization challenges* and improve test stability.

Habchi et al. [35] conducted a literature review and interviewed 20 software professionals to explore the issue of flaky tests in software testing environments. They found that waiting points in GUI tests are a significant cause of flakiness. The study highlights the impact of flaky tests on testing practices and product quality, underscoring the need for better strategies to address these challenges. The findings suggest that automation tools, such as logging and monitoring, can play a crucial role in detecting and preventing flakiness.

Pei et al. [90] proposed TRaf, an automated framework addressing flaky tests caused by asynchronous waits in web GUI testing. TRaf uses code similarity and past changes to recommend optimal wait times for asynchronous calls. Their study of 49 flaky tests from 26 JavaScript web projects found that developers often add or increase wait time to mitigate flakiness. TRaf's approach involves localization and tuning, outperforming developer-written fixes by reducing test execution time by up to 20.2% with dynamic optimization. This highlights TRaf's potential to improve testing efficiency and reliability.

Feng et al. [31] introduced AdaT, a novel lightweight approach for enhancing Android GUI testing efficiency by dynamically adapting event timing according to the current GUI rendering status. This innovative methodology integrates deep learning techniques and real-time GUI streaming to precisely determine rendering states, enabling synchronized testing events. AdaT's implementation showcases notable advancements in automated testing, emphasizing the seamless integration of image-based inference to optimize testing processes for Android applications.

Olianas et al. [87, 88] investigated ways to address flakiness in Selenium-based GUI testing. They replaced static waits with explicit waits, improving synchronization between test scripts and the AUT. The authors developed both a manual procedure and a tool-based approach to optimize test suites by substituting thread sleeps with more reliable WebDriver-Wait commands. These efforts reduced flakiness and enhanced the overall reliability and efficiency of the test suite, providing insights to improve automated testing practices.

Liu et al. [66] introduced WEFix, a tool that automatically generates explicit waits to address flaky tests in web GUI testing. By examining DOM changes on the browser, WEFix can predict client-side operations and strategically insert explicit waits at waiting points, reducing flakiness and runtime overhead. This innovative approach has proven highly effective, resolving 98% of flaky tests in evaluations. Compared to traditional implicit wait methods, WEFix enhances test reliability and minimizes runtime overhead, making it valuable for web GUI testing.

These studies underscore the prevalence and severity of synchronization issues, highlighting their detrimental effects on test reliability and system quality. Notably, only the work of Liu et al. has explored this challenge within the Cypress context, focusing solely on applying the *Explicit Wait* mechanism. Our work differentiates itself by compiling a catalog of waiting mechanisms for Cypress (Appendix C) and proposing a new one (*Network Wait* in Section 3.2), which is equally effective and appropriate for the non-deterministic scenario of scriptless testing [78].

6.3 Related to Challenge 3: Efficient Systematic Exploration

Takala et al. [109] explore model-based GUI testing of Android applications through a case study with the BBC News Widget. They implemented a keyword-based test automation tool for the Android emulator and compared its effectiveness to traditional GUI testing. Their results indicate potential benefits of model-based testing, with all models and tools made available as open source.

Kilinc-Ceker et al. [52] introduce a model-based ideal testing approach for GUI programs, merging Holistic Testing and Model-based Testing to create comprehensive test suites that effectively identify functional faults. Their methodology enhances testing efficiency by automating procedures and ensuring model correctness through rigorous selection criteria. Experimental evaluations validate its effectiveness in improving GUI program testing.

Amalfitano et al. [11] introduce a technique for efficient Android app testing using a crawler to automate crash and regression testing. Their methodology involves automatically building a GUI model and generating test cases for automatic execution. They demonstrate

the technique's effectiveness by testing a small Android application, highlighting its usability and potential for improving testing processes.

Weise et al. [115] conducted a study on the importance of ontology in defining parameter semantics and efficient web service discovery. In their analysis, the uninformed search performed by IDS was found to be inefficient due to excessive costs and algorithm limitations compared to other methods for locating composite semantics in web services.

Jiang et al. [47] emphasize the significance of GUI testing in Android apps and examines the impact of GUI state equivalence choices on error detection. It compares random search and systematic search with BFS and DFS algorithms using 33 real applications to study their effects on fault detection rate and code coverage. Their findings indicate that both random search and systematic search are equally effective, while state equivalence has a significant impact on fault detection rate and coverage.

Wen [116] presents a new methodology for testing web-based applications and technologies, the URL-Driven Automated Testing (URL-DAT). This method involves using previously known URLs and data-driven testing to guide data through the automation of test execution, thereby combining them. However, no search algorithm is used since navigation through the AUT is not the goal.

Hu et al. [43] suggest that automated testing can improve software testing efficiency by using test automation tools such as Selenium and QTP to enhance test case accuracy. It involves representing the software project workflow as a directed graph and traversing it with the DFS algorithm to generate test paths, aiming to increase maintainability and reuse of tests. The conclusion presents promising results in industrial tests, such as in a scientific research clinical management project.

Yuan et al. [124] introduce ALT, a GUI testing approach that uses execution feedback and event semantic interaction analysis for better fault detection. An empirical study shows that ALT is effective in detecting GUI event interaction faults, emphasizing the importance of feedback-driven testing. ALT differs from traditional methods by using runtime feedback for refining test cases and generating batches with Event Semantic Interaction (ESI) relationships and batch-wise test case generation to systematically explore event interactions.

Lim et al. [65] introduce Boundary Iterative-Deepening Depth-First Search (BIDDFS), an algorithm that combines the IDS and Dijkstra algorithms to optimize pathfinding by set-

ting node storage limits and following a specific expansion pattern. Through simulation experiments, BIDDFS showed superior performance when performing blind searches in unknown environments, evidencing its potential for real-world pathfinding efficiency improvements. However, it does not share chain information from previous nodes or directly access any node in the graph.

The mentioned studies cover a wide range of topics, such as the use of model-based GUI testing, IDS in GUI testing, DFS and BFS in GUI testing, direct URL access in tests, and algorithms that enhance IDS. Our work is distinct in proposing an effective way to reduce the costs related to GUI testing focusing in the web context with IDUBS but preserving its testing power.

6.4 Automated GUI Testing Tools

TESTAR [113] is a state-of-the-art tool that shares similar goals with Cytestion and was used as a baseline in our empirical studies. Both tools aim to detect faults causing visible failures, but their GUI testing strategies differ. Cytestion employs a systematic exploration approach, creating and executing test cases for each actionable element discovered. In contrast, TESTAR uses a random approach and stops upon detecting a fault, often requiring multiple runs. Additionally, Cytestion extends its search for failures beyond the GUI to include server response requests and the browser console. It leverages the Cypress framework, offering faster executions, readable test scripts, and artifacts like videos and screenshots for test failures, whereas TESTAR provides a limited HTML report.

Crawljax [75] dynamically analyzes GUI state changes in AJAX applications by automatically exploring web applications to derive a testing model that captures GUI states and event-based transitions. This enables the generation of test cases to uncover potential failures during user interactions. However, Crawljax may not specifically address input forms, potentially making some pages inaccessible without appropriate entries. The tool also heavily relies on the DOM and XPath expressions to locate web elements, resulting in fragile tests [93].

WEBMATE [27] is a commercial GUI test generator that uses the Selenium API to control the browser and create a usage model capturing all interaction paths within the appli-

cation. The model, implemented as a finite state machine, ensures that previously visited states are not re-explored, with each state comprising elements for interacting with the application. However, it does not differentiate between static and dynamic elements, reducing its potential to differentiate elements in modern applications.

GUITAR [123] automates GUI exploration by interacting with widgets to create an event flow graph, from which test cases are generated using coverage criteria to detect GUI-related failures. However, it produces short test sequences (3-20 actions) and has mainly been tested on simple systems, limiting its use for complex industrial web applications. Additionally, GUITAR may not prioritize the uniqueness of GUI elements, leading to difficulties in accurately identifying and interacting with specific elements.

Murphy [4] automates GUI testing using intelligent agents triggered by specific GUI states to detect failures in behavior and functionality. By leveraging AUT-specific knowledge and predefined triggers, Murphy identifies discrepancies in application behavior, uncovering potential faults affecting user experience or system reliability. Although the open-source project is not actively maintained, Murphy's approach to intelligent GUI interaction remains a valuable contribution to automated testing.

AUGUSTO [69] automates GUI testing by generating test cases from a high-level model of the GUI and its expected behavior. This model-based approach reduces manual effort and improves test script maintainability and adaptability to GUI changes. While creating these models requires expertise, AUGUSTO enhances the efficiency and effectiveness of GUI testing by enabling precise behavior specifications and automated test oracles to verify application functionality.

Sapienz [67] is a multi-objective automated testing tool for Android apps that combines random fuzzing, systematic exploration, and search-based testing to optimize test sequences for coverage, fault detection, and length. Using adaptive strategies and multi-level instrumentation, it handles app response times by modifying the source code. Sapienz operates with a 30-minute time limit but can run indefinitely. The study shows it outperforms traditional tools like Android Monkey, highlighting its potential to revolutionize Android app testing.

WebTestingExplorer [71] is a feedback-driven automated testing tool for web applications that utilizes runtime state to search for defects in real-time and generate test cases with

oracles. Unlike Cytestion, WebTestingExplorer is based on the Selenium framework and aims to automate test generation and maintenance. However, the lack of scientific evidence and experiments hinders the ability to validate the tool's effectiveness, highlighting the need for further research and validation.

A variety of automated GUI testing solutions exist, some aiming to find faults and others to generate meaningful test cases. However, no usable tool focuses on systematic and automatic GUI testing for web applications. We also note the antiquity of the related works, all dated eight years or more. The systematic literature review [83] shows that the challenges exposed for GUI testing on the Web are mainly concentrated in publications of the last seven years. Thus, these tools likely do not address these latent problems or have mitigated them at the expense of fault detection effectiveness.

Chapter 7

Concluding Remarks

In this work, we presented Cytestion, an automated approach and tool for systematic GUI testing in web applications. Cytestion adopts a scriptless and progressive strategy, discovering and interacting with actionable elements, and generating new tests to detect faults that cause visible failures. The tool utilizes browser messages, HTTP request statuses, and failure GUI messages as the oracle for fault detection. The generated test suite, along with fault summaries and demonstration videos, are provided as outputs.

In order to enable effective GUI testing, we needed to face latent challenges reported in the literature related to creating a scriptless GUI testing tool, particularly with a focus on systematic exploration. The unique discovery of actionable web elements through systematic exploration was a critical advancement that had not been addressed previously. The UAES approach was developed and empirically evaluated, yielding good results compared to the Markup Approach, detecting 95.30% of the manual elements identified by testers. Integrating the UAES approach into Cytestion was essential to ensure complete and comprehensive coverage of actionable elements.

Addressing synchronization with the AUT was another significant challenge. Synchronization is a common problem in automated GUI testing and becomes more complex when generating and executing tests fully automatically. To tackle this issue, waiting mechanisms are employed, which involve introducing deliberate pauses to allow the AUT to reach a stable state before interactions. Our literature review identified various mechanisms, including *Static Waits*, *Implicit Waits*, and *Explicit Waits*, commonly used in the Selenium framework. Each method has its pros and cons: *Static Waits* are easier to implement but less precise,

while *Explicit Waits* offer greater accuracy but require more manual configuration.

Related works point to *Explicit Wait* as the most robust mechanism due to its accuracy, given well-established waiting conditions defined by humans. The *Network Wait* mechanism was designed to eliminate this manual overhead by waiting for communication between the client side and the web server. Studies using *Network Wait* showed results equivalent to *Explicit Wait*, mitigating flaky tests that cause false positives. *Network Wait* is a suitable replacement for *Explicit Wait* in the non-deterministic context of a scriptless GUI testing tool.

Another challenge was the lengthy process of systematic exploration, which can take hours in an industrial context. We addressed this by viewing the problem as graph exploration and presenting the IDUBS algorithm, which uses page URLs to provide new starting points and reduce execution time and redundant state revisits. Empirical studies showed that IDUBS reduced test execution time by 43.41% and decreased test case redundancy by 49.30% compared to the IDS algorithm. Moreover, direct access to certain URLs exercised different parts of the code, exposing real faults not found by IDS.

After integrating all these solutions, we conducted two empirical studies to evaluate Cytestion compared to a state-of-the-art GUI testing tool called TESTAR. The first involved injected faults in four open-source projects, while the second involved twenty industrial projects, revealing several real faults, with 30 issues reported, 28 of which were detected only by Cytestion. The results demonstrated the effectiveness of Cytestion in detecting faults and its potential for practical application in web development. Our approach and tool contribute to addressing the challenges of automated GUI testing, providing a scriptless and systematic testing solution.

The implementation of Cytestion has had a significant impact both in practice and within the academic literature. Practically, Cytestion has demonstrated substantial value in identifying real-world issues in production applications, leading to improved system performance and user experience. The tool's ability to automate the detection of various failures—ranging from critical errors that disrupt system functionality to usability problems, has been shown to be essential for enhancing software quality. By integrating automated testing into the CI/CD pipelines and providing detailed video documentation of errors, Cytestion has streamlined the debugging process, reduced manual effort, and accelerated issue resolution.

From a literature perspective, this work advances the field of automated GUI testing by introducing and empirically validating novel techniques such as the UAES approach and the Network Wait mechanism. These contributions address critical gaps in existing testing methodologies, particularly in scriptless and systematic exploration. The empirical studies conducted demonstrate the effectiveness of these methods in reducing execution time and improving fault detection, thus providing a robust theoretical foundation for future research. The findings offer valuable insights into the challenges of automated testing and pave the way for further innovations in testing tools and techniques, contributing to the ongoing evolution of the field.

Although our evaluation studies show promising results, there are areas for improvement. Future work includes incorporating objective testing with semantic search to target specific functionalities. By leveraging Large Language Models (LLMs) and our catalog of actionable elements, we can generate specialized test cases for critical functionalities, enhancing Cytestion's ability to exercise various features and navigate alternative flows, thereby improving fault detection rates. This catalog will serve as a valuable resource for testers, aiding in the generation of context-aware test scripts and enhancing the overall efficiency and comprehensiveness of the testing process.

Another critical improvement involves addressing the number of false positives, many of which are related to intercepting requests. While this strategy enhances detection power, it also increases false positives, necessitating strategies to automatically evaluate results. We plan to investigate advanced filtering and validation techniques, possibly incorporating AI methods like Convolutional Neural Networks (CNN). These models can classify false positives using previously analyzed and annotated data, such as images and videos, to train the model. This improvement will enhance the reliability of Cytestion's fault detection capabilities and reduce the manual effort required to analyze test results.

We explore the integration of parallelism in Cytestion, and initial efforts have demonstrated a significant reduction in execution time by utilizing two instances of Cypress simultaneously. However, there remains an opportunity for empirical evaluation of additional parallel configurations. Specifically, experimenting with more than two threads or instances could provide deeper insights into the scalability of parallelism and its impact on overall performance. Furthermore, future research should focus on analyzing the computational

overhead associated with various parallel configurations to determine the most cost-effective solution. This investigation would contribute to identifying the optimal balance between execution efficiency and resource utilization, ultimately enhancing Cytestion's effectiveness in automated GUI testing.

Additionally, we plan to expand the empirical studies to include more open-source projects and industrial case studies, which would provide a broader perspective on Cytestion's performance and applicability across different web applications and development environments. By testing Cytestion in a wider variety of real-world scenarios, we can identify potential limitations and areas for improvement, ensuring that the tool remains robust and effective in diverse contexts.

Finally, investigating how Cytestion can be integrated into existing software development workflows, particularly in continuous integration and delivery pipelines, would involve automating the execution of Cytestion tests as part of the software development lifecycle. This integration would enable seamless and frequent testing, ensuring that any faults introduced during development are quickly identified and addressed. By embedding Cytestion into the continuous integration process, we can provide developers with immediate feedback, fostering a more efficient and responsive development environment.

References

[1] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Pekka Aho. Automated state model extraction, testing and change detection through graphical user interface. *University of Oulu*, 2019.

[3] Pekka Aho, Teemu Kanstren, Tomi Räty, and Juha Röning. Automated extraction of gui models for testing. In *Advances in Computers*, volume 95, pages 49–112. Elsevier, 2014.

[4] Pekka Aho, Matias Suarez, Teemu Kanstrén, and Atif M Memon. Murphy tools: Utilizing extracted gui models for industrial software testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 343–348. IEEE, 2014.

[5] Pekka Aho and Tanja Vos. Challenges in automated testing through graphical user interface. In *2018 ieee international conference on software testing, verification and validation workshops (icstw)*, pages 118–121. IEEE, 2018.

[6] Pekka Aho, Tanja EJ Vos, Sami Ahonen, Tomi Piirainen, Perttu Moilanen, and Fernando Pastor Ricos. Continuous piloting of an open source test automation tool in an industrial environment. *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 1–4, 2019.

[7] Emil Alégroth. *Visual gui testing: Automating high-level software testing in industrial practice*. Chalmers Tekniska Hogskola (Sweden), 2015.

[8] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. Visual gui testing in practice: challenges, problems and limitations. *Empirical Software Engineering*, 20:694–744, 2015.

[9] Francisco Almenar, Anna I Esparcia-Alcázar, Mirella Martínez, and Urko Rueda. Automated testing of web applications with testar: Lessons learned testing the odoo tool. In *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings 8*, pages 218–223, Raleigh, North Carolina, USA, 2016. Springer, Springer.

[10] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Techniques and tools for rich internet applications testing. In *2010 12th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 63–72. IEEE, 2010.

[11] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *2011 IEEE fourth international conference on software testing, verification and validation workshops*, pages 252–261. IEEE, 2011.

[12] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411, St. Louis MO USA, 2005. Association for Computing Machinery.

[13] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[14] Yauhen Leanidavich Arnatovich and Lipo Wang. A systematic literature review of automated techniques for functional gui testing of mobile applications. *arXiv preprint arXiv:1812.11470*, 2018.

[15] Satya Avasarala. *Selenium WebDriver practical guide*. PACKT publishing, 2014.

[16] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international*

conference on Object oriented programming systems languages & applications, pages 641–660, New York, NY, USA, 2013. Association for Computing Machinery.

[17] Alexander Bainczyk, Alexander Schieweck, Bernhard Steffen, and Falk Howar. Model-based testing without models: the todomvc case study. *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, pages 125–144, 2017.

[18] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679–1694, 2013.

[19] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[20] Sebastian Bauersfeld, Tanja EJ Vos, Nelly Condori-Fernández, Alessandra Bagnato, and Etienne Brosse. Evaluating the testar tool in an industrial case study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–9, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Axel Bons, Beatriz Marín, Pekka Aho, and Tanja EJ Vos. Scripted and scriptless gui testing for web applications: An industrial case. *Information and Software Technology*, 158:107172, 2023.

[22] Maura Cerioli, Maurizio Leotta, and Filippo Ricca. What 5 million job advertisements tell us about testing: a preliminary empirical investigation. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1586–1594, 2020.

[23] Hatim Chahim, Mehmet Duran, Tanja EJ Vos, Pekka Aho, and Nelly Condori Fernandez. Scriptless testing at the gui level in an industrial setting. In *Research Challenges in Information Science: 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23–25, 2020, Proceedings 14*, pages 267–284. Springer, 2020.

[24] Peter Chapman and David Evans. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 263–274, 2011.

[25] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020.

[26] Instituto Nacional da Propriedade Industrial. Certificado de registro de programa de computador. Processo Nº BR512023000403-0, 2023. Expedido em 28 de fevereiro de 2023.

[27] Valentin Dallmeier, Bernd Pohl, Martin Burger, Michael Mirold, and Andreas Zeller. Webmate: Web application test generation in the real world. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 413–418. IEEE, 2014.

[28] Dmitry Davidov and Shaul Markovitch. Multiple-goal search algorithms and their application to web crawling. In *AAAI/IAAI*, pages 713–718, 2002.

[29] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 296–306, 2019.

[30] Markus Ermuth and Michael Pradel. Monkey see, monkey do: Effective generation of gui tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 82–93, 2016.

[31] Sidong Feng, Mulong Xie, and Chunyang Chen. Efficiency matters: Speeding up automated testing with gui rendering inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 906–918. IEEE, 2023.

[32] Boni García. *Hands-On Selenium WebDriver with Java*. " O'Reilly Media, Inc.", 2022.

[33] Boni García, Micael Gallego, Francisco Gortázar, and Mario Munoz-Organero. A survey of the selenium ecosystem. *Electronics*, 9(7):1067, 2020.

[34] Mark Grechanik, Qing Xie, and Chen Fu. Creating gui testing tools using accessibility technologies. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 243–250. IEEE, 2009.

[35] Sarra Habchi, Guillaume Haben, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 244–255. IEEE, 2022.

[36] William GJ Halfond and Alessandro Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 145–154, 2007.

[37] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 751–762, 2016.

[38] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. Why do record/replay tests of web applications break? In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 180–190. IEEE, 2016.

[39] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217, 2014.

[40] Liu Hongyun, Jiang Xiao, and Ju Hehua. Multi-goal path planning algorithm for mobile robots in grid space. In *2013 25th Chinese Control and Decision Conference (CCDC)*, pages 2872–2876. IEEE, 2013.

[41] Md Hossain, Hyunsook Do, and Ravi Eda. Regression testing for web applications using reusable constraint values. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 312–321. IEEE, 2014.

[42] Shannon C Houck, Lucian Gideon Conway III, and Laura Janelle Gornick. Automated integrative complexity: Current challenges and future directions. *Political Psychology*, 35(5):647–659, 2014.

[43] Xiaoming Hu and Yibo Huang. Research and application of software automated testing based on directed graph. In *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC)*, pages 661–664. IEEE, 2021.

[44] Taufan Fadhilah Iskandar, Muharman Lubis, Tien Fabrianti Kusumasari, and Arif Ridho Lubis. Comparison between client-side and server-side rendering in the web development. In *IOP Conference Series: Materials Science and Engineering*, volume 801, page 012136. IOP Publishing, 2020.

[45] Gunel Jahangirova. Oracle problem in software testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 444–447, 2017.

[46] Thorn Jansen, Fernando Pastor Ricós, Yaping Luo, Kevin Van Der Vlist, Robbert Van Dalen, Pekka Aho, and Tanja EJ Vos. Scriptless gui testing on mobile applications. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 1103–1112. IEEE, 2022.

[47] Bo Jiang, Yaoyue Zhang, Wing Kwong Chan, and Zhenyu Zhang. A systematic study on factors impacting gui traversal-based test case generation techniques for android applications. *IEEE Transactions on Reliability*, 68(3):913–926, 2019.

[48] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[49] Mohammadparsa Karimi. Automated scriptless gui testing aligned with requirements and user stories. In *International Conference on Research Challenges in Information Science*, pages 131–140. Springer, 2024.

[50] Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, and Mika Katara. Model-based testing through a gui. In *International Workshop on Formal Approaches to Software Testing*, pages 16–31. Springer, 2005.

[51] Imran Akhtar Khan and Roopa Singh. Quality assurance and integration testing aspects in web based applications. *ArXiv*, abs/1207.3213, 2012.

[52] Onur Kilincceker, Alper Silistre, Fevzi Belli, and Moharram Challenger. Model-based ideal testing of gui programs–approach and case studies. *Ieee Access*, 9:68966–68984, 2021.

[53] Hiroyuki Kirinuki, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. Nlp-assisted web element identification toward script-free testing. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 639–643. IEEE, 2021.

[54] Inessa V Krasnokutska and Oleksandr S Krasnokutskyi. Implementing e2e tests with cypress and page object model: evolution of approaches. In *CS&SE@ SW*, pages 101–110, 2023.

[55] Rebecca Krosnick and Steve Oney. Understanding the challenges and needs of programmers writing web automation scripts. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9. IEEE, 2021.

[56] 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)*, pages 272–281. IEEE, 2013.

[57] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Approaches and tools for automated end-to-end web testing. In *Advances in Computers*, volume 101, pages 193–237. Elsevier, 2016.

[58] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. Challenges of end-to-end testing with selenium webdriver and how to face them: A survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 339–350. IEEE, 2023.

[59] Maurizio Leotta, Filippo Ricca, and Paolo Tonella. Sidereal: Statistical adaptive generation of robust locators for web testing. *Software Testing, Verification and Reliability*, 31(3):e1767, 2021.

[60] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.

[61] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Pesto: Automated migration of dom-based web tests towards the visual approach. *Software Testing, Verification And Reliability*, 28(4):e1665, 2018.

[62] Grischa Liebel, Emil Alégroth, and Robert Feldt. State-of-practice in gui-based system and acceptance testing: An industrial multiple-case study. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 17–24. IEEE, 2013.

[63] Kai Li Lim, Kah Phooi Seng, Lee Seng Yeong, Li-Minn Ang, and Sue Inn Ch'ng. Pathfinding for the navigation of visually impaired people. *International Journal of Computational Complexity and Intelligent Algorithms*, 1(1):99–114, 2016.

[64] Kai Li Lim, Kah Phooi Seng, Lee Seng Yeong, Li-Minn Ang, and Sue Inn Ch'ng. Uninformed pathfinding: A new approach. *Expert systems with applications*, 42(5):2722–2730, 2015.

[65] Kai Li Lim, Kah Phooi Seng, LS Yeong, SI Ch'ng, and K Ang Li-minn. The boundary iterative-deepening depth-first search algorithm. In *Second International Conference on Advances in Computer and Information Technology: ACIT 2013*, pages 119–124. Institute of Research Engineers and Doctors, LLC, 2013.

[66] Xinyue Liu, Zihe Song, Weike Fang, Wei Yang, and Weihang Wang. Wefix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests. *arXiv preprint arXiv:2402.09745*, 2024.

[67] 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*, pages 94–105, 2016.

[68] 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*, pages 121–130. IEEE, 2008.

[69] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *Proceedings of the 40th international conference on software engineering*, pages 280–290, 2018.

[70] Mirella Martínez, Anna I Esparcia-Alcázar, Tanja EJ Vos, Pekka Aho, and Joan Fons i Cors. Towards automated testing of the internet of things: Results obtained with the testar tool. In *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III 8*, pages 375–385. Springer, 2018.

[71] Scott McMaster and Xun Yuan. Developing a feedback-driven automated testing tool for web applications. In *2012 12th International Conference on Quality Software*, pages 210–213. IEEE, 2012.

[72] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 260–269. IEEE, 2003.

[73] Atif M Memon and Myra B Cohen. Automated testing of gui applications: models, tools, and controlling flakiness. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1479–1480. IEEE, 2013.

[74] Atif M Memon, Mary Lou Soffa, and Martha E Pollack. Coverage criteria for gui testing. In *Proceedings of the 8th European software engineering conference held*

jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pages 256–267, 2001.

[75] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):1–30, 2012.

[76] Fatini Mobaraya, Shahid Ali, et al. Technical analysis of selenium and cypress as functional automation framework for modern web application testing. In *9th International Conference on Computer Science*, 2019.

[77] Thiago Santos de Moura. Automação de testes em aplicações web utilizando uma abordagem ad-hoc. *Universidade Federal de Campina Grande*, 2021.

[78] Thiago Santos de Moura, Everton L. G. Alves, Regina Letícia Santos Felipe, Cláudio de Souza Baptista, Ismael Raimundo da Silva Neto, and Hugo Feitosa de Figueirêdo. Addressing the synchronization challenge in cypress end-to-end tests. In *Proceedings of the XXXVIII Brazilian Symposium on Software Engineering*, 2024.

[79] Thiago Santos de Moura, Everton L. G. Alves, Hugo Feitosa de Figueirêdo, and Cláudio de Souza Baptista. Cytestion: Automated gui testing for web applications. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 388–397, 2023.

[80] Thiago Santos de Moura, Regina Letícia Santos Felipe, Everton L. G. Alves, Pedro Henrique S. C. Gregório, Cláudio de Souza Baptista, and Hugo Feitosa de Figueirêdo. Iterative deepening url-based search: Enhancing gui testing for web applications. In *Proceedings of the XXXVIII Brazilian Symposium on Software Engineering*, 2024.

[81] Thiago Santos de Moura, Francisco Igor de Lima Mendes, Everton L. G. Alves, Ismael Raimundo da Silva Neto, and Cláudio de Souza Baptista. An automatic approach for uniquely discovering actionable elements for systematic gui testing in web applications. In *2024 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2024.

[82] Ad Mulders, Olivia Rodriguez Valdes, Fernando Pastor Ricós, Pekka Aho, Beatriz Marín, and Tanja EJ Vos. State model inference through the gui using run-time test generation. In *International Conference on Research Challenges in Information Science*, pages 546–563. Springer, 2022.

[83] Michel Nass, Emil Alégroth, and Robert Feldt. Why many challenges with gui test automation (will) remain. *Information and Software Technology*, 138:106625, 2021.

[84] Michel Nass, Emil Alegroth, and Robert Feldt. Improving web element localization by using a large language model. *arXiv preprint arXiv:2310.02046*, 2023.

[85] Michel Nass, Emil Alégroth, Robert Feldt, and Riccardo Coppola. Robust web element identification for evolving applications by considering visual overlaps. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 258–268. IEEE, 2023.

[86] Michel Nass, Emil Alégroth, Robert Feldt, Maurizio Leotta, and Filippo Ricca. Similarity-based web element localization for robust test automation. *arXiv preprint arXiv:2208.00677*, 2022.

[87] Dario Olianas, Maurizio Leotta, and Filippo Ricca. Sleepreplacer: a novel tool-based approach for replacing thread sleeps in selenium webdriver test code. *Software Quality Journal*, pages 1–33, 2022.

[88] Dario Olianas, Maurizio Leotta, Filippo Ricca, and Luca Villa. Reducing flakiness in end-to-end test suites: An experience report. In *International Conference on the Quality of Information and Communications Technology*, pages 3–17. Springer, 2021.

[89] Rafael AP Oliveira, Upulee Kanewala, and Paulo A Nardi. Automated test oracles: State of the art, taxonomies, and trends. *Advances in computers*, 95:113–199, 2014.

[90] Yu Pei, Jeongju Sohn, Sarra Habchi, and Mike Papadakis. Traf: Time-based repair for asynchronous wait flaky tests in web testing. *arXiv preprint arXiv:2305.08592*, 2023.

[91] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn Stolee. Wait, wait. no, tell me. analyzing selenium configuration effects on test flakiness. In *2019*

IEEE/ACM 14th International Workshop on Automation of Software Test (AST), pages 7–13. IEEE, 2019.

[92] Sujay Raghavendra. *Python Testing with Selenium: Learn to Implement Different Testing Techniques Using the Selenium WebDriver*. Springer, 2021.

[93] 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*, volume 113, pages 89–133. Elsevier, 2019.

[94] Fernando Pastor Ricós, Pekka Aho, Tanja Vos, Ismael Torres Boigues, Ernesto Calás Blasco, and Héctor Martínez Martínez. Deploying testar to enable remote testing in an industrial ci pipeline: a case-based evaluation. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part I 9*, pages 543–557. Springer, 2020.

[95] Fernando Pastor Ricós, Arend Slomp, Beatriz Marín, Pekka Aho, and Tanja EJ Vos. Distributed state model inference for scriptless gui testing. *Journal of Systems and Software*, 200:111645, 2023.

[96] Olivia Rodríguez-Valdés, Tanja EJ Vos, Pekka Aho, and Beatriz Marín. 30 years of automated gui testing: a bibliometric analysis. In *Quality of Information and Communications Technology: 14th International Conference, QUATIC 2021, Algarve, Portugal, September 8–11, 2021, Proceedings 14*, pages 473–488. Springer, 2021.

[97] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. An empirical analysis of ui-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1585–1597. IEEE, 2021.

[98] Urko Rueda, Tanja EJ Vos, Francisco Almenar, MO Martınez, and Anna I Esparcia-Alcázar. Testar: from academic prototype towards an industry-ready tool for automated testing at the user interface level. *Actas de las XX Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2015)*, pages 236–245, 2015.

[99] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

[100] Leandro N Sabaren, Maximiliano Agustín Mascheroni, Cristina L Greiner, and Emanuel Irrazábal. A systematic literature review in cross-browser testing. *Journal of Computer Science & Technology*, 18, 2018.

[101] Nema Salem, Hala Haneya, Hanin Balbaid, and Manal Asrar. Exploring the maze: A comparative study of path finding algorithms for pac-man game. In *2024 21st Learning and Technology Conference (L&T)*, pages 92–97. IEEE, 2024.

[102] 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*, pages 50–54, 2016.

[103] Shahaf S Shperberg, Steven Danishevski, Ariel Felner, and Nathan R Sturtevant. Iterative-deepening bidirectional heuristic search with restricted memory. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 331–339, 2021.

[104] Alper Silistre, Onur Kilincceker, Fevzi Belli, Moharram Challenger, and Geylani Kardas. Models in graphical user interface testing: Study design. In *2020 Turkish National Software Engineering Symposium (UYMS)*, pages 1–6. IEEE, 2020.

[105] Érica Sousa, Carla Bezerra, and Ivan Machado. Flaky tests in ui: Understanding causes and applying correction strategies. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 398–406, 2023.

[106] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. Pesto: A tool for migrating dom-based to visual web tests. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 65–70. IEEE, 2014.

[107] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 503–514, 2018.

[108] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based gui testing of an android application. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 377–386. IEEE, 2011.

[109] Arne-Michael Torsel. Automated test case generation for web applications from a domain specific model. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pages 137–142. IEEE, 2011.

[110] Pradeep Udupa and S Nithyanandam. An efficient method for testing source code by using test case reduction, prioritization and prioritized parallelization. In *2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS)*, pages 1192–1196. IEEE, 2019.

[111] Arie Van Deursen. Testing web applications with state objects. *Communications of the ACM*, 58(8):36–43, 2015.

[112] Tanja EJ Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, and Ad Mulders. testar–scriptless testing through graphical user interface. *Software Testing, Verification and Reliability*, 31(3):e1771, 2021.

[113] Tanja EJ Vos, Peter M Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015.

[114] Yan Wang, Jianguo Lu, and Jessica Chen. Ts-ids algorithm for query selection in the deep web crawling. In *Web Technologies and Applications: 16th Asia-Pacific Web Conference, APWeb 2014, Changsha, China, September 5-7, 2014. Proceedings 16*, pages 189–200. Springer, 2014.

[115] Thomas Weise, Steffen Bleul, Diana Comes, and Kurt Geihs. Different approaches to semantic web service composition. In *2008 Third International Conference on Internet and Web Applications and Services*, pages 90–96. IEEE, 2008.

[116] Robert B Wen. Url-driven automated testing. In *Proceedings Second Asia-Pacific Conference on Quality Software*, pages 268–272. IEEE, 2001.

[117] Thomas Wetzlmaier, Rudolf Ramler, and Werner Putschögl. A framework for monkey gui testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 416–423. IEEE, 2016.

[118] Lee White and Husain Almezen. Generating test cases for gui responsibilities using complete interaction sequences. In *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*, pages 110–121. IEEE, 2000.

[119] Thomas D White, Gordon Fraser, and Guy J Brown. Improving random gui testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 307–317, 2019.

[120] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[121] Feng Xue, Junsheng Wu, and Tao Zhang. Visual identification of mobile app gui elements for automated robotic testing. *Computational Intelligence and Neuroscience*, 2022(1):4471455, 2022.

[122] Dacong Yan, Shengqian Yang, and Atanas Rountev. Systematic testing for resource leaks in android applications. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 411–420. IEEE, 2013.

[123] Xun Yuan and Atif M Memon. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95, 2009.

[124] Xun Yuan and Atif M Memon. Iterative execution-feedback model-directed gui testing. *Information and Software Technology*, 52(5):559–575, 2010.

[125] Daniel Zimmermann and Anne Koziolek. Gui-based software testing: An automated approach using gpt-4 and selenium webdriver. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 171–174. IEEE, 2023.

Appendix A

The Markup Approach

As there is no formal definition for a web GUI, focusing solely on HTML outcomes can lead to incomplete findings [36]. In practice, a common strategy is to rely on the tester to manually inspect the source code of the web pages and identify (markup) the regions that will become actionable elements in the GUI during the tests. This is the Markup approach. The markups can then be used by the test generation tools.

Figure A.1 exemplifies a React component with a markup. As long as this component is imported and used, the *data-cy* property will exist. To ensure uniqueness for dynamic elements, a hash value could be generated based on the page URL concatenated with a tester-defined value for each instance carrying this property (the *hashCode* function returns along with the *buttonId*). The reuse of components may speed up the marking process. However, it may be impractical to apply this approach in third-party code. For example, it is impossible to apply a markup in a modal component that not allow customization options[1].

```
import React from 'react';
import { hashCode } from './utils';
const CustomButton = ({ buttonId, children }) => (
  <button data-cy={`${hashCode()}-${buttonId}`}>
    {children}
  </button>
);
export default CustomButton;
```

Figure A.1: Example of a React component with markups.

Static and dynamic elements should be annotated with different values. When static ele-

[1] https://ant.design/components/modal

ments have a hash value based on the URL page, every time that URL changes, the element will have a different value and can be considered as a new one. With only the tester-defined value, static elements always maintain the same identification, allowing them to be discovered just once regardless of the GUI state. In contrast, dynamic elements are uniquely discovered based on their current state; they often assume different values after a user action or a change in the URL.

Figure A.2 illustrates how static and dynamic elements should be marked. The menu item *Home Page* is marked with only `data-cy="home"` (a), while the *Download* button has a specific hash associated with the current state (b). This distinction allows for multiple download buttons to exist in the system, each with a unique identification, while the *Home Page* remains consistent regardless of the achieved state.

By using custom properties to signify which elements of a particular GUI state can be acted upon, their properties can guide a scriptless tool. During the test generation process, a tool can keep track of previously used locators to manage which *data-cy* values have not yet been explored. This ensures that dynamic elements are properly explored depending on the state in which they are generated, while static elements are triggered only once.

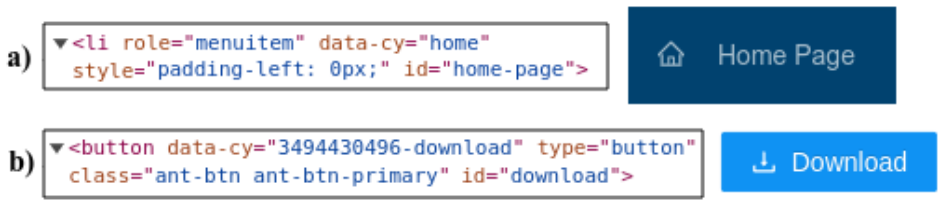

Figure A.2: Example of HTML elements with markups.

The Markup approach can address the challenge of discovery actionable elements and deals with the issue of localization by assuming that testers have assigned unique values to the added custom property. However, it encounters important limitations: i) certain parts of the AUT might be inaccessible to introduce markups (e.g., third-party code); ii) the time required to introduce markups in complex systems may be unfeasible; iii) the testers need to be familiar with the source code in order to introduce proper test values; and iv) the approach relies on the tester to ensure the uniqueness of values representing dynamic elements, which can be error-prone.

Appendix B

Evaluation Studies of UAES

We present two empirical studies designed with the goal of assessing the accuracy of the UAES approach in uniquely discovering actionable elements to enable systematic GUI exploration. For this, we established the following research question:

- RQ: Does UAES help in detecting and selecting actionable elements in web systems?

We compare UAES to the Markup approach. We are using the Markup approach (presented in Appendix A) as a baseline since it relies on experts to manually identify the actionable elements of pages. Therefore, the closer UAES gets to the results of Markup, the better. Moreover, we use the Cytestion version presented in Chapter 4 to explore the systems generating GUI test suites using either Markups or UAES.

To answer the research question, we conducted two empirical studies. The first was carried out using four open-source projects, while the second considers twenty industrial web applications. Both studies follow the same procedure: we select a set of systems as objects. For each object, we requested real testers to manually identify and mark parts of the source codes that will become actionable elements in the pages using the Markup approach. We then ran the Cytestion tool using both approaches. Cytestion systematically explores the pages, dynamically and uniquely discovering all marked elements and storing those found by UAES too. Finally, we compare the set of found actionable elements for both approaches.

It is important to highlight we instrumented the Cytestion tool to be able to simultaneously execute both approaches (Markup and UAES). By doing so, we mitigate any distortion of data collection and randomness that might occur due to individual executions (e.g., class

changing between sections).

To support our analysis, we classify the actionable elements found either by the Markup and UAES approaches in three sets: *same*, elements found by both approaches; *new*, elements discovered only by UAES; and *missed*, elements found by the markup approach but not by UAES.

Our empirical studies ran on a desktop with an Intel Core i7 10700KF processor, 32GB of RAM DDR4 3200MHz, an Nvidia GTX 1060 6GB GDDR5 video card and a SATA SSD 1TB 500Mpbs/s.

B.1 A Study with Open Source Applications

In our first study, four open-source web applications were selected as objects: i) *petclinic* is a SpringBoot application to manage pet owners' registration and scheduling veterinarian visits; ii) *bistro restaurant* is a website developed with HTML, Javascript, and CSS to display restaurant portfolios; iii) *learn educational* is a responsive website that showcases online educational course portfolios; iv) *school educational* is an HTML5 website that implements common functionalities found in school applications. These applications are basic web systems used for academic purposes without utilizing reusable components[1].

Table B.1 provides information on the projects, including their size (KLOC), and the number of test cases generated and executed by Cytestion. Despite their simplicity, those systems offer navigation functionalities, expose relevant information, and support registration operations. This is evidenced by the number of test cases generated.

Project	**KLOC**	**# of Generated Tests**
petclinic	25.7	50
bistro restaurant	33.4	212
learn educational	19	225
school educational	30.2	231

Table B.1: Projects, KLOC and number of generated tests.

[1] `https://gitlab.com/lsi-ufcg/cytestion/loc-study/applications`

We asked two testers to apply the Markup approach in all four systems. The testers had over three years of experience in web application development and GUI testing with Cypress. We instructed them to systematically navigate the code, pinpoint all elements that could be interacted with in a GUI testing scenario, and validate the effects of their actions. The estimated time for completing this task was two days.

Then, we compared the number of discovered elements by Markup and UAES.

B.1.1 Results and Discussion

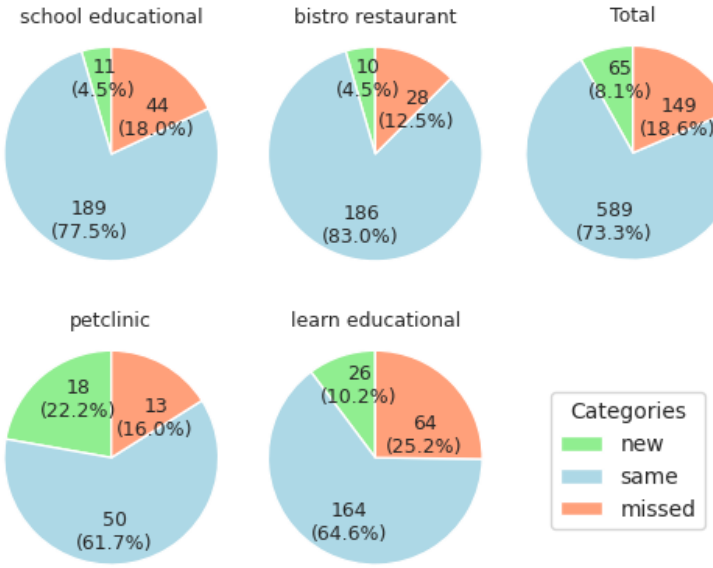

Figure B.1: Open source results: actionable elements equal, new and missed.

Figure B.1 displays all actionable elements found, both in total and per system. We categorized them as *same*, *new*, and *missed*. 73.3% of the elements were discovered by both approaches (*same*). When considering only the elements found by the Markup method (*same* + *missed* = 738 elements), which serves as our baseline, UAES was able to automatically discovery 79.81% of them.

8.1% of the elements were discovered solely by UAES. These were elements overlooked by the testers and thus were not included in the generated test cases. We manually reviewed the newly found elements, all of which were considered valid actionable elements. They included essential functionalities such as table pagination, which had been overlooked by the testers during manual marking. These findings significantly contribute to demonstrate the

effectiveness of UAES in avoiding the error-proneness of manual marking and the associated risks of such strategy.

We investigate the remaining missed elements (20.19%) and find that most of them do not adhere to basic development standards (best practices). For example, we found elements in the same state sharing locators, which is not recommended as they cannot be uniquely discovered. Figure B.2 illustrates such a case, where static navigation elements (Home, About, Courses, Fees, Portfolio, and Contact) appear both in the navigation menu and in the footer. While duplicating these elements for user navigation assistance is common, they lack other attributes such as unique IDs, making it impossible to differentiate between them based solely on text.

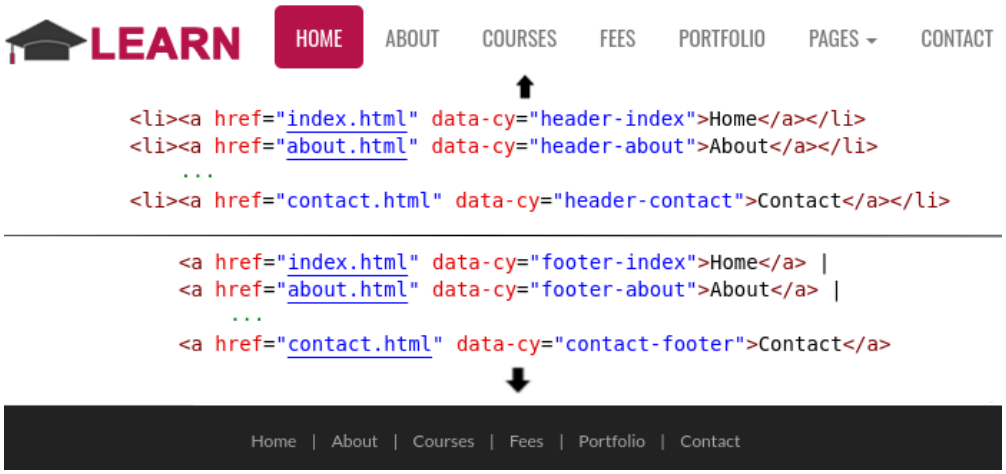

Figure B.2: Example of missed elements due to static navigation elements presented in the navigation menu and the footer.

UAES missed some elements by wrongly considering some dynamic elements as static, which should be processed only once. That occurred because those elements closely resembled elements from previous states. For example, in Figure B.3, the *Add Owner* button appears in both *Find Owners* and the *Owner* states with identical attributes and text. Although the state URLs are different, using this as a distinguishing factor in UAES would result in rediscovering every static element (e.g., menu items) after any change in URL, as demonstrated in Section 3.1. UAES always looks for the elements found in the current URL and the previous one to uniquely discover the elements. Therefore, it is up to developers to address this issue by implementing differentiators for similar elements. Back to the example

(Figure B.3), if either buttons had an ID or different text (e.g., *Save*), UAES would be able to distinguish them and discovery both as different actionable elements.

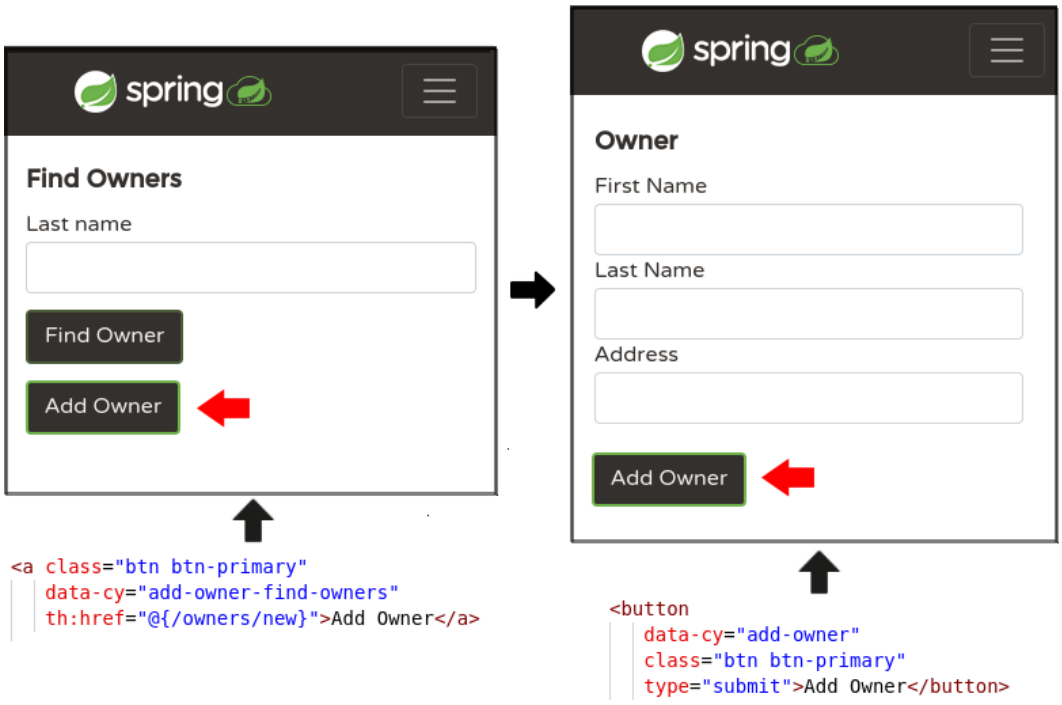

Figure B.3: Example of missed elements due to the lacking of distinct locators.

Finally, we found that some of the missed elements presented tags that lacked any valid locator, indicating development issues and, therefore, the lack of usable information for localization.

Those findings suggest that UAES showed comparable performance to the Markup approach in open-source projects. Moreover, elements that were not identified by the testers, were discovered by UAES, which can improve the generated test suites. Finally, most of the missed elements could be detected by UAES if good programming practices were followed during development. It is crucial to incorporate distinct attributes in the source code for each actionable element to enable effective automation testing. These attributes serve as locators for test scripts or any automated testing tools.

B.2 A Study with Industrial Applications

Application	KLOC	# of Generated Tests
A1	73	316
A2	62	189
A3	52.8	219
A4	75	329
A5	65.9	108
A6	41.4	298
A7	109.1	407
A8	43.7	261
A9	228.9	714
A10	78.1	652
A11	306.8	1683
A12	82	454
A13	77.6	447
A14	58.5	122
A15	42.1	159
A16	68.5	357
A17	37	56
A18	32.4	87
A19	397.9	444
A20	178.5	676

Table B.2: Industrial applications used in our study.

In our second study, we used twenty industrial applications from a partner company. All systems were implemented using React, the Ant Design component library[2], and contain react-based custom components developed by the organization. The systems work as encapsulated modules that provide specialized services within a composite of various fiscal branch

[2] `https://ant.design/`

services offered by the organization and integrated on the web. These applications have been under development and refinement for three years with contributions from six distinct teams within the organization. The development efforts were distributed across 30 repositories, including dependent submodules.

The selected industrial applications demonstrate the defining characteristics of modern web applications, such as responsive design, dynamic content generation, and seamless user interactions. In addition, their teams follow structured implementation methodologies such as component-based architecture to ensure quality, maintainability, and scalability across projects.

Table B.2 shows the size and number of test cases generated and executed by the Cytestion tool. For confidentiality reasons, we name the systems A1 - A20. All projects are already in production and receive weekly updates.

All projects already utilize the Markup approach and Cytestion to manage GUI testing in their development process. Therefore, it falls upon the testers in these teams to manually add the markups to enable the use of the Cytestion tool and execute the GUI test suites. However, time constraints often compel testers to prioritize specific actionable elements when adding markups, leaving certain parts of the system untested. Additionally, third-party code cannot be marked and therefore remains untested as well. Despite that, the strategy of using Cytestion and Markup has been successfully used in the partner company. Over 40 issues in production code have recently been opened due to faults detected using Cytestion.

B.2.1 Results and Discussion

Figure B.4 presents the actionable elements discovered in our second study grouped as *same*, *new*, and *missed*. Again, *same* represents the elements found by both approaches (Markup and UAES); *new* are the ones discovered only by UAES; and *missed* are the elements found by the Markup approach but not by UAES. We can observe that 49.1% of the elements were detected by both approaches. However, UAES was able to discover 95.30% of the elements detected by the Markup approach (*same* / *same* + *missed*), which highlights its significant capability in discovering actionable elements that testers often detect manually.

Based solely on the elements detected by the Markup approach, only 4.7% were not automatically discovered by UAES. This rate is lower than the one captured in our first study

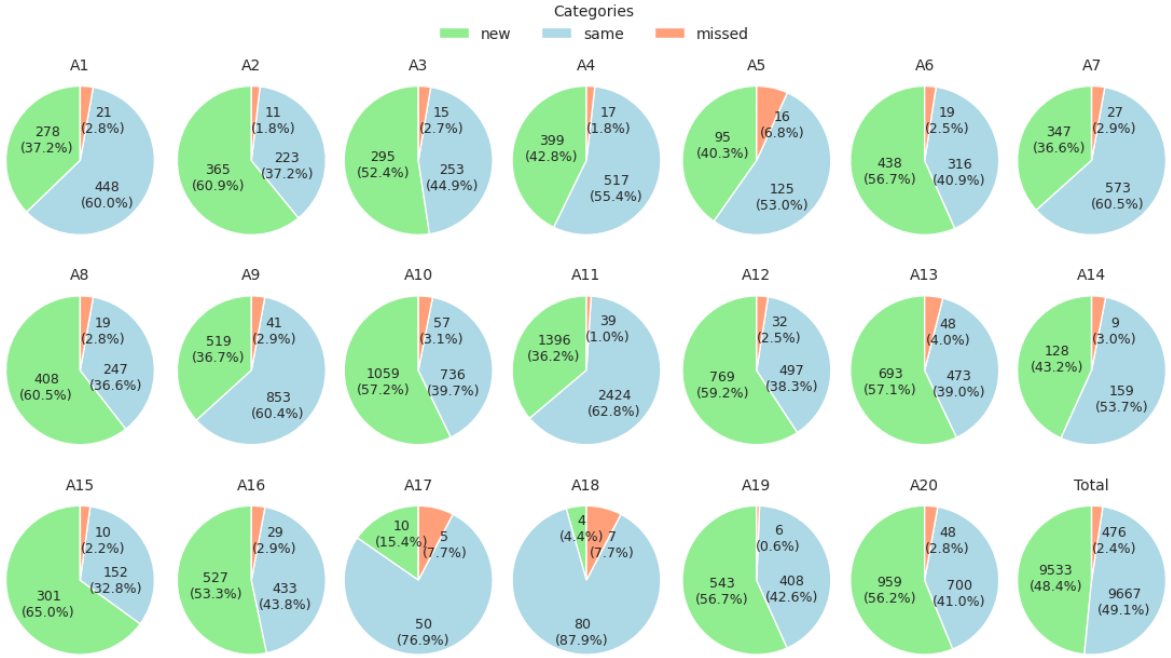

Figure B.4: Results from the industrial study: actionable elements equal, new and missed.

(20.19%). We believe that this drop is due to the strict development policies followed by the project teams when creating web pages. For instance, it is recommended to use attributes like a tooltip for accessibility reasons, which makes a tooltip visible when the user hovers over the element and provides another available locator in most cases.

We investigated those missing elements and found that, in some cases, UAES failed to discover static elements with only inner text as their location, while having other samples at the same state. Similarly, there were instances where dynamic elements were mistakenly classified as static because they existed in the immediately preceding state. Again, this limitation was anticipated and deliberately not treated by UAES because we need to consider the previous state to provide a uniquely discovering, and in these cases, it is up to the developer to implement some differentiation between two entirely distinct elements from related states.

48.4% of the discovered elements were found exclusively by UAES. We believe this high rate is mainly due to the error-prone nature of manual markup. Testers often work under tight schedules, and industrial systems contain thousands of lines of code that need inspection to be properly marked, often resulting in neglected elements. For example, in system A11, which has 3,859 actionable elements, only a subset (2,424) was marked by testers due to these challenges, leaving parts of the application untested. Additionally, third-party com-

ponents cannot be manually marked, but can be accessed by UAES since it considers the resulting page's HTML.

We validated these additional elements (*new*), which were confirmed to be valid actionable elements that were not annotated by the testers: 62% of them were *input* elements, with 71% of those being checkboxes. This highlights the versatility and remarkable capability of UAES in identifying form fields that are difficult to annotate consistently. Furthermore, 22% of the new elements consisted of *buttons* and *anchors*, while another 16% comprised unstructured elements such as actionable list items (*li*), *divs*, and *spans*, which were identified through the class value provided by the tester.

Part of the new elements refer to instances that could not be manually marked by the tester because they are part of a third-party code. For instance, in Figure B.5, the modal component displayed includes confirmation buttons like *Delete*, the modal component is responsible for rendering. Developers who only utilize this component within its default configuration would not be able to add the necessary markup, and consequently, actions on this button would not be considered when generating the GUI test suite.

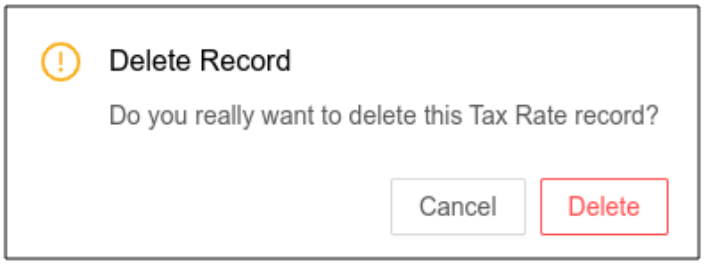

Figure B.5: Buttons identified only using UAES.

It is important to highlight the great value of an automatic approach. The Markup approach requires the tester to manually update the page's source code in order to assist a GUI testing tool in accessing the page elements. This manual strategy can be very challenging in complex systems due to the large number of elements and components, which often consume a significant amount of time and are prone to errors. Our study with industrial applications reflects such a scenario, where a large number of elements were not marked by the tester (*new*) due to practical time constraints or because they were overlooked. UAES is a fully automatic strategy that discovers actionable elements. It does not require any manual intervention and has proven to discover most of the elements to be used in the test generation

process, outperforming a manual strategy.

Based on the results discussed in Sections B.1 and B.2.1, we can conclude that the UAES approach facilitates the discovery and selection of actionable elements for systematic GUI testing with an accuracy of 94.25% ($|same|/|same + missed|$ from both studies), yielding results that outperform the approach that relies on testers' expertise.

Furthermore, the tests that covered the *new* elements detected two new faults in production code within functionalities related to the exposed modal in Figure B.5. The found faults were presented to the projects QA teams and project managers, who acknowledged the issues and allocated resources towards their resolution. This highlights the practical benefits of UAES in enabling systematic GUI testing and enhancing fault detection.

Regarding performance, the execution time of UAES is primarily related to the string search algorithm used for discovering locator keys. In our studies, we used the well-known Aho-Corasick algorithm. The execution time of UAES did not exceed two seconds per system, which we consider acceptable for complex and industrial applications. Additionally, it is important to note that UAES is not tied to the Aho-Corasick algorithm. Therefore, other search algorithms can also be explored.

B.3 Threats to Validity

Our results do not generalize beyond the projects used in our study. However, since we examined a combination of open-source and industrial projects, we consider the set of projects to be a reliable sample representing web applications. Furthermore, it is essential to highlight the substantial representation of industrial projects in our analysis.

The evaluation of UAES effectiveness is primarily based on discovering actionable elements, which may overlook other important aspects of GUI testing such as test coverage, fault detection capabilities, and scalability. However, accurate discovery is fundamental for effective GUI testing because it lays the groundwork for comprehensive test coverage. By automating element discovery, UAES enhances efficiency and reduces human error, contributing to overall testing quality.

The reliance on manual identification of actionable elements by testers, albeit necessary for comparison purposes, introduces the possibility of human error and subjectivity in the

Markup approach, which could impact the accuracy of the comparative analysis. However, the testers who added the markups had over three years of experience in developing web applications and were involved in the team's development process. The annotations were made with the aim of using the Cytestion tool as effectively as possible, and the results obtained so far in fault detection using this tool demonstrate the robustness of the marking carried out.

The rapid evolution of web technologies and frameworks could be seen as a threat to the long-term effectiveness of UAES. Concerns may center on its compatibility with emerging technologies and the complexity of integrating it into existing testing frameworks. However, UAES uses a string-based strategy to search the DOM of pages and extract relevant elements, allowing for seamless integration with scriptless testing tools. Additionally, testers can easily extend the list of actionable elements by defining custom class names, ensuring both effectiveness and ease of integration.

Appendix C

Catalog of Waiting Mechanisms

The *synchronization challenge* is well-known in automated GUI tests. In this context, waiting mechanisms can act as traffic lights for testing, controlling when to stop, when to go, and when to wait. This orchestration may help avoid synchronization issues.

To systematically identify relevant waiting mechanisms, we conducted a literature mapping using common keyword searches combined with snowballing, a technique in which references from selected articles are reviewed to identify additional relevant studies, ensuring a comprehensive review [120]. We subsequently validated the importance of these mechanisms among experienced testers from a partner company. Our review of the literature identified four waiting mechanisms applicable in this context: *Implicit Wait*, *Static Wait*, *Explicit Wait*, and *Fluent Wait* [33, 87, 15]. However, the referenced works focus solely on Selenium test scripts. An exploration of the grey literature introduced a new mechanism specific to Cypress: *Stable DOM Wait*[1]. Furthermore, we introduce a novel mechanism exclusive to Cypress: the *Network Wait*. As a result, we compile a catalog of waiting mechanisms for Cypress tests[*].

Although some mechanisms are not native to Cypress, they can be implemented. For instance, we discuss the implementation of *Explicit Waits* using external Cypress dependencies. Additionally, mechanisms such as *Stable DOM*, which have not been addressed in any related work using Selenium, are currently exclusive to Cypress. Furthermore, the proposed mechanism incorporates a concept unique to Cypress: intercepting requests. Consequently,

[1] `https://github.com/narinluangrath/cypress-wait-for-stable-dom`

[*] A detailed version of the catalog is available on our website: `https://noto.li/mhRfQe`

this new mechanism is only supported by Cypress.

In this section, we present our catalog by discussing each mechanism, exploring its pros and cons, relevance, and implementation details within the Cypress context. To the best of our knowledge, our catalog is the first to compile and demonstrate the use of these waiting mechanisms in the Cypress framework. For each mechanism, we provide a general description, followed by an example of how to implement it with Cypress, and a discussion on implication, benefits, and possible drawbacks. We hope this catalog aids testers in gaining a better understanding and handling of synchronization issues in Cypress tests.

C.1 Implicit Wait

Description: The *Implicit Wait* acts as a single traffic light overseeing waits for the entire test script. Unlike other waiting mechanisms inserted before specific commands, the *Implicit Wait* is a global setting that uniformly impacts all commands. It enforces a time limit for every interaction, ensuring that no command proceeds until it is either feasible to continue or a specified waiting period has elapsed. Selenium employs an *Implicit Wait* [33]. For Cypress, although this mechanism is not officially labeled as such, it includes a general automatic waiting feature.

Usage Scenario with Cypress: When a tester utilizes commands like *click*, *type*, or *clear*, Cypress implicitly enforces those commands to wait for the respective operations to complete. The default timeout for Cypress commands is 4000 milliseconds, but it can be customized based on specific requirements, as exemplified in Listing C.1 (line 1). In this example, the `defaultCommandTimeout` is set to two seconds. Consequently, when a test script interacts with a page element, Cypress waits two seconds to acquire the element before proceeding with the *click* command.

Additionally, Cypress offers a timeout option within commands, allowing custom waiting times for individual commands. For instance, in Listing C.1 (line 4), the timeout for the command that acquires the desired item was set to five seconds. This gives the tester control over how long to wait for the asynchronous call to be fulfilled and for the item to appear on the page. It is important to note that altering the timeout for a specific command extends

the basic characteristics of an *Implicit Wait*, as it solely evaluates the action of the command without considering any other condition.

```
1 Cypress.config('defaultCommandTimeout', 2000);
2
3 cy.get('[id="product-select"]').click();
4 cy.get('[id="item-10"]', { timeout: 5000 }).click();
```

Listing C.1: Example of Implicit Wait configuration in Cypress.

Implications, Benefits and Drawbacks: The use of *Implicit Waits* can simplify wait management by offering a consistent approach across test commands, which may reduce the need for extensive wait coding and improve script readability and maintenance. However, they may cause inefficiencies and unreliable test outcomes due to the use of fixed waiting times that may not match the actual application response times, potentially slowing the tests and leading to flakiness [91, 97]. Complex synchronization issues, such as waiting for visible elements with specific attributes, may require alternative strategies for more precise handling.

C.2 Static Wait

Description: In Selenium, testers often use `Thread.sleep()` to pause the execution of a test script for a specific duration. This pause is not dependent on any conditions or external factors, and the script resumes only after the pre-defined time has elapsed. For Cypress, a similar behavior is achieved using the `wait` command.

Usage Scenario with Cypress: In Listing C.2, we exemplify the use of this mechanism. After interacting with the *Select products* element, we use the `wait` command to pause the test execution for one second (line 2). After the waiting period is over, the test execution performs the click on the item (line 3).

```
1 cy.get('[id="product-select"]').click();
2 cy.wait(1000);
3 cy.get('[id="item-10"]').click();
```

Listing C.2: Example of Static Wait use in a Cypress test script.

Implications, Benefits and Drawbacks: The use of *Static Waits* can impact reliability and efficiency. Although it provides a straightforward synchronization method by pausing execution for a fixed duration, this can lead to inefficiencies in dynamic web applications with varying loading times. Fixed wait times may cause unnecessary delays if the application is ready early or result in test breakages if it takes longer than expected, increasing test flakiness [87, 97]. It may also mask underlying synchronization issues, making it less effective in dynamic testing environments. *Static Waits* should be used moderately and combined with adaptive waiting strategies.

C.3 Explicit Wait

Description: In Selenium, *Explicit Waits* are a powerful mechanism for ensuring that a test script proceeds only when specific conditions are met [92, 105]. This feature is indispensable when dealing with web elements that might take an unforeseeable amount of time to load or become interactive.

For implementing *Explicit Waits* testers often use external dependencies such as *cypress-wait-until*[3]. This dependency introduces a custom command called `waitUntil` that enables implementing *Explicit Waits*. This command helps the tester to use common conditions (e.g., the presence of an element, specific values for global variables) or custom conditions.

```
1 cy.get('[id="product-select"]').click();
2
3 cy.waitUntil(() =>
4   cy.get('body').then(($body) =>
5     $body.find('[id="item-8"]').length > 0),
6       { timeout: 10000 });
7
8 cy.get('[id="item-10"]').click();
```

Listing C.3: Example of Explicit Wait use in Cypress.

Usage Scenario with Cypress: Listing C.3 (line 3) uses `waitUntil` to make the test execution wait up to 10 seconds for the desired item to exist on the web page. The arrow

[3] `https://www.npmjs.com/package/cypress-wait-until`

function encapsulates the condition, ensuring that the button is both present and acquirable. Upon meeting this condition, the script subsequently simulates a user click action on the button using the `click` function (line 8). This approach ensures reliable user interaction simulations in dynamic web environments.

Implications, Benefits and Drawbacks: *Explicit Waits* can enhance test reliability and accuracy by allowing tests to wait for specific conditions. However, this approach also increases script complexity and maintenance requirements, as defining and managing waiting conditions can make test scripts more challenging to read and maintain [93]. Additionally, if not properly implemented, *Explicit Waits* can still result in timeouts or missed conditions, leading to test breakages despite their improved control capabilities.

C.4 Fluent Wait

Description: A *Fluent Wait* can be seen as a customizable version of an *Explicit Wait* [87]. It leverages the same commands as *Explicit Waits*, but with additional parameters that testers can tweak to tailor the waiting strategy. These parameters include options such as polling frequency and custom error messages for timeouts.

```
1  cy.get('[id="product-select"]').click();
2
3  cy.waitUntil(() =>
4    cy.get('body').then(($body) =>
5      $body.find('[id="item-10"]').length > 0),
6        { timeout: 10000,
7          interval: 1000,
8          errorMsg: 'This is a custom error message'
9        });
10
11 cy.get('[id="item-10"]').click();
```

Listing C.4: Example of Fluent Wait use in Cypress.

Usage Scenario with Cypress: In Listing C.4, the tester wants to ensure that the execution of the test continuously checks for the presence of a specific item on the web page over a 10-second period. It evaluates this condition at one-second intervals. The `waitUntil` command

(line 3) encapsulates this condition and introduces the `interval` and `errorMsg` parameters. The `interval` parameter specifies that the evaluation should happen every one second, and the `errorMsg` parameter provides a custom error message in case of a timeout.

Implications, Benefits and Drawbacks: *Fluent Wait's* flexibility enables customization of wait durations and check frequencies, enhancing test efficiency and reducing flakiness by adapting to dynamic application behavior. To fully exploit this flexibility, it is important to test various parameter configurations [87]. However, this adaptability can also increase script complexity and maintenance challenges. Poorly designed polling strategies may result in excessive resource use or longer waiting times, affecting overall performance. Thus, while *Fluent Wait* offers significant benefits, careful management is crucial.

C.5 Stable DOM Wait

Description: This mechanism addresses a critical need in GUI testing, which is to ensure the stability of the Document Object Model (DOM) for a specified duration before allowing the test flow to proceed. This is achieved by using the MutationObserver[4] interface to detect alterations in the DOM tree. This mechanism is particularly useful for visual regression testing, where minor DOM changes can impact a web page's visual appearance. Ensuring a stable DOM helps in capturing accurate visual snapshots.

Usage Scenario with Cypress: For Cypress users, this mechanism is accessible through an external library known as *cypress-wait-for-stable-dom*[5]. After importing this library, a tester can access the `waitForStableDOM` function, as shown in Listing C.5.

```
1 cy.get('[id="product-select"]').click();
2 cy.waitForStableDOM({ pollInterval: 500, timeout: 5000 });
3 cy.get('[id="item-10"]').click();
```

Listing C.5: Example of Stable DOM Wait use in Cypress.

[4] https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver
[5] https://www.npmjs.com/package/cypress-wait-for-stable-dom

In this example, the `waitForStableDOM` function is called with various parameters, including `pollInterval`, which specifies the duration for which the DOM must remain stable, and `timeout`, which sets the time limit for waiting. Importantly, the code in line 3 is executed only when the preceding command has completed its wait execution. This ensures that a test script interacts with a stable DOM, enhancing the reliability of its results.

Implications, Benefits and Drawbacks: *Stable DOM Waits* play a crucial role in test automation, particularly in scenarios requiring visual consistency and accurate DOM interactions. By ensuring that the DOM remains stable for a specified duration, this approach reduces false positives from transient changes and provides reliable test results. However, if the DOM is frequently updated, this mechanism may introduce delays in test execution. Balancing poll intervals and timeouts is essential to maintain both reliability and efficiency.

C.6 Network Wait

Description: *Network Wait* is a novel mechanism that closely observes requests made during test execution. Its purpose is to ensure that a test proceeds only when all these requests are completed. As new requests are initiated, a counter is incremented; upon request completion, the same counter is decremented. The test execution resumes its course when this counter reaches zero, remaining in this state for a predetermined period of time.

To the best of our knowledge, the *Network Wait* mechanism is unique to Cypress because it can intercept requests - a feature not available in Selenium. This mechanism monitors client-server communication during test execution, whereas current wait mechanisms focus solely on the DOM. Therefore, existing methods cannot replicate *Network Wait* behavior. We developed and published our own external NPM dependency for *Network Wait*, which is available to the Cypress testing community[6].

Usage Scenario with Cypress: Listing C.6 presents an implementation of this mechanism through Cypress native `intercept` command. The `routeHandler` function manages the `pendingCount` variable, which decreases with each response, ensuring patient waiting until

[6] `https://www.npmjs.com/package/cypress-network-wait`

all requests are completed. In our example, intercepts are configured to cover all URLs and HTTP methods (line 7), allowing the process to wait until the counter reaches zero before the test proceeds (lines 10-13).

```
1  let pendingCount = 0;
2  function routeHandler(request) {
3    pendingAPICount++;
4    request.on('response', () => pendingCount--);
5  }
6
7  cy.intercept('*', '*', routeHandler);
8
9  Cypress.Commands.add('waitNetworkFinished', () => {
10   while (pendingCount > 0) {
11     cy.log('Waiting for pending requests.');
12     cy.wait(500);
13   }
14   cy.log('All requests completed!');
15 });
16
17 cy.get('[id="product-select"]').click();
18 cy.waitNetworkFinished();
19 cy.get('[id="item-10"]').click();
```

Listing C.6: Simplified implementation of Network Wait and its use in Cypress.

This *Network Wait* is recommended in situations where the subsequent testing step is uncertain and a more generic waiting strategy is required, which is often the case for scriptless GUI testing [21]. It dynamically adapts to network conditions, reducing waiting times on faster networks and extending them on slower networks. This adaptability may lead to an effective balance between precision and efficiency during test execution.

Implications, Benefits and Drawbacks: The *Network Wait* mechanism dynamically adjusts to network conditions, ensuring tests are executed accurately without premature execution or unnecessary delays. This approach is beneficial for scenarios with unpredictable network behavior and multiple asynchronous requests. However, it requires careful monitoring to manage edge cases like incomplete or stalled requests. Additionally, the use of polling loops (e.g., `cy.wait(500)`) can introduce minor delays, affecting overall test execution time.

Appendix D

Studies on Synchronization Issues and Waiting Mechanisms

We present the empirical studies performed to assess the impact that synchronization issues can cause in test suites. We also evaluate the efficiency of various waiting mechanisms to address this challenge. Our investigation is guided by the following research questions:

- RQ_1: How much of a test suite can break due to synchronization issues?
- RQ_2: How effective are waiting strategies to mitigate synchronization issues?

RQ_1 relates to our hypothesis that synchronization issues may cause test breakages due to temporal misalignments between test execution and system response, especially in repeated executions (e.g., 50 times) or less controlled environments. By addressing RQ_1, we hope to provide a practical understanding of how synchronization issues affect test suites. Our second hypothesis, related to RQ_2, is that waiting mechanisms could minimize synchronization issues in Cypress tests. Therefore, we aim to assess the performance of various waiting strategies in resolving these challenges.

To answer the research questions, we conducted empirical studies in two different scenarios. The first considers a test suite developed for an open-source scalable online store application, while the second considers a suite from an industrial project. In both cases, we investigated the impact of synchronization issues on the test suites and compared the effectiveness of different waiting mechanisms. Specifically, we compared *Static Wait* (with a

default one-second delay), *Stable DOM Wait* (with a default one-second interval), *Network Wait*, and *Explicit Wait*. Although *Fluent Wait* and *Implicit Wait* were listed in our catalog (Appendix C), we did not include them in our studies because the former can be seen as a customizable *Explicit Wait*, and the latter is an inherent setting in Cypress, automatically applied to all test scripts.

In our studies, we used Cypress version 12.17.2 in its default configuration (default timeout command of four seconds). The empirical studies were executed on a Desktop equipped with an Intel Core i7 10700KF processor, 32GB DDR4 3200MHz RAM, Nvidia GTX 1060 6GB GDDR5 graphics card, and a 1TB SATA SSD with 500Mbps/s.

D.1 Investigating the Impact of Synchronization Issues

For the first study, we selected the Sylius Standard[1] (version 1.12.4). Sylius is an open-source eCommerce framework known for its modular and flexible architecture, making it well-suited for developing customized online stores. It offers extensive configuration options and advanced features to manage catalogs, orders, payments, and shipping, among other aspects of eCommerce. We selected this application as object due to its wide range of functionalities. Additionally, this project is easy to use and popular, as evidenced by its high number of stars and active community engagement on GitHub. During the study, we used the sample data provided by the framework to populate the database and executed the entire web application within a single Docker container using an image built from its repository[2].

Synchronization issues often occur in applications where modules are potentially hosted on distinct machines, including the front-end, back-end, and database. Another potential scenario involves complex functionalities that trigger asynchronous calls and lead to prolonged database queries. To have a controlled experiment in which we investigate the potential degradation of the test suites, we emulate such scenarios. We deliberately introduced various levels of network delays into our testing environment. This process was inspired from a related work that discusses how varying loading times can contribute to test flakiness [87]. For that, we used the command-line utility tool Traffic Control[3]. Traffic Control is instru-

[1] `https://github.com/Sylius/Sylius-Standard`, `https://github.com/Sylius/Sylius`

[2] `https://gitlab.com/lsi-ufcg/cytestion/sync-study/sylius-showcase`

[3] `https://linux.die.net/man/8/tc`

mental for network traffic management and manipulation, with a specific focus on bandwidth regulation and control.

A total of five Sylius application instances were created for the purpose of our study. Four of these were equipped with Traffic Control to introduce delays, resulting in the following treatments: *Without delay*, *500 ms delay*, *1000 ms delay*, *1500 ms delay*, and *2000 ms delay*. These values were introduced with a reasonably significant gradual increase to simulate different network delays. Each instance has a unique Docker image tag, facilitating effortless execution and repeated database restoration. The artifacts of our study are available on our website[4].

To create the test suite, we recruited 48 students. They are last year Computer Science students with a solid programming and testing background. Prior the study, they went through a comprehensive two-hour training session on GUI testing and Cypress. The students generated test cases for 25 distinct Sylius features. Each feature corresponded to a different part of the application, such as products, payments, and more. To maintain flexibility and emulate real-world test suite creation, we intentionally refrained from imposing specific requirements on what elements should be tested within each section. Instead, each student had the freedom to create test cases based on their understanding of the assigned section and the system behavior.

From the test cases created by the students, we validated and selected 200 tests to be used in our study. This selection was manually done by the first author to create a suite with valid test cases. We considered valid test cases that include interactions with multiple actionable elements and assertions confirming the expected behaviors. Notably, some students incorporated static waits in their tests, which resembles the behavior of experienced testers who would try to predict and treat possible synchronization issues. Also, it is important to highlight that we neither intentionally designed the testing sections nor guided the test case creation to include synchronization issues, as those issues could potentially arise in any part of an application or test case.

For the five created Sylius instances (*Without delay*, *500 ms delay*, *1000 ms delay*, *1500 ms delay*, and *2000 ms delay*), we executed the created suite 50 times, resulting in a total of 10,000 test case executions per instance. By repeating a test case execution for a given system

[4] `https://gitlab.com/lsi-ufcg/cytestion/sync-study/execute-study`

instance, we intended to be able to detect possible synchronization issues and evaluate the suite reliability. A similar approach was performed in related studies [87, 88].

D.1.1 Results and Discussion

Out of the 200 test cases, 35 (17.5%) exhibited flakiness during at least one execution. In Figure D.1, these 35 test cases are ordered by the total number of breakages per instance. Interestingly, we found ten test cases that experienced the highest breakage rates, which means they were the most affected by delays (test cases 1-10). As the delays increased, the number of executions experiencing breakage drastically increased for those tests. We manually investigated all 35 tests and found that tests 1-10 interact with elements that trigger asynchronous calls. For instance, some of these test cases involve selecting a product, which is listed only after clicking the *Select products* drop-down input, creating a test breakage similar to the one described in Section 3.2. Another example is a test case that opens and closes a large image. This image is loaded after a clicking action. This test encountered breakage when a close action was performed when the image was not yet fully loaded. Tests 11-35, on the other hand, present a lower number of breakages and their flakiness were due to issues not related to synchronization, such as logic faults and dependencies on randomly generated data. Therefore, our second study (Section D.2) focuses only on the ten test cases that include synchronization issues (tests 1-10).

Even though delays were applied universally, the impact on the remaining 165 test cases may have been less evident due to Cypress's native waiting mechanisms for page loading and the default implicit waits. However, the presence of asynchronous calls can still introduce synchronization issues, emphasizing the need to address such problems to maintain the robustness of a test suite.

Synchronization issues often reflects in test flakiness. Therefore, the same test case may present different outputs (pass or fail) in different executions. Figure D.2 illustrates test case and suite breakages for test cases 1-10 across different configurations. Each bar represents the breakage rate of test cases (out of 500 = 10 test cases x 50 runs) and test suite executions (out of 50 runs). While Figure D.2-a refers to test cases runs that experienced breakage, Figure D.2-b presents test suite breakages. A test suite breaks when at least one of the 10 tests within the suite experiences breakage.

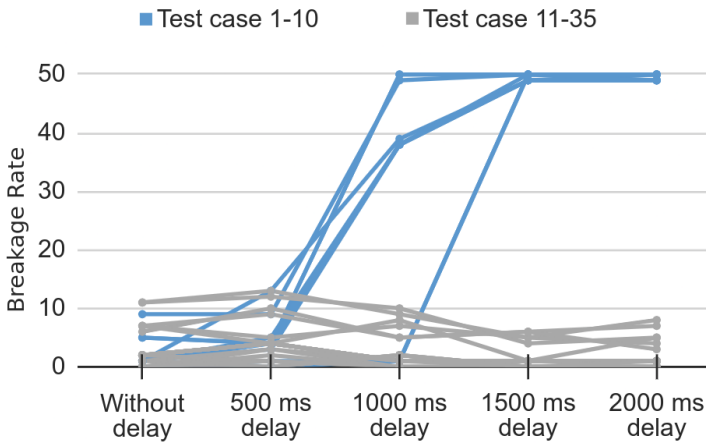

Figure D.1: Number of test case breakages by Sylius instance.

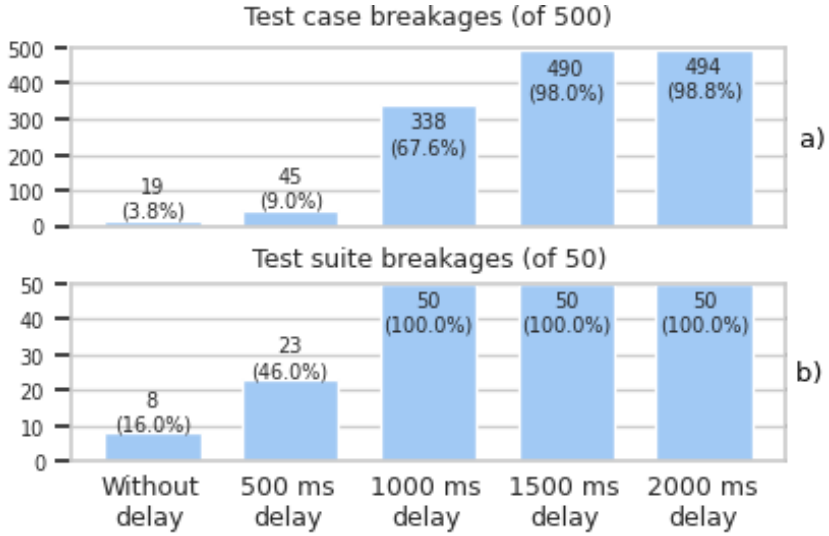

Figure D.2: Breakage rates by test case and suite runs.

We can see that, even for the scenario without delays, synchronization issues were found (19 breakages across 8 suite executions), compromising the test suite reliability by 16%. When a delay of 500 ms was introduced, this rate escalated to 46% across 23 suite runs. For greater delays (1000 ms, 1500 ms, and 2000 ms) all suite executions experienced breakage. These results evidence the substantial impact that synchronization issues might have on the reliability of a test suite (RQ_1). Even minor delays can cause significant increases in both individual test cases and overall suite breakages.

D.2 Evaluating Different Waiting Mechanisms

Based on the results of our first study, we selected the 10 test cases most affected by synchronization issues (test cases 1-10). For each of those tests, we created four refactored versions. In each version, we applied a different waiting mechanism (*Static Wait*, *Stable DOM Wait*, *Network Wait*, and *Explicit Wait*) trying to fix the found synchronization issues. Finally, we reran the suite 50 times on the five Sylius instances. The goal of this second study is to identify the best waiting mechanism that could help a tester cope with the synchronization challenge (RQ_2). It is important to highlight that the refactorings were manually applied by the first author in all locations identified as having a synchronization issue in our first study. The refactorings were later revised and confirmed by the third and fourth authors.

D.2.1 Results and Discussion

We present the results of our second study in Figure D.3 that depicts the occurrences of test case and test suite breakages, along with the average test suite execution time. These results are showcased for the five delay settings (Without delay, 500 ms, 1000 ms, 1500 ms, and 2000 ms) and four waiting mechanisms (*Static Wait*, *Stable DOM Wait*, *Network Wait*, and *Explicit Wait*).

For the *Without delay* scenario, only the test artifacts that used *Static Wait* exhibited test case (0.02%) and suite breakages (2.0%). However, for the *500 ms delay*, all waiting mechanisms mitigated all synchronization issues, showing 0.0% of breakage rate for test case and test suite.

As for the *1000 ms delay* setting, *Static Wait* experienced breakage rates of 16.8% for test cases and 70.0% for test suites. Furthermore, *Stable DOM Wait* showed breakages at 2.0% for test cases and 16.0% for test suites. On the other hand, *Network Wait* and *Explicit Wait* remained with no breakages.

For the *1500 ms delay*, breakages notably increased: 44.2% and 98.0% test case and suite breakage rates, respectively, for *Static Wait*. *Stable DOM Wait* rates increase to 27.0% for test cases and 94.0% for the test suite. On the other hand, *Network Wait* and *Explicit Wait* remained without breakages.

Finally, for the most degraded setting (*2000 ms delay*), breakage rates rose to 60.2% for

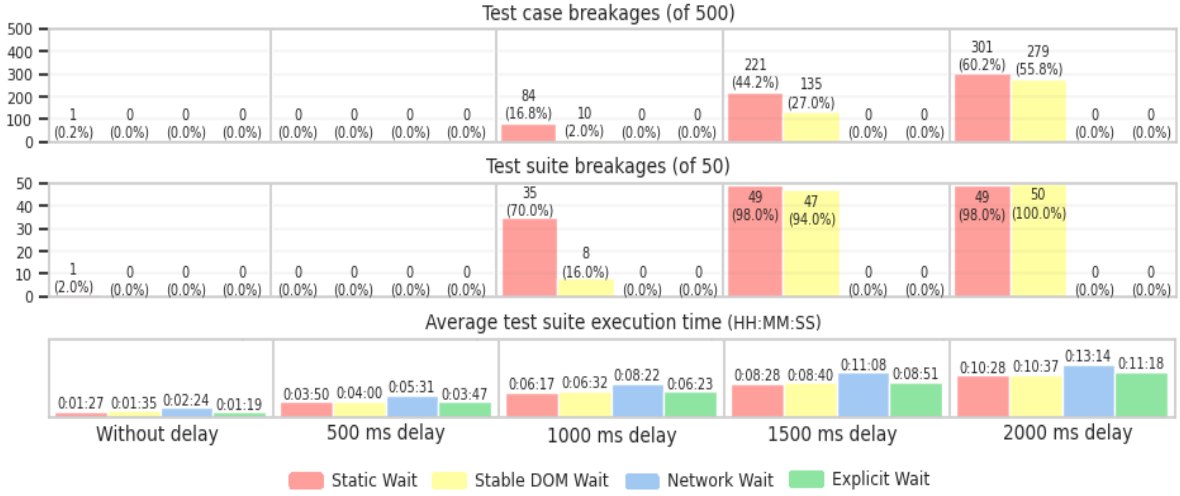

Figure D.3: Results from the Sylius study: test case breakages, suite run breakages, and average suite execution time.

Static Wait and 98% for test cases and test suite, respectively. Similarly, the *Stable DOM Wait* mechanism increased breakage rates to 55.8% for test cases and 100% for the test suite. Again, *Network Wait* and *Explicit Wait* maintained their effectiveness without presenting any breakages.

We conducted a manual investigation of these breakages. For *Static Wait*, most of the breakages occurred because the set waiting time was not sufficient to overcome the synchronization issues. As for the *Stable DOM Wait* breakages, the common found issue was due to the solution inherent assumption of DOM stability while rendering the loading of the executed action. For extended periods, it led to premature execution release and breakages.

To assess breakage rate differences, we applied the Fisher's exact test, along with odds ratios for effect size evaluation [13]. For the scenarios *Without delay* and *500 ms delay*, all mechanisms showed similar performance (p-value = 1). However, at delays of 1000 ms and 1500 ms, *Network Wait* and *Explicit Wait* outperformed *Stable DOM Wait*, which in turn surpassed *Static Wait* (p-value < 0.05). The same pattern occurred at a *2000 ms* delay, except where *Stable DOM Wait* equaled *Static Wait* (p-value = 0.1784347).

Network Wait and *Explicit Wait* exhibited no breakages during the experiment. However, it is essential to weigh trade-offs. On average, *Network Wait* took slightly longer to execute the test suite compared to *Explicit Wait* across all delay settings, with a range of 65 to 137 seconds. When evaluating waiting mechanisms, it is crucial to consider both breakage rates

and execution time, which can vary across projects. The prolonged execution times with *Network Wait* can be attributed to waiting for non-essential requests. Although effective in reducing breakages, it may come at the cost of longer execution times.

We conducted a Mann-Whitney U test for each delay configuration to evaluate execution time differences between *Network Wait* and *Explicit Wait* [13]. In all delay configurations, significant p-values (all $< 10^{-17}$) were observed, indicating substantial differences. Vargha-Delaney effect sizes ranged from -16.28471 to -8.055329, indicating consistently longer execution times for Network Wait compared to *Explicit Wait*.

These results contribute to answer RQ_2 by highlighting the diverse effectiveness of waiting mechanisms in handling synchronization issues. Notably, both *Network Wait* and *Explicit Wait* demonstrate resilience, exhibiting no breakages across various delay scenarios. However, the observed differences in execution times underscore the significance of a strategic selection tailored to each scenario.

D.3 A Case Study with an Industrial Application

To complement the previous results and assess the impact of synchronization related issues and the use of various waiting mechanisms in a real-world scenario, we conducted a case study involving an industrial application from a partner company. This project is a React-based[5] application designed for managing government auditing processes and features a robust Cypress test suite that runs multiple times a day. It works as a regression suite that the development team runs before any modification is integrated into the main codebase.

Table D.1 presents the project metrics, including lines of code, lines of test code, test cases, and waiting points. Waiting points are specific code areas in the test suite where synchronization between the test script and the AUT is necessary. The reported 866 waiting points were identified by the project testing team. It was reported that synchronization issues have caused several practical problems to the team such as release delays and wasted efforts evaluating false bugs. To address this, the team applied a one-second *Static Wait* to each identified waiting point.

The web application operates entirely within a single Docker container using the desig-

[5] `https://react.dev`

Metric	**Value**
Lines of Code	136,128
Lines of Test Code	5,933
Test Cases	169
Waiting Points	866

Table D.1: Metrics for the study object.

nated code version. Database, front- and back-end components are preloaded before running the Cypress test suite. This scenario resembles the environment reported in our first studies with Sylius (Sections D.1 and D.2).

For this study, we used the following procedure: the existing project suite already uses the *Static Wait* mechanism, which we called the *Static Wait* version. We created three additional versions of this test suite, refactoring all waiting points to apply *Stable DOM Wait*, *Network Wait*, and *Explicit Wait*. Similarly to our previous studies, we executed each version of the suite 50 times within the AUT, and we registered the number of test case breakages, test suite breakages, as well as the average test suite execution time.

D.3.1 Results and Discussion

Regarding RQ_1, the data obtained from this study reinforces that the synchronization challenge can significantly affect the reliability of a test suite. As shown by the test results in Figure D.4, the *Static Wait* version (original version) experienced 70 test case breakages out of 8450 (0.83%). While this percentage might seem relatively low, it becomes concerning when considering that these breakages occurred in 16 different test suite executions, representing 32%. We interviewed the project developers and testers that confirmed the occurrence of these flaky tests on a daily basis and that their common practice to deal with that issue is to re-execute the test suite, which often is found as very costly. This underscores the impact of synchronization challenges on the reliability of the test suite.

Regarding RQ_2, while both the *Network Wait* and *Explicit Wait* versions resulted in identical results for test case and test suite breakages, significant differences emerged in average

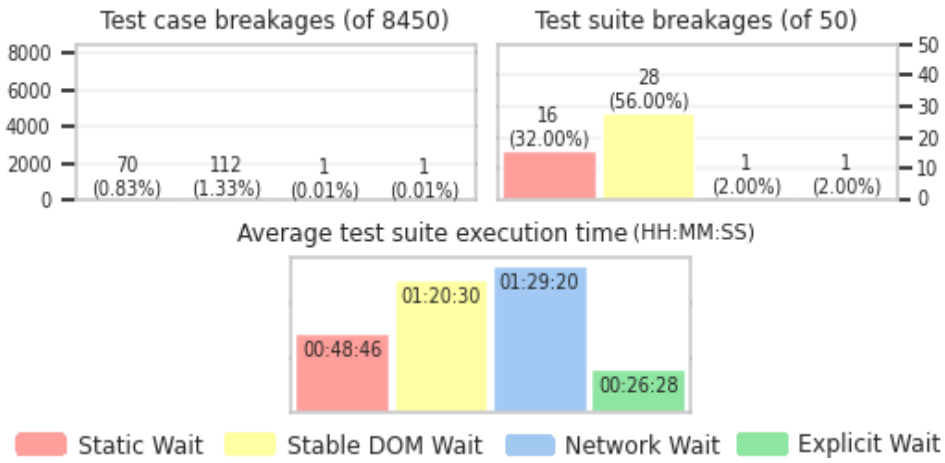

Figure D.4: Case study results, including test cases breakages, suite run breakages and average test suite execution time.

execution times. *Explicit Wait* outperformed with an average test suite execution time of 00:26:28, surpassing even the *Static Wait* version (00:48:26), while *Network Wait* exhibited the longest average execution time at 01:29:20.

However, it is crucial to consider the implementation complexities associated with these mechanisms. While implementing the *Network Wait* involved a simple replace operation, implementing the *Explicit Wait* was a very complex task. This complexity stemmed from the necessity to establish suitable wait conditions, requiring coordinated efforts from multiple authors over approximately four hours to address all 866 waiting points. Such complexities should be factored into the decision-making process when selecting the preferred mechanism.

Although the results regarding breakage rates were similar in our case study and in the Sylius one, they differed for execution time. We attribute this variation to the inherent nature of the *Network Wait* mechanism, which waits for all requests, including unnecessary ones, to be fulfilled before proceeding with the test execution. This difference may be attributed to the industrial case study inherently having more network requests than Sylius, influencing the overall execution time. In contrast, *Explicit Wait* sets conditions that indirectly require minimal or no requests before a specific element becomes explicitly available on the GUI.

Finally, we analyzed the two breakages that remained for both *Network Wait* and *Explicit Wait*. We observed that those breakages refer to waiting points that were not noticed/identified by the project testing team. They only became apparent after undergoing 100 test suite

executions. We later included those point to the refactored versions of *Network Wait* and *Explicit Wait* and no breakages were found. This revelation was attributed to a temporary limitation of hardware resources during the execution, exposing a waiting point that went unnoticed because it normally loads quickly.

D.4 Learned Lessons

Taking into account the results of our studies, we can answer RQ_1 and RQ_2:

RQ*$_1$*: *Synchronization issues can significantly impact a test suite reliability. For some scenarios, we found that up to 32% of a suite can fail due to those issues.*

RQ*$_2$*: *Network Wait and Explicit Wait were the most effective mechanisms, with equal breakage rates for test cases and suites, but Explicit Wait had better execution times.*

We believe that the results achieved can be valuable for testers implementing GUI Cypress suite. They evidence the importance of caution when writing test scenarios involving asynchronous calls. By understanding the common pitfalls and sources of flakiness, testers can act proactively to reduce breakages, leading to more robust and reliable tests. Moreover, by knowing how to apply the different waiting mechanisms in Cypress scripts, testers can have the proper tools to deal with such challenges and fine-tune their test cases.

Explicit Wait is a well-known mechanism for Selenium suites. Our findings showed its effectiveness for Cypress suites as well. However, it is important to weight the associated complexity to implement it, requiring thorough understanding of subsequent testing actions and conditions, which can be error-prone and affect the overall script efficiency.

Testers can identify waiting points reactively or proactively. Reactively, after a breakage, testers can rerun the test with Cypress's GUI, monitoring the logging to pinpoint and address synchronization problems. On the other hand, when working proactively, testers can identify waiting points before they cause breakages, especially for elements triggering asynchronous events like select input and dropdown interactions. Incorporating waiting mechanisms in relevant test cases can preempt potential issues.

We believe our results can also benefit tool builders and researches. New tools can in-

tegrate effective waiting mechanisms like *Network Wait* and *Explicit Wait*, providing testers with more reliable synchronization solutions. The new *Network Wait* mechanism enhances software testing with its generic, cost-effective approach, offering opportunities for further exploration by tool makers. For instance, scriptless GUI testing often deals with non-deterministic scenarios where the goal is to navigate through the application in search of visible failures [79]. Traditional waiting mechanisms often fall short due to the difficulty in determining the appropriate wait conditions. *Network Wait* addresses this challenge, providing a solution that does not require the definition of a specific condition. Additionally, researchers can use our studies to validate existing techniques and develop advanced synchronization algorithms and tools for automated testing practices, resulting in more robust solutions for real-world applications.

D.5 Threats to Validity

In terms of external validity, our results cannot be generalized beyond the context of the specific projects used in our study. Since we based our study on two web applications and their test suites, one open-source and one industrial, we acknowledge that the sample lacks diversity. Nevertheless, given the substantial size of these systems and the number of test cases they encompass, we argue that they serve as good representatives of the broader web application and GUI testing domain. Furthermore, in the Sylius study, we utilize a test suite generated by students, whereas the second study employs a test suite developed by a team of professionals. Despite their differences, both suites include synchronization issues, indicating that tests with synchronization issues are not solely attributable to lack of experience and can manifest in any GUI test suite.

In terms of internal validity, we utilized the Cypress default settings to configure command timeout. This implies that our evaluations of waiting mechanisms can be seen as a blend of implicit and other waits. We acknowledge that different configurations for those settings could impact our results. Nevertheless, these settings are intrinsic to the framework's API that requires some value assignment. We opted to use Cypress default settings to resemble how most testers use it. Furthermore, our empirical study involved two manual actions by the first author, potentially threatening internal validity: the manual validation

and selection of tests, as well as the subsequent manual refactoring to incorporate waiting mechanisms. To mitigate these threats, the selected tests and refactored code were reviewed and validated by the third and fifth authors.

In terms of conclusion validity, the use of external libraries, such as *cypress-wait-for-stable-dom* and *cypress-wait-until*, poses a potential validity threat to our study conclusions due to their impact on our results. We acknowledge that errors or limitations in these libraries could influence our findings, and the validity of our conclusions depends on this factor. To address this concern, we meticulously reviewed the documentation and source code of these libraries and performed additional validation testing.

In terms of construct validity, we compared *Static Wait* (one-second delay) and *Stable DOM Wait* (one-second interval). While different values could produce different outcomes, we chose one-second delays because of their common usage in the projects we studied. However, this choice might have affected our findings, and varying delay times could influence our conclusions.

Appendix E

Evaluation Studies of IDUBS

We present the empirical studies conducted to evaluate IDUBS in the context of GUI testing. For that, we compared IDUBS with a baseline strategy (IDS) focusing on four key aspects: test case execution time, number of revisited states, test suite coverage, and number of detected faults. To guide our investigation we established two research questions:

- RQ_1: Can IDUBS effectively reduce GUI testing costs?
- RQ_2: Does IDUBS maintain test suite performance?

RQ_1 explores the redundancy of GUI state visits on IDUBS tests, which can directly influence test case execution time, a critical factor when considering costs. Meanwhile, RQ_2 compares the generated suites performance regarding code coverage and capabilities to detect faults, when compared to IDS.

We conducted two empirical studies to address these questions. The first examined a diverse set of industrial projects, while the second focused on open-source projects. Both studies used the Cytestion tool for generating GUI test suites, using the same configuration. The prototype version of Cytestion employs IDS for test case generation. We extended the Cytestion infrastructure by implementing IDUBS, creating a new version (Cytestion IDUBS). This new version is available in our repository[1]. With Cytestion IDUBS, we compared the performance of the IDS and IDUBS algorithms across different projects. Each algorithm was executed separately, as they do not incorporate aleatory aspects. The generated suites systematically and exhaustively explored the AUTs.

[1] https://gitlab.com/lsi-ufcg/cytestion/cytestion/-/tags/2.0

E.1 Metrics and Configuration

We established four metrics to address our research questions. For RQ_1, we use *execution time for each test case* and *frequency of visited states in a test suite*. As our goal is to minimize testing costs, we assess this aspect considering test execution time and test suite redundancy. Faster execution and fewer visited states signify a more efficient and less repetitive test suite.

A cost-effective test suite should maintain its testing efficacy. For RQ_2, we evaluate performance using code coverage and the number of visible failures detected. Code coverage is measured with an official Cypress dependency[2] that quantifies frontend code elements. This dependency uses Istanbul[3] to instrument the source code, enabling Cypress to analyze it during execution.

We perform different statistical tests to support our conclusions based on the collected data [13]. We used the Wilcoxon rank sum test to compare the top 5 most visited states. For execution times, we used the Mann-Whitney U test, which assesses differences in continuous measurements. To compare coverage rates, we applied the Wilcoxon signed-rank test, suitable for paired data and non-normal distributions.

In our Cytestion setup, we need to configure a generic oracle to assess the identified states. The default configuration includes checking for: (i) failure messages in the browser console; (ii) HTTP status codes in the 400 or 500 families following server requests; or (iii) default error messages in the GUI such as “Error” and “Exception”. However, due to its generic nature, this approach may result in false positives and required additional manual analysis to confirm the presence of actual faults.

Despite their deterministic nature, the algorithms may produce varying numbers of test cases due to different exploration strategies. To compare directly, we map the corresponding tests of the generated suites. Moreover, to mitigate execution time outliers caused by external factors like network latency changes, we applied the Winsorization transformation [13], which limits extreme values to reduce the impact of spurious outliers.

Our empirical studies executed on a desktop with an Intel Core i7 10700KF processor, 32GB of RAM DDR4 3200MHz, an Nvidia GTX 1060 6GB GDDR5 video card and a SATA

[2] `https://github.com/cypress-io/code-coverage`

[3] `https://istanbul.js.org/`

SSD 1TB 500Mpbs/s.

E.2 A Study with Industrial Applications

Application	KLOC	# of IDS Tests	# of IDUBS Tests
A1	68.5	346	340
A2	82	463	447
A3	52.8	229	231
A4	77.6	443	450
A5	306.8	1756	1780
A6	178.5	794	847
A7	65.9	101	97
A8	75	363	366
A9	37	251	248
A10	228.9	1283	1179
A11	78.1	800	802
A12	32.4	90	90
A13	109.1	420	407
A14	43.7	262	270
A15	62	174	171
A16	73	410	362
A17	58.5	112	116
A18	41.4	357	361
A19	42.1	191	165
A20	397.9	444	444

Table E.1: Industrial apps: KLOC, IDS, and IDUBS test counts.

In our first study, we examined twenty industrial React-based applications from a partner company, each developed by different teams. The applications handle specific fiscal and cost management tasks for companies. Table E.1 shows the size (KLOC), and the number

of test cases generated and executed by Cytestion with IDS and IDUBS. For confidentiality reasons, the applications are labeled A1 - A20. It is important to highlight that all projects are in production, having been tested by both their development teams and the company QA team. Any discovered faults were reviewed and, if confirmed, registered as bugs.

E.2.1 Results and Discussion

Figure E.1 presents the frequency of visits of the top-5 most visited states of each generated test suite using IDS and IDUBS. The initial state is the most accessed GUI state across all projects. With IDS, every test case starts at the root, therefore, each test case visits the initial state. Except for the A12 project, IDUBS effectively reduced revisits to the initial state. This reduction was anticipated as home pages typically serve as starting points with access to various features of the system and often lead to new URLs being accessed in subsequent iterations of IDUBS.

When investigating the A12 executions, we found that the generated test cases did not reach new URLs due to the project's unique characteristic: the URLs simply do not exist. This project has few features, all accessed under the same URL, unlike other applications. In the other 19 projects, IDUBS showed a noticeable decrease in repetitions in the 2nd through 5th states. This was anticipated, as industrial applications often have many intermediate states that must be reached to access deeper functionality. Consequently, these states are repeatedly accessed by IDS, while IDUBS partially avoids them.

In total, IDS accessed 41,710 states, while IDUBS accessed 20,853 states, achieving a 50% reduction in access for industrial projects. This indicates that IDUBS significantly reduced redundancy compared to IDS. The Wilcoxon rank sum tests on the top 5 most visited states revealed significant differences for all systems ($p < 0.05$), except A12, with large Cohen's d values (1.022 to 1.690), indicating a statistical difference between IDS and IDUBS.

Figure E.2 shows the execution time of each test case (x axis) for the IDS and IDUBS suites in each project. The blue lines refer to IDS tests, while the green lines refer to IDUBS tests. It is important to highlight that we used in this analysis only the tests found in both suites, in the same order. Each blue point has a corresponding green point, and the execution time is measured in seconds (y-axis).

Our analysis shows that execution times vary across projects, with IDUBS consistently

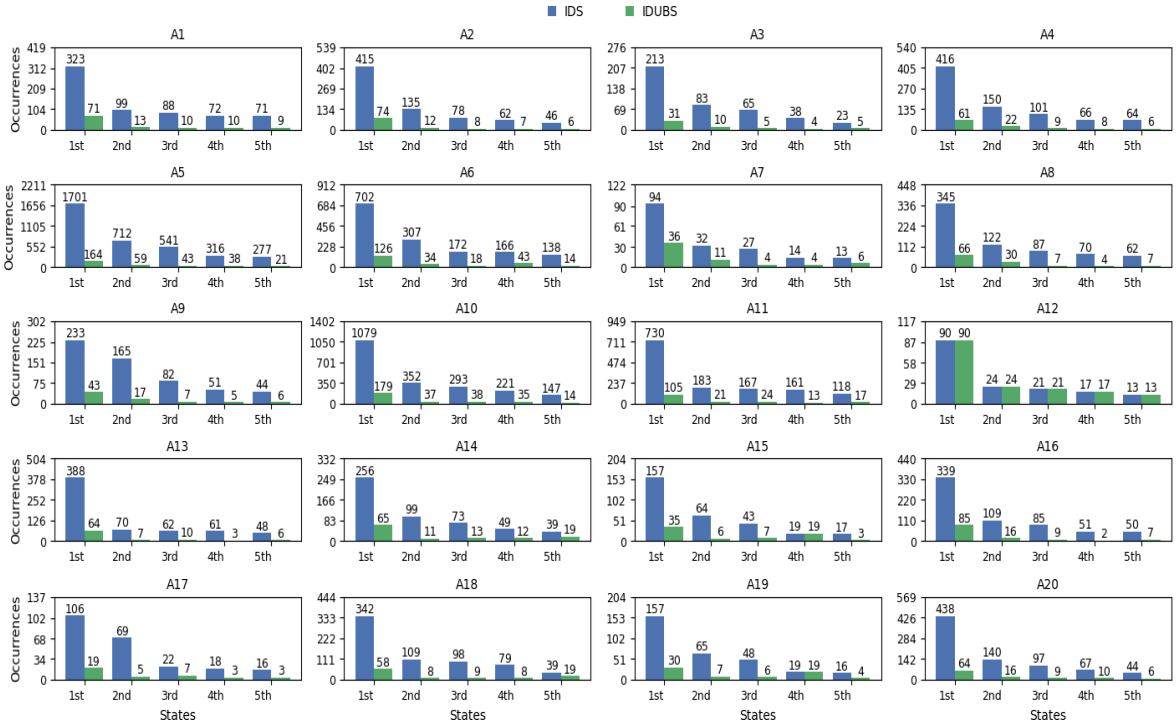

Figure E.1: Number of access occurrences in most accessed states.

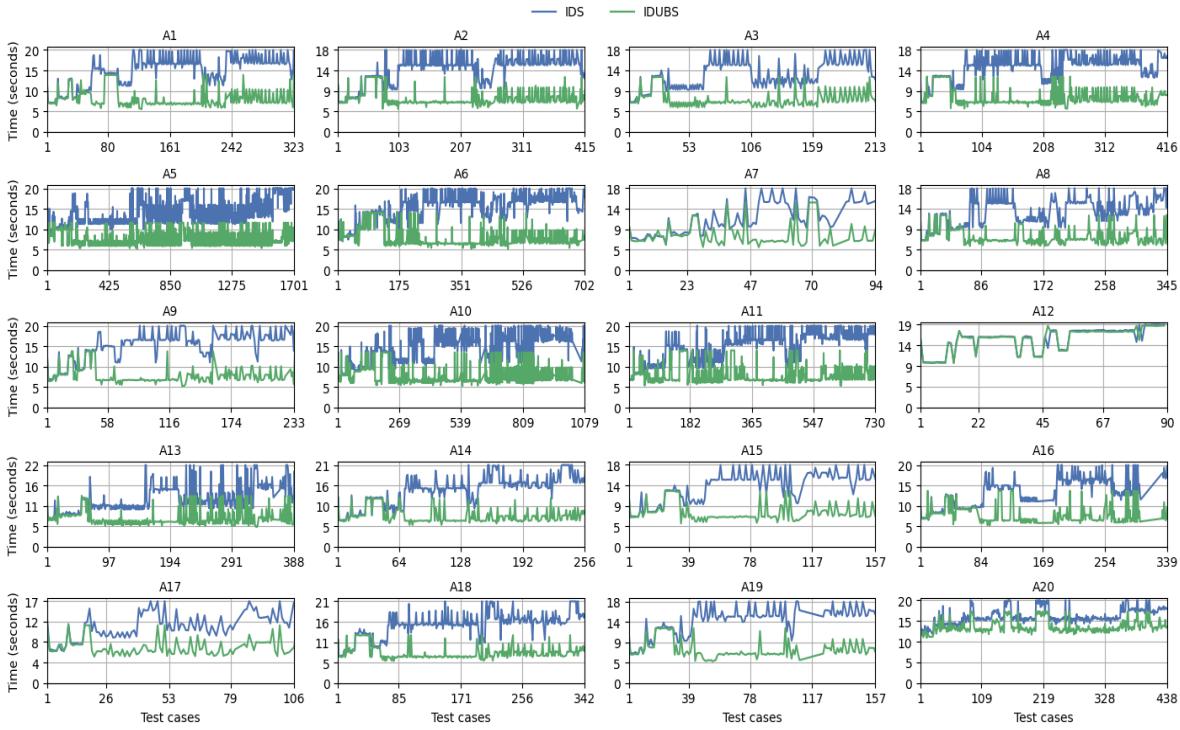

Figure E.2: Test case execution times for IDS and IDUBS.

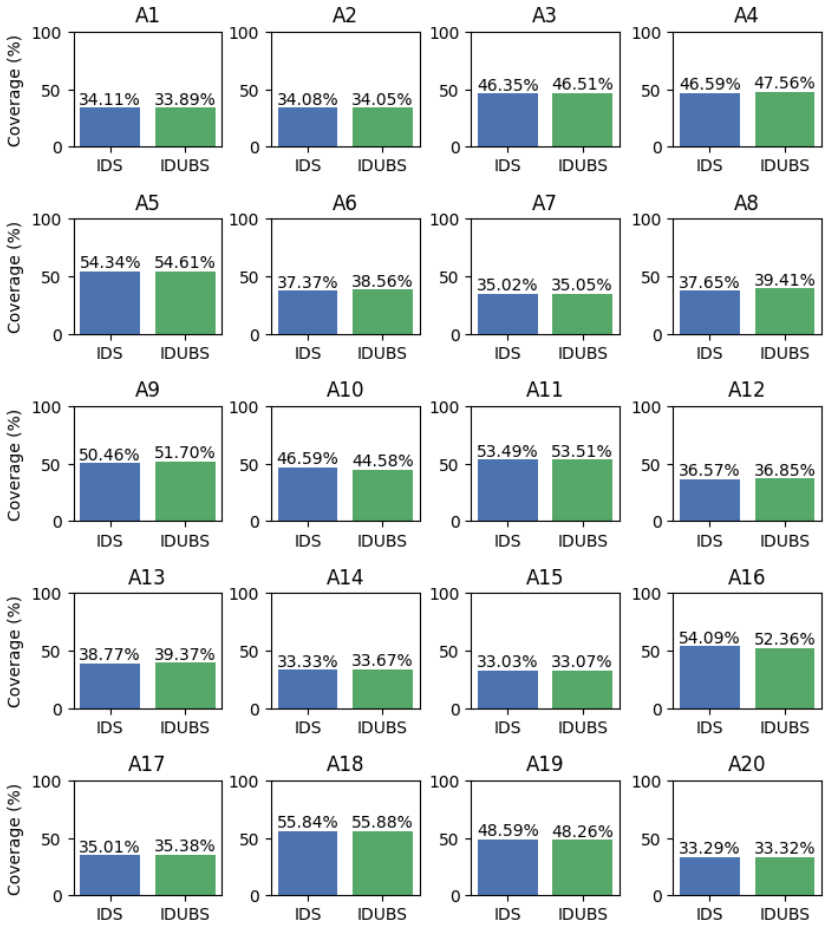

Figure E.3: Frontend code coverage results.

performing faster. IDS fluctuates between 21.8 seconds and 6.2 seconds, while IDUBS ranges from 18.2 to 4.7 seconds. IDUBS reduced the total execution time by 43.60% in the industry setting. Although both suites initially had similar execution times, IDUBS improved over time by discovering and utilizing new URLs. The exception is A12, where IDUBS did not reduce the execution times. This conclusion was supported by the Mann-Whitney U test where we found significant p-values $< 10^{-15}$ and large Vargha-Delaney effect sizes ($d > 1.5$) for all systems, excepting A12, indicating that IDUBS generally outperforms IDS.

It is possible to observe a gap effect in the executions. As test cases start accessing more complex functionalities that involve intricate database queries, it leads to slow server responses and results in execution peaks. This situation was observed in both executions, but IDUBS consistently showed lower values compared to IDS due to its ability to avoid revisits and shorten paths. These findings help us answer RQ_1 by providing evidence of cost reduction in both state access redundancy and execution time, thus affirming that IDUBS can

effectively reduce costs.

Figure E.3 shows that both IDS and IDUBS achieve similar coverage levels for frontend code lines across all systems (Wilcoxon signed-rank test p-value of 0.1004). Despite using shorter test cases, IDUBS produces test suites with coverage nearly equivalent to IDS. This suggests both algorithms offer comparable coverage efficacy. Although the coverage levels range from 33.03% to 55.88%, it is noteworthy that these suites were automatically generated. Additionally, IDS has proven effective in detecting visible GUI faults in real-world scenarios [79], making it a valuable option.

We analyzed the visible failures detected by both suites. The IDS suites identified 48 faulty states, which we manually inspected. These faults correspond to six actual issues related to various types of bugs, including button-triggered processes displaying the error message "An unexpected error occurred" and page crashes when the edit button is clicked.

The IDUBS suites identified 317 states with visible failures. Moreover, all found IDS failed states were also detected by the IDUBS. We carefully analyzed each failure and discovered that a fault in one of the horizontal components of the applications was exposed only when the page was reloaded or accessed directly via the URL. Consequently, this fault appeared on all pages using this component exclusively when running the IDUBS suite.

In total, seven faults were registered, six found by both suites (IDS and IDUBS), and one detected only by the IDUBS suite. The faults were presented to the QA team and managers, who provided positive feedback. They noted that these issues had been overlooked by the company's quality process and could impact the user experience.

The findings discussed here demonstrate the benefits of using IDUBS in industrial settings. The generated suites provide similar coverage while detecting new faults and significantly reduce the costs associated with test execution, including time and redundancy.

E.3 A Study with Open Source Applications

In our second study, four open-source web applications were selected as objects: i) *school educational*, an HTML5 website that implements common functionalities found in school applications; ii) *petclinic*, a SpringBoot application to manage pet owners' registration and scheduling veterinarian visits; iii) *learn educational*, a responsive website that showcases

online educational course portfolios; and iv) *bistro restaurant*, a website developed with HTML, JavaScript, and CSS to display restaurant portfolios. They are available in our repository[4]. Since our first study (Section E.2) dealt with React-based projects, here we selected projects that do not use any modern web framework. This decision is motivated by our objective to ascertain the continued relevance of our findings across a wider spectrum of applications.

Project	KLOC	# of IDS Tests	# of IDUBS Tests
school educational	30.2	231	231
petclinic	25.7	50	50
learn educational	19	225	225
bistro restaurant	33.4	212	212

Table E.2: Open projects: KLOC, IDS, and IDUBS counts.

Table E.2 provides information on the projects, including their size (KLOC), and the number of test cases generated and executed by Cytestion with IDS and with IDUBS. Despite their simplicity, these systems offer navigation features with a wide range of potential GUI states, display important information, and facilitate registration operations that can result in visible failures. This is evidenced by the number of test cases generated.

E.3.1 Results and Discussion

Figure E.4 shows the frequency of visits of the top-5 most visited states of each generated test suite using IDS and IDUBS. Again, the initial state is the most accessed GUI state across all four projects. IDUBS effectively reduced revisits to the initial state, decreasing redundancy by at least 85% across all projects.

When we consider the 3rd, 4th, and 5th most accessed states, we noticed less variation in repetition. With the exception of the *petclinic* project (Wilcoxon rank sum tests, $p-value = 0.01193$, and Cohen's $d = 2.298$), all other projects had a similar number of accesses in these three states using both algorithms. This happened because these states offer numerous actions that do not change the URL, leading all test cases to revisit them in subsequent

[4] `https://gitlab.com/lsi-ufcg/cytestion/opt-study/applications`

iterations. Finally, considering only open sources, IDS accessed a total of 2191 states while IDUBS accessed 1402 states. Therefore, IDUBS resulted in an access reduction of 36.01%.

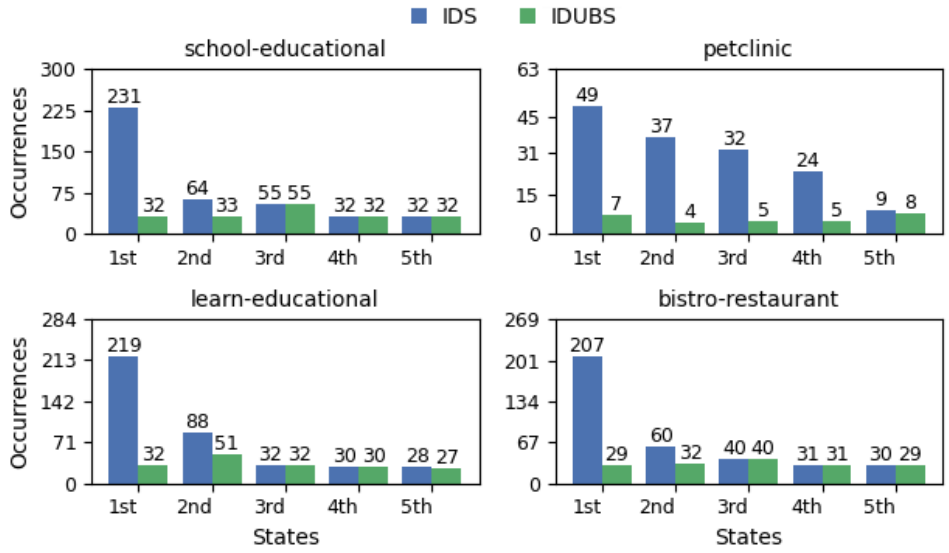

Figure E.4: Frequency of accesses in highly accessed states.

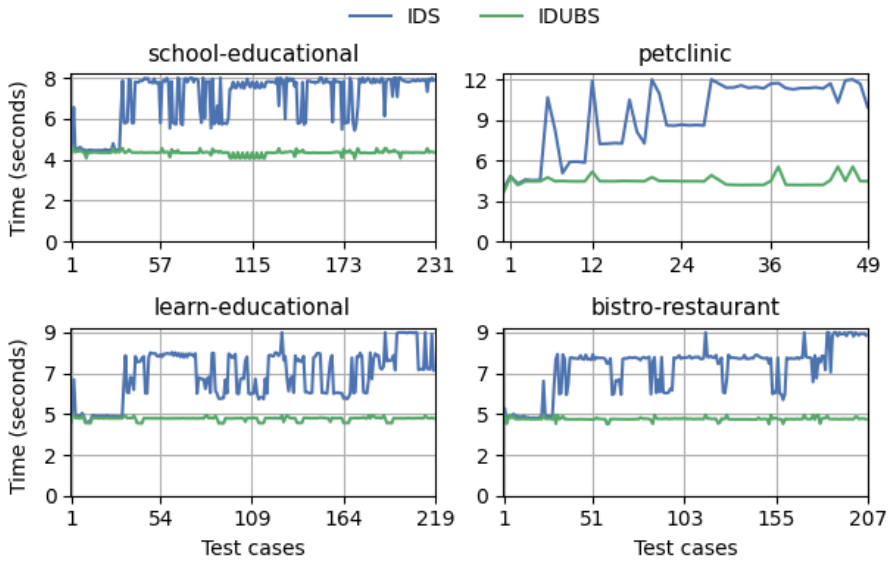

Figure E.5: Test case execution times by algorithm.

Figure E.5 presents the execution time of each test case for IDS and IDUBS per project. Our analysis reveals a consistent decrease in execution times for all four projects. In the *school-educational*, IDUBS tests took between 4.1 and 4.6 seconds, compared to IDS tests which ranged from 4.2 to 8.1 seconds, resulting in up to a 3.4-second reduction in execution time. Additionally, the *petclinic* project experienced the most significant drop, with a reduc-

tion of 6 seconds, while the *learn-educational* and *bistro-restaurant* projects saw decreases of 4.6 and 4.7 seconds, respectively. The Mann-Whitney U test confirms this conclusion by presenting significant differences between the two strategies, with p-values $< 10^{-14}$ and large Vargha-Delaney effect sizes ($d > 2$).

IDUBS has demonstrated stable runtimes in all test cases across various projects. The approach of accessing the URL every time it changes has helped maintain consistent execution times. If the URL changes after each action, every new iteration will always contain a visit to the new URL and one action. IDS, on the other hand, had to continuously access the home page, perform a series of actions in the AUT, and wait for API requests to finish. As the interaction with actionable elements of the AUT naturally demands a variable response time influenced by API request efficiency responses, this variability directly impacts execution times. In contrast, IDUBS direct URL access requires fewer actions to perform tests. These findings help us answer RQ_1 by providing evidence of cost reduction in both state access and execution time, thus affirming that IDUBS can reduce costs effectively.

Regarding performance, we were unable to measure frontend code coverage due to compatibility issues with the Istanbul dependency, which supports only projects using frameworks like React that use JavaScript ES5. Therefore, we focus our analysis on the found faulty states. Each suite identified nine states with visible failures. We manually investigated the states and found that all failures were false positives. They involved actionable elements linked to external websites with failing requests. Cytestion deals with the exploration limit to avoid exploring states that do not belong to the AUT. However, test cases that try to access such states are still evaluated by the generic oracle. Despite generation not continuing in that branch, faults can still be found on this external site. This situation can be viewed as a limitation of the generic oracle implemented by the Cytestion tool. However, for the purpose of our investigation the executions show an equivalence in fault detection of the two algorithms.

Based on the results discussed in Sections E.2 and E.3, we can answer RQ_1 and RQ_2 by stating that IDUBS can effectively reduce GUI testing costs (execution time and test redundancy) while maintaining or improving the performance (coverage and new faults), when compared to IDS.

E.4 Threats to Validity

Our results are based on the specific projects examined in our studies. However, we analyzed a set that combined open-source and industrial projects, which we consider to be a reliable sample of web applications. It is important to emphasize the substantial representation of industrial projects in our analysis, enhancing the relevance of our findings to similar industrial contexts.

Computational overhead can be a key aspect when evaluating an algorithm. In our studies, we indirectly analyze this aspect by comparing the execution time of IDS and IDUBS. However, other metrics are yet to be analyzed in the future (e.g., memory usage).

The performance analysis (RQ_2) in the study on open-source applications was limited because we were unable to collect coverage information, and no real faults were detected. Nevertheless, we contend that greater significance lies in the evaluation carried out in the industrial study. Considering the diverse sizes and complexities of the industrial objects, we believe they offer robust evidence regarding the stability of IDUBS concerning testing efficacy. Industrial settings adhere to rigorous quality standards and involve various stakeholders, thereby ensuring the reliability and applicability of the results. Additionally, the open-source study further validated IDUBS's ability to reduce GUI costs (RQ_1).

Our findings rely on the utilization of IDS and IDUBS within the Cytestion tool. The authors meticulously validated both implementations through a series of testing scenarios. Furthermore, the fundamental principles of these algorithms can be applied autonomously, irrespective of any particular tool. This implies that the found IDUBS advantages go beyond a singular implementation, as other implementations or tools can likewise harness their benefits.

The IDS algorithm highlights its combination of BFS and DFS. While IDS inherently performs a BFS through multiple DFS executions, alternative methods like Bidirectional Search and Heuristic-Enhanced IDS can be used to enhance efficiency [103].

External factors, such as network conditions, or changes in the web application environment, could introduce variability in the results and impact both algorithm performance. To mitigate this risk, we executed the test suites in a controlled environment on a dedicated machine, running each suite only once with minimal delay between them. Moreover, we used

the Winsorization transformation to mitigate possible outliers. The consistent results found across different projects indicate IDUBS's resilience to external factors.