On overcoming challenges with GUI-based test automation

Michel Nass

Blekinge Institute of Technology Doctoral Dissertation Series
No 2024:02

On overcoming challenges with GUI-based test automation

Michel Nass

Doctoral Dissertation in Software Engineering

Department of Software Engineering
Blekinge Institute of Technology
SWEDEN

Department of Software Engineering
Publisher: Blekinge Institute of Technology,
SE-371 79 Karlskrona, Sweden
Printed by MEDIA-TRYCK, Lund, Sweden 2024
ISBN 978-91-7295-473-1
ISSN 1653-2090
urn:nbn:se:bth-25638

"Chase knowledge with passion, not milestones with haste, for true understanding lies in the journey, not just the embrace of the finish line."
- ChatGPT

Abstract

Background: Automated testing is widely used in modern software development to check if the software, including its graphical user interface (GUI), meets the expectations in terms of quality and functionality. GUI-based test automation, like other automation, aims to save time and money compared to manual testing without reducing the software quality. While automation has successfully reduced costs for other types of testing (e.g., unit- or integration tests), GUI-based testing has faced technical challenges, some of which have lingered for over a decade.

Objective: This thesis work aims to contribute to the software engineering body of knowledge by (1) identifying the main challenges in GUI-based test automation and (2) finding technical solutions to mitigate some of the main challenges. One such challenge is to reliably identify GUI elements during test execution to prevent unnecessary repairs. Another problem is the demand for test automation and programming skills when designing stable automated tests at scale.

Method: We conducted several studies by adopting a multi-methodological approach. First, we performed a systematic literature review to identify the main challenges in GUI-based test automation, followed by multiple studies that propose and evaluate novel approaches to mitigate the main challenges.

Results: Our first contribution is mapping the challenges in GUI-based test automation reported in academic literature. We mapped the main challenges (i.e., most reported) on a timeline and classified them as essential or accidental. This classification is valuable since future research can focus on the main challenges that we are more likely to mitigate using a technical solution (i.e., accidental).

Our second contribution is several approaches that explore novel concepts or advance state-of-the-art techniques to mitigate some of the main acciden-

tal challenges. Testing an application through an augmented layer (Augmented Testing) can reduce the demand for test automation and programming skills and mitigate the challenges of creating and maintaining model-based tests. Our proposed approach for locating web elements (Similo) can increase the robustness of automated test execution.

Conclusion: Our results provide alternative approaches and concepts that can mitigate some of the main accidental challenges in GUI-based test automation. With a more robust test execution and tool support for test modeling, we can help reduce the manual labor spent on creating and maintaining automated GUI-based tests. With a reduced cost of automation, testers can focus more on other tasks like requirements, test design, and exploratory testing.

Keywords: *GUI Testing, Test Automation, Augmented Testing, Test Case Robustness, Web Element Locators, Large Language Models*

Acknowledgements

First, I would like to thank my supervisors, Emil Alégroth and Robert Feldt, who spent countless hours helping me transition from an industrial mindset to becoming more academic. Next, I would like to express my gratitude to Riccardo Coppola, Maurizio Leotta, and Filippo Ricca for an amazing collaboration in two of the studies. I feel truly blessed to be able to work with researchers of your caliber. Finally, I would like to thank my colleagues at BTH and my family for their encouragement and support.

Overview of Papers

Papers in this Thesis

- **Chapter 2: Michel Nass**, Emil Alégroth, and Robert Feldt. "Why many challenges with GUI Test Automation (will) remain". Published in *Information and Software Technology (IST)*, 2021.
- **Chapter 3: Michel Nass**, Emil Alégroth, and Robert Feldt. "Augmented Testing: Industry Feedback To Shape a New Testing Technology". Published in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019.
- **Chapter 4: Michel Nass**, Emil Alégroth, and Robert Feldt. "On the Industrial Applicability of Augmented Testing: An Empirical Study". Published in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020.
- **Chapter 5: Michel Nass**, Emil Alégroth, Robert Feldt, Maurizio Leotta, and Filippo Ricca. "Similarity-based web element localization for robust test automation". Submitted to *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022.
- **Chapter 6: Michel Nass**, Emil Alégroth, Robert Feldt, and Riccardo Coppola. "Robust web element identification for evolving applications by considering visual overlaps". Published in *IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2023.
- **Chapter 7: Michel Nass**, Emil Alégroth, and Robert Feldt. "Improving web element localization by using a large language model". Submitted to *Software Testing, Verification and Reliability (STVR)*, 2023.

Contribution Statement

We used the Contributor Roles Taxonomy (CRediT) [93] to formulate individual author contributions to papers included in this thesis.

Chapter 2:

Michel Nass: Conceptualization (lead), data curation (lead), formal analysis(lead), investigation (lead), methodology (lead), visualization (lead), writing - original-draft (lead), writing - review and editing (lead).

Emil Alégroth: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Robert Feldt: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Chapter 3:

Michel Nass: Conceptualization (lead), data curation (lead), formal analysis(lead), investigation (lead), methodology (lead), visualization (lead), writing - original-draft (lead), writing - review and editing (lead).

Emil Alégroth: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Robert Feldt: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Chapter 4:

Michel Nass: Conceptualization (lead), data curation (lead), formal analysis(lead), investigation (lead), methodology (lead), visualization (lead), writing - original-draft (lead), writing - review and editing (lead).

Emil Alégroth: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (sup-

porting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Robert Feldt: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Chapter 5:

Michel Nass: Conceptualization (lead), data curation (lead), formal analysis(lead), investigation (lead), methodology (lead), visualization (lead), writing - original-draft (lead), writing - review and editing (lead).

Emil Alégroth: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Robert Feldt: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Maurizio Leotta: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Filippo Ricca: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Chapter 6:

Michel Nass: Conceptualization (equal), data curation (equal), formal analysis(supporting), investigation (equal), methodology (supporting), visualization (lead), writing - original-draft (equal), writing - review and editing (supporting).

Emil Alégroth: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Robert Feldt: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Riccardo Coppola: Conceptualization (equal), data curation (equal), formal analysis(lead), investigation (equal), methodology (lead), visualization (supporting), writing - original-draft (equal), writing - review and editing (lead).

Chapter 7:

Michel Nass: Conceptualization (lead), data curation (lead), formal analysis(lead), investigation (lead), methodology (lead), visualization (lead), writing - original-draft (lead), writing - review and editing (lead).

Emil Alégroth: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Robert Feldt: Conceptualization (supporting), data curation (supporting), formal analysis(supporting), investigation (supporting), methodology (supporting), visualization (supporting), writing - original-draft (supporting), writing - review and editing (supporting).

Other Papers not in this Thesis

- Emil Alégroth, and Michel Nass. "JAutomate: A Tool for System- and Acceptance-test Automation". Published in *IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013.

Funding

This work was supported by the KKS foundation through the S.E.R.T. Research Profile project at Blekinge Institute of Technology.

Contents

List of Abbreviations

Abbreviation	Definition
AT	Augmented Testing
DOM	Document Object Model
GPT	Generative Pre-trained Transformers
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
LLM	Large Language Model
OCR	Optical Character Recognition
REST	REpresentational State Transfer
SLR	Systematic Literature Review
SOAP	Simple Object Access Protocol
SUT	System Under Test
UFT	Unified Functional Tester
UI	User Interface
VGT	Visual GUI Testing
VON	Visually Overlapping Nodes

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Software testing aims to ensure that the developed software works correctly, but this process can be time-consuming and is therefore associated with considerable cost [71, 73]. To alleviate these challenges, test automation has been proposed to speed up the testing process, allowing for more frequent test runs, boosting delivery speed, and helping to improve software quality [17, 28, 131]. One primary use of automation is regression testing, which aims to assess that changes to the system under test (SUT) have not introduced new faults or otherwise degraded the quality of the latest software version. Testers often use automated regression test scripts, which, on a Graphical User Interface (GUI) level, aim to mimic end-user interactions to verify the software's behavior [103, 108]. However, each new software version can cause the scripts to fail, necessitating time and resources to investigate the cause, report the problem, and repair/update the script. This challenge is particularly true for GUI-level testing since GUI elements often change with updates in the user interface but are also affected by underlying system changes [26, 58, 152]. Another challenge with GUI testing is that GUIs are designed for humans, not machines, making the interface less ideal for machine comprehension, causing problems after GUI updates. A common problem is not being able to locate the correct widget (i.e., visible GUI component, like a button) during test execution, often resulting in a failed or invalid test. Such a problem is less common in foundational tests like unit test-

ing [131] that are designed for machine-to-machine interaction (i.e., test code checking source code).

Test automation relieves the human testers from the tedious and time-consuming effort of performing the tests manually [73] [71]. However, automation introduces new challenges in developing, maintaining, and running automated test scripts, some of which pertain to the technical aspects of automated GUI testing [39, 72, 81, 115]. These challenges can outweigh the advantages of automating the GUI test cases, especially when considering the creation or maintenance costs of the tests. This research aims to dig deeper into the challenges associated with GUI-based test automation by identifying the key (i.e., most prominent) challenges (and their root causes) and finding technical solutions to mitigate them. Reducing the challenges might lead to a more widespread adoption of GUI-based test automation in the industry.

We formulated two overall goals to guide our research:

- **G1:** Identify the key challenges related to GUI-based test automation.
- **G2:** Mitigate some of the key challenges using technical solutions.

The remainder of this Chapter is structured as follows. Section 1.2 gives a background to software testing and GUI-based regression testing. Section 1.3 describes the problem and motivation behind this research. Research objectives and questions are described in Section 1.4. We outline the research methods used in this thesis work in Section 1.5. Section 1.6 presents an overview of the studies we included in this thesis, and we discuss some threats to the validity in Section 1.7. Finally, we synthesize the results and discuss the implications of our work in Section 1.8 and outline possible future research in Section 1.9.

1.2 Background and Related Work

1.2.1 Software Testing

Figure 1.1 shows the classical V-model of software development that associates phases of testing activities with software design phases [13]. The V-model can be helpful when identifying test activities regardless if the team is following a waterfall or agile software development approach. Each test phase in the V-model corresponds to a phase on the software design side. While there are many variations of the V-model, and each phase might have a different name, there are typically at least these four test phases in the V-model:

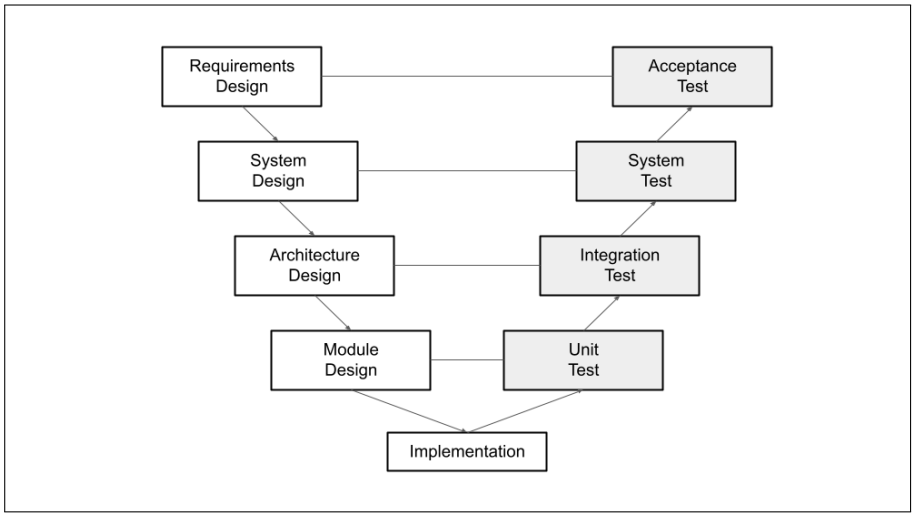

Figure 1.1: The V-model of software development.

Unit testing: Unit tests are used to eliminate issues at the lowest level of the code, here called a unit. A unit is the smallest independent entity and is typically implemented as a function or procedure. The unit test verifies that the unit is working as expected when isolated from any dependencies (e.g., other functions) [107, 131].

Integration testing: Integration testing is the process of testing how different units of a system work together [82, 107]. This type of testing is valuable because it can help to identify errors and bugs when two or more units are connected, rather than just testing individual units.

System testing: System testing is used to evaluate the overall functionality of a system [107]. This testing type is often done by a team of testers independent of the development team. Anand et al. described such an example when testing a large mission-critical software system [34].

User acceptance testing: User acceptance testing is used to determine whether or not a software application is acceptable to the end-user [107]. This type of testing is typically performed by actual software application users rather than by testing professionals.

While unit- and integration testing are mainly designed based on architecture and code (i.e., technical), the system- and user acceptance tests are primarily created from end-user and system documentation (i.e., user-centered).

Regression testing is a type of testing that is used to verify that changes to a system or application did not introduce new faults [132]. This type of testing is typically done after changes have been made to the code base (e.g., after a source code update or at the end of an iteration/sprint), and it is used to ensure that the changes have not caused any new problems. The regression testing typically involves running tests related to all the four types of tests displayed in Figure 1.1 since a change might theoretically affect any part of the system. While tests on a higher level of abstraction (e.g., acceptance tests) can find an issue, a lower-level test (e.g., a unit test) might provide a quicker (i.e., since it is typically run more often) and a more detailed clue to find the root cause of the issue. In practice, only a subset of the tests in the complete regression test suite might be selected for execution since there might not be time to run them all. Using a subset of the complete regression test suite is especially common when some tests must be performed manually due to time constraints. The subset is selected based on many factors, e.g., the severity of a possible defect and the likelihood that the changes have affected the tested functionality.

Exploratory testing is a type of software testing that is conducted without following a specific test plan or script [59, 83]. This means that testers are free to explore the software to find bugs and issues without being restricted by a predetermined set of test cases. Exploratory testing is often used to supplement regression testing to pinpoint faults the scripted tests do not find since they do not follow a predefined path through the SUT, increasing the test coverage. Some drawbacks of exploratory testing include the difficulties of replicating the test results and the dependency of testers with domain knowledge [18].

Until now, we have only covered testing the functionality of a system (i.e., functional testing). However, to thoroughly test a system, we also need to consider its quality aspects (i.e., non-functional testing), e.g., its performance, usability, scalability, and reliability. Non-functional testing typically requires a different set of tools and skill sets than are usually employed with functional testing. However, we will not go into more detail about non-functional testing since this research focuses on testing a system's functionality through the user interface.

1.2.2 Automated Software Testing

Test automation is a process of entirely or partially automating the creation, maintenance, and execution of test cases for software and is a replacement or a supplement to manual testing. Automated tests are typically executed faster than manual tests and can cover more scenarios than manual execution (given the same time constraints). The drawbacks of automated tests are that they can be expensive to create and maintain [94, 108] and typically require specialized skills, like programming, to develop and maintain the scripts [55, 108]. Automated tests can fail to detect some issues since they typically only check a subset of the SUT output. For example, an output field indicates an error by changing color (e.g., red), but there is no step in the test case that checks the color of the text field. In contrast, with manual tests, humans can use their cognitive ability to identify defective behaviors that are not explicitly defined in the test case.

The goal of software testing is to assess the quality of the software so that we know when the software is ready for release [107]. Until the software is released, we must also identify issues and drawbacks to know what to fix. To assess the quality of the software, we need sufficient test coverage [130]. A high test coverage becomes increasingly challenging to achieve as the software increases in size and when the calendar time and resources (e.g., the number of testers) for testing activities are limited. Test automation is valuable since we can achieve a higher test coverage, resulting in a better quality estimate. The cost of maintaining (i.e., repairing and updating) the tests will also increase when the software grows since the regression test suite increases in size proportional to the software (when targeting the same test coverage). Therefore, keeping the maintenance cost of tests low is vital since it might exceed the benefits of higher test coverage.

One popular concept mentioned in a book by Mike Cohn [52] is to divide tests into three layers visualized as a pyramid in Figure 1.2. The three layers are (1) Graphical User Interface (GUI) Tests, (2) Service Tests, and (3) Unit Tests. Although simple, the test pyramid implies that we should combine tests of various granularity and that it likely contains more unit tests than GUI tests since they are less complex and faster to execute. While Unit- and Service Tests are typically automated in a regression test suite, it is still a common practice in the industry to execute GUI Tests manually despite the possible benefits of automation.

A common practice in object-oriented programming is to create one test class for each unit (e.g., a class) of the system. The test class needs to be maintained

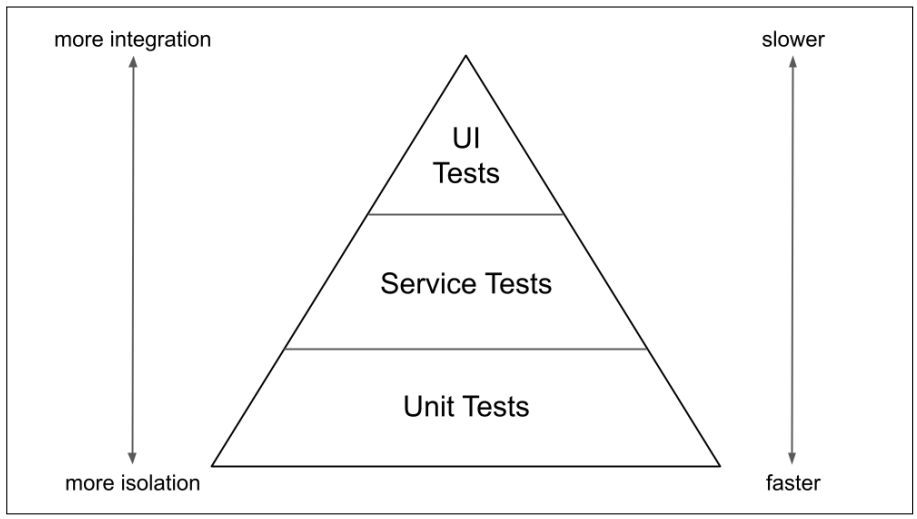

Figure 1.2: The test pyramid.

when the class that contains the functionality is updated, resulting in an additional maintenance cost. While unit testing requires skills in programming and unit testing frameworks (e.g., jUnit or nUnit), unit testing plays an important role during software development [173].

Service tests check that a service works as intended. These tests are usually done by sending a request to the service and checking the response using some form of protocol (e.g., HTTP, SOAP, or REST). While service testing is sometimes performed manually, there are a wide variety of tools for automated service testing, like SoapUI or Postman.

1.2.3 Automated GUI testing

GUI testing tests an application using its graphical user interface to ensure it functions and behaves correctly. Sometimes, we also include testing that the user interface is user-friendly or follows specific rules, but this type of testing is not the main focus of this research. Many tools are available for GUI testing, like Selenium, Appium, Eggplant, Sikuli, EyeAutomate, TestComplete, Ranorex, and Watir. We can divide GUI test tools into three generations based on their technology for locating and interacting with the GUI components [23].

Coordinate based: The first generation of GUI test automation tools uses screen coordinates when deciding where, for example, to click on the screen. This technique is simple and works on any user interface, but relying on screen (or window) coordinates makes the scripts sensitive to GUI components that change location. Coordinate-based localization might have been feasible for full-screen applications but more problematic for window-based applications that we can resize and move to any part of the screen. Moving or resizing the window would cause all the window's GUI components to change location, thereby likely making a coordinate-based script fail during test execution.

Component based: Second-generation tools extract GUI components from the user interface of the application to test. The test tool interacts with the GUI components to simulate actions (e.g., a click) or extract information (e.g., the value from a text field) suitable for checking an outcome. The type of the GUI components extracted depends on the System Under Test (SUT). For a web application, web elements are extracted from the document object model (DOM) of a web page. Second-generation tools do not have the same weakness as first-generation tools since the coordinate is only one of many properties we can extract from a GUI component. However, the technique requires extracting GUI components from the user interface, which makes the technique dependent on the implementation of the SUT. Tool vendors often try to handle this dependency by providing an interface (or API) to use when interacting with different types of applications (e.g., web, Windows, Mac, Android, Flash). However, the problem is that each interface comes with its own limitations. In some cases, it might be simple (e.g., for a pure HTML application), and in other cases, it is difficult or almost impossible to extract all of the GUI components (e.g., a Flash application) using the interface. Some examples of component-based tools for web applications are Selenium[1], Protractor[2], and Playwright[3].

Second-generation tools often support recording (and replaying) test scripts since manually extracting all the properties of the GUI components is time-consuming [17]. While recording a test script is fast compared to writing the script and extracting properties manually, it typically produces fragile test scripts that require constant maintenance when the tested application evolves (i.e., is updated or refined) [94]. This maintenance work increases with the size of the test suite for several reasons. One reason is that the initially recorded scripts need to be re-recorded when the SUT changes since the layout or GUI component properties might have changed. To avoid replacing many (or all)

[1] https://www.selenium.dev/
[2] https://www.protractortest.org/
[3] https://playwright.dev/

recordings when the SUT changes, it is a common practice to restructure the test scripts to reuse commonly used scripts and GUI components. This work requires programming knowledge since it resembles structuring code in a typical software project. Another problem with maintaining an ever-increasing set of recorded tests is the repository containing the recorded properties extracted from the GUI components. We need to merge the components in an older repository with those in the newly recorded repository to avoid re-recording each test since we would only like to maintain one instance of each GUI component, avoiding many copies.

Visual GUI Testing: Third-generation tools use the pictorial user interface (e.g., the pixels on the screen) and image recognition to locate GUI components and to check the expected outcome [23]. As with tools based on coordinates, we can use Visual GUI Testing (VGT) tools with any application, regardless of implementation. Scripts, including images, are straightforward for a manual tester to understand and can be read as manual step-by-step instructions. One example script that uses the Windows calculator application can be seen in Figure 1.3.

The script contains four mouse-click actions using images to find the coordinate to click on the screen. Note that this technique resembles coordinate-based testing with the difference that the coordinates are retrieved dynamically by searching for an image on the screen. Given that the image recognition returns with the correct coordinate, this technique compensates for moving GUI components and reduces the maintenance cost caused by manually updating the target coordinates. The last command in the VGT script verifies that a certain image is visible on the screen. Checking images makes it possible to assert expected conditions in VGT tools. Many VGT tools also support optical character recognition (OCR) to read text or values into the script. There are several tools that support VGT, like Sikuli [168], EyeAutomate (previously called jAutomate) [31], EggPlant [5], and Unified Functional Tester (UFT) [91].

1.3 Problem and Research Motivation

Record and replay tools were popular many years ago, being advertised as recording once and replaying the recorded scripts any number of times. What began as an appealing vision of test automation at a low cost ended when people realized that the recorded scripts were not robust and needed restructuring and constant maintenance [94, 118].

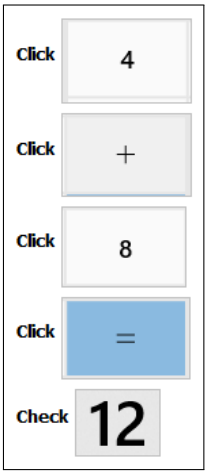

Figure 1.3: A VGT script testing the Windows calculator.

Despite the widespread adoption of automated code and API verification testing through unit and integration tests, manual testing via the GUI remains a common practice. The preference for manual testing before implementing test automation implies that there are challenges associated with automating GUI-based tests. These challenges, which diminish the effectiveness of GUI test automation compared to manual methods, are well-documented in the literature [138, 156]. Unit testing consists of test code interacting with code, and service tests simulate requests from external systems (i.e., application interfaces). In contrast, GUI tests are triggered by human interaction with the system using the user interface. In other words, unit and service tests interact with functions or interfaces designed to be used by machines, while GUI tests interact with interfaces designed for humans, giving us one possible explanation (i.e., hypothesis) as to why GUI testing is different from unit and service testing that the industry has successfully adopted.

By using test automation, we take advantage of the speed and accuracy of the machines. Still, we hypothesize that when it comes to automated GUI testing, we experience a lack of tolerance to GUI changes in the machine-executed scripts that humans possess, making the scripts seem fragile (i.e., non-robust) during automated test execution.

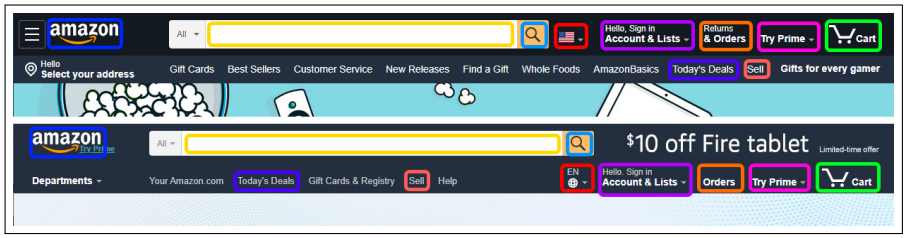

Figure 1.4: Two versions of the amazon.com website.

Figure 1.4 shows two versions (newer and older) of the amazon.com website menu (i.e., excluding content). Menu options with corresponding functionality have been given a frame with identical colors. In this example, we note that some menu options have moved (i.e., changed location within the GUI), and some options have a different caption (e.g., "Returns & Orders" vs. "Orders"). A coordinate-based tool would have failed to locate most of the menu options in this example since they have changed location. However, component-based and VGT tools might succeed in locating most of the GUI components, at least the ones that have not both changed location and caption. In this case, the challenge regards how to successfully locate GUI components that have both been relocated and changed the caption.

Another problem with GUI test automation is the need for programming skills required when maintaining the automated test cases. Programming skills are especially needed when the test suite increases in size since maintaining an automated script resembles structuring source code in a software development project where you need reusable functions and follow good design patterns [32]. Automated tests are often sensitive (i.e., fragile) to changes in the GUI of the SUT. Setting up a rigid test environment and mastering the test automation tools and frameworks requires skills and knowledge. Failing to set up a stable foundation for GUI test automation will likely result in unstable test scripts that require constant maintenance. We could reduce the cost of creating and maintaining automated GUI tests by finding a way (i.e., a tool or an approach) to lower the technological threshold, making automation easier, faster, and more accessible for a broader range of human resources (i.e., both developers and non-developers).

1.4 Research Objectives and Questions

Figure 1.5 shows the relationship between this thesis's goals, objectives, and research questions. The overall research goals, introduced in Section 1.1, are foremost to identify the key (most prominent) challenges of GUI-based test automation and, secondly, to mitigate some of the key challenges through technical solutions. Although we realize that there are also non-technical solutions to the challenges, we decided to begin by addressing the technical issues that will likely affect non-technical aspects like guidelines or processes.

Objectives O2 and O3 were formulated based on results gathered from a systematic literature review (see Section 2) designed to identify the key challenges in GUI-based test automation (objective O1):

- **O1:** Identify the key (most prominent) challenges of GUI-based test automation.
- **O2:** Propose and evaluate an approach to mitigate the challenges related to automation skills and model-based test approaches.
- **O3:** Develop and evaluate an approach to mitigate the challenge of robust identification of GUI widgets.

For the second objective (O2), we ended up with the concept of Augmented Testing that utilizes an Augmented Layer placed between the manual tester and the SUT. The Augmented Layer provides two theoretical benefits. The first is an easy and reliable way of recording user actions (e.g., mouse clicks and keystrokes). Secondly, we can use the Augmented Layer to insert information in the eyesight of the manual tester to, for example, highlight the next action to perform (e.g., the next test step in a test case).

For the third objective (O3), we devised an approach that uses a GUI widget's multiple properties (e.g., name, ID, text) when evaluating the similarity of two widgets. This approach, named Similo, can be utilized to find the next widget to interact with (i.e., among available widgets on the screen) that is most similar to the target widget (i.e., that contains the recorded properties).

The objectives were addressed by the following research questions:

- **RQ1:** What are the key (most prominent) challenges of GUI-based test automation?
- **RQ2:** What impact can our proposed concept (i.e., Augmented Testing) have on mitigating the challenges related to automation skills and model-based test approaches?

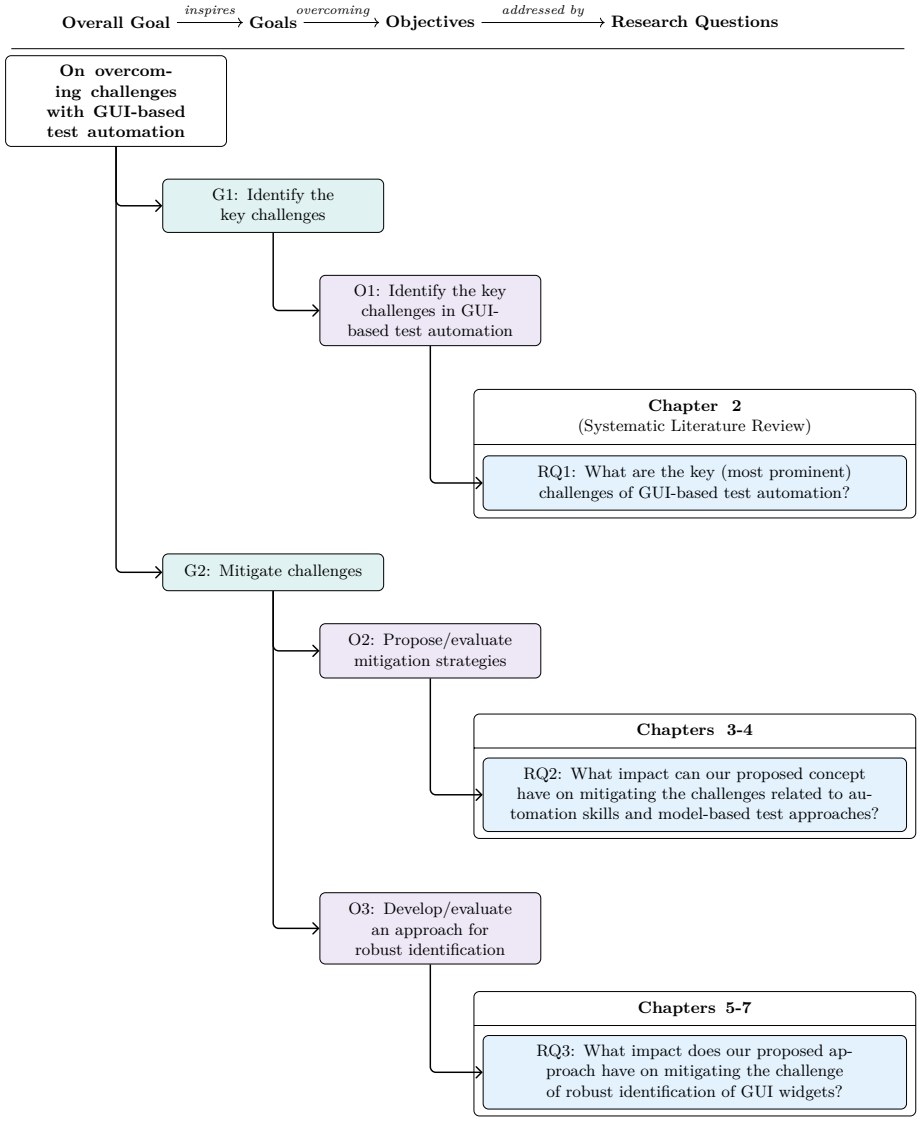

Figure 1.5: An overview of research goals, objectives, and questions.

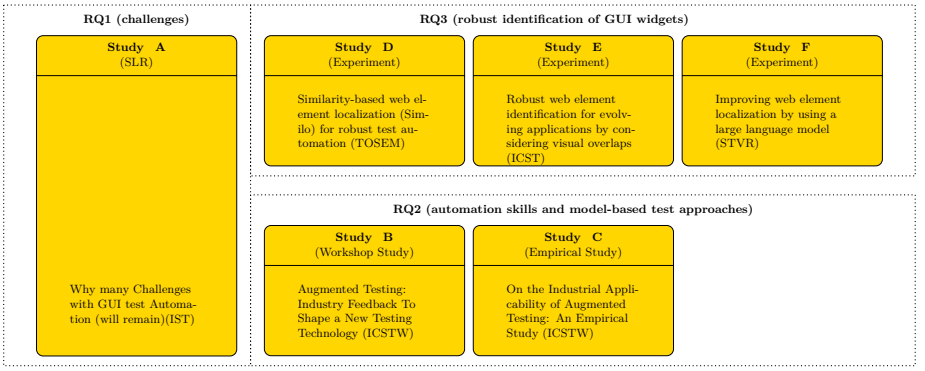

Figure 1.6: An overview of studies answering the research questions.

- **RQ3:** What impact does our proposed approach (i.e., Similo) have on mitigating the challenge of robust identification of GUI widgets?

The remainder of this Section elaborates on the objectives and associated research questions. Figure 1.6 presents an overview of the studies designed to address the objectives and answer our research questions. Included studies are presented chronologically on a timeline (i.e., from left to right). Our research began with a systematic literature review (SLR) to identify the challenges reported in academic papers that contain empirical evaluations in collaboration with the industry (Study A, detailed in Section 2). We started working on the SLR and had the results before conducting the remaining studies. However, the long process of publishing the SLR resulted in a publication date later than studies B and C.

Study A answers the first research question (RQ1), while the remaining studies answer the second and third research questions (RQ2 and RQ3). The SLR classifies the challenges ranging from essential to accidental, visualized in Figure 1.7. We consider challenges essential if they are inherent to a specific technology or approach and as accidental when it is possible to eliminate the challenge through technology or improved ways of working [43].

The SLR pinpoints four accidental challenges (i.e., that we can mitigate with technical solutions) that have remained a problem for the industry for many years and that still exist today (challenges C2, C3, C18, and C19 in Figure 1.7). Three of the four (C3, C18, and C19) accidental challenges are

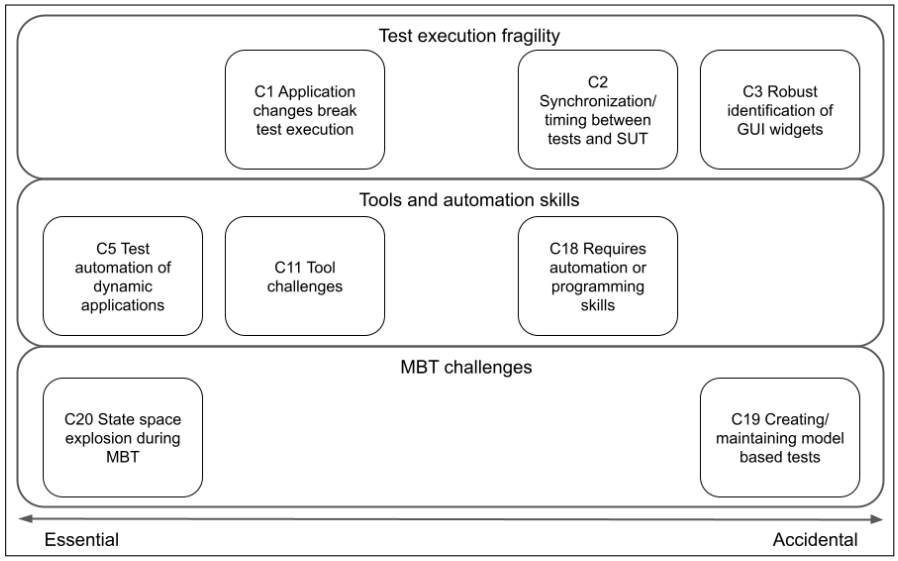

Figure 1.7: The key challenges related to GUI-based test automation arranged from essential to accidental difficulties.

addressed in studies B, C, D, E, and F. Study B answers RQ2 and proposes a novel technique called Augmented Testing and an academic tool called Scout that addresses the challenges related to the need for automation and programming skills (C18) and the problem of creating and maintaining model-based tests (C19). While Study B only evaluates the technique and the tool by receiving feedback from the industry, Study C continues by testing the tool on real applications downloaded from SourceForge. Study D answers RQ3 and targets the challenge related to locating GUI elements in a robust way (C3) by evaluating the proposed algorithm, called Similo, on 48 public web applications. The web element localization algorithm Similo is improved further in Study E (now called VON Similo), where we validate the performance on a similar set of web applications. Study F proposes and evaluates VON Similo LLM, which improves upon the VON Similo approach by utilizing the semantic understanding and context awareness of large language models.

1.5 Research Methodology

A research methodology encompasses a study's approach, strategy, and reasoning to gain new knowledge [158]. While methodology is the broad approach to acquiring knowledge, research methods are the specific tools and techniques for gathering and analyzing data. Validity in a study refers to the extent to which results are accurate and unbiased [143]. In this thesis, we follow the guidelines from Runesson and Höst that categorized four aspects of validity: construct validity, internal validity, external validity, and reliability [143]. Each research methodology has different characteristics and, therefore, varying levels of inherent beneficial/detrimental research validity traits [112]. Runesson and Höst formulated four types of research methodologies based on Robson's classification: descriptive, exploratory, explanatory, and improving [143]. *Descriptive research* aims to describe a contemporary situation or phenomenon. A case study is one example of descriptive research [143]. *Exploratory research* focuses on observing subjects in their natural environment, forming findings from observations. The data, often collected using interviews or surveys, is mainly qualitative [159]. *Explanatory research* quantifies a relationship or compares groups to identify a cause-effect relationship. The collected data is mainly quantitative and often gathered from a controlled experiment [159]. *Improving research* attempts to identify improvements to a studied phenomenon. Examples include design science and action research [142, 146]. Quantitative research uses numerical data, while qualitative research uses non-numerical data to evaluate a phenomenon.

Table 1.1: Overview of the research methods, data collection methods, and data analysis methods presented in this thesis.

	Chapters					
Research Method	**2**	**3**	**4**	**5**	**6**	**7**
Systematic Literature Review	X					
Workshop Study		X				
Quasi-Experiment			X			
Experiment				X	X	X
Data Collection Method	**2**	**3**	**4**	**5**	**6**	**7**
Snowballing	X					
Workshop		X				
Questionnaire Survey			X			
Quantitative data collection				X	X	X
Data Analysis Method	**2**	**3**	**4**	**5**	**6**	**7**
Thematic Analysis	X	X				
Statistical Analysis			X		X	

This research initially collected qualitative data about challenges (that are still valid today) with GUI-based test automation using a systematic literature review (SLR). The focus of the research than changed to aim towards finding novel solutions that target the challenges identified by the SLR and evaluate the solutions mainly using quantitative methods. Table 1.1 shows an overview of the research methods, data collection methods, and data analysis methods used in this thesis work. The remainder of this Section outlines and motivates our selection of research methodologies.

1.5.1 Systematic Literature Review

A systematic literature review (SLR) is a secondary study that uses a well-defined methodology to identify, analyze, and interpret evidence in an unbiased and repeatable way [87]. We used an SLR as the first method to gather challenges of GUI test automation from the literature and to answer research question one (RQ1). The SLR followed the guidelines provided by Kitchenham et al. [87] when establishing the research goal, defining research questions, identifying search strings, defining inclusion/exclusion criteria, performing Fleiss Kappa analysis, including/excluding publications, snowballing [157], performing data analysis and coding, and presenting the results.

The SLR found eight key challenges, four of which were classified as possible to solve or mitigate using technical solutions. These became the focus of the research as the remaining challenges are classified as inherent to the GUI test automation area and are not as easily, or perhaps even impossible, to solve. The remaining research methods in the thesis are used when attempting to mitigate the four chosen challenges identified by the SLR.

1.5.2 Case Study

A case study is a research method that involves an in-depth, detailed examination of a phenomenon, which can be a single unit, such as a person, group, event, or organization [143]. Case studies are usually conducted over some time, and they often involve extensive data collection and analysis. While case studies do not typically generate causal relationships like experiments, they can provide a deeper understanding of the phenomena due to being conducted in a real-world environment [143]. This thesis work does not contain any case studies, as initially planned, due to the outbreak of COVID-19, which forced us to interrupt an ongoing case study at Ericsson designed to compare Augmented Testing and Scout with a contemporary approach. At the same time, we abandoned our initial ambition of following the technology transfer model suggested by Gorschek et al. [69].

1.5.3 Workshop Study

A workshop study is a research method where experts meet to discuss and generate ideas or solve problems around a particular topic, similar to focus groups [88]. In Study B, we used workshops with consultants from the industry to evaluate the novel concept of Augmented Testing (AT). We selected this method to collect creative solutions, ideas, and opinions in a collaborative and time-efficient way. The downside of a workshop study is that group dynamics can bias the results, and therefore, the outcome may not represent all testers. In Study B, a prototype for augmented testing (i.e., Scout) was first demonstrated and later used by the participants in a workshop study involving ten software developers to gather their perceptions about the benefits and drawbacks of augmented testing and the tool Scout.

1.5.4 Experiment

An experiment is a scientific procedure performed to test a hypothesis or verify a phenomenon in a controlled environment [159]. While experiments offer more control over variables and a stronger basis for causal inference than case studies and quasi-experiments, they may lack real-world applicability (i.e., lower construct validity). Still, we decided to go for an experiment in a controlled environment when testing our hypothesis that our proposed web localization solution had higher effectiveness (i.e., more correctly located web elements) than the baseline approaches. A benefit of using experiments, apart from testing a hypothesis, is the convenience of replicating our design when comparing a novel solution or approach to ours. Using a case study would also be possible, but since it is conducted in a real-world environment, it would be harder to isolate the study results from the interference of confounding variables. We selected 48 (40 in Study E) of the most popular web applications for the experiments to enhance the construct validity and performed three experiments in this thesis work. The experiment in Study D evaluates the difference in robustness and time efficiency (i.e., dependent variables) between web element localization approaches (i.e., independent variable) on two different versions of 48 web applications. In Study E, we evaluated the concept of Visually Overlapping Nodes (VON) using a similar experiment that compared VON Similo (i.e., a version of Similo enhanced by the VON concept) with the original version of Similo. The experiment used a ground truth set of 1163 manually collected web element pairs extracted from 40 web applications. Results were evaluated based on the approach's precision, recall, and accuracy. A final experiment was conducted in Study F, using 804 web element pairs extracted from 48 real-world web applications to compare VON Similo LLM against the baseline algorithm (i.e., VON Similo). Again, we compared the effectiveness and efficiency (i.e., dependent variables) of both approaches (i.e., independent variable). We also analyzed motivations from the LLM for all instances where the original approach (i.e., VON Similo) failed to find the right web element.

1.5.5 Quasi-Experiment

A quasi-experiment is an observational study, similar to an experiment, that uses a treatment or intervention that is not randomly assigned [159]. This means that subjects are not randomly assigned to the control groups, which can introduce bias into the results. Quasi-experiments were used in Study C to evaluate the AT concept and the Scout prototype in collaboration with practitioners from the

industry. In the evaluation, quasi-experiments and questionnaire surveys were performed in three workshops with 12 practitioners from two Swedish companies (Ericsson and Inceptive). The quasi-experiments were based on guidelines for software engineering experiments with dependent and independent variables [159]. The quasi-experiments were used to measure time (i.e., dependent variable) given an approach (i.e., independent variable). The workshops in Study C involve human subjects within their domain, making it impossible to control for all confounding variables (e.g., the participant's hardware, previous knowledge, and allocated resources).

Quasi-experiments are realistic and relevant when conducted in a real-world environment. Another benefit is the convenience of using existing subjects and conditions instead of relying on random assignment. The drawbacks are lower internal validity (i.e., compared to experiments) due to random assignment and more challenging to make causal inferences due to confounding variables.

1.6 Overview of Chapters

This section contains an overview of the publications included in this thesis, from the initial SLR to the papers that propose and evaluate novel approaches and concepts.

Study A: Systematic Literature Review

Study A, presented in Chapter 2, is titled "Why many challenges with GUI Test Automation (will) remain".

Context: Automated testing is ubiquitous in modern software development and used to verify requirement conformance on all levels of system abstraction, including the system's graphical user interface (GUI). GUI-based test automation, like other automation, aims to reduce the cost and time for testing compared to alternative, manual approaches. Automation has been successful in reducing costs for other forms of testing (like unit- or integration testing) in industrial practice. However, we have not yet seen the same convincing results for automated GUI-based testing, which has instead been associated with multiple technical challenges. Furthermore, the software industry has struggled with some of these challenges for more than a decade with what seems like only limited progress.

Objective: This systematic literature review takes a longitudinal perspective on GUI test automation challenges by identifying them and then investigating why the field has been unable to mitigate them for so many years.

Method: The review is based on a final set of 49 publications, all reporting empirical evidence from practice or industrial studies. Statements from the publications are synthesized, based on a thematic coding, into 24 challenges related to GUI test automation. Key challenges (i.e., the most reported) were also classified as essential or accidental difficulties (see Figure 1.7).

Results: The most reported challenges were mapped chronologically (i.e., in time) and further analyzed to determine how they and their proposed solutions have evolved over time. This chronological mapping of reported challenges shows that four of them have existed for almost two decades.

Conclusion: Based on the analysis, we discuss why the key challenges with GUI-based test automation are still present and why some will likely remain in the future. For others, we discuss possible ways of how the challenges can be addressed. Further research should focus on finding solutions to the identified technical challenges with GUI-based test automation that can be resolved or mitigated. However, in parallel, we also need to acknowledge and try to overcome non-technical challenges.

Contributions: The specific contributions of this literature review are:

- The technical challenges of test automation through the GUI reported by academic literature.
- A longitudinal perspective of how the key challenges have evolved.
- To what extent the key challenges can be resolved or mitigated using technical solutions.

Study B: Workshop Study

Study B, presented in Chapter 3, is titled "Augmented Testing: Industry Feedback To Shape a New Testing Technology".

Context: Manual testing remains the predominant approach for software acceptance and system testing in the industry. While test automation has been proposed as a solution, both manual and automated testing face challenges that limit their effectiveness and return on investment. There is a pressing need for a new testing approach that can overcome these challenges and enhance efficiency and cost-effectiveness.

Objective: In this paper, we introduce a novel approach referred to as Augmented Testing (AT). AT involves testing by adding a visual layer between the tester and the System Under Test (SUT), superimposing information on the GUI.

Method: To explore the benefits and drawbacks of AT, we developed a prototype and conducted an industrial workshop study involving 10 software developers. The study aimed to gather the practitioners' perceptions and insights regarding AT's potential advantages and limitations.

Results: The workshop study revealed a higher number of benefits associated with AT than drawbacks. Notable benefits included improved clarity on what to test and what had been tested, as well as a reduction in manual work.

Conclusion: The findings from this study suggest that AT holds promise as a valuable testing technique. Its potential benefits, especially in terms of test clarity and reduced manual effort, warrant further research and development. AT could offer the industry new advantages that are currently lacking in existing testing approaches.

Contributions: The specific contributions of this study are:

- A presentation of a novel technique we refer to as Augmented Testing (AT) and its realization in a prototype.
- The collection of perceived benefits and drawbacks of AT from an industrial workshop study that can be used for further development of the technique and prototype.

Study C: Empirical Study

Study C, presented in Chapter 4, is titled "On the Industrial Applicability of Augmented Testing: An Empirical Study".

Context: Testing applications with graphical user interfaces (GUI) is a vital yet time-consuming task in software development. GUI test automation tools aim to streamline this process but often come with challenges related to script development and maintenance.

Objective: To address these challenges, a novel technique called Augmented Testing (AT) has been proposed. AT involves testing the System Under Test (SUT) using an Augmented GUI that provides guidance and records interactions.

Method: In this study, we evaluate a prototype tool called Scout that follows the AT concept. We conducted an industrial empirical study involving

12 practitioners from two Swedish companies, Ericsson and Inceptive. The evaluation included quasi-experiments and questionnaire surveys.

Results: The results indicate that Scout allows for the creation of equivalent test cases faster than using two popular state-of-practice tools, with statistical significance. This suggests that AT offers cost-value benefits and can be applied to industrial-grade software. It also addresses deficiencies in state-of-practice GUI testing technologies, particularly in terms of ease-of-use.

Conclusion: AT, as exemplified by the Scout prototype, presents a promising approach to improving GUI test automation. It offers advantages in terms of efficiency and ease of use, making it a valuable addition to the toolkit of software testers.

Contributions: The specific contributions of this study are:

- An overview of a novel technique for GUI test automation called Augmented Testing.
- Results from an empirical evaluation of AT with industrial practitioners of the efficiency of Scout, a prototype for AT, compared to two state-of-practice approaches for GUI test automation.

Study D: Experimental Study

Study D, presented in Chapter 5, is titled "Similarity-based web element localization for robust test automation".

Context: Non-robust (fragile) test execution is a commonly reported challenge in GUI-based test automation despite much research and several proposed solutions. A test script needs to be resilient to (minor) changes in the tested application but, at the same time, fail when detecting potential issues that require investigation. Test script fragility is a multi-faceted problem. However, one crucial challenge is how to reliably identify and locate the correct target web elements when the website evolves between releases or otherwise fail and report an issue.

Objective: This study proposes and evaluates a novel approach called similarity-based web element localization (Similo), which leverages information from multiple web element locator parameters to identify a target element using a weighted similarity score.

Method: This experimental study compares Similo to a baseline approach for web element localization. To get an extensive empirical basis, we target 48 of the most popular websites on the Internet in our evaluation. Robustness is

considered by counting the number of web elements found in a recent website version compared to how many of these existed in an older version.

Results: Results of the experiment show that Similo outperforms the baseline; it failed to locate the correct target web element in 91 out of 801 considered cases (i.e., 11%) compared to 214 failed cases (i.e., 27%) for the baseline approach. The time efficiency of Similo was also considered, where the average time to locate a web element was determined to be four milliseconds. However, since the cost of web interactions (e.g., a click) is typically on the order of hundreds of milliseconds, the additional computational demands of Similo can be considered negligible.

Conclusion: This study presents evidence that quantifying the similarity between multiple attributes of web elements when trying to locate them, as in our proposed Similo approach, is beneficial. With acceptable efficiency, Similo gives significantly higher effectiveness (i.e., robustness) than the baseline web element localization approach.

Contributions: The specific contributions of this study are:

- A novel approach for more robust web element localization based on comparison of the similarity of web element locator parameters.
- An empirical study that shows the effectiveness and time efficiency of the proposed approach compared to the baseline approach.

Study E: Experimental Study

Study E, presented in Chapter 6, is titled "Robust web element identification for evolving applications by considering visual overlaps".

Context: Fragile (i.e., non-robust) test execution is a common challenge for automated GUI-based testing of web applications as they evolve. Despite recent progress, there is still room for improvement since test execution failures caused by technical limitations result in unnecessary maintenance costs that limit its effectiveness and efficiency. One of the most reported technical challenges for web-based tests concerns how to locate a web element used by a test script reliably.

Objective: This study proposes the novel concept of Visually Overlapping Nodes (VON) that reduces fragility by utilizing the phenomenon that visual web elements (observed by the user) are constructed from multiple web elements in the DOM that overlap visually.

Method: We demonstrate the approach in a tool, VON Similo, which extends the state-of-the-art multi-locator approach (Similo) that is also used as

the baseline for an experiment. In the experiment, a ground truth set of 1163 manually collected web element pairs from different releases of the 40 most popular web applications on the internet are used to compare the approaches' precision, recall, and accuracy.

Results: Our results show that VON Similo provides 94.7% accuracy in identifying a web element in a new release of the same SUT. In comparison, Similo provides 83.8% accuracy.

Conclusion: These results demonstrate the applicability of the visually overlapping nodes concept/tool for web element localization in evolving web applications and contribute a novel way of thinking about web element localization in future research on GUI-based testing.

Contributions: The main contributions of this work are:

- Insights into the relative power of different web element attributes for web element localization.
- A generally applicable, yet novel, concept called Visually Overlapping Nodes (VON).
- An improved version of similarity-based web element localization (Similo) that implements VON (VON Similo).

Study F: Experimental Study

Study F, presented in Chapter 7, is titled "Improving web element localization by using a large language model".

Context: Web-based test automation heavily relies on accurately finding web elements. Traditional methods compare attributes but don't grasp the context and meaning of elements and words. The emergence of Large Language Models (LLMs) like GPT-4, which can show human-like reasoning abilities on some tasks, offers new opportunities for software engineering and web element localization.

Objective: This study introduces and evaluates VON Similo LLM, an enhanced web element localization approach. Using an LLM, it selects the most likely web element from the top-ranked ones identified by the existing VON Similo method, ideally aiming to get closer to human-like selection accuracy.

Method: An experimental study was conducted using 804 web element pairs from 48 real-world web applications. We measured the number of correctly identified elements as well as the execution times, comparing the effectiveness and efficiency of VON Similo LLM against the baseline algorithm. In addition,

motivations from the LLM were recorded and analyzed for all instances where the original approach failed to find the right web element.

Results: VON Similo LLM demonstrated improved performance, reducing failed localizations from 70 to 39 (out of 804), a 44% reduction. Despite its slower execution time and additional costs of using the GPT-4 model, the LLM's human-like reasoning showed promise in enhancing web element localization.

Conclusion: LLM technology can enhance web element identification in GUI test automation, reducing false positives and potentially lowering maintenance costs. However, further research is necessary to fully understand LLMs' capabilities, limitations, and practical use in GUI testing.

Contributions: The specific contributions of this study are:

- A novel approach that can improve web element localization by utilizing a large language model.
- An empirical study that shows the effectiveness and efficiency of the proposed approach compared to the baseline approach.
- A qualitative content analysis on the motivations gathered from the LLM, explaining the main aspects used when comparing the similarity of two web elements.

1.7 Threats to Validity

In this section, we present some of the primary threats to the validity of the results presented in this thesis, divided into four aspects suggested by Runeson et al. [143].

Construct validity: How we selected databases and constructed the search query in our systematic literature review could impact the publications included. We took steps like breaking down the query, using a full-text search tool, and manual review to minimize this impact.

We included four publications during snowballing, suggesting that our search query string could have been better. Still, we decided it was good enough since we argue that a few missed publications will likely not significantly impact the results. In the worst case, we have missed some challenges of importance, but given the authors' extensive empirical understanding of the area, we perceive it as unlikely. Additionally, for a challenge to be classified as "key", it had to be mentioned by four or more publications, making it even less likely that we missed a key challenge.

Internal validity: The authors of this study possess significant industrial experience, which presents a potential issue for the study's internal validity. We recognized that this experience could introduce bias in synthesizing and coding challenges (Chapter 2). Despite our awareness of this risk and our efforts to make impartial judgments, there remains a possibility of bias.

External validity: Although this thesis work aims to overcome some of the challenges related to the entire field of GUI-based test automation, our surveys, quasi-experiments, and experiments were all performed using web applications only, leaving out other types of GUI applications, such as Android and Windows apps. In a previous version of Scout (i.e., older than the version used in Chapters 3 and 4), we used image recognition instead of Selenium WebDriver with the benefit of working with any application having a GUI regardless of its implementation. However, using image recognition has a few drawbacks: (1) the results will be affected by the reliability of the selected image recognition algorithm, and (2) the test environment needs to be carefully controlled since results rely on the size of the screen or window. Therefore, using Selenium WebDriver as the driver was a more practical choice that mitigated some confounding variables. In the Similo study (Chapter 5), we selected to use web applications in the experiment when comparing Similo to other approaches since the selected baseline approaches (i.e., absolute XPath, relative ID-based XPath, Selenium IDE, Montoto, and Robula+) were all designed and evaluated by locating web elements in web applications. By convenience, it was easiest to continue using the same set of web applications (with minor changes) in the two studies, improving the Similo approach (Chapters 6 and 7).

While Android and Windows applications do not have a DOM structure like web applications do, they typically arrange GUI elements in a tree structure where GUI elements have properties (e.g., position, size, label, id, name). Both Scout (our demonstrator tool for Augmented Testing) and Similo (all versions) could conceptually work with any GUI model containing GUI elements with some arbitrary set of properties arranged in a hierarchy. Therefore, we postulate that our solutions would generalize to most GUI applications, but more research is needed.

Reliability: The author's expertise and familiarity with GUI-based test automation influence the analysis and categorization of challenges in the SLR (Chapter 2). This personal influence poses a risk to the consistency and repeatability of our study. If other researchers were to replicate the SLR, there might be slight variations in how they synthesize and describe the challenges. However, such differences would be minor and wouldn't significantly alter our main findings.

1.8 Discussion

In this Section, we synthesize the results from the six studies and discuss the implications and limitations of our findings.

Challenges of GUI-based Test Automation: Our SLR (Chapter 2) found that many of the challenges in GUI-based test automation have been around for a long time (i.e., often more than ten years). We classified four of the most reported challenges as possible to mitigate using a technical solution (i.e., accidental). One of those challenges is the need for automation and programming skills when designing reliable and maintainable GUI-based test automation tests. Another is the time and cost associated with creating and maintaining model-based tests caused by SUT changes. Fragile test scripts caused by non-robust identification of GUI widgets (e.g., web elements) is a third challenge and one of the most frequently reported. The fourth challenge is the need to synchronize the test execution between the SUT and the test runner, which is closely related to the challenge of robust widget identification. This thesis work proposes and evaluates solutions for the first three challenges. In contrast, the fourth synchronization challenge is only affected (i.e., a side effect) by a more reliable way of identifying GUI widgets since it is a common practice to add dynamic delays halting the script execution until a widget appears.

Augmented Testing Advantages: We proposed and evaluated a concept called Augmented Testing (AT) and a prototype research tool called Scout (Chapters 3-4) in an attempt to mitigate the challenges associated with the need for programming and test automation skills when creating and maintaining model-based tests (i.e., two of the key accidental challenges). The aim was to reduce or eliminate the need for programming skills by offering an alternative way of creating and maintaining tests utilizing an Augmented Layer (i.e., augmented information projected on top of the SUT GUI). Our hypothesis was that creating and maintaining tests using an Augmented Layer is a more efficient and user-friendly way than the more traditional script-based approach that requires programming skills while still providing similar benefits. We performed workshops, quasi-experiments, and questionnaire surveys in collaboration with participants represented by three industrial companies and concluded that the participants find AT and Scout promising and that the concept can provide cost-value benefits when compared to state-of-practice approaches. However, we have only gathered perceptions or empirical data from a small number of participants (i.e., 22 in total from three companies) and compared Scout with two state-of-practice tools (i.e., Selenium and Protractor). More research is

needed by, for example, evaluating AT in real-world scenarios in collaboration with testers from the industry.

More Robust Web Element Localization: We proposed and evaluated approaches (i.e., Similo, VON Similo, and VON Similo LLM) that attempt to improve the robustness of identifying GUI widgets in comparison to existing approaches like the Robula+ algorithm suggested by Leotta et al. [100] (Chapters 5-7). A more robust approach to locating web elements can reduce maintenance efforts (e.g., due to fewer broken scripts to repair). However, we must still modify the automated tests as the SUT changes. We can integrate an approach like Similo into existing test scripts to, for example, replace the findElement method in scripts that utilize Selenium WebDriver [11]. Such an approach (i.e., tool) could also automatically manage (i.e., store and repair) the properties used by Similo (i.e., when finding the best match), reducing manual maintenance even further. Repairing the properties could be done by replacing existing ones with new ones as they change. However, this may also destroy correctly stored property values with invalid values in the case of a failed test. We have not evaluated repair in this thesis work since we only decided to focus on the localization problem.

With a more robust test execution, testers can shift their focus from maintaining and repairing failing test cases to designing and creating tests that ensure the software delivers value to the end-user. They could also spend more time on other test activities like exploratory, performance, and usability testing.

In this thesis work, we have only evaluated the effectiveness of locating web elements based on oracles created by the authors, i.e., the web element pairs selected in an older and newer version of the same web page. However, the definition of the same web element is not trivial. We could define it as a web element with the exact same set of attributes located at the same location (XPath). Another definition can compare pixels to ensure they have all identical RGB values. With a less strict definition, it is more complicated to determine when two elements are close enough to represent the same web element (e.g., a button or an input field). Simply selecting the most similar (i.e., according to some measurements) is not always adequate since that would result in false positives, likely causing a failed (or inaccurate) test execution. One solution to this problem is to locate the most similar web element when performing actions (e.g., a button click) and mitigate the risk of false positives by including checks verifying that we are still on the correct test execution path. However, we did not investigate such a solution in this thesis work.

The test synchronization problem is closely related and sometimes difficult to distinguish from the challenge of robust localization of web elements. The-

oretically, we could add a long (or infinite) delay after each performed action to ensure that we are in the following application state (e.g., the next page). To be in the correct application state is crucial for ensuring that the performed action will get the same result as a previous test run (also assuming that the SUT is deterministic). We can only make that assumption when assuming that the SUT behaves deterministically. Without determinism, the SUT can go into an unknown application state with unpredictable outcomes. However, waiting for a long time is impractical since it lowers the efficiency of the tests. The challenge is to wait long enough (but not longer than necessary) to ensure that we are in the expected application state and that the SUT is ready to receive the next action. When the delay is not long enough, we might end up in a situation where the web element that we aim to interact with is not yet loaded or visible in the GUI, with the result that the most similar web element is not the correct one (i.e., since it is not yet available). Triggering an action (e.g., a click) on an incorrect web element will likely take us on a different path, eventually resulting in a failed test execution. In this thesis, we have deliberately ignored this problem by assuming we are already in the correct application state, leaving the test synchronization challenge to future work.

1.9 Future Work

The integration of large language models, as seen with the VON Similo LLM approach, indicates the potential role of AI in revolutionizing GUI-based test automation. Recent work by Park et al. and Feldt et al. suggests that the AI-driven approach to GUI testing is gaining traction [63, 135]. In the near future, we anticipate AI agents able to handle all aspects of regression testing for a GUI application (i.e., from design to test execution). These agents would operate based on straightforward instructions in natural language. The instructions could range from detailed step-by-step test cases (e.g., click on the save button) to high-level goals (e.g., test this website). This transformative approach could enhance efficiency and reduce the need for human intervention. The shift would also make the testing process more accessible, as the reliance on specialized scripting would diminish, paving the way for a more user-friendly approach to GUI regression testing.

In conclusion, while our research provides significant advancements in GUI-based test automation, the journey has just begun. The potential use of AI when mitigating accidental and perhaps even some of the essential challenges signals

an opportunity for future research advancements in GUI-based test automation that can benefit the industry.

Chapter 2

Why many challenges with GUI Test Automation (will) remain

Abstract

Context: Automated testing is ubiquitous in modern software development and used to verify requirement conformance on all levels of system abstraction, including the system's graphical user interface (GUI). GUI-based test automation, like other automation, aims to reduce the cost and time for testing compared to alternative, manual approaches. Automation has been successful in reducing costs for other forms of testing (like unit- or integration testing) in industrial practice. However, we have not yet seen the same convincing results for automated GUI-based testing, which has instead been associated with multiple technical challenges. Furthermore, the software industry has struggled with some of these challenges for more than a decade with what seems like only limited progress.

Objective: This systematic literature review takes a longitudinal perspective on GUI test automation challenges by identifying them and then investigating why the field has been unable to mitigate them for so many years.

Method: The review is based on a final set of 49 publications, all reporting empirical evidence from practice or industrial studies. Statements from the

publications are synthesized, based on a thematic coding, into 24 challenges related to GUI test automation.

Results: The most reported challenges were mapped chronologically and further analyzed to determine how they and their proposed solutions have evolved over time. This chronological mapping of reported challenges shows that four of them have existed for almost two decades.

Conclusion: Based on the analysis, we discuss why the key challenges with GUI-based test automation are still present and why some will likely remain in the future. For others, we discuss possible ways of how the challenges can be addressed. Further research should focus on finding solutions to the identified technical challenges with GUI-based test automation that can be resolved or mitigated. However, in parallel, we also need to acknowledge and try to overcome non-technical challenges.

Keywords: System Testing, GUI Testing, Test Automation, Systematic Literature Review

2.1 Introduction

Manual testing is time-consuming, repetitive, and error-prone to perform for a human tester [73] [71]. Test automation, using techniques like unit testing [131] and record-replay [17], have been suggested as solutions to these challenges and proven successful in practice. Another reason for test automation is that the tests can be executed faster and more frequently and, therefore, deliver faster feedback about the quality of the software under development [108]. While automated testing is the state of practice for checking code and APIs (using unit- and integration testing), it is still common to manually test a system under test (SUT) through the graphical user interface (GUI) without any automation. That manual testing, using the GUI, is preferred before test automation indicates that there are challenges related to GUI-based test automation that reduces its applicability compared to manual GUI-based testing. This observation is supported by literature, which has frequently reported on challenges, problems, and limitations with GUI test automation.

This systematic literature review (SLR) [87] aims to take a longitudinal perspective to find the technical challenges reported by academic literature with GUI-based test automation during the last 20-years to determine how the most reported (key) challenges have evolved and if there have been attempts to mitigate them. In this SLR, the focus is mainly on testing the functionality of

the SUT by using the GUI, not testing or checking that the user interface is presented, or rendered, correctly on the screen.

There have been previous attempts that try to identify the challenges (technical or otherwise) of software test automation. For instance, in 2012, Rafi et al. presented an SLR, which identifies 9 benefits and 7 limitations of automated software testing [138]. The literature review identified several benefits: "Less human effort" and "Increased fault detection". However, "Automation can not replace manual testing" and "False expectations" were identified as some of the limitations. A more recent SLR study by Wiklund et al., published in 2017, reported impediments, both technical and non-technical, provided similar results on software test automation [156].

In this SLR, unlike the previous that studied the entire field of test automation, the focus is on test automation performed using the GUI only. Hence, a subset of the software test automation field. Additionally, the SLR is delimited to the technical challenges of the approach that can be addressed using technical solutions, leaving out any non-technical perspectives such as behavioral, process, or business aspects. Furthermore, the SLRs published by Rafi et al. and Wiklund et al. present limitations and impediments on a relatively high abstraction level, an effect of their studies looking at such a large field of study. As an example, Rafi et al. reported the limitation: "Difficulty in maintenance of test automation" and Wiklund et al., the impediments: "Maintenance Costs" and "Fragile Test Scripts". In contrast, the goal of this SLR is to dig deeper into the technical challenges to identify their root causes to acquire further insights into their solution. Finally, the previous SLRs do not present solution attempts, how the challenges have evolved, and if the challenges are possible to solve or mitigate, thereby motivating the need for this SLR that investigates the complexities and longevity of each unresolved challenge in more detail. Knowing how consistently the challenges were described over a more extended time period, and any solution attempts, will make it possible to identify solutions or mitigation strategies for the key (most commonly reported) technical challenges.

The specific contributions of this literature review are:

- The technical challenges of test automation through the GUI reported by academic literature.
- A longitudinal perspective of how the key challenges have evolved.
- To what extent the key challenges can be resolved or mitigated using a technical solution.

This paper is structured as follows: The research process behind the systematic literature review is described in Section 2.2. Section 2.3.1 contains the results obtained by gathering information from the included publications. Section 2.4 discusses the results presented in Section 2.3.1 and synthesised in Section 2.3.2. We will give our conclusions and some work to consider for the future in section 2.6.

2.2 Systematic Literature Review

The SLR presented in this work is based on the guidelines for conducting systematic literature reviews in software engineering, presented by Kitchenham et al. [87]. An overview of the literature review process is shown in Figure 2.1. The number of included publications after each step is presented in Table 2.1.

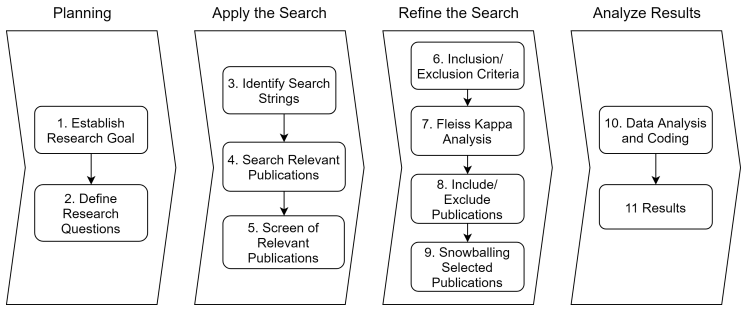

Figure 2.1: Literature review process

Table 2.1: Overview showing the number of included publications per step.

Step	Description	No included
4	Search Relevant Publications	185
5	Screen of Relevant Publications	95
8	Include/Exclude Publications	39
9	Snowballing Selected Publications	49

1. Establish Research Goal: The research goal of this SLR is to find the key challenges, based on empirical evidence found in literature, associated with testing a system automatically through its GUI. Knowing the key challenges and their root-causes is crucial for future research to find effective solutions to mitigate them efficiently.

2. Define Research Questions: The research goal has been broken down into the following research questions:

- RQ1: What are the academically reported technical challenges for automated testing of an application through its GUI?
- RQ2: How have key challenges for GUI-based test automation evolved in the last 20 years?
- RQ3: To what extent can the key challenges be resolved or mitigated using a technical solution?

3. Identify Search Strings: The search string sought to find publications that contain empirical results about the challenges of automated testing/checking of software applications/programs through the GUI. Keywords related to benefits were included in the search since a discussion about benefits is likely to contain statements concerning challenges as well.

The query string was constructed from keywords and synonyms to these keywords, which were formulated based on the authors' industrial experience and taking keywords used in previous literature reviews into consideration [138, 156]. Below, we present a decomposition of the search string.

Focus on software applications or programs: (software OR application OR program) AND

Focus on automated testing/checking: ("test automation" OR "automatic testing" OR "automated testing" OR "automated test" OR "automatic checking" OR "automated checking" OR "automated regression") AND

Focus on user-interfaces or human-machine interfaces: ("user interface" OR GUI OR "human machine interface" OR HMI) AND

Contains benefits and/or challenges: (challenge OR benefit OR advantage OR improvement OR limitation OR drawback OR problem OR pitfall) AND

Contains empirical studies or experiments: (empirical OR industry OR industrial OR practice OR "case study" OR survey OR experiment)

Resulting in the final query string: (software OR application OR program) AND ("test automation" OR "automatic testing" OR "automated testing"

OR "automated test" OR "automatic checking" OR "automated checking" OR "automated regression") AND ("user interface" OR GUI OR "human machine interface" OR HMI) AND (challenge OR benefit OR advantage OR improvement OR limitation OR drawback OR problem OR pitfall) AND (empirical OR industry OR industrial OR practice OR "case study" OR survey OR experiment)

Exclude publication released before year 2000: Due to the rapid evolution of the software engineering field, we decided to exclude all publications published before the year 2000. This choice was made to delimit the number of found papers and because papers older than this are considered out-of-date and not applicable to modern software systems.

4. Search Relevant Publications: Four of the most widely used literature databases available today for the software engineering field were selected for the search: Scopus, IEEE Xplore, ACM, and Wiley. The final search string and the filter that excludes all publications published before the year 2000 were used in the selected databases. The database searches resulted in a total of 185 included publications distributed chronologically, as shown in Table 2.2.

Table 2.2: Search Results

Scopus	**IEEE Xplore**	**ACM**	**Wiley**
69 matches	58 matches	35 matches	23 matches

5. Screen of Relevant Publications: All publications included in the search results were downloaded and imported into the tool Mendeley [170]. Mendeley was used to remove any duplicate publications found in more than one database. Non-research publications, publications that were not peer-reviewed, or not written in English were also excluded resulting in 95 remaining publications.

6. Inclusion/Exclusion Criteria: We formulated three tiers of inclusion/exclusion criteria as shown in Table 2.4, 2.5 and 2.6. Publications that satisfied all of the inclusion criteria and did not match any of the exclusion criteria for a tier were included and evaluated by the criteria in the next tier until all the tiers were covered. Table 2.3 contains definitions for some of the terms that we used in the inclusion/exclusion criteria.

The three tiers were defined as:

- **Tier 0 - Check Preconditions:** Analysis of paper metadata to ensure that the paper followed the base criteria to be eligible for further analysis.

- **Tier 1 - Check Title and Abstract:** Analysis of the title and abstract of each paper to get an indication that it would include support evidence usable to meet the research objective.

- **Tier 2 - Read Full Text:** Analysis of the complete paper to ensure it would provide evidence to meet the research objective.

Table 2.3: Definitions

Term:	**Definition:**
peer-reviewed article	Peer-reviewed articles are written by experts and are reviewed by several other experts in the field before the article is published to ensure the article's quality.
Empirical evidence	Any result acquired or observed by a human (researcher) in a reported experiment or case study (regardless of size).
Graphical User Interface (GUI)	A form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators.
System Under Test (SUT)	The system that is being tested for correct operation.
Document Object Model (DOM)	A cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document.
Widget	A component of a GUI, that enables a user to perform a function or access a service.
Automated Testing	Automated testing or test automation is a method in software testing that makes use of special software tools to control the execution of tests and then compares actual test results with predicted or expected results. Test execution is done automatically with little or no intervention from the test engineer. For this study, automated testing regards only tests of the functionality or features of the SUT, not its quality characteristics (like usability or performance) nor the visual appearance (that widgets are rendered correctly) on the SUT's user interface.
Automated Testing through the GUI	Automated testing of a system through emulation of human/user usage of the SUT. The emulation can be performed using physical robotics (e.g. a robot arm) or purely through software (e.g. using OS-level events to type on the keyboard or move the mouse cursor), or a combination of these. Access to widgets is given by the accessibility API of the operating system, DOM objects in the SUT, internal GUI controls or other technical interfaces. These are all considered valid methods for emulating a human/user.

Table 2.4: Inclusion/Exclusion criteria tier 0

Inclusion criteria:	**Exclusion criteria:**
Included publications must be peer-reviewed. See the definition of peer-reviewed article in Table 2.3.	Exclude all publications published before year 2000 since they are considered out-of-date and not applicable to modern software systems.
Include workshop, conferences, book chapters, and journal publications.	Exclude short publications (less than 6 pages).
Include publications written in the English language only.	Exclude gray literature (non-peer-reviewed books, blogs, etc)
	Exclude bachelor/master/Ph.D. theses.

Table 2.5: Inclusion/Exclusion criteria tier 1

Inclusion criteria:	**Exclusion criteria:**
Include publications that indicate that they report empirical evidence about benefits or challenges of automated testing/checking through the GUI.	Exclude publications that do not study automated testing/checking through the GUI.
	Exclude publications that do not cover benefits or challenges of automated testing/checking through the GUI.
	Exclude publications that do not contain any empirical evidence.
	Exclude publications where the empirical evidence does not report benefits or challenges of automated testing/checking through the GUI.

Table 2.6: Inclusion/Exclusion criteria tier 2

Inclusion criteria:	**Exclusion criteria:**
Include publications that contain empirical evidence about the benefits or challenges of automated testing/checking through the GUI.	Exclude publications that do not study automated testing/checking through the GUI.
	Exclude publications that do not cover benefits or challenges of automated testing/checking through the GUI.
	Exclude publications that do not contain any empirical evidence.
	Exclude publications where the empirical evidence does not report benefits and/or challenges of automated testing/checking through the GUI.

7. Fleiss Kappa Analysis: The leading researcher evaluated the remaining 95 publications using the inclusion/exclusion criteria for tier 0 and randomly selected 20 (more than 20 percent) of the included publications for Fleiss Kappa analysis [64]. The authors of this publication reviewed the randomly selected publications based on the inclusion/exclusion criteria for Tier 1 and Tier 2, marking papers that they believed should be included, excluding the rest. Fleiss Kappa analysis, a measure of the agreement between reviewers (inter-rater agreement), was then calculated. The result was 0.60, indicating a moderate or close to a substantial agreement between the reviewers [92]. This agreement among the three reviewers was considered good enough for the leading researcher to continue to review the remaining publications and gave us confidence that the inclusion/exclusion criteria were suitably defined. Of course, a threat remains that a few papers are overlooked, but we argue that the impact of a few incorrectly excluded or included publications would not likely have a significant impact on the results.

8. Include/Exclude Publications: All publications that were not randomly selected and reviewed during the Fleiss Kappa analysis were reviewed, by the leading researcher, using the inclusion/exclusion criteria for tier 1 and 2.

The review resulted in 39 included publications, counting also the publications there were included during the Fleiss Kappa analysis.

9. Snowballing Selected Publications: There are many possible reasons for not finding a relevant publication during the database searches. One is that the author of the publication uses a different vocabulary than the one used in the search string. Another reason is that the search keywords are not found since most literature databases only search a portion of the publication, like the title and abstract, but not the full text.

To verify that all relevant papers had been captured by the search, backward and forward snowballing, as described by Wohlin [157], was performed on the 39 included publications. Publications identified during the snowballing session that seemed relevant for the study, based on reading their title or abstract, were collected into a set of candidates. The leading researcher then assessed all the candidates using the inclusion/exclusion criteria to ensure that we analyzed all added or discarded candidates with the same rigor and systematic review as previously included publications. Using the existing inclusion/exclusion criteria is perceived to strengthen the approach's reliability since these criteria had been previously agreed upon by the researchers. This snowballing session resulted in five additional publications that passed the inclusion/exclusion criteria. Additionally, since performing an SLR is a lengthy process, another five publications were later discovered that had been published while writing this paper, resulting in a final total of 49 included publications.

10. Data Analysis and Coding: Analysis of the included publications was performed by the leading researcher using thematic analysis and coding [54]. 186 statements related to technical challenges about GUI-based test automation were identified and extracted from the 49 included publications. A statement is defined as a sentence, quote, or paragraph from the author of the publication based on experience, a conviction, or concluded from a result. All statements are extracted from publications that report empirical results making it more likely that the statements are relevant and valid. The codes (C1...Cn) were formulated iterative, i.e., without any starting set, from the statements, focusing on challenges. Each extracted statement was mapped to an existing code when the leading researcher believed that the statement related to the same challenge or mapped to a new code otherwise. After coding, the codes were analyzed for semantic equivalence and merged if such was found. The merger was done by formulating a new code that replaced both existing codes, or by removing one of the existing codes.

The resulting set of codes and statements were then used as evidence to draw the study's conclusions and answer its research questions.

2.3 Results and Synthesis

Here we present an overview of the results and answer the first research question (RQ1) in Section 2.3.1. The answers to the remaining research questions (RQ2 and RQ3) are presented in Section 2.3.2 after further analysis and synthesis of the reported challenges.

2.3.1 Results

Table 2.7 contains a list of the 49 included publications that contain one or more statements related to challenges associated with GUI-based test automation. The coded and mapped challenges from the reviewed literature are listed in Table 2.9 with references to this paper's reference list. This table thereby represents a summary of all the currently known challenges explicit to automated testing through an application's GUI, representing an answer to RQ1.

To provide an overview, Figure 2.2 visualizes the key (most commonly reported) challenges on a timeline sorted on the year that the publication, which reports the challenge, was published. The criteria for a challenge to be considered "key" is that four or more publications can triangulate it. Each dot in the timeline represents one, or more, publications that were published during that year. An index number specifies the number of publications when more than one. The figure shows that several of the challenges have existed for almost 20 years but that the majority of challenges are newer, i.e., published within the last seven years.

To gain further insights, Figure 2.3 also visualizes what application platform (desktop, web, or mobile), the empirical evaluation was performed on and how these platforms appear over time. As shown in the figure, the first period, from the year 2000 to 2004, only contains two empirical evaluations, both on desktop applications. In the second and third periods (2005-2015), research on desktop applications is dominant, while in the fourth period, web-applications become dominant, and research in mobile applications are on the rise. We also note that the number of published papers increase over time, indicating a growing interest in empirical studies of GUI-based test automation.

Table 2.7: Included publications

No:	Title:	Year:
1	Hierarchical GUI Test Case Generation Using Automated Planning [115]	2001
2	Effective Automated Testing: A Solution of Graphical Object Verification [148]	2002
3	Experimental Assessment of Manual Versus Tool-Based Maintenance of GUI-Directed Test Scripts [72]	2009
4	Automated GUI Testing for J2ME Software Based on FSM [77]	2009
5	Repairing GUI Test Suites Using a Genetic Algorithm [78]	2010
6	Debug Support for Model-Based GUI Testing [76]	2010
7	Experiences of System-Level Model-Based GUI Testing of an Android Application [149]	2011
8	AutoBlackTest: Automatic Black-Box Testing of Interactive Applications [109]	2012
9	Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study [28]	2013
10	Graphical User Interface Testing Using Evolutionary Algorithms [94]	2013
11	Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing [19]	2014
12	On the Industrial Applicability of TextTest: An Empirical Case Study [25]	2016
13	An analysis of automated tests for mobile Android applications [145]	2016
14	Continuous Integration and Visual GUI Testing: Benefits and Drawbacks in Industrial Practice [30]	2018
15	Introducing automated GUI testing and observing its benefits: an industrial case study in the context of law-practice management software [67]	2018
16	Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles [111]	2018
17	Using a Pilot Study to Derive a GUI Model for Automated Testing [164]	2008
18	A “More Intelligent” Test Case Generation Approach through Task Models Manipulation [46]	2017
19	Evaluating the TESTAR tool in an industrial case study [37]	2014
20	Comparing Automated Visual GUI Testing Tools: An Industrial Case Study [66]	2017
21	Robust Test Automation Using Contextual Clues [166]	2014
22	Obstacles and opportunities in deploying model-based GUI testing of mobile software: a survey [80]	2012
23	PESTO: Automated migration of DOM-based Web tests towards the visual approach [101]	2018
24	Design and industrial evaluation of a tool supporting semi-automated website testing [108]	2014
25	Automatic testing of GUI-based applications [110]	2014
26	An event-flow model of GUI-based applications for testing [114]	2007
27	Pattern-based GUI testing: Bridging the gap between design and quality assurance [118]	2017
28	Model-based Approach to Assist Test Case Creation, Execution, and Maintenance for Test Automation [74]	2011
29	Efficient and Change-Resilient Test Automation: An Industrial Case Study [150]	2013
30	Reuse of model-based tests in mobile apps [55]	2017
31	Automated Testing of Software-as-a-Service Configurations using a Variability Language [136]	2015
32	Reducing GUI Test Suites via Program Slicing [36]	2014
33	Improved GUI Testing using Task Parallel Library [134]	2016
34	Introducing Model-Based Testing in an Industrial Scrum Project [62]	2012
35	Development and Maintenance Efforts Testing Graphical User Interfaces: A Comparison [90]	2016
36	Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case? [171]	2016
37	Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation [35]	2011
38	Using combinatorial testing to build navigation graphs for dynamic web applications [155]	2016
39	Automating Web Application Testing from the Ground Up: Experiences and Lessons Learned in an Industrial Setting [57]	2016

Table 2.8: Included publications, continued

No:	Title:	Year:
40	Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning [33]	2019
41	Using Multi-Locators to Increase the Robustness of Web Test Cases [99]	2015
42	Visual GUI Testing in Practice: limitations, Problems and Limitations [23]	2015
43	Call Stack Coverage for GUI Test-Suite Reduction [113]	2006
44	A Black-Box Based Script Repair Method for GUI Regression Test [81]	2018
45	A New Algorithm for Repairing Web-Locators using Optimization Techniques [60]	2018
46	Apply computer vision in GUI automation for industrial applications [48]	2019
47	ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing [100]	2016
48	Offline Oracles for Accessibility Evaluation with the TESTAR Tool [56]	2019
49	Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: an Empirical Study [29]	2015

Table 2.9: Reported challenges related to GUI-based test automation

Challenge:	Reported by publication no:
C1 Application changes break execution	1, 3, 10, 12, 14, 21, 24, 27, 29, 31, 35, 39, 41, 43, 44, 46, 49
C2 Synchronizing/timing between tests and SUT	1, 6, 19, 20, 32, 35, 39, 40, 42, 45
C3 Robust identification of GUI widgets	2, 7, 9, 20, 23, 24, 29, 42, 44, 45, 46, 49
C4 Changed screen resolution	13, 29, 40
C5 Test automation of dynamic applications	11, 13, 19, 21, 29, 32, 38, 39
C6 Fails for unknown reason	14, 24
C7 Non-determinism	40, 49
C8 Sensitivity to business logic	20
C9 Cascading error	29
C10 Environment/Hardware configuration	13, 21, 29
C11 Tool challenges	9, 11, 12, 13, 19, 22, 24, 29, 36, 42
C12 High maintenance cost of test cases	10, 24
C13 Long setup time	22, 24
C14 HMI must be available	10, 34
C15 Hard to reproduce the error	6, 19, 24
C16 Less effective in detecting faults	15
C17 Requires workflow changes	22
C18 Requires automation or programming skills	7, 9, 12, 19, 22, 24, 30, 31, 39, 46
C19 Creating/maintaining model based tests	6, 8, 11, 16, 22, 26, 27, 28, 31, 34, 40, 48, 49
C20 State space explosion during MBT	1, 4, 5, 16, 17, 18, 25, 26, 27, 32, 37, 38, 43, 49
C21 Low test coverage during MBT	19, 36
C22 Identify less faults when using an oracle	19
C23 Limited applicability of models	26, 49
C24 Slow test execution during MBT	19, 30

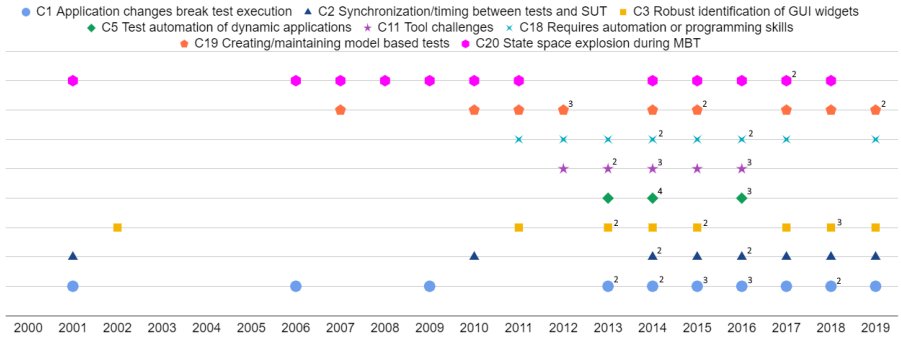

Figure 2.2: The key challenges mapped on a timeline. Each dot represents a statement from one or more publications published during the year.

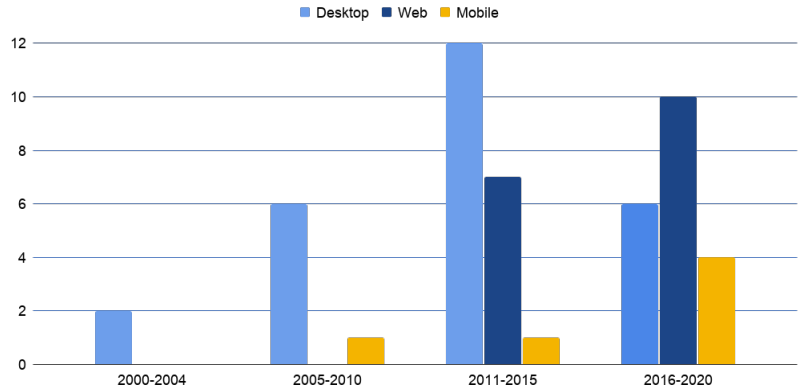

Figure 2.3: The type of software application selected for the empirical evaluation distributed over four time-periods.

2.3.2 Synthesis

To answer RQ2, the SLR results were analyzed further and synthesized to identify how consistent the challenges remained (e.g., how consistently they were described) over the studied time period. Four of the challenges were reported repeatedly during 17 years or more, as can be seen in Figure 2.2, and one (C5) was only reported during a four-year-long time-period.

The analysis also showed that the level of description of the challenges varied and that some reported challenges were so generally defined that it is unlikely that their description can aid in finding any candidate solutions. Such challenges must first be broken down, i.e., further detailed, to be adequately understood. As an example, Mahmud et al. reported in 2014, that practitioners with test automation experience mentioned that automated tests had to be rewritten or re-recorded when the application changed. Mahmud et al. did not document what explicit changes the practitioners perceived to be the underlying reason for tests to be rewritten or re-recorded or if all changes had the same impact. Such information would be valuable in guiding both development and research efforts towards finding candidate solutions to aid the practitioners.

To answer RQ3 and better understand which challenges to address with more focus and effort and which challenges to defer from or focus less effort on, we need to distinguish different challenges. In the book "No silver bullet" by Brooks et al., software engineering challenges are classified as essential or accidental [43]. We argue that a similar analysis can add value here. Essential challenges are inherent in the nature of software, whilst accidental challenges are challenges today but not inherent. This classification implies that accidental challenges can be resolved or mitigated through technical, or other, means. In contrast, essential challenges are not necessarily solvable because they are so fundamental, or significant, that no general solution can be perceived. In the following descriptions of found challenges, we use Brooks et al. definitions to classify the identified key challenges in an attempt to provide guidance for future research. However, we decided to limit the definition of accidental challenges to challenges that can be mitigated using a technical solution to match the scope of this SLR. Therefore, our definition of an accidental challenge is that it can be solved using a technical solution. Also, our definition of an essential challenge is that any technological solution cannot completely resolve it. Hence, excluding challenges related to human, process, or organizational aspects of GUI-based test automation.

Application changes break test execution (C1): The most reported challenge for automated GUI-based tests is that application changes cause tests to fail, stated by seventeen publications between 2001 to 2018. Memon et al. wrote in 2001 that regression testing of GUI applications is a challenge since the user interface does not remain constant across successive versions of the software [115]. A similar conclusion came from Jiang et al., in 2018, that scripts made for the previous version of the SUT do no longer work well when the GUI layouts change [81].

This challenge is mainly essential since a significant change of the SUT will, and should, break the test execution. Hence, a prerequisite of any technical solution that seeks to eliminates this challenge must fail the test when the test execution is expected to break, but otherwise pass. In particular, this challenge relates to the oracle problem, i.e. what information, and how much information, to use to determine correct SUT behavior [56].

As an essential challenge, we can conclude that this challenge is present today and likely will not be entirely removed in the future. However, when discussing this challenge, we must also consider the background to why the SUT was changed. Some changes to the SUT are intentional, and the tests might need to be adjusted accordingly to prevent the tests from failing during execution. However, other changes are unintentionally caused by intentional modifications to the SUT and might break the test execution since the tests are not adjusted according to the unintentional changes. Furthermore, different changes range in ease of detection. Significant changes, like a missing widget, are simple for a human tester to spot, while minor changes, like the size or position of GUI widgets, might be impossible for a human to notice. Unintentional minor changes that cause the test execution to fail can be frustrating for the human tester since the automated test scripts seem to break for no apparent reason. Since the human tester would expect an intentional change to impact the tests, further research should focus on finding solutions that reduce failures caused by unintentional, and primarily minor, changes. For example, a technical solution that provides a more robust identification of GUI widgets (C3) might prevent minor changes to the SUT from breaking the test execution.

Synchronizing/timing between tests and SUT (C2): The challenge of keeping the automated test scripts, or states in a model, in sync with the tested application was reported by ten publications published between 2001 and 2019. This challenge stems from the need for tests to wait until the SUT is ready to receive the next action, or event, to avoid the risk that the action does not have the desired effect on the SUT. As an example, let us study the following pseudo test script:

```
Step 1: Open the Report Dialog.
Step 2: Click on the Create Report button in the Report Dialog.
Step 3: Check that the Report has been created.
```

Since it may take some time to open the Report Dialog (step 1), there is a risk that step 2 is performed before the Report Dialog and Create Report button are visible. The Report might not be created if step 2 is performed before the

Create Report button is ready to receive the click action, causing step 3 to fail. A human, replicating the failure, would not perceive the same challenge, and we observe that the script, in this case, would have reported a "false positive" test result. However, and regardless, the test result still required investigation, leading to unnecessary root-cause analysis costs. The solution to making the test more robust would be to add synchronization, like static or dynamic delays, between the steps.

However, as stated by Heiskanen et al., in 2010, a common cause of failure in test runs are the delays between executing certain keywords on the SUT [76]. This statement is supported by Cheng et al. [48], in 2019, which stated that "The timing to trigger a series of events may not always be the same between two different runs. So, a straightforward replay is often infeasible."

The synchronization challenge has been reported repeatedly, from empirical evaluations on all types of applications (desktop, web, and mobile), during an 18-year time-period and is likely still a challenge today, supported by two publications published as recently as 2019. Since human testers do not seem to experience this challenge when testing manually, but machines and software often face issues, it is perceivable that this is a purely accidental, and technical, challenge. Debroy et al. [57] suggest that static (or fixed) delays, e.g., a delay of a few seconds, will solve this challenge but at the cost of making the script execution unnecessarily slow. For instance, assume that the transition between GUI states is 1.9 seconds. A delay of 2 seconds would then be sufficient to synchronize the test execution and ensure that the next script action would happen after the GUI state change, with only a penalty of 100-millisecond unnecessary delay. Of course, these penalties should be kept as small as possible, but (1) fine-tuning synchronization is time-consuming [25], and (2) having minor penalties increases the risk that performance degradation in the application leads to new false positives. For instance, in our example above that, the state transition time increases from 1.9 to 2.1 seconds. Debroy et al., therefore instead suggests the use of dynamic waits, e.g., waiting for a specific page title, or until the title changes from a particular title. Another option is to wait for GUI widgets to appear, for example, a headline, a text field, or a button.

The synchronization challenge is related to, or partly dependent on, the challenge of robust identification of GUI widgets (C3) since the test execution needs to be synchronized by waiting for correctly identified widgets. Therefore, we pose that these two challenges (C2 and C3) are connected and that a solution to C3 might also provide a possible solution, or mitigation, for the synchronization challenges (C2).

Robust identification of GUI widgets (C3): As stated, this challenge is possibly related, or even the leading cause of many of the other reported challenges. It might be one of the challenges that cause the test execution to fail when the SUT changes (C1). A more robust widget detection would, per definition, be more tolerant and possibly able to prevent unexpected failure. We differentiate between expected and unexpected failures here since some changes to the application are expected to cause failure while other changes are not. For example, a missing widget is expected to fail the test execution, but a minor change in a widget's position should not cause the script to fail. Robust widget identification of this type would possibly also mitigate the synchronization challenge (C2) since dynamic waits rely on waiting for located widgets. The challenges related to a changed screen resolution (C4) might also be reduced since a reliable widget identification technique might be able to find the correct widgets even if the widgets have been rearranged. Thummalapenta et al. [150] give the example that some elements (widgets) might not appear in the same location when the screen resolution changes, thus preventing the test execution from continuing. Even tests that fail due to unknown reasons (C6) might be partly caused by non-robust identification of widgets since failure due to timing and widget identification can be hard to understand and explain. Hence, challenges reported in research may be incorrectly classified even though caused by this underlying challenge.

The correct widget, like an input field or a button, first needs to be identified before verifying some of its properties or triggering an action. Failing to identify the correct widget will most likely result in a failed script. The correct widget is, in this case, defined as the widget that the human tester that created the script or model would select in a given situation. Note that the automated test execution should fail when the correct widget is not present since that would have also prevented the human tester from continuing. Widgets can, however, be located in several ways. Alégroth et al. [24] describe three generations of techniques for automated GUI-based testing: coordinate-based, component/widget-based, and Visual GUI Testing (image recognition). These techniques are fundamentally different and, as such, have various benefits and drawbacks. Research has shown that combinations of these techniques can have positive net effects, which represent one possible solution to the problem [29].

Twelve publications reported challenges with reliable identification of widgets from 2002 to 2019. Two of the papers bring up the challenge of using screen coordinates. Four publications mention the challenge with component/widget-based identification, seven of the publications talk about robustness issues related to image recognition, and one does not go into any technical details. Solv-

ing or significantly reducing the challenges with robust identification of widgets could have a notable impact on the success of the GUI-based test automation field. There have been many attempts to solve or mitigate this challenge over the years. Leotta et al. proposed, in 2015, a new type of locator, named multi-locator, that uses a voting procedure for selecting the best candidate from a set of locators [99]. Multi-locators might be one solution for reducing some of the fragility issues when trying to locate widgets when testing an application through the GUI. Another exciting technique, proposed by Yandrapally et al., for addressing the fragility challenge by improving the reliability of widget localization, is using contextual clues [166]. The idea is to mimic how humans perceive an application by evaluating the target widget and the surrounding widgets. Taking advantage of the surrounding widgets would, in theory, make it possible to locate a widget even in the worst case when all of the locators, associated with a widget, failed by using contextual information about the widget's neighbors. To automatically compensate for frequent changes and be able to identify widgets in a continually changing GUI would perhaps increase the stability of the automated tests and lower the maintenance cost of automated tests. Therefore, we perceive this challenge to be accidental since it should be possible to come up with a technical solution that can identify widgets with an accuracy comparable to human testers. We base this assumption on recent advancements in oracle and GUI locator research, where significant improvements to test execution robustness can be observed [99].

Test automation of dynamic applications (C5): Controlling the dynamic aspects of an application is not only a challenge for GUI-based test automation but rather a challenge shared by the entire field of test automation and all types of applications. Dynamic aspects include, continuously changing databases, dynamic GUI contents, application state transitions, and dependency of external dynamic applications. The challenge is mentioned by eight publications during a four-year-long time-period between 2013 and 2016. Thummalapenta et al. [150] give an example: "if test setup does not remove such entities created during the previous test run, scripts in the new run fail while attempting to create an existing entity". Hence, an additional complexity with testing dynamic applications is that the tests themselves may also be affecting the system state when tests are executed. This implies that tests might need to be executed in a specific order since the state change of previous states set the preconditions for later tests.

Automating a dynamic application is an essential challenge compared to a static application, placing significantly higher demands on the time and resources needed to handle and prepare tests to deal with dynamic data in a

robust, deterministic way. For instance, the application database might need to be set to a specific state before starting the test execution, external systems might need to be mocked, and test cases might need to be executed in a particular order to lay the foundation for stable test automation. While some of these issues might be mitigated through technological advances, the core challenge of non-deterministic SUT behavior will remain, making this a lasting challenge.

Tool challenges (C11): Tool related challenges were reported, for all platforms, by ten publications published between 2012 and 2016. These challenges are however different from each other, ranging from difficult installation [108] [37] or lacking documentation [23] to technical issues like the one mentioned by Alégroth et al. in 2015 [25]: "The main drawbacks are that the tool requires hooks into the AUT's GUI library which means TextTest is currently restricted to Java and Python.".

These challenges range from minor drawbacks to severe maturity or stability issues with the software products. While challenges with unique tools might be accidental and possible to solve, the higher-level concept of "tool challenges", in general, is an essential challenge since it is difficult, or even impossible, to find one technical solution that addresses them all. For example, general solutions how to make the user interface more user-friendly or how to make the application more responsive.

Requires automation or programming skills (C18): Successful test automation requires knowledge, experience and skills to master. Between 2011 and 2019, ten publications mentioned the lack of skills as a challenge in regards to either test automation, test automation tools, or programming when automating an application through the GUI. The need for development skills were further explicitly specified by eight of the included papers. The need for programming skills were highlighted by five papers [108] [136] [37] [55] [57] and two papers stated a need for experience in writing regular expressions [25] or XPath locators [100].

Minor skills in test automation or programming can be sufficient when only creating a few test cases but, as with the case of software development, more skills are needed when creating and maintaining an extensive suite of tests as complexity increases with size. Many tools have targeted this challenge during the years because skilled resources are typically hard to find and are more expensive to hire. Capture/replay (CR), or record/replay, tools is one approach that attempts to simplify and speed up the process of creating and running automated GUI-based test cases. However, as explained by Moreira et al., the main drawback with CR tools is the cost of script maintenance [118]. Previously recorded tests break easily when the GUI changes and this leads to consider-

able manual labor to repair them. According to Moreira et al., this is often the reason the CR methods are abandoned after a few software releases. Another challenge emerges with CR when newly recorded test cases need to be merged with existing test suites since this generally requires programming and skills to deal with growing code complexity. The lack of skill is mostly an accidental challenge. It might be possible to find a technical solution that makes CR tools capable of creating and maintaining stable test suites that can be used by testers with less programming skills than today. One possible solution could be to create and maintain the tests using an Augmented GUI proposed by Nass et al. [119]. Lack of skills could also be seen as an essential challenge since test automation skills will always be required regardless of any technical advances. Hence, even if user-friendly tools can be developed, the purpose of the tools, and their contextual use, must still be taught to the users.

Creating/maintaining model based tests (C19): The time and cost to create and keep the model up to date with a continually changing GUI application resemble the challenge of maintaining the test scripts to prevent application changes from breaking the test execution (C1). This challenge is probably the main reason why the industry has not yet embraced MBT. Memon et al. wrote, in 2007, that models used for automated GUI testing are expensive to create [114] and was confirmed by Aho et al. [19], in 2014, Patel et al. [136], in 2015, and Moira et al. [118], in 2017. That model-based tests also require maintenance effort, to keep the models up to date when the SUT changes, is also mentioned by three publications [76] [80] [62], from 2010 to 2012.

One option for avoiding the time-consuming work of manually creating the models is to generate the models by traversing the SUT automatically. While this approach makes it possible to generate thousands of test cases quickly, they might not be effective or efficient in finding defects or providing adequate coverage of the SUT. The dilemma is especially valid for GUI-testing since it takes many magnitudes more time to execute tests on a GUI-level than on a unit-level. Gupta et al. claimed, in 2011, that model-based approaches mainly focus on test case generation but that it is not effective in real-life industry scenarios [74]. Aho et al. provide supporting statements to Gupta, in 2014, that industry adoption of model-based approaches for GUI-based test automation has been insignificant despite many academic approaches to target the challenge [19].

The challenge reported on all application platforms, associated with the creation/maintenance of model-based tests (C19), is accidental since automated ways of generating models, like GUI crawling, lower model development cost. It should be possible to find new or improve existing technical solutions that target this challenge through further research. Instead, especially for GUI-based

test automation, the underlying challenge is that the industry can not afford to maintain and run several test technologies at the same time, e.g., script-based and model-based tests. Thus, although the technologies have different benefits and drawbacks, script-based testing is considered less costly and therefore comes up on top. As such, there is still a need to find solutions for efficient modeling of GUI applications to make MBT attractive for the industry.

State space explosion during MBT (C20): There are close to infinite paths or scenarios through a typical GUI application. This presents a core challenge for model-based testing in terms of possible state-space to explore during testing (also known as state-space explosion). Memon et al. [115] reported in 2001 that the space of possible interactions with a GUI is high, and twelve other publications have provided supporting observations of the challenge for both desktop and web applications from 2006 to 2018.

As such, the state-space explosion challenge is essential. Any technical mitigation attempt to it must be able to navigate the state-space in such a manner that it only finds scenarios that within the state-space that, for instance, provide the best test coverage of the SUT or the best defect finding ability. However, these solutions are workarounds to the problem since they still do not cover the entire state-space, which, in theory, could be infinite and, therefore, never completely covered. Mariani et al. proposed a machine-learning based solution, called Augusto, that can efficiently and effectively test application-independent functionalities (AIFs) while the rest of the SUT needs to be covered using existing approaches [111]. Examples of AIFs are common user interface patterns, like authentication operations and CRUD (Create, Read, Update, Delete) operations. While it is still too early to know if the approach, proposed by Mariani et al., works for the industry, it implies that machines can learn from human behavior how to efficiently and effectively navigate through GUI applications and thus avoid the essential state space explosion challenge. However, as the challenge is considered essential, we perceive that even advanced solutions will only avoid the challenge, never solve it entirely.

2.4 Discussion

While it might be tempting to find a solution to a challenge, we should first determine if the challenge is possible to solve or if the challenge is inherent. Figure 2.4 contains the key challenges identified in our study related to GUI-based test automation, arranged from essential to accidental difficulties. Challenges are considered as essential if they are inherent to a specific technology

or approach and as accidental when it is possible to eliminate the challenge through technology or improved ways of working [43]. However, as discussed in the synthesis of the results, the view of the challenge as accidental or essential also relates to from what perspective we view the challenge. Therefore, some challenges are arranged in Figure 2.4 in between essential and accidental, for example, tool challenges.

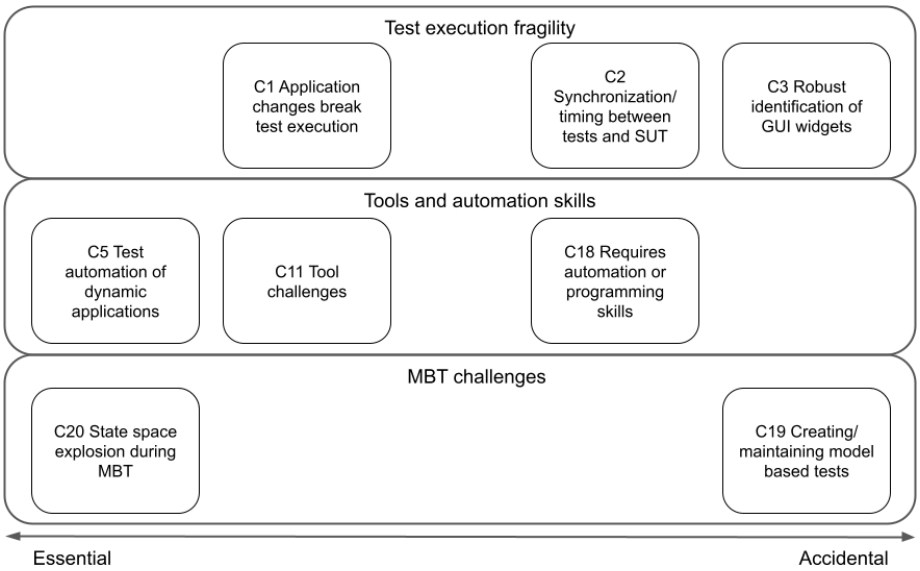

Figure 2.4: The key challenges related to GUI-based test automation arranged from essential to accidental difficulties.

In the figure, we also attempt to cluster the challenges into three different groups related to (1) test execution fragility, (2) tools and automation skills, and (3) MBT challenges. This grouping is based on the semantic similarities between the challenges of each group, as presented in the literature, e.g., in what papers the challenges are presented. In the following sections, we discuss these groups in more detail.

Test execution fragility: Three of the challenges have been grouped together as they are related to the fragility of test execution. As discussed, they may be all caused by the challenge to reliably identify widgets when running automated tests, i.e. (C3). There have been, at least, nine academic papers

that attempt to solve or mitigate the challenge with robust widget identification in the past two decades [166] [99] [21] [102] [60] [49] [150] [100] [172], and publications still report the same challenge. Despite these efforts, the challenges remain, possibly because the academic solutions have been insufficient in solving the challenges faced by the industry. Another possible explanation is that academia has found possible solutions but not communicated these in a suitable way to industry. In the latter case, academia might have failed in transferring the knowledge to the industry, or the industry has not yet been able to incorporate the solution into their products or processes.

Tools and automation skills: Three of the challenges form a group related to test automation tools or skills required for efficient/effective test automation or programming (C5, C11, and C18). The challenge that testers need automation and programming skills to succeed with GUI-based testing (C18) is the only challenge, in this group, that we defined as mainly accidental and, therefore, the challenge that should be devoted to the most research efforts in the future. The current solution to the challenge is training the team members or by hiring someone with the required skills. However, people with in-depth knowledge of test automation and programming are difficult to find, and training is both time-consuming and costly. Therefore, the industry would benefit from a GUI-based test automation approach that does not require the same amount of skills as needed today.

CR tools was a good attempt for such an approach since it allowed anyone with domain knowledge to record valid test scripts. However, the challenge with CR tools is that the recorded scripts are fragile and need programming skills to be enhanced or merged into reliable test suites. Also, CR tools can only record new test scripts and require programming or scripting knowledge when maintaining existing test scripts at a high cost. This first challenge, related to the script fragility, can be partly mitigated with a more robust identification of GUI widgets (C3). The remaining challenges could be addressed by a possible future solution for reducing the need for automation and programming skills. It could enhance CR tools with functionality, methods, or processes that can aid the manual tester when merging and maintaining the recorded tests.

Aho et al. divide GUI-based test automation into three levels: Script-based, Model-based, and Scriptless [20]. According to Aho et al., the benefits of scriptless testing are that the initial investment and maintenance effort is low compared to the other approaches. Still, a challenge is effective action selection since actions are typically selected randomly. Combining scriptless with the recording functionality from CR tools would perhaps be an interesting approach to investigate in further research since recorded actions from real test scenarios could

provide useful hints when selecting actions instead of picking actions randomly. Hence, a more machine-learning based approach based on models of the system to be tested, recorded from human users.

MBT challenges: The final group is about challenges with model-based testing. Since the state-space explosion challenge (C20) is essential, research should concentrate on possible solutions for creating and maintaining model-based tests (C19). Many of the reported MBT challenges are caused by the large number of possible ways to traverse a GUI application and to be able to extract the paths that provide the best test coverage and the highest chance of finding a defect. Instead of trying to extract high-quality test scenarios by randomly traversing the application, the solution might instead be similar to the one suggested by Mariani et al., that we need high-level guidance from typical patterns of working with GUI applications [111]. Perhaps it is even possible to create the patterns, or the models themselves, by recording or observing actual end-users or testers of the SUT. Recording the models from the users would be a feasible solution to handle both the state space challenge and the high cost of creating the models. Theoretically, these models would include meta information of what the primary states of the application under test are, assuming that they are covered by the human that recorded them. This information would provide guidance for heuristics of how to generate suitable test cases, which could trump existing random approaches.

The challenge that models need maintenance when the SUT changes (C19) are very similar to the challenge that application changes break the test execution (C1), which in turn depends a lot on robust identification of GUI widgets (C3). Finding a solution to C3 might, consequently, have a positive impact on C19.

Summary: Synthesis of the acquired results, and grouping of the identified challenges, reveal some challenges to be accidental (solvable) and some essential (only possible to mitigate or solvable in specific contexts). Additionally, many challenges share the common issue of a lack of robust identification of GUI widgets, implying that this is a central and critical challenge to solve.

Furthermore, the grouping of essential and accidental challenges helps us explain why certain challenges are chronologically reoccurring, and cannot be solved, in research on different test techniques and platforms. This conclusion provides interesting insights for both industry and academia. It should influence our mindset regarding what challenges we target and how we reason about the proposed solution- or mitigation strategies in the future.

2.5 Threats to Validity

This section covers some of the threats to the validity of this literature review.

Construct validity: The selection of databases and construction of the query string for the literature search has a significant impact on the publications included in this study and is a threat to be considered. We divided the query string into several parts and tested each part separately to avoid missing any relevant publications that contained benefits or challenges about GUI-based test automation. A search tool was created to perform full-text searches inside the downloaded publications to make sure that they contained all the keyword combinations. This was complemented by manual review to ensure the inclusion of semantically suitable papers. However, four publications were included during the snowballing procedure after checking the inclusion/exclusion criteria and suggests that our search query string was not perfect. However, we decided that it was good enough to continue with the study since we argue that a few missed publications will likely not have a significant impact on the results. We argue that in the worst case, we may have missed some challenge of importance, but given the authors' extensive empirical understanding of the area, we perceive it as unlikely. However, it cannot be ruled out since the challenges were identified and formulated by the leading researcher based on the statements extracted from the included publications and is therefore affected by the leading researcher's knowledge and perception. However, the acquired challenges were reviewed by the other authors, and no explicit challenges have been observed as left out.

Internal validity: The extensive industrial experience held by the authors of this study is a threat to internal validity. Bias from industrial experience can undoubtedly affect the synthesis and coding of challenges, even though we were aware of this challenge and actively tried to make unbiased judgments. There is also a risk that the challenges extracted from the literature are biased or that our interpretation of them are affected by our own experience from the field and that it might have an impact on the synthesis.

External validity: We decided to exclude publications that did not contain any empirical evidence to try to reduce the positive-results bias, a type of publication bias that occurs when the authors are more likely to submit positive results than negative or inconclusive results [144].

Also, a well-defined methodology, like a systematic literature review in our case, makes it less likely that the literature results are biased even though it does not protect against publication bias in the primary studies, according to Kitchenham et al. [87]. To eliminate possible bias from our own analysis, the results and conclusions were verified within the research group through

continuous discussion throughout the study process. Although not eliminating analysis bias, this approach mitigates the personal bias of one or several of the authors and thereby improves the validity of the results.

Reliability: Data analysis and coding of challenges are affected by the author's experience and knowledge in the GUI-based test automation field. They are a threat to the reliability and reproducibility of this literature review. Repeating this literature review would likely result in slightly different descriptions of the challenges, but we consider these potential deviations small and not significantly affect our results. Even though our definitions of essential and accidental challenges were based on the definition by Brooks et al., we decided to limit the definition of accidental challenges as something that can be mitigated using a technical solution to get a clear definition and match the scope of this SLR. Our redefinition and interpretation of essential and accidental challenges impact the classification of the reported challenges. However, we note that it does not affect essential challenges as these, per definition, cannot be completely solved with technical solutions, or otherwise.

2.6 Conclusions

Test automation through the GUI is still a challenge despite many emerging tools and technologies during the last two decades. This SLR pinpoints 24 challenges of GUI-based test automation synthesized from 49 publications that build on empirical evidence. Eight key challenges were mapped on a timeline to determine how the challenges have evolved and if there have been attempts to mitigate them.

A novel contribution of this work is that all identified key challenges have been classified as essential or accidental in an attempt to provide guidance for future research and to avoid time expenditure on solutions for inherent challenges that cannot be solved entirely. Instead, we urge a focus on the challenges of a more accidental nature that we can solve by further research. Additionally, we urge a change in mindset, with this new understanding in mind, to approach proposed solutions or mitigation strategies in the future.

We might never find any solutions to four of the key challenges since they are essential. The remaining four are accidental challenges that are possible to mitigate and therefore deserving further attention. The challenge of robust identification of GUI widgets (C3) could be the root cause of the synchronization challenge (C2) and many of the other challenges reported. A solution, to C3, could make the test execution less fragile and reduce the maintenance time and

cost of both test scripts and models. Both the challenge that GUI-based test automation requires automation or programming skills (C18) and the challenge of creating/maintaining model-based tests (C19) could be targeted by a tool, method, or process that provide scriptless creation and maintenance of the tests instead of recording and maintaining the tests by coding or scripting like a conventional CR tool. A scriptless approach could provide a more efficient and effective way of creating and maintaining automated GUI tests without extensive programming skills.

Future work should be conducted to find solutions that enhance script execution robustness (C2 and C3), reduces the skills required for successful GUI-based test automation (C18), and improves the efficiency and effectiveness of creating/maintaining model-based tests (C19).

2.7 Acknowledgements

We would like to acknowledge that this work was supported by the KKS foundation through the S.E.R.T. Research Profile project at Blekinge Institute of Technology.

Chapter 3

Augmented Testing: Industry Feedback To Shape a New Testing Technology

Abstract

Manual testing is the most commonly used approach in the industry today for acceptance- and system-testing of software applications. Test automation has been suggested to address drawbacks with manual testing but both test automation and manual testing have several challenges that limit their return of investment for system- and acceptance-test automation. Hence, there is still an industrial need for another approach to testing that can mitigate the challenges associated with system- and acceptance-testing and make it more efficient and cost effective for the industry.

In this paper we present a novel technique we refer to as Augmented Testing (AT). AT is defined as testing through a visual layer between the tester and the System Under Test (SUT) that superimposes information on top of the GUI.

We created a prototype for AT and performed an industrial workshop study with 10 software developers to get their perceived benefits and drawbacks of AT. The benefits and drawbacks will be useful for further development of the

technique and prototype for AT. The workshop study identified more benefits than drawbacks with AT. Two of the identified benefits were: "Know what to test and what has been tested" and "Less manual work".

Due to these results, we believe that AT is a promising technique that deserves more research since it may provide industry with new benefits that current techniques lack.

Keywords: System Testing, Test Automation, Industrial Workshop Study, Augmented Testing

3.1 Introduction

Manual testing is the most commonly used approach in the industry today for acceptance- and system-testing of software applications. Tests are conducted by the human by communicating through the Human Machine Interface (HMI) or Graphical User Interface (GUI) of the System Under Test (SUT). Manual tests are however time consuming, repetitive and error-prone to perform for a human tester [71, 73]. Test automation, using techniques like unit testing [131] and record-replay [17], has been suggested as a solution to the drawbacks found in manual testing. These techniques also include tools that use image recognition to emulate manual testing [23, 47].

However, all the proposed techniques, especially on the GUI level, have several reported challenges and drawbacks that limit their return of investment for system- and acceptance-test automation.

One common drawback reported, in an industrial case study performed, by Thummalapenta at al. [150] is that the widget recognition failed because of the high degree of dynamism in the application GUI. Another drawback reported by Thummalapenta at al. is that the GUI is affected by changes in the server-side data and gives the example: "there is no guarantee that the book will appear in the second position during playback as well".

Hence, there is still a need for research on efficient and effective testing to mitigate the challenges of both manual- and automated-GUI testing.

In this paper we present a novel technique we refer to as Augmented Testing (AT). AT can be explained as a technique that realizes the concept of human-machine symbiosis [68] for software testing through the GUI. AT is defined as testing through a visual layer between the tester and the SUT that superimposes information on top of the GUI. The superimposed information can be anything from actions to perform, results to check, identified issues, comments, statistics etc. An example of how the augmented information can look like is illustrated

in figure 3.4. The visual layer, hereby called the Augmentation Layer (AL), relays all information, visual and non-visual, back and forth between the tester and the SUT.

We believe that AT can have a positive impact on the efficiency of testing applications through the user interface since the AL enables a faster and more intuitive communication interface between the human and the machine.

One of the drawbacks of this technique is additional overhead since the AL is placed in between the tester and the SUT and drains system resources. Another drawback is that the interaction with the AL might be slightly different than when using the actual SUT GUI. This drawback might have a negative impact on the usability of the SUT.

A common approach to computer science and software engineering research is to first propose and refine new technology and then later evaluate its effects in industry. Here we aim to involve industry throughout the development of augmented testing and to iterate and refine it based on industry feedback. We thus implemented a prototype and carried out an industrial workshop with 10 participants to collect their perceptions of the technique.

The specific contributions of this paper are:

- A presentation of a novel technique we refer to as Augmented Testing (AT) and its realization in a prototype.
- The collection of perceived benefits and drawbacks of AT from an industrial workshop study that can be used for further development of the technique and prototype.

This paper is structured as follows. In Section 3.2 the theory behind AT and the basic features of the prototype will be described. Section 3.3 presents work related to the AT technique. Section 3.4 describes the method of the industrial workshop study were the participants evaluated our prototype for AT. The results from that study can be found in Section 3.5. We will discuss the results in section 3.6 and give our conclusions in section 3.7. Finally some work to consider for the future is presented in section 3.8.

3.2 Background

The workflow of Augmented Testing (AT) is illustrated in figure 3.1. The human-machine interactions can be divided into four steps:

1. The machine retrieves the GUI bitmap and widget information, if available, from the SUT.
2. The Augmented Layer (AL) is displayed for the human and consists of the retrieved GUI bitmap augmented with actions, checks, defects, suggestions etc. from the machine.
3. The human interacts with the AL through the augmented actions, checks, defects, suggestions etc.
4. The machine receives the human interaction and updates the model. The human interaction is also relayed to the SUT normally using mouse or keyboard events but may also interact with widgets directly or indirectly using an Application Programming Interface (API).

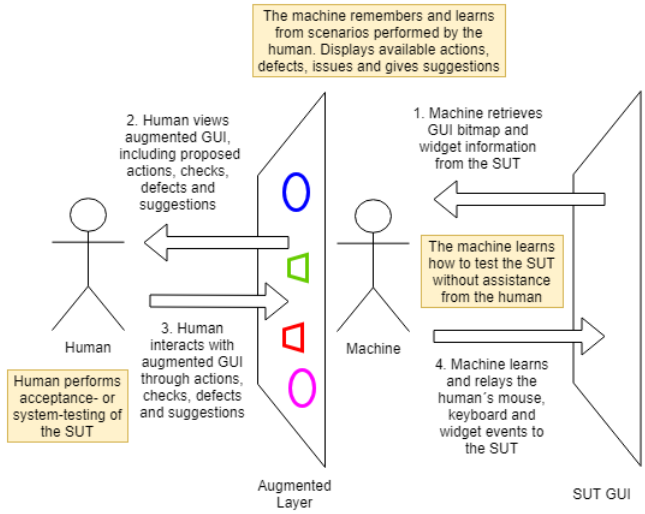

Figure 3.1: Workflow of Augmented Testing

A prototype for AT was created in Java. A screenshot of the prototype can be found in figure 3.4. In this example we are testing the Wikipedia [14] website and the screenshot is taken during an ongoing test session.

A typical test session with the prototype starts by entering or updating a few settings like the name of the product, the product version, the name of the

tester, the type of application (desktop or web), the type of browser for web applications and the Home Locator. The Home Locator is a URL for a website or the path to an executable file for a desktop application that is launched when the session is initiated. A screenshot of the start session dialog can be found in figure 3.2.

Figure 3.2: Screenshot of the start session dialog

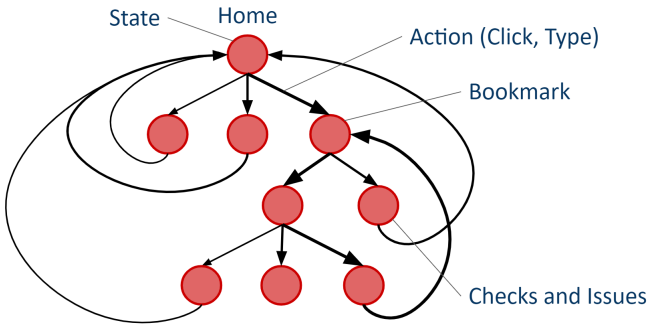

Figure 3.3: A state model tree

All actions performed by the tester, like mouse clicks and keyboard input, are recorded during the test session. Previously recorded actions are augmented using a blue circle on the AL. Hovering over the augmented action will reveal more information, like the type of action and coverage. The recommended action is larger than the other actions. Suggestions from the machine are indicated using purple circles. The suggestions include scenarios to try in order to improve the test coverage or boundary values to enter. Results to check are indicated

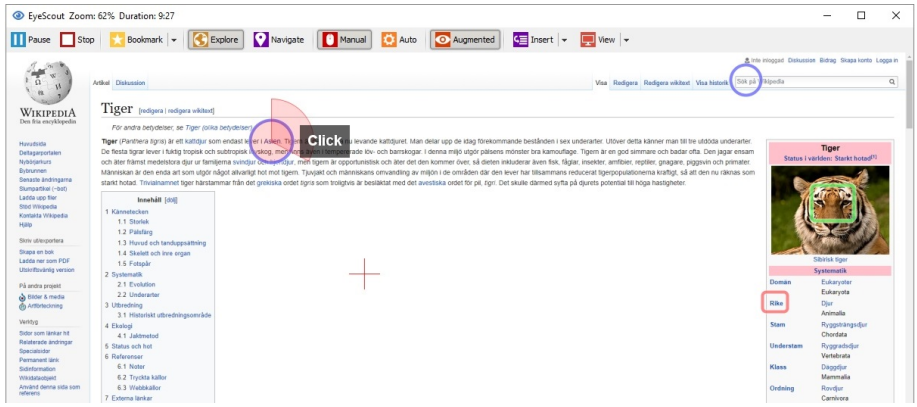

Figure 3.4: Screenshot of the prototype for Augmented Testing

by a green rectangle in the prototype. Checks may use image recognition [126] or compare properties retrieved from GUI widgets [99, 101]. Valid results are indicated with a green rectangle and invalid checks are indicated with a yellow rectangle since they are either still valid or invalid. Checks that have not yet been evaluated are indicated with a gray rectangle. Hovering over a check will reveal more detailed information. Issues are indicated with a red rectangle. The issues that cannot be confirmed are indicated by a yellow rectangle since they can either be fixed or still be an issue. Non-evaluated issues are indicated with a gray rectangle.

The tester may either select a previously recorded action displayed in the AL or perform a new action, like a mouse click, double-click, mouse scroll or enter a value using the keyboard. Performing a previously recorded action might reveal a new set of actions to select from while performing a new action will not display any actions to select from since that path has not yet been recorded.

The application state is important in AT since a given sequence of actions must give the same predictable output that can be checked by the tool. An application state is the state of an application and all it's dependencies like internal memory, stored data and external applications at a given time. Figure 4.2 shows an example of a state model tree. The home state is the state of the application when the session was started. A bookmark is a known application state that can be useful, similar to a bookmark in a web-browser. Bookmarks can be added at any time during a test session. The tester may at any time select to go back to the home state, or a bookmark, to begin a new path through the SUT. A check

is a result that needs to be verified, by the human or the machine, like a value or an image. Issues are things that should be corrected. Comments are notes that makes the test scenario easier to understand. Checks, issues or comments may be added, by the tester, at any time during a session.

All actions are recorded into the prototypes state model tree, which means that new actions create new branches within the state model tree that upon replay of the test case will become available to the tester. The state model tree has many things in common with a weighted neural network and is used to identify patterns on input data to enable the prototype to give suggestions to the user. Actions with a stronger connection weight are more common paths to take when transitioning from one state to another. Stronger connections are visualized by a thicker arrow in the state model tree in figure 4.2.

The tester has the option to select the Auto button to go from manual mode into auto mode. In auto mode, the prototype will automatically click on the recommended action until done. The tool will automatically go back into manual mode if it finds a check or issue that is different than recorded. A tester may also select the Navigate button to navigate to a previously reported issue or bookmark. The prototype will then highlight the next action to select until the target application state is reached. During auto mode, the tool will auto click on the highlighted actions. The tester can stop the test session at any time by pressing the Stop button.

3.3 Related Work

Mariani et al. [109] presented a technique and a tool, called AutoBlackTest, for automatically generating system test cases for interactive applications. The goal is to generate test cases that exercise the functionalities of the application by navigating its GUI. Aho at al. [19] introduces a process of using Murphy tools to extract models of GUI applications and utilizing the extracted models to support various GUI testing activities, and share their experiences of using that approach in industrial development and testing environments. Pham et al. proposed concepts for linking debugging information with a video/screen-recording of an application. They claim that it is a useful concept for debugging the application under test, that it helped in debugging failures and that it can save a lot of time in the testing and debugging phase of a software development process. Amalfitano et al. [33] presents a technique for detecting, using machine learning, and unlocking something they call a Gate GUI. A Gate GUI is, according to Amalfitano et al., a graphical user interface that prevents a random exploration

technique from advancing to relevant parts of the SUT. A typical example of a Gate GUI is a login dialog where a valid user name and password have to be entered before allowing access to other features of the application.

We use a combination of all of the techniques mentioned above in the prototype for AT. The prototype records and models valid application scenarios, by observing the end-user, like the tools used by Mariani et al. and Aho at al. The SUT can be recorded very accurately since all interactions go through the AL and never directly to the SUT. Information from the current state is also merged visually with a screenshot from the SUT similar to the concept described by Pham et al. to produce an image that contains the screenshot augmented with information from the current state, like actions, checks, issues, comments etc. Gate GUIs, investigated by Amalfitano et al., can be handled by the prototype without any manual coding or modeling since we know the application state, and all application views, of the SUT and have recorded valid application scenarios that are required to transition between them.

3.4 Industrial Workshop Study

This study was performed at a Swedish bank and finance company, hence forth referred to as "the bank". This study was conducted at one of the banks mobile app development locations, the research site, where 15 developers work.

3.4.1 Company Description

The bank is one of the largest based in Sweden, with almost 8 million customers in 2018, mainly from the northern part of Europe. The bank has more than 14 000 employees and has collaboration with about 60 local banks. The IT organization is a cross-border company with 1500 IT employees spread across all home markets, located in several Swedish cities but also several other cities in northern Europe. The IT organization is responsible for IT management in ensuring the long-term focus of IT and adaptation to the bank's business strategies. The IT organization deliver a range of IT services to the bank, including development and maintenance.

We performed a workshop study with 10 software developers from a team located at the research site. The software developers creates mobile applications, both iOS and Android, used by the customers of the bank. The developers collaborate with other teams in Sweden that provide them with requirements, UX-design and back-end functionality etc.

3.4.2 Selecting an Application to Test

For the workshop we wanted to select an application to test that the developers found relevant and familiar. The most relevant choice would have been to test the mobile applications that they are developing. We could however not use a mobile application since the prototype only supports desktop- and web-applications, but not mobile-applications. Instead, we decided that the developers should create test cases on their own banks public website since that would be familiar to them. However, we did not want to affect the participants choice of test cases too much, so we decided to give the introduction and demonstration on another website, namely a competitor bank's website. We selected this website since it uses a familiar vocabulary and contained similar functionality, but did not use the same structure as their own banks public website.

3.4.3 Workshop Study

The study consisted of two workshops, guided by the research questions:

- RQ1: What are the perceived benefits in industry of AT as demonstrated by the prototype?
- RQ2: What are the perceived drawbacks in industry of AT as demonstrated by the prototype?
- RQ3: What additional benefits can industrial practitioners perceive with AT as a technique?

Both workshops were performed in the same way and lasted for two hours. Each workshop was divided into four parts:

1. Introduction and demonstration of a prototype tool for AT during 30 minutes. We used the public website of a competitor bank as an example application to test during the introduction and demonstration. The demonstration covered how to start a new test session, how to create actions by using the mouse and keyboard and how to stop the session. The participants also got instructions of how to add checks, issues and how to return back to the home state. We also demonstrated the auto test feature and how to navigate to a previously reported issue.

2. The participants got time to evaluate the tool freely for 30 minutes, but this time using their own banks public website as an example application to test. Only one of the developers could use the prototype at a time but they were allowed to talk, collaborate and switch users at any time during the, 30 minute, evaluation period. They were instructed to remember any observed benefits or drawbacks they saw during the evaluation period but not to disclose or discuss them with the other participants.

3. Each participant was instructed to write down all the observed benefits and drawbacks of the prototype tool or technique for AT. They were instructed not to discuss the benefits and drawbacks with the other participants and were also instructed not to write down any minor defects directly related to shortcomings of the prototype since the main objective is to get feedback on the technique of AT rather than minor problems with the prototype. This part was limited to 30 minutes.

4. The participants were finally instructed to try to imagine a state-of-the-art tool, based on the same technique as the prototype, created by a well known software developer. They were also instructed to stick to technology that existed today in order to avoid speculations about future technologies such as general artificial intelligence. The purpose was to find yet unknown or possible benefits of the technique for AT that might be implemented in future interactions of the prototype. Each participant was then instructed to write down any additional benefits that a user could get when using such a tool. They were also instructed to clearly separate the additional benefits from the benefits they noted in part 3. This part was limited to 30 minutes.

The first workshop had 4 participants and the second workshop was attended by 6 participants. It would have been better to have the same number of participants in both workshops since that would have given the participants the same amount of time per person of evaluating the prototype. However, we overlooked this potential threat since we deemed it more important to let the developers select the workshop that best suited their work schedule to get as many participants as possible.

We decided to begin the workshop with an introduction of AT and the prototype even though we also introduced bias about the presented features of the technique and prototype. To minimize the introduced bias, we demonstrated the prototype and the theoretical concepts that we believed was required to be able to create a few test cases without emphasizing one feature or concept over

another. An alternative workshop design would have been to let the developers figure out how to use the prototype without instructions but that option was discarded since AT is a new concept that would have taken a long time to understand without some kind of introduction.

3.5 Results

The coded and mapped perceived benefits (RQ1) from the two workshops are illustrated in figure 3.5 and figure 3.6. Benefits from a hypothetical state-of-the-art implementation of AT (RQ3) is marked with an asterisk (*). The numbers in the rightmost box are references to the stated benefits or drawbacks from the 10 participants, which can be found in Tables 3.1 and 3.2. The coded and mapped drawbacks (RQ2) from the two workshops are illustrated in figure 3.7.

Tables 3.1 and 3.2 contains all the collected comments about perceived benefits and drawbacks of AT from all of the 10 participants. Comments given in the Swedish language were interpreted into English. Additionally, as can be seen in the third column of Table 1, the comments have all been mapped to codes that aim to group the comments together into higher-level concepts. Benefits from a hypothetical state-of-the-art implementation of AT is marked with an asterisk (*).

3.5.1 Threats to Validity

A threat to the external validity of the study's results is that all participants in the study were mobile app developers. We would likely have got a different result if the participants belonged to a different role, like testers or requirement engineers, or a combination of all. Another threat to the external validity is that we only performed one study at one location with one company. Additional results from several companies, and especially from different types of companies that does not belong to the bank sector, would have provided us with a more valid result.

Since our prototype for AT did not support mobile applications we also have a treat to the construct validity. All the participants were mobile app developers and would probably have responded differently when testing a mobile app as an example instead of a website.

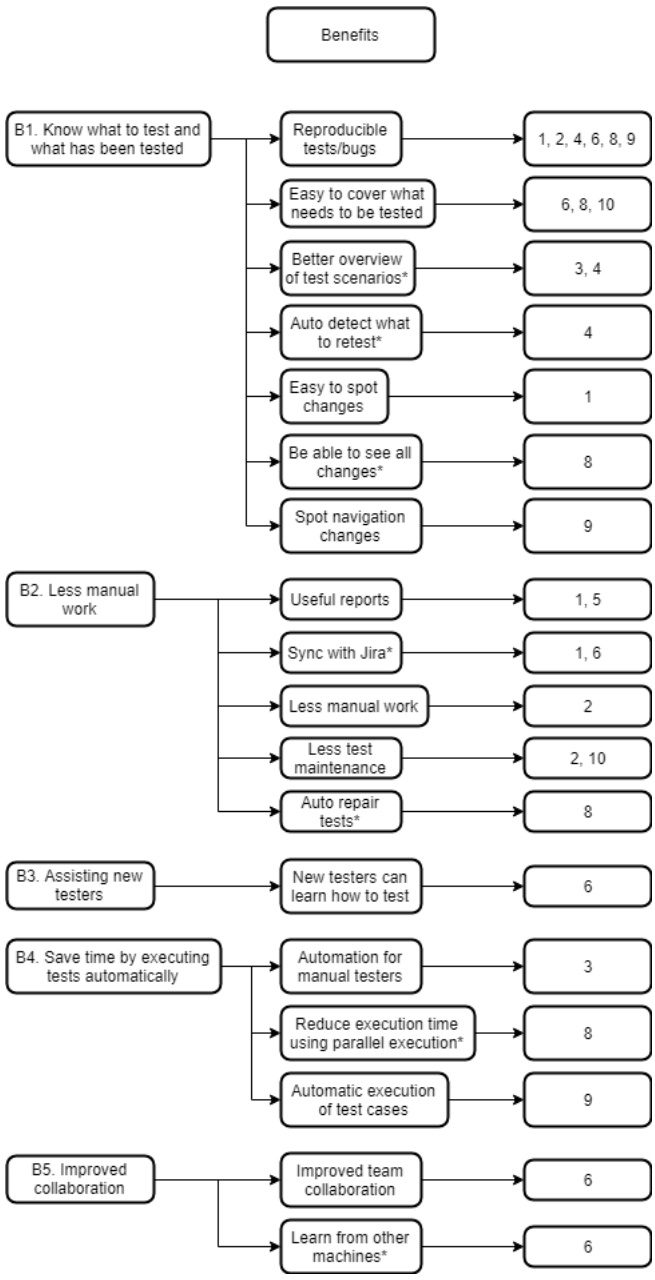

Figure 3.5: Code mapping of perceived benefits

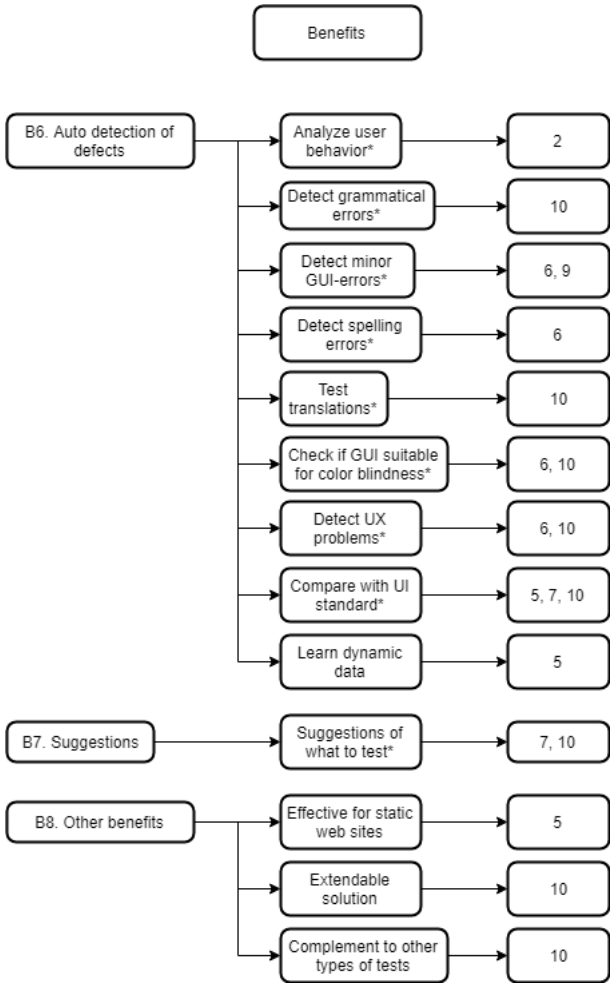

Figure 3.6: Code mapping of perceived benefits

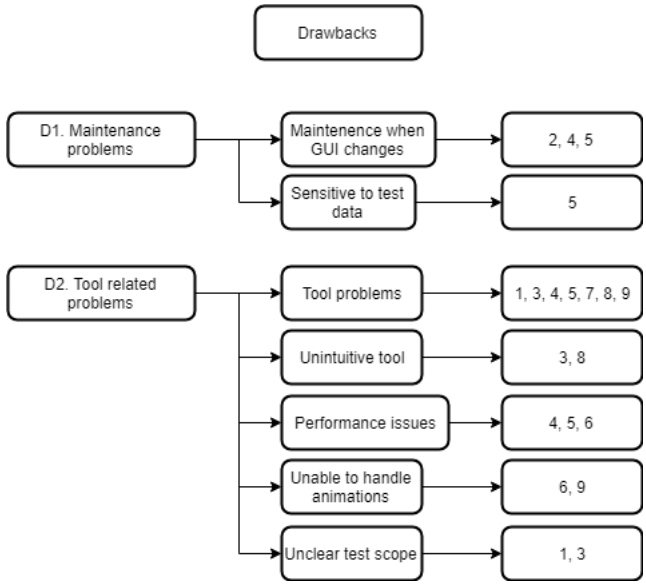

Figure 3.7: Code mapping of perceived drawbacks

Table 3.1: Identified perceived benefits and drawbacks

Part:	Benefit or drawback:	Mapped to code:
1	"Consistent tests" and "Reproducible"	Reproducible tests/bugs
1	"Easy to spot things that changed"	Easy to spot changes
1	"Pretty reports (dashboard, coverage mm)"	Useful reports
1	"Jira support"*	Sync with Jira*
1	"Collaboration between testers and developers"	Improved team collaboration
1	"Bigger checks (larger than 400x200 pixels), a bug?"	Tool problems
1	"Scope of test unclear, unclear what we are testing, only do relevant tests."	Unclear test scope
2	"Better feedback on how to reproduce a bug."	Reproducible tests/bugs
2	"Less manual work."	Less manual work
2	"Lower problems with maintenance."*	Less test maintenance
2	"Be able to analyze user behavior."*	Analyze user behavior*
2	"A lot of maintenance of tests when the GUI changes."	Maintenance when GUI changes
2	"Should not test the application through the GUI since our tests change too often."	Maintenance when GUI changes
3	"Better overview of test scenarios."*	Better overview of test scenarios*
3	"Automated tests for manual testers (low technical)."	Automation for manual testers
3	"Scroll does not work.", "User feedback after a click." and "Easy to click in the wrong place."	Tool problems
3	"Hard to know when to use auto or semi auto."	Non-intuitive tool
3	"Trace-ability of test and feedback." and "Need a better overview of test flows and checks."	Unclear test scope
4	"Testers can create repeatable test cases."	Reproducible tests/bugs
4	"Convenient reporting and overview of available tests."*	Better overview of test scenarios*
4	"Auto detect what parts to retest in a new version."*	Auto detect what to retest*
4	"Testers can share reproducible defects with developers and others."	Improved team collaboration
4	"Recorded flows can easily become fragile (break in new versions)."	Maintenance when GUI changes
4	"The coverage term is confusing (perhaps use touched instead)."	Tool problems
4	"Hard to create tests with good performance (executes slowly)."	Performance issues
5	"With AI be able to generate specific and grouped change reports."*	Useful reports
5	"Be able to analyze the state of UI widgets from a component library to detect bugs in the layout."*	Compare with UI standard*
5	"With AI be able to learn to differentiate between dynamic data and other changes."	Learn dynamic data
5	"Good and effective for static web-sites and applications."	Effective for static web sites
5	"A lot of work with maintaining tests in a "living" program/app."	Maintenance when GUI changes
5	"Sensitive to changes in test data, that changes often in test environments."	Sensitive to test data
5	"Window in window, better with mouse control + screen-recording?"	Tool problems
5	"Difficult to see a good solution/implementation for app (spec iOS)."	Tool problems
5	"Heavy on performance."	Performance issues
6	"Nice to have the test cases in the same tool as they are executed by."	Reproducible tests/bugs
6	"Feels like a good way to make sure that we cover everything that needs to be tested."	Easy to cover what needs to be tested
6	"Fast to learn existing tests for new testers."	New testers can learn how to test
6	"Connection with tools like Jira."*	Sync with Jira*

Table 3.2: Identified perceived benefits and drawbacks

Part:	Benefit or drawback:	Mapped to code:
6	"Detect common UI-errors, like misaligned widgets or too small text size."*	Detect minor GUI-errors*
6	"Connection with Invision for retrieving wire-frames."*	Detect minor GUI-errors*
6	"Possible to detect spelling errors or incorrect sen tenses."*	Detect spelling errors*
6	"Detect color schemes unsuitable for color blind people."*	Check if GUI suitable for color blindness*
6	"Detect too long scenarios (too many clicks - bad UX)."*	Detect UX problems*
6	"Needs to have better performance than the prototype."	Performance issues
6	"Unable to handle animations."	Unable to handle animations
7	"Show widget sizes to compare with desired measurements."*	Compare with UI standard*
7	"Get suggestions on widgets to test and list options to select from."*	Suggestions of what to test*
7	"Actions should be squares instead of rings."	Tool problems
7	"Action rings could block other elements."	Tool problems
8	"Can test everything in a flow."	Reproducible tests/bugs
8	"Good documentation of how to navigate regardless of resolution etc."	Easy to cover what needs to be tested
8	"Be able to show changes on all widgets regardless if you have selected to test them."*	Be able to see all changes*
8	"Be able to recognize flows and "repair" even if there are big changes in the scenario."*	Auto repair tests*
8	"Be able to run parallel flows to save time."*	Reduce time using parallel execution*
8	"Be able to 'learn' new flows automatically on other units."*	Learn from other machines*
8	"No feedback after performing an action."	Tool problems
8	"No way of automatically executing the test again without updating the product version."	Tool problems
8	"Cannot select parameters for test."	Tool problems
8	"No documentation (like an info dialog) that describes the changes between new product versions."	Non-intuitive tool
9	"To catch bugs in regression testing."	Reproducible tests/bugs
9	""Detects changes in navigation behaviour."	Spot navigation changes
9	"Can perform visual repetitive tasks/checks."	Automatic execution of test cases
9	"Can detect visual changes better than a human."	Detect minor GUI-errors*
9	"Large changes in a higher application state might result in a removed sub-tree."	Tool problems
9	"Cannot check animations."	Unable to handle animations
10	"Reduces the risk of missing something for repetitive tests."	Easy to cover what needs to be tested
10	"Requires less maintenance for smaller changes in the GUI that traditional testing."	Less test maintenance
10	"Detects flows after larger GUI changes."*	Less test maintenance
10	"Detect grammatical errors."*	Detect grammatical errors*
10	"Test translations."*	Test translations*
10	"Test for color-blindness."*	Check if GUI suitable for color blindness*
10	"Can give general feedback about the UX or UI, like a non-intuitive GUI."*	Detect UX problems*
10	"Sanity checks based an a library of valid GUI widgets."*	Compare with UI standard*
10	"Gives suggestion of what to check, like an invalid image."*	Suggestions of what to test*
10	"Plugins makes it possible to tailor the solution."	Extendable solution
10	"A good complement to other types of tests."	Complement to other types of tests

3.6 Discussion

Rafi et al. [138] performed a literature review, that identified the benefits and limitations of automated software testing, for example: ”Difficulty in maintenance of test automation”, ”Automation can not replace manual testing” and ”False expectations” and included results from both experience reports and empirical evidence.

We compared our results from the workshop study with the results from the literature study by Rafi et al. and identified a few benefits and drawbacks that we could find in both studies, like "B4. Save time by executing test automatically", "B2. Less manual work" and "D1. Maintenance problems" in our study maps well to "B3. Reduced testing time", "B7. Less human effort" and "L3. Difficulty in maintenance of test automation" from the study by Rafi et al. This suggests that AT has benefits and drawbacks in common with test automation.

More interesting is that we find a number of benefits in the study about AT that are not presented in the study by Rafi et al. These benefits were: "B1. Know what to test and what has been tested", "B3. Assisting new testers", "B5. Improved collaboration" and "B6. Auto detection of defects". This result implies that AT can provide new benefits to the field of software testing. Alternatively, the discrepancies between our result and the results found by Rafi et al. can be explained by the closer human-machine interaction that AT enables. This close interaction allows the user to apply cognitive functions on top of the automated scenarios, which effectively augments them to potentially make them more efficient and effective than manual or automated testing by themselves.

The identified perceived drawbacks (D1 and D2) of AT relate to both the tool prototype and the technique. Since the prototype is in a very early stage of development, far from being a full featured product, drawbacks like the ones reported in "D2. Tool related problems" are expected but should be possible to resolve in future iterations of the prototype. The drawbacks reported regarding maintenance problems (D1) are known problems in the test automation community and was also identified as a limitation (L3) in the study by Rafi et al. Tools based on the AT technique will likely also suffer, to some extent, from this problem since they rely on the same under-laying technologies as current tools for test automation, for example widget identification, image recognition, etc. However, to what extent is still a subject of future research.

Another interesting observation is that we identified more perceived benefits than drawbacks with AT in the workshop. Additionally, the perceived drawbacks were mostly related to the limitations of the prototype or were common drawbacks found in test automation. Due to these results, we believe that AT

is a promising technique that deserves more research since it may provide industry with new benefits that current techniques lack. Especially since this line of research may uncover solutions to problems common to automated testing.

We also found that the two main perceived benefits (B1 and B2) observed during the workshop were associated more with the AT prototype tool than the technique itself. One possible reason for this might be that the participants did not get enough information during the workshop to evaluate the technique as a technique. Hence, their perceptions of the technique are biased by the AT prototype tool's performance, features and how it was demonstrated. Whilst this conclusion highlights a threat to the validity of this work, the study also did identify AT specific benefits and one drawback which then rules out that the participants were completely unable to grasp the concept of AT as a technique.

One of the strongest perceived benefits "B6. Auto detection of defects", like "Detect spelling errors" or "Detect UX problems", is not implemented in the prototype for AT. This suggest that we failed to anticipate an important use case of the AT technique when creating the prototype. The ability to find defects or issues automatically has been discussed many times before using model based testing techniques [109] [19] and should be considered in more detail in future iterations of the prototype. The workshop study also identified another possible benefit of a future tool: "B5. Improved collaboration". The prototype does not contain team collaboration features today since it is not a key feature in the concept of AT even though the concept could be extended to encompass it. Team collaboration features, like crowd-sourcing of input actions and input data, is interesting since it opens up new possibilities for generation of suggestions to the user. However, since this feature also adds complexity to the tool, future research is required to evaluate its efficiency and effectiveness.

3.7 Conclusions

We presented a novel technique we refer to as Augmented Testing (AT) and performed an industrial workshop study with 10 software developers to get their perceptions about benefits and drawbacks with the technique.

When we compared the perceived benefits and drawbacks collected from the participants during the workshop study with the results from a literature study by Rafi et al. several common benefits and drawbacks were identified. The results indicate that AT has benefits and drawbacks in common with test automation. However, we also found a number of perceived benefits that were

not present in the literature study by Rafi et al., implying that AT can provide new benefits to the field of software testing.

Some of the perceived benefits reported in the workshop study were: "B1. Know what to test and what has been tested", "B2. Less manual work" and "B6. Auto detection of defects". The perceived drawbacks reported were: "D1. Maintenance problems" and "D2. Tool related problems". The identified benefits and drawbacks will be useful for further development of the technique and prototype for AT.

Since more perceived benefits than drawbacks were identified by the workshop, and the perceived drawbacks were mostly related to the limitations of the prototype or common problems found in test automation, we believe that the technique for AT is promising and deserves more research.

3.8 Future Work

Due to the observed common benefits and drawbacks between automated testing and AT we believe that a more in-depth analysis of such benefits and drawbacks would be beneficial for further research into the technique. In particular, the benefits and drawbacks associated with GUI-based testing, which is currently lacking in academic literature, would give insights into features that AT could focus on to maximize its complementary value to automated testing in practice.

Since we have only identified perceived benefits and drawbacks, future work is also required to identify the actual benefits and drawbacks of AT when used in industrial practice. This work implies analysis of AT in actual use, first in a controlled academic setting but later also in an industrial setting to evaluate its efficiency and effectiveness.

In summary, AT is a novel technique for software testing with perceived benefits that complement current automated testing techniques. These benefits could help solve industrial challenges associated with software testing and therefore warrant future research and development of the technique.

3.9 Acknowledgements

This work was supported by the KKS foundation through the S.E.R.T. Research Profile project at Blekinge Institute of Technology and by a research grant for the ORION project (ref. 20140218) from The Knowledge Foundation in Sweden.

Chapter 4

On the Industrial Applicability of Augmented Testing: An Empirical Study

Abstract

Testing applications with graphical user Interfaces (GUI) is an important but also a time-consuming task in practice. Tools and frameworks for GUI test automation can make the test execution more efficient and lower the manual labor required for regression testing. However, the test scripts used for automated GUI-based testing still require a substantial development effort and are often reported as sensitive to change, leading to frequent and costly maintenance. The efficiency of development, maintenance, and evolution of such tests are thereby dependent on the readability of scripts and the ease-of-use of test tools/frameworks in which the test scripts are defined.

To address these shortcomings in existing state-of-practice techniques, a novel technique referred to as Augmented Testing (AT) has been proposed. AT is defined as testing the System Under Test (SUT) through an Augmented GUI that superimposes information on top of the SUT GUI. The Augmented GUI

can provide the user with hints, test data, or other support while also observing and recording the tester's interactions.

For this study, a prototype tool, called Scout, has been used that adheres to the AT concept that is evaluated in an industrial empirical study. In the evaluation, quasi-experiments and questionnaire surveys are performed in two workshops, with 12 practitioners from two Swedish companies (Ericsson and Inceptive). Results show that Scout can be used to create equivalent test cases faster, with statistical significance, than creating automated scripts in two popular state-of-practice tools. The study concludes that AT has cost-value benefits, applies to industrial-grade software, and overcomes several deficiencies of state-of-practice GUI testing technologies in terms of ease-of-use.

Keywords: System Testing, Test Automation, Industrial Case Study, Augmented Testing

4.1 Introduction

Manual software testing of an application is labor-intensive, requiring both technical and domain knowledge to be effective, and is therefore costly [71, 73]. Test automation has been proposed as a complement to reducing the error-prone, repetitive, and labor-intensive work performed using manual testing. Automation can lead to shorter lead time since automated tests are often executed faster than manual test cases, which is particularly important in agile projects. Automated testing has been proposed for all levels of system abstraction, but testing of an application through its user interface has been reported to have many limitations that reduce its applicability in industrial practice [90, 138, 150]. One of the most commonly reported limitations is test script fragility that in turn leads to a high cost of creating and maintaining automated test cases. These costs then become dependent on the time required to understand, write, or rewrite the tests.

To address these challenges, a novel technique that we call Augmented Testing (AT) [119], has been proposed as a possible solution that enhances and increases the communication speed between the human tester and the machine. The intention is to make it easier for the tester to create effective tests and more easily co-evolve them with the SUT and its requirements.

Previous work on AT has, however, not explored the technique's applicability in industrial practice, only it's perceived applicability [119]. To evaluate the actual applicability, this paper presents an industrial empirical study where Scout,

a prototype tool for AT, is compared against two commonly used frameworks, in the industry, for GUI test automation.

Following the technology transfer model defined by Gorschek et al. [69], this study aims to evaluate if AT has reached a stage of research to make it industrially applicable.

The specific contributions of this paper are, therefore:

- An overview of a novel technique for GUI test automation, called Augmented Testing (AT).
- Results from an empirical evaluation of AT with industrial practitioners of the efficiency of Scout, a prototype for AT, compared to two state-of-practice approaches for GUI test automation.

4.1.1 Augmented Testing

Augmented Testing (AT) is a novel technique that aims to improve communication between the human tester and the machine to improve the efficiency and effectiveness of GUI testing. AT is defined as testing the System Under Test (SUT) through an Augmented GUI, which contains superimposed information on top of the SUT GUI. The Augmented GUI may be used to highlight important areas as well as give hints to the tester (hereby called Superimposing), but can also observe the testers' interactions with the SUT GUI and use this to learn how to perform actions and verify conditions (hereby called Observing). Superimposing and Observing are thereby core concepts of AT and mandatory features of any AT tool or framework. AT can be utilized in many potential areas, including giving the tester guidance through suggestions of what to verify, how to test (e.g. new test data) or what to test (e.g. new test paths to explore). This guidance aims to reduce cognitive load by removing, for instance, context switching between the computer monitor and manual test case descriptions, recording automated test scenarios or providing inputs for exploratory testing.

4.1.2 Scout

Scout is a prototype tool that follows the definition of AT and that has been developed for evaluating the technique in collaboration with industrial practitioners. Figure 4.1 gives an overview of the core components in Scout.

The Augmented GUI is the only user interface presented to the tester, whilst the actual GUI of the SUT is made unavailable. This implies that all actions performed by the tester are received from the Augmented GUI and is relayed

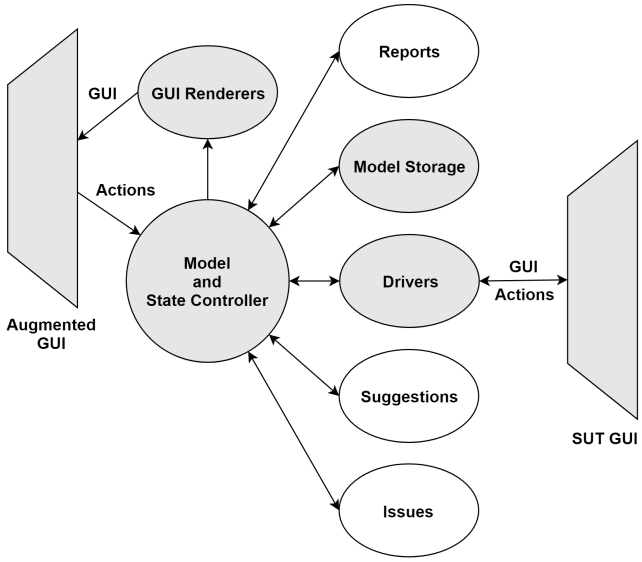

Figure 4.1: Scout Overview

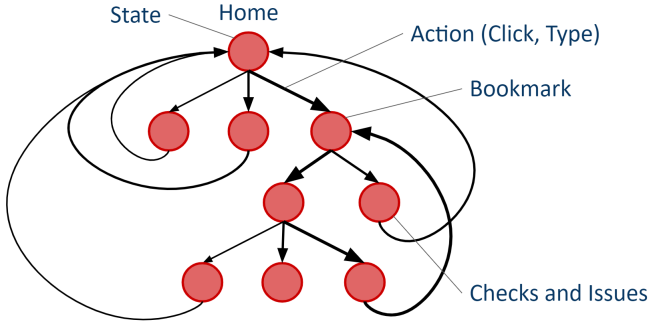

Figure 4.2: State Model

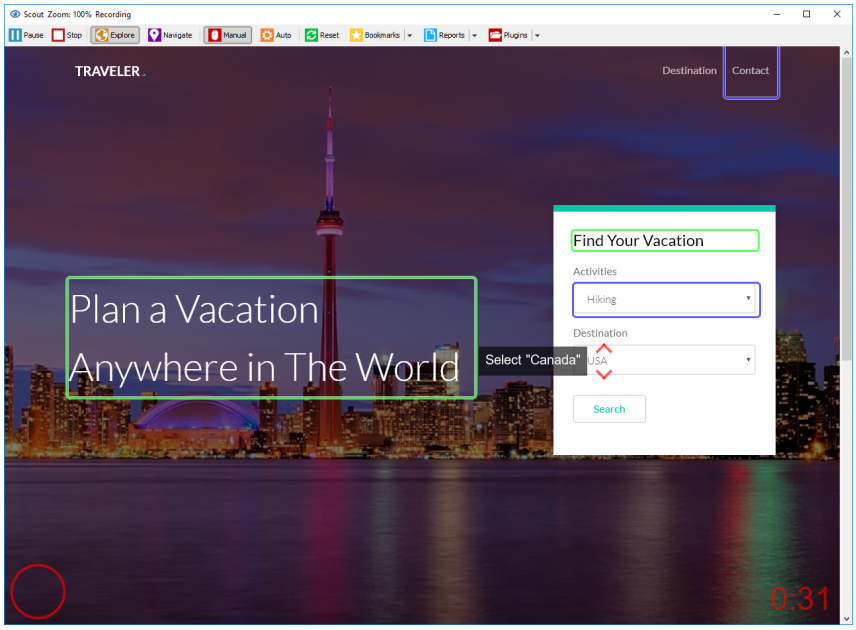

Figure 4.3: Screenshot of the Scout Prototype

down to the actual SUT GUI through an application *Driver* that translates the actions received from the tester to the corresponding actions on the SUT.

An Augmented GUI thereby shows the SUT's GUI but modified with additional checks, suggestions, issues and other information that is rendered, on top, of it. A sample screenshot of the Scout prototype, during a test session with a web application, can be seen in Figure 4.3. In the figure, the web site is augmented with previously recorded actions (blue squares) and automated checks (green squares).

Unlike conventional test tools that use scripts to store test scenarios, Scout stores actions, checks, suggestions and issues in a Model. The Model is implemented as a weighted graph, where each node in the graph represents an application state. An example of a small Model is visualized in Figure 4.2. We define the application state to be the state of an application, including all its dependencies like internal memory, stored data and external applications at a given time. Nodes are connected by Actions that represent an event performed by the tester, like a mouse press or keyboard input. Utilizing a Model to store

information is suitable for SUT state and user behavior information. However, this is a Scout specific implementation, not necessarily shared by other, future, AT tools that may store information in other ways, e.g. scripts.

Another component within Scout is the State Controller that keeps track of the current application state. Keeping track of the application state is crucial since the augmented information needs to be synchronized with the GUI of the SUT.

The Scout prototype has similarities to previous record and replay (R&R), also called capture and replay, tools but differ in several fundamental areas. R&R tools have several benefits, like the ease-of-use of recording tests. However, they also have many drawbacks, such as a high cost of understanding and thereby maintaining the generated test cases [108]. Another drawback is that many recording tools record human interactions by hooking into the SUT and capturing its responses, which may not be possible to replay, leading to failed test cases. The R&R tools were also limited by their underlying technologies that made the test cases brittle [118, 150]. This brittleness caused tests to require significant maintenance efforts that diminished their return on investment.

Scout addresses the recording challenge by recording actions directly when they are performed by the tester before they are relayed to the SUT's GUI. In this way, Scout is able to replay the test case in exactly the same way it was recorded. A more important difference between Scout and previous R&R tools is that test cases can be maintained or extended in the same way as they were initially recorded. Hence, by following existing paths, highlighted in the Augmented GUI, and occasionally exploring new ones, the tester can easily create new branches of existing test cases. Albeit sometimes these actions are performed differently than in the real SUT's GUI due to the implementation of Scout, for instance, text may be written before assigning it to a text field. However, and in contrast to R&R tools that may require manual merging of several recordings through programming, Scout requires no such technical knowledge. The knowledge that was previously required to maintain the recorded test suite by creating reusable, often parameterized, functions or methods [108].

4.2 Related Work

In this section, we'll present an overview of related work in advanced GUI-based testing with similarities to AT and the Scout prototype. The previous works will then be discussed and related to our work at the end of the Section.

Pham et al. [137] proposed a concept for linking debugging information with a video- or screen-recording of an application. They stated that it is a useful concept for debugging the SUT but also that it helped in debugging failures and that it can save time in the testing and debugging phase of a software development process. Vos et al. [154] presented a technique and a tool, called TESTAR, to reduce the fragility of test scripts caused by constant changes in the SUT GUI by automatically detecting and executing tests generated from the widgets currently available. Aho at al. [19] introduced a process of using Murphy tools to extract GUI application models and using the extracted models to support different GUI testing activities but also share their experiences of using that approach in industrial development and testing environments. Mariani et al. [109] presented a tool and a technique that they call AutoBlackTest, to be able to automatically generate system test cases for interactive applications. The goal of the tool and technique was to generate test cases that exercise the functionalities of the SUT by navigating the SUT GUI. Amalfitano et al. [33] presented a technique for detecting and unlocking something they call a Gate GUI, using machine learning. A Gate GUI is, according to Amalfitano et al., a GUI that prevents a random exploration technique from advancing to relevant parts of the SUT. An example of a Gate GUI is a login dialog where a valid name and password have to be entered before access is allowed to certain features of the SUT.

Scout and AT are inspired by all of the techniques mentioned above. Scout records and models valid application scenarios, by observing and learning from the end-user, similar to the tools described by Aho at al. and Mariani et al. The interactions with the SUT can also be recorded very accurately since all the end-user interactions go through the Augmented GUI. Instead of just selecting actions to perform randomly or using a heuristic, like the TESTAR tool presented by Vos et al., Scout has the benefit of both knowing all available actions in a given state but also the actions it learned from observing the end-user. Furthermore, similar to Pham et al. actions, checks, issues and other information from the current application state is augmented/visualized on top of a screenshot from the SUT to ease failure identification and debugging. Gate GUIs, investigated by Amalfitano et al., can also be handled in Scout without manual scripting or modeling since these are captured in user scenarios and connected to the application's state. Thus, albeit not providing the same solutions, Scout provides a combined solution similar to the related works and their benefits.

4.3 Methodology

The goal of this study is to evaluate the industrial applicability of the Superimposing and Observing concepts of AT by comparing Scout against state-of-practice (SOP) approaches for testing an application through the GUI. This research goal has been broken down into three research questions where AT is represented by the research prototype tool Scout:

- **RQ1:** How does the efficiency of creating tests using AT compare to scenario-based SOP approaches?
- **RQ2:** How do industrial practitioners perceive the issue finding ability when using AT compared to SOP approaches?
- **RQ3:** How much experience in test automation and programming do industrial practitioners perceive to be required to successfully create tests using AT compared to scenario-based SOP approaches?

We aim to answer our first research question using quasi-experiments. Quasi-experiments were chosen over controlled experiments since the research objective is to evaluate AT's applicability in practice. This objective infers a need to involve human subjects, within their domain, making it impossible to control for all confounding variables, e.g. the participants' hardware (computer), previous knowledge and allocated resources (e.g. number of subjects and time). The experiments were however based on guidelines for Software Engineering experiments [159] with dependent and independent variables and aim to test the hypothesis:

- H_{01}: The efficiency of creating tests, measured as time, is greater when using AT compared to scenario-based SOP approaches.

4.3.1 Phase 1: Scout vs Protractor

Protractor [9] is an end-to-end test automation framework for Angular [2] applications. Protractor runs tests against an application running in a web browser, interacting with it as a user would. The Protractor framework is commonly used by the industry and therefore a good state-of-practice candidate to compare against Scout.

This phase of the study was performed in collaboration with Ericsson, a telecommunication company that is one of the largest manufacturers of mobile

communication equipment in the world with more than 94 000 employees worldwide. The company was chosen partly due to convenience sampling and because its applications include rich web-GUIs that could benefit from AT. The study was performed using a quasi-experiment on a system developed and maintained by approximately 66 developers in 11 teams at the company. The purpose of the quasi-experiment was to measure the efficiency, measured as time, of creating test cases for the system (dependent variable) using Scout and Protractor (independent variable), and provide support for hypothesis H_{01} to answer RQ1.

The web interface of the system is currently tested using the Protractor framework that uses Selenium WebDriver to perform low-level events, like button presses and sending keystrokes to the application. As such, Protractor is comparable to Scout that uses this same means of interaction and enables us to measure efficiency as a function of the tools' different means of test case development, i.e. recording through the Augmented GUI (Scout) and scripting (Protractor).

Before the workshop, to make Scout applicable to the system, the tool required some customization since many of the web components in the system were not identified correctly by Scout's test drivers. As an example, Scout was unable to detect some HTML headings (h1, h2, h3, etc.) as clickable since they were not surrounded by an anchor element. The leading researcher spent approximately 8 hours to identify and add support for the unrecognized web components, which made Scout applicable to most components (estimated to 90 percent). To avoid issues during the workshop, only recognized components were used.

In preparation for the workshop, a test case was defined in natural language text that was relevant to the SUT and representative for tests in the existing Protractor test suite. A constraint for the test was, however, to make it small enough to be created during the limited time, allocated by the company, for the workshop (i.e. 1.5 hours). The leading researcher, in collaboration with one of the developers from the team, used the web interface of the system to specify a test case of 16 steps that were similar, but not identical, to one of their existing test cases.

The comparison of Scout to Protractor was performed in a workshop divided into two sessions, 30 and 60 minutes respectively, with six developers from different teams. During the first session of the workshop, the participants were provided with print-outs of the step-by-step instructions for the test case and were given the task to develop a corresponding test case using Protractor and take note of the time to create a working (i.e. running) test. They were not allowed to take any breaks or consult anyone while creating the test case, however,

using online documentation or forums was allowed. This task was time-boxed to 20 minutes in the first 30-minute session due to the limited time of the workshop. Participants that failed to complete the task during the allocated time (i.e. complete the entire test case) were instructed to estimate how much, in percent, of the test case that was remaining and also how much time that it would take to complete the remaining part. This made it possible to not only get an estimate of the time to create the entire test case but also get a hint of how reliable that estimation was. A participant that, for example, was 90 percent finished would likely be able to make a fairly accurate approximation of the total time since only 10 percent of the work would need to be estimated.

At the beginning of the second session of the workshop, the participants were instructed to download and install the Scout tool. After completing the Scout installation process, the participants were given a short introduction to the tool (10 minutes) and then instructed to create the same test case as in the first session but this time using Scout. However, instead of a manual print-out as in the first session, the test steps were rendered in Scout's augmented GUI as text-boxes beneath the mouse cursor, a feature supported by the tool for superimposing information. The feature automatically traverses the imported, manual, test sequence and displays one step of the sequence for each action the user should make. Since Scout keeps track of the session time automatically, the participants were instructed to simply write down the session time, displayed by the tool, when finished recording the test.

Finally, at the end of the second session of the workshop, the participants were asked to fill out a questionnaire survey, with 6 point Likert-scale [40] questions, that contained statements about Scout and Protractor that should be answered with a number where 1 was "strongly disagree" and 6 was "strongly agree". The questions used in the questionnaire are found in Tables 4.4 and 4.5.

4.3.2 Phase 2: Scout vs Selenium

Selenium WebDriver [11] is a well-known open-source tool for automating web applications. The tool can be used directly using its application interface (API), which is provided in many different programming languages, or indirectly through some kind of frameworks like Scout or Protractor.

The quasi-experiment was performed as a workshop with Inceptive, a consultant company that is focused on software testing and requirements. The company has a total of 38 employees distributed on two locations in Sweden of which six participated from their Gothenburg office.

Six participants attended the workshop, all with some experience of software development but almost no previous knowledge in test automation, Selenium WebDriver or Scout.

The workshop began with a two-hour seminar about Selenium WebDriver since we decided that the participants needed an introduction to be able to create a test case. Additionally, after downloading and completing the Scout installation process, the participants were given a ten-minute introduction to the tool by the leading researcher.

The quasi-experiment aimed to compare the time to create a test case from a textual step-by-step instruction (dependent variable) using the Selenium WebDriver API and Scout (independent variable) for the same website used in Phase 1. This quasi-experiment aimed to test hypothesis H_{01} and answer RQ1. The participants were instructed to create a test using Scout from a manual instruction that was presented, one instruction at a time, in the Augmented GUI, i.e. superimposed information. Next, the same test instructions, this time in the form of natural language text, were used by the participants to create an automated test using Java and the Selenium WebDriver API. The participants were also instructed to note the time needed to create each test case.

Finally, at the end of the workshop, the participants were asked to fill out a questionnaire survey, with 6 point Likert-scale questions, which contained statements about Scout and Selenium WebDriver that should be answered with a number where 1 denoted "strongly disagree" and 6 denoted "strongly agree". The questions can be found in Table 4.4 and 4.5.

4.4 Results

The results of the study were acquired by comparing Scout to two different state-of-practice approaches to GUI testing in industrial practice. Protractor was chosen out of convenience since it was the framework currently used by Ericsson. Selenium, was explicitly chosen since it represent a commonly used technique in industrial practice.

4.4.1 Phase 1: Scout vs Protractor

Table 4.1 presents the time, in minutes, for the participants to create one test case using Scout and then by using Protractor. In the table, partly estimated values are marked with an asterisk (*) while completely estimated values are marked with two asterisks (**). The (***) denotes the result of the leading

researcher, added to the table as a base-line value compared to the practitioners. The leading researcher did not perform the Protractor tests due to the lack of experience with the tool.

Table 4.1: Create test using Scout and Protractor

Part. no:	**Scout (min):**	**Protractor (min):**
1	3.3	23*
2	3.7	90*
3	4.6	60*
4	3.5	31
5	3.9	60**
6	2.6	16
7***	3	NA
Average:	**3.5**	**46.7**
StDev:	**0.65**	**28.2**

Table 4.2: Create test using Scout and Selenium

Part. no:	**Scout (min):**	**Selenium (min):**
1	2.43	24.25
2	1.98	8.15
3	2.38	14.78
4	1.85	39.92
5	2.53	13
6	1.43	NA
7***	1.63	10.27
Average:	**2.03**	**18.4**
StDev:	**0.43**	**11.9**

Creating a test using Scout required, on average, about 7.5% of the time to create a similar test case using Protractor (3.5/46.7 minutes). Formal statistical analysis with the Wilcoxon signed-rank test also showed this result to be statistically significant different (p=0.03125). Worth noting is the high standard deviation of the time required to create Protractor test cases, which can be explained by the varying skill of the participants, which also influences their estimates. However, even if the average time required to create tests in Scout is compared to the best time for creating the test in Protractor, Scout still only required 21% as much time (3.5/16 minutes). Additionally, we note that the leading researcher's time (3 minutes) was roughly equivalent to the participant's average (3.5 minutes).

4.4.2 Phase 2: Scout vs Selenium

Results from the workshop, that required the participants to create tests using the two tools, are presented in Table 4.2. Creating a test using Scout required, on average, about 11% (2.03/18.4 minutes) of the time to create a similar test case using Selenium. Formal statistical analysis with the Wilcoxon signed-rank test also showed this result to be statistically significant different (p=0.03125). Additionally, we note that the leading researcher's time to create a Scout test (1.63 minutes) was comparable to the participants' average (2.03 minutes). However, the leading researcher's time to create the test case in Selenium (10.27 minutes) was almost twice as fast as the average of the participants (18.4). Thus, it is possible that, on average, more skilled Selenium users would perform better.

One of the Selenium results were marked as NA since the participant failed to create a test case due to problems with the development environment. The (***) denotes the result of the leading researcher, added to provide a base-line value compared to the practitioners.

4.4.3 Survey

A questionnaire survey was performed with a total of 12 participants, 6 from each workshop in Phase 1 and 2, who created test cases using Scout and either Protractor or Selenium WebDriver. Results from the survey are presented in Table 4.4 and 4.5. The participants in Phase 2 also got to answer two additional questions (denoted Question 8 for Scout and Selenium) concerning the ease of understanding Scout and Selenium test cases. The survey contained statements about Scout and Protractor or Selenium WebDriver that should be answered according to a 6 point Likert-scale, where 1 denoted "strongly disagree" and 6 denoted "strongly agree".

The tables also contains calculated averages, hereby referred to as the Likert-average (LA), of the responses from the 12 participants. The LA value gives us a number to use for comparing the responses from the participants, even though we are aware of the fact that numbers in a Likert-scales should not be used like metrics.

When comparing question 3 in Table 4.4 with the corresponding question in Table 4.5 we see that the LA value for Scout is much higher (5.66 vs 1.5) than for Protractor/Selenium and that gives us an indication that the participants perceive that Scout require less programming skills than Protractor/Selenium (RQ3). Question 4 shows a similar result, even though less convincing (5.26 vs 3.25), but gives an indication that the participants perceive that Scout requires

less test automation skills than Protractor/Selenium (RQ3). Question 6 was designed to answer RQ2 and shows that the participants perceive that Scout is almost as good as Protractor/Selenium for locating issues (4.08 vs 4.8). However, this is only based on perception and further research is required to verify the tool's ability to do so in practice. Question 7 confirms the results, from the quasi-experiments designed to answer RQ1, that the participants perceive that it is less time consuming to create tests using Scout than using Protractor/Selenium (2 vs 5.17). Question 8 indicates that the participants perceive that Scout tests are easier to understand and follow than Selenium tests (5 vs 3.33). Finally, question 1 and 2, are not connected to any research question but, indicates that the participants perceive that Scout is both easier to use and learn than Protractor/Selenium (5.08 vs 3.67 and 5.33 vs 4).

Table 4.3: Comparing all approaches of creating tests

Approaches:	**Time (min):**	**Percent:**
Scout vs Protractor	3.5 vs 46.7	7.5%
Scout vs Selenium	2.03 vs 18.4	11%

Table 4.4: Survey about Scout. LA - Likert Average, PX - Person X.

Question	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**LA**
1. Scout is easy to use.	6	6	3	5	5	6	4	5	5	6	6	4	**5.08**
2. Scout is easy to learn.	6	6	4	5	5	6	5	6	6	6	5	4	**5.33**
3. Scout requires little or no programming skills.	6	6	6	6	6	6	6	6	6	3	6	5	**5.66**
4. Scout requires little or no test automation skills.	5	6	6	4	5	5	6	5	6	5	6	4	**5.25**
5. Scout is fun to use.	5	5	5	4	3	6	3	6	4	5	3	3	**4.33**
6. Scout can spot the same kind of issues as other test automation tools.	5	6	5	5	4	3	4	3	3	3	4	4	**4.08**
7. Creating automated tests using Scout is time consuming.	1	2	2	2	3	2	2	1	2	1	3	3	**2**
8. Scout tests are easy to understand and follow.							5	5	4	6	5	5	**5**

Table 4.5: Survey about Protractor / Selenium. P1-P6 used Protractor whilst P7-P12 used Selenium. LA - Likert Average, PX - Person X.

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	LA
1. Selenium / Protractor is easy to use.	3	3	5	4	4	3	3	2	5	3	4	5	**3.67**
2. Selenium / Protractor is easy to learn.	2	4	5	3	4	3	4	3	5	6	4	5	**4**
3. Selenium / Protractor requires little or no programming skills.	2	1	3	1	1	1	1	2	2	2	1	1	**1.5**
4. Selenium / Protractor requires little or no test automation skills.	4	1	4	2	3	2	3	3	5	5	5	2	**3.25**
5. Selenium / Protractor is fun to use.	3	2	3	3	3	2	5	5	4	4	4	3	**3.41**
6. Selenium / Protractor can spot the same kind of issues as other test automation tools.	6	4	6	6	6	2	4	6	5	5	4	4	**4.8**
7. Creating automated tests using Selenium / Protractor is time consuming.	5	6	5	4	6	5	5	5	6	4	6	5	**5.17**
8. Selenium tests are easy to understand and follow.							5	3	3	3	3	3	**3.33**

4.4.4 Threats to Validity

In this section we will go through some threats to the validity of this paper. We stress that the focus of the study was industrial applicability of GUI Augmentation and that, like most industrial studies, resource constraints and other confounding factors played a role in the study's design and execution.

Construct validity: The studies were performed with industrial practitioners on one large industry-grade system but also on a website with a lower complexity than a real system. To be able to only use industry-grade systems would have provided us with more valid results.

Internal validity: Due to limited time available for the workshops, we decided not to break up the participants into two groups to enable a cross-experimental design, in phase 1 and 2, even though we would get a learning bias when the participants created the same test cases again but with a different framework. Another threat to the internal validity is that we have not measured the issue finding ability of the tests created using Scout. We have only asked the participants, using a survey question, if they think that Scout can spot the

same kind of issues as other test automation tools. Extending each phase by also measuring the issue finding ability would have increased the time to perform the workshops and since the time, with the participants, was very limited we decided to leave that to further research.

External validity: A threat to external validity is that we have only results from testing web applications but not desktop- or mobile-applications, etc., that might give completely different results due to the underlying technology. To also be able to compare Scout with SOP frameworks on a wide variety of industrial applications would improve external validity, something that we aim to do in further research. Another threat is that we have only evaluated AT using one tool, our prototype Scout, since we are not yet aware of any other tool that meets the definition of AT.

Reliability: There was a high variance between the results for creating Protractor and Selenium WebDriver tests in phases 1 and 2 even though we could not identify a correlation with the previous experience of the participants. A possible explanation to that observation is that the results depend on a confounding variable, that we have failed to identify, and would be a treat to the reliability since it affects the average results.

4.5 Analysis

4.5.1 RQ1: How does the efficiency of creating tests using AT compare to scenario-based SOP approaches?

We performed two quasi-experiments to answer this question and had the hypothesis that; the efficiency of creating tests, measured as time, is greater when using AT compared to scenario-based SOP approaches (H_{01}).

A summary of the results from the quasi-experiments, where we compared Scout with other state-of-practice approaches, can be found in Table 4.3. When comparing the Augmented GUI in Scout with two commonly used frameworks for automated GUI testing, the participants were able to create a test case using augmentation in 7.5% of the time, on average, compared to Protractor and in 11% of the time, on average, when compared to Selenium. Both results showing, with statistical significance, that tests could be created quicker in Scout's Augmented GUI.

These results show that AT, as it is currently implemented by Scout, is more efficient for test case creation than the two state-of-practice approaches that we investigated. However, this claim is delimited to the investigated approaches,

the number of participants in the study and the high variance in the results that prohibit us from confidently stating with what factor Scout is faster. We also stress that the research question only focused on providing a result regarding if AT is more efficient for test case creation, not more efficient overall. Overall efficiency would require additional research, as presented in Section 4.8. This result, regardless, shows initial cost-benefit and industrial value, which opens up for additional research into AT to explore what additional benefits more advanced superimposition and observation of information can achieve. Such research should focus on the overall value of AT as a concept can achieve and also compare Scout to other AT tools (when made available) and similar techniques such and R&R.

4.5.2 RQ2: How do industrial practitioners perceive the issue finding ability when using AT compared to SOP approaches?

We performed a questionnaire survey with a total of 12 participants, 6 from each workshop during phases 1 and 2, to be able to answer this question.

The responses from question 6 in the survey give us the answer that the participants find Scout almost as good as Protractor/Selenium for locating issues, with a Likert-average (LA) of 4.08 compared to 4.8 for Protractor/Selenium. This is a logical conclusion since Scout currently utilizes the same driver framework as the other tools, i.e. Selenium. However, since the participants did not rank the tool as high, and have experience with Protractor and/or Selenium but not with Scout, they might need more time with Scout to be able to trust its ability to find issues. An alternative explanation is that the participants are correct in their perception since a tool like Scout has flexibility limitations compared to an API based approach. In the current prototype of Scout it was, for example, not possible to automatically verify the number of rows presented in a table, something that would be fairly easy to do using a programming language.

4.5.3 RQ3: How much experience in test automation and programming do industrial practitioners perceive to be required to successfully create tests using AT compared to SOP approaches?

Questions 3 and 4 from the survey were designed to give us an answer to this question. The LA value for question 3, regarding the requirement on program-

ming skills, is much higher for Scout than for Protractor/Selenium (5.66 vs 1.5). Additionally, for question 4, regarding test automation skills, participants indicate that Scout requires fewer skills in test automation than Protractor/Selenium (5.26 vs 3.25).

Skilled programmers or test automation experts are expensive and a tool or technique, like Scout and AT, which require less technical skills could thereby save a lot of money for the industry. Additionally, the lower skill requirements imply that domain experts, which usually lack the technical skill (e.g. nurses in the medical domain) can perform test automation.

These results are further supported by the fact that practitioners, of varying knowledge and skill, were all able to finish the test creation tasks in Scout in the allocated time-frames, whilst many did not finish them in the compared state-of-practice approaches.

4.6 Discussion

The results and analysis from our first research question (RQ1) show that AT, represented by the tool Scout, was more efficient, in creating tests, than the two SOP approaches that we compared with. We have also indications that the industrial participants think the need for programming or test automation skills are lower for successfully creating tests using Scout compared to the SOP approaches (RQ3). The participants also believe that Scout has almost the same issue finding ability as the SOP approaches (RQ2).

We believe that one explanation of the observed gains in efficiency of creating tests can be that the test steps are created by just clicking, inside the Augmented GUI, instead of typing commands using some form of script or programming language. As an example, a typical action like a click on a button or link is, in most cases, added using one mouse click on the Augmented GUI while it would typically require two or more lines of code to perform the same thing.

That testers, without programming skills, can quickly create effective tests would be valuable for the industry since more tests result in better test coverage that in turn helps improve the quality of the SUT. Also AT would help lower overall cost, not only because of the shorter time to create the tests, but also that less skills and training of the testers, e.g. in programming, is required.

Tests in Scout are however defined on a higher abstraction level than tools that rely on a programming language or scripts, making tests in Scout easier to understand but also less flexible to use. This limitation can however be handled, to some extent, in Scout since it has a modular design that can be customized

for the unique characteristics of the SUT, but the general impact of the modular design is still not known.

In future research, as presented in Section 4.8, the most pressing focus is the need to evaluate the maintenance cost of tests created using Scout since a high cost for maintenance could reduce the tool's and AT's viability for industrial use.

4.7 Conclusions

Testing an application through the GUI is important but also a time consuming and costly task to perform in practice. Tools and frameworks for GUI test automation can be used to lower the manual labor but the tests produced requires a substantial development and maintenance effort that reduces the cost savings from the automation effort. Augmented Testing (AT) and the tool prototype Scout has been proposed as a solution to lower the cost of both creating and maintaining test cases for testing an application through its GUI.

We performed two quasi-experiments and a survey with a total of 12 participants from from two Swedish companies (Ericsson and Inceptive) to measure the efficiency of Scout's Augmented GUI, on two different web applications. The results show that Scout requires less programming and test automation skills, that the tests are easier to understand and follow and that creating effective test cases is more efficient compared to the two state-of-practice approaches that we investigated.

4.8 Future Work

Scout uses the same under-laying techniques, as other state-of-practice tools for GUI test automation. Since both Protractor and Scout uses Selenium WebDriver as a driver for executing the tests, we have decided not to measure and compare the time to automatically execute regression tests, in this study, since an optimized tool should, at least in theory, have a similar execution speed. We should however measure and compare the test execution speed with other tools and techniques just to verify our assumptions, by additional research.

The results indicate that AT is a very promising technique for GUI test automation, but before we can draw any conclusions about the industrial viability of AT, we also need to research the long-term maintenance cost of the tests

since that is one of the most reported problems for similar techniques and tools designed for GUI test automation.

To further improve the effectiveness and quality of the test cases, in Scout, we also need to research how to provide the tester with suggestions of, for example, new paths to try or new values to enter that might reveal an issue or increase the test coverage.

4.9 Acknowledgments

This work was supported by the KKS foundation through the S.E.R.T. Research Profile project at Blekinge Institute of Technology.

Chapter 5

Similarity-based Web Element Localization for Robust Test Automation

Abstract

Non-robust (fragile) test execution is a commonly reported challenge in GUI-based test automation, despite much research and several proposed solutions. A test script needs to be resilient to (minor) changes in the tested application but, at the same time, fail when detecting potential issues that require investigation. Test script fragility is a multi-faceted problem. However, one crucial challenge is how to reliably identify and locate the correct target web elements when the website evolves between releases or otherwise fail and report an issue. This paper proposes and evaluates a novel approach called similarity-based web element localization (Similo), which leverages information from multiple web element locator parameters to identify a target element using a weighted similarity score. This experimental study compares Similo to a baseline approach for web element localization. To get an extensive empirical basis, we target 48 of the most popular websites on the Internet in our evaluation. Robustness is considered by counting the number of web elements found in a recent website version compared to how many of these existed in an older version. Results of the experiment show that Similo outperforms the baseline; it failed to locate

the correct target web element in 91 out of 801 considered cases (i.e., 11%) compared to 214 failed cases (i.e., 27%) for the baseline approach. The time efficiency of Similo was also considered, where the average time to locate a web element was determined to be four milliseconds. However, since the cost of web interactions (e.g., a click) is typically on the order of hundreds of milliseconds, the additional computational demands of Similo can be considered negligible. This study presents evidence that quantifying the similarity between multiple attributes of web elements when trying to locate them, as in our proposed Similo approach, is beneficial. With acceptable efficiency, Similo gives significantly higher effectiveness (i.e., robustness) than the baseline web element localization approach.

Keywords: GUI Testing, Test Automation, Test Case Robustness, Web Element Locators, XPath Locators

5.1 Introduction

Software testing is vital to ensure a software application's quality, but it is also time-consuming and costly in practice [71, 73]. Still, numerous reports highlight test automation's efficiency and ability to lower costs while ensuring high quality of the released application [17, 28, 131].

Although automated testing has been proposed for different types of testing, one of its main application areas in practice is in automated regression testing. Automated regression testing is a way for testers to ensure each software release's quality. Typically, on higher levels of system abstraction, e.g. Graphical User Interface (GUI) level, it involves creating a suite of test scripts that emulate user scenarios while checking, using oracles, that the application behaves correctly [103, 108]. However, it is natural that new software releases contain changes that can then break the automated regression tests. This necessitates test suite maintenance which incur additional effort and costs to repair the test scripts to ensure the test suite remains up-to-date. This maintenance cost is especially high when testing an application through its GUI, since it frequently changes between releases [26, 58, 152]. Additionally, these tests are affected both by visual changes to the GUI and by changes to its underlying logic and application under test (AUT) architecture. GUIs are also primarily designed for humans, i.e., they are not designed for machine-to-machine communication, which presents additional challenges for automation, e.g., synchronization between scripts and the AUT, which are not as prominent in lower-level test techniques such as unit-testing [131].

There are several different techniques for automated testing of a GUI application [24], but one of the most commonly used approaches in practice when testing websites (i.e., web applications) is to use the Document Object Model (DOM) [4]. Although DOM-based approaches are specific to websites, similar approaches can be found for testing GUI-based desktop and mobile applications, for which meta-information about GUI elements can be accessed via the operating system or GUI library used by the application. In a DOM-based approach, GUI web elements (buttons, anchors, text fields, labels, etc.) are located using DOM properties, which include web element attributes, element text, unique IDs, XPath's [15], and CSS selectors [3]. DOM properties are, however, sensitive to changes in the GUI of the website, which affect the robustness of the automated test execution as the website evolves from release to release. This observation is often referenced as (test) *script fragility* (i.e., a lack of robustness) and frequently reported as a challenge by researchers [25, 30, 72, 94, 108, 115, 118, 166], resulting in increased test maintenance, costs, and lower AUT quality.

Naturally, significant changes to the website under test *should* cause test execution to break since the cause of such test failures may indicate a defect that needs to be addressed. However, minor changes might also break the test execution, even though a manual tester might have considered the test execution to succeed. Such minor changes that cause automated test execution failures are thereby a source of unnecessary debugging and maintenance work, especially since the test execution may seem to break for no apparent reason, e.g., when the change is small and difficult to recognize for the human user. This phenomenon of tests unpredictably failing has, as mentioned in the literature, been summarized as GUI tests being fragile/lacking robustness to AUT changes [60, 121].

There have been many attempts to address the fragility problem in the past two decades [21, 49, 60, 100, 102, 150, 172]. Several approaches try to limit the fragility problem by trying to build robust locators (e.g., [100]), i.e., locators capable of identifying the correct element even if the page has changed. One of the more recent attempts, proposed by Leotta et al., is to use multiple locators, instead of just one locator, to identify a web element [99] in a website. The basis of this approach is to utilize multiple sources of information to triangulate the correct web element. Research has shown that the multi-locator approach can effectively increase the probability of finding the correct web element since it is unlikely that all locators used for localization of the web element are changed simultaneously between two releases of a website.

A web element locator is defined as a method, function, approach, or algorithm that locates a web element in a web page given a locator parameter. The locator parameter is defined as a tuple that consists of a name and a value that the locator can use when locating one or more web element(s). Common types of single-locators use an XPath (path expression) or CSS expression as a parameter but could also use the tag name or a web element attribute e.g., ID, name, or class name. XPath locators select one or more nodes in an HTML DOM-tree when provided with a path expression as a locator parameter.

For this work, we refer to these first level locators as single-locators to avoid confusing it with multi-locators. A multi-locator (ML) approach (e.g., the approach proposed by Leotta et al.) uses more than one single-locator when localizing one or more web element(s) to increase the chance of finding the correct web element(s). Since we refer to Leotta et al.'s approach frequently in the paper, we will refer to it as Leotta's Multi-Locator (LML) to distinguish it from the more general concept of multi-locator (ML). The LML approach is also our selected baseline (see Section 5.4.2) that we compare with Similo.

To support the reasoning that using multiple sources of information improves the effectiveness of web element localization, Leotta et al. showed in their study [99] that the LML approach could reduce the number of failed localization attempts of existing web elements in six websites, from 12% down to 8%. In their study, they explicitly looked at failures caused by modifications to, or rearrangement of, the GUI's layout, look and feel, or DOM structure. While the LML approach resulted in an impressive 30% reduction of failed localization attempts, manually repairing locators due to technical limitations in the localization technique is still associated with considerable effort (i.e., cost) and warrants continued research.

This paper proposes a novel approach to web element localization for websites realized in a locator approach that we call *similarity-based web element localization* (in short **Similo**). Like the LML approach, Similo takes advantage of information from multiple sources. Similo is not, by definition, a multi-locator since it does not use the result gathered from a selection of single-locators like the LML approach. Instead, Similo quantifies the similarity between multiple *attributes* (locator parameters) of each candidate web element (i.e., possible candidates) and the target element (i.e., the element with the desired locator parameters) to identify the candidate element with the highest similarity to the target element, i.e., the candidate element with the highest probability of being the correct match for the target element. The Similo approach makes it possible to take advantage of any locator parameters regardless if the locator parameters can find a unique match or not. In comparison, the LML approach can only

take advantage of locators that can identify a candidate web element uniquely. However, since Similo targets the same challenge, and returns the same result, it is natural to compare their performance in an experiment.

In summary, the purpose of Similo is to increase the robustness of locating web elements in a website by comparing the similarity of web element locator parameters to achieve more stable test execution of GUI-based tests over time as the website evolves. In the reported study, we compare our approach with the LML approach (i.e., baseline) in a controlled experiment where we measure how many web elements could no longer be located between two releases, by either approach, in 48 websites. Results of the experiment show that Similo outperforms the baseline approach in terms of web element localization after website change at reasonable execution times for practical applications.

The specific contributions of this paper are:

- A novel approach for more robust web element localization based on comparison of the similarity of web element locator parameters;
- An empirical study that shows the effectiveness and time efficiency of the proposed approach compared to the baseline approach.

This paper is structured as follows. Section 5.2 gives a background of web element locators and presents the LML approach. Section 5.3 covers the details of the proposed Similo approach. The design, research questions, and procedure of the empirical study we conducted are presented in Section 5.4. The results are sketched in Section 5.5 and discussed in Section 5.6. Section 5.7 covers some threats to the validity of this study. We present related work in Section 5.8, and state conclusions and future work in Section 5.9.

A package for replicating the experiment is available for download from [10].

5.2 Locating Web Elements

Listing 5.1 shows an example of a simple test script, implemented in Java using Selenium WebDriver [11], which checks the functionality of a contact form in Figure 5.1. To improve the script's readability, we removed all the synchronization code needed to synchronize the script execution against the website by delaying the script execution to match website events.

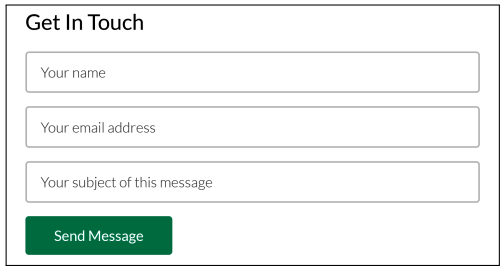

Figure 5.1: A contact form.

```
1  import org.openqa.selenium.By;
2  import org.openqa.selenium.WebDriver;
3  import org.openqa.selenium.chrome.ChromeDriver;
4  import static org.junit.jupiter.api.Assertions.*;
5  import org.junit.jupiter.api.Test;

7  public class ContactTests{

9      @Test
10     public void sendMessageTest(){
11         System.setProperty("webdriver.chrome.driver", "C:\\...\\
               chromedriver.exe");
12         WebDriver webDriver = new ChromeDriver();
13         webDriver.get("http://mimicservice.com/traveler");
14         webDriver.findElement(By.linkText("Contact")).click();
15         String text = webDriver.findElement(By.tagName("H1")).getText();
16         assertTrue(text.contains("Get In Touch"));
17         webDriver.findElement(By.id("name")).sendKeys("Michel");
18         webDriver.findElement(By.id("email")).sendKeys("michel.nass@bth.
               se");
19         webDriver.findElement(By.id("subject")).sendKeys("Contact me");
20         webDriver.findElement(By.linkText("Send Message")).click();
21         text = webDriver.findElement(By.tagName("H1")).getText();
22         assertTrue(text.contains("we will contact you shortly"));
23         webDriver.quit();
24     }
25 }
```

Listing 5.1: Sample test script implemented in Java using Selenium WebDriver.

Following is a description of the steps taken by the script in Listing 5.1. The test script begins by starting a new Chrome browser and navigating to the website "mimicservice.com/traveler". Next, the script clicks on the "Contact" link and verifies that the form's heading is "Get In Touch". The script continues by finding all the web elements that make up the form and fills it in by sending a

text to each input field. Finally, the script clicks the "Send Message" button and checks that the "we will contact you shortly" message appears on the webpage before, finally, closing the browser. As such, the script evaluates the webpage behavior by assuming that certain labels, i.e. the oracle, can only be checked if the website is operating correctly.

As can be seen from the test script example, the findElement method in Selenium WebDriver is used frequently for locating each of the web elements that the test script interacts with. In fact, the method is used every time an action is performed, a web element retrieved, or the value of a web element acquired to check (or assert) the AUT's behavior. The findElement method locates and returns the first web element that matches the supplied locator parameter. When there is no match in the current webpage, the findElement method throws a NoSuchElementException that breaks script execution.

Broken locators occur due to one out of two primary reasons; (1) that the web element is no longer present, or that the DOM-structure (or HTML code) of the application has been modified such that the web element has other properties [60, 81, 99], or (2) that the requested web element is not yet available during runtime of the application (synchronization between application and test runner) [37, 57]. We can address the first problem by deleting the reference to the removed/changed web element in the test or by updating the locator parameter (By class option) used by the findElement method, e.g., updating the ID attribute. A tester can correct the second problem in an automated test script by adding or modifying its synchronization code (generally a wait command, Implicit, Explicit and Fluent Wait in Selenium WebDriver), i.e., halt the test execution for a more extended time period to ensure that the locator is available.

Both types of problems are common in practice and also the leading cause of script maintenance costs [121]. Therefore, to reduce these costs, it is crucial to select locators that are resilient to changes in the AUT to make them robust.

Figure 5.2 shows a newer (to the left) and older (to the right) version of the same website (the homepage of YouTube.com). Some target web elements are marked using colored rectangles in the old version of the website (to the right). In this paper, we refer to the web elements in the older version, which we are trying to locate in the newer version, as target web elements. All the web elements in the newer version of the website are referred to as candidate web elements (i.e., the candidates that might be our target). In the example, each target web element has a corresponding candidate web element in the newer version of the website (marked with the complementary color).

There are eight different locators available in Selenium WebDriver, designed for finding elements by ID, name, class, tag, link text, partial link text, XPath,

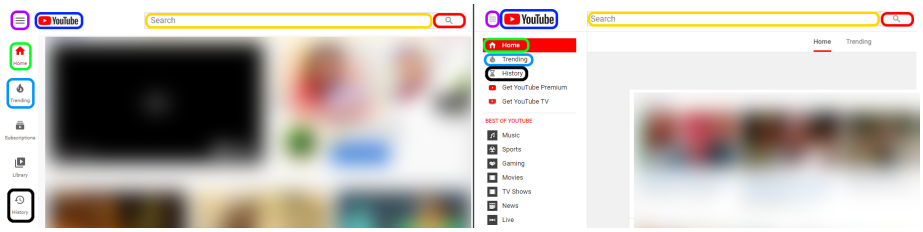

Figure 5.2: Web elements present in both the newer (left) and older (right) versions of the YouTube.com website. Some of the content is blurred since it could be sensitive or copyrighted.

and by CSS. We refer to these locators, individually, as single-locators since they try to locate one or many web elements using only one locator parameter (e.g., a single XPath expression or "ID" value).

Locating a web element using an absolute XPath is a common use of single-locators. In the example illustrated by Figure 5.2, we observe that an absolute XPath extracted from the YouTube logo in the older website is likely to work also in the newer version of the website since the GUI has a similar appearance. However, we cannot guarantee that the absolute XPath needed to locate the web element is identical among the two versions of the website without looking at the DOM structure. We also note that the History menu item, marked with a black rectangle, has been moved from the third item in the older menu to the fifth item in the newer menu. Therefore, it is likely that the absolute XPath has changed for that web element (since the child index changed, such as, for example, div[3] to div[5]). Any change in the absolute XPath, used by a single-locator, would result in a failed localization attempt and a failed test script. Some studies [95] shows that absolute XPath locators are very fragile since they contain the entire specification of how to traverse the DOM tree, from the root to the target element. However, the other kinds of locators that are more robust (e.g., the ones based on the ID attribute) can be broken by some web app evolution patterns (e.g., a modification to the app's IDs). This happens since they all represent a single point of failure, even if with a lower associate probability w.r.t. absolute XPath. For this reason, considering multi-locators can help further reduce the fragility of the web element localization steps.

5.2.1 Multi-Locator (LML) approach

Leotta et al. proposed the Multi-Locator (LML) approach [99], which, instead of using a single-locator, takes advantage of the results from several single-locators and a voting procedure to combine their outputs and improve the accuracy of locating the correct web element across website's or app's evolution. In the worst case, even one working single-locator might be enough to find the desired web element. This approach is valuable since a more reliable way of locating web elements improves the robustness of test execution which, in turn, reduces the need for script maintenance and thereby cost.

The idea of Leotta et al. is based on the assumption that the various algorithms for the creation of locators have different strengths and weaknesses; they often exhibit complementary performance. For this reason, their approach uses a voting decision procedure to aggregate the results of multiple alternative locators for producing a consolidated locator.

Leotta et al. experimented with four different variants of the voting decision procedure for the LML approach: (1) unweighted worst order, (2) unweighted best order, (3) weighted, and (4) theoretical limit. These variants produce slightly different results. For the unweighted variants (1 and 2), each kind of single-locator is of equal importance (one vote each), and both will only give a different result when more than one candidate receives the same number of votes (a tie). Each kind of locator in the weighted variant (3) is assigned a weight based on resilience to change, i.e., computed on a corpus of web applications for which successive versions are available. Each vote is proportional to that weight. The candidate web element with the highest sum of weighted votes will be selected as the best matching web element. The theoretical limit (4) is a particular case where we assume that the approach can pick the correct web element if any single-locator returns the right web element. As the name suggests, this variant is only possible in theory but is still something to aim for and compare against since it is guaranteed to perform at least as good as the other three (i.e., the best absolute performance achievable with the LML approach). In their study, Leotta et al. confirmed that the weighted LML approach performed better (about 30% fewer broken locators) than the most robust single-locator included in the experiment (i.e., ROBULA+ [100]), thus confirming the hypothesis that using multiple sources of information is valuable for web element localization. As expected, the theoretical limit variant performed the best results, with about 16% fewer broken locators, than the weighted variant. We decided to compare our proposed approach against the theoretical limit variant of the LML approach in our experiment to avoid the

possible bias of selecting or calculating a new set of weights required by the weighted variant.

Even though the LML approach increases the robustness compared to the best of the single-locators with up to 30 percent w.r.t. the state of the art solutions, with our further studies we discovered that, in certain cases, the approach still fails to find a significant number of web elements. As such, further research is warranted since advances in locating the correct web elements impact test script robustness and, thereby, maintenance costs.

5.3 The Similo approach

The similarity-based web element localization (Similo) approach attempts to increase the robustness even further than the LML approach. Similar to the LML approach, Similo tries to take advantage of multiple sources of information instead of just one as a single-locator. When comparing Similo with the LML approach, Similo can take advantage of locators that pinpoint more than one web element, unlike LML, which can only be used with locators that can identify one unique web element. For example, an XPath locator pinpoints an element within the DOM D_1 model by defining a set of predicates on such element properties. The DOM can change during the app evolution (D_2), and the web element of interest can have some of its element properties changed. In such a case, the single-locator returns no web element. Contrary, Similo looks separately at each of the properties (in this paper called locator parameters) of each element in the DOM D_2 model. It returns all web elements that have a partial match. The core functionality of the approach consists of finding the web element among a set of candidate web elements (e.g., web elements extracted from a webpage), which has the most similar locator parameters to the target web element (i.e., desired capabilities). This is achieved by comparing the locator parameters of the target web element (from the DOM D_1) with the locator parameters of each of the candidate web elements (in the DOM D_2). Each comparison results in a similarity score, a sum of the outcomes of the individual comparisons multiplied with a weight. The candidate web element with the highest similarity score is returned as the most similar web element found in the DOM D_2.

Figure 5.3 contains an overview of how locator parameters are compared, weighted, and summarized into a similarity score. A locator parameter can be any feature, visible or non-visible, of the web element, e.g., text, ID, XPath, size, or location. Each locator parameter from the target web element is compared with the corresponding locator parameter in the candidate web element

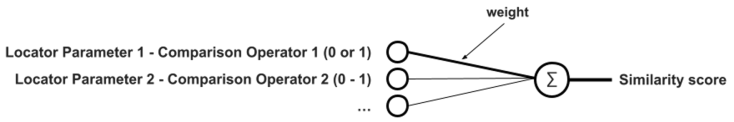

Figure 5.3: Overview of how to calculate the similarity score between two sets of locator parameters.

using a comparison operator. A comparison operator can be any function that can compute the similarity of two locator parameter values and return a value between zero and one (or binary zero or one). Zero if there is no similarity between the compared locator parameters (target and candidate), one when the compared locator parameters are identical, and, if reasonable for the specific operator, a value between zero and one if there is some degree of similarity between the compared locator parameters. The outcome from each comparison is multiplied by its weight (representing the reliability across DOM version of each kind of locator parameter) and summarized into the similarity score. Weights can be based on experience, be calculated, or learned from empirical data. A high similarity score indicates high similarity between the target and candidate web element. When all candidate web elements have been associated with a similarity score, the candidate web element with the highest score is selected as the most similar (i.e., best matching locator parameters) web element.

Algorithm 1 Similo finds only the best candidate

```
Require: targetWebElement
Require: candidateWebElements
  mostSimilarWebElement = null
  highestSimilarityScore = 0
  for all candidateWebElement in candidateWebElements do
    similarityScore = calculateSimilarityScore(targetWebElement, candi-
    dateWebElement)
    if similarityScore > highestSimilarityScore then
      mostSimilarWebElement = candidateWebElement
      highestSimilarityScore = similarityScore
    end if
  end for
  return mostSimilarWebElement
```

Algorithm 2 Similo finds all candidates and ranks them

```
Require: targetWebElement
Require: candidateWebElements
  rankedCandidates = new List()
  for all candidateWebElement in candidateWebElements do
    similarityScore = calculateSimilarityScore(targetWebElement, candidateWebElement)
    rankedCandidates.add(candidateWebElement, similarityScore)
  end for
  sortedCandidates = rankedCandidates.sort(highest similarity score first)
  return sortedCandidates (highest similarity score first)
```

There are, at least, two ways of realizing the Similo approach. The first is to iterate through all the candidate web elements, compare each candidate with the target (i.e., using the locator parameters computed for each web element) to get the similarity score, and remember the candidate with the highest score, as in Algorithm 1. Another way is to calculate a similarity score for all the candidates (i.e., by comparing locator parameters) and then sorting all the candidates based on the similarity score (highest score first), as in Algorithm 2. While the first variant is slightly more efficient (no need to sort the list of candidates), the second variant will not only give us the most similar web element (i.e., highest similarity score) but also the runners-up. A ranked list of similar web elements could be helpful when evaluating or exploring other candidates, e.g., when the most similar web element is not adequate (e.g., a test raises an error following the interaction with such element).

5.3.1 Selecting Locator Parameters and Comparison Operators for Similo

In the study presented by Leotta et al., all the XPath locators were designed to identify single web elements uniquely. However, Similo is not restricted to this behavior; locator parameters that do not identify unique matches can also be used. Hence, the locator parameters selected for the experiment include absolute XPath that can uniquely identify one web element in a webpage and the Tag locator that can only locate one unique web element when there is only one web element with a specific Tag present in the entire webpage.

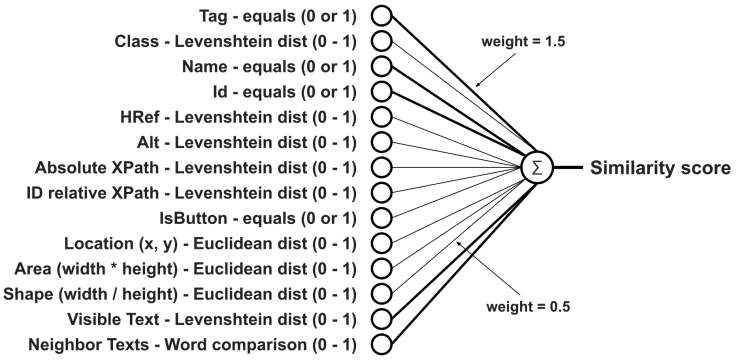

Figure 5.4: Overview of how to calculate the similarity score between two sets of locator parameters in our experiment.

We selected 14 different locator parameters that could be of value when calculating the similarity score and a corresponding comparison operator to use when comparing the locator parameter values. The selected locator parameters aim to cover the majority (with a few exceptions explained below) of commonly used properties from various tools and approaches for web element localization and script repair. Selenium WebDriver API [11] contains eight locator types (id, name, class, tag, link text, partial link text, XPath, and CSS), while Selenium IDE [11] selects the first unique locator from a prioritized list (id, link text, name, and various XPaths). Test script repair tools WATER [49] and COLOR [86] used ten (id, xpath, class, linkText, name, tagname, coord, clickable, visible, zindex, and hash) and nineteen properties respectively (id, class, name, value, type, tag name, alt, src, href, size, onclick, height, width, XPath, X-axis, Y-axis, link text, label, and image) when suggesting a repair for the broken script. WATER and COLOR are further described in Related Work (Section 4.2).

Table 5.1 contains a mapping of locator parameters used by the four approaches to the selection used by Similo. We decided to use DOM properties only in this study, leaving out the image hash (respectively called hash in WATER [49] and image in COLOR [86]) created from the pictorial user interface [24] (i.e., the page rendering produced by the browser) for two reasons: (1) the pictorial user interface is not present in the DOM and could not be generated by our Javascript function that extracts locator parameters (as said is generated by the browser and so could show minor differences across browsers or even different versions of the same browser); (2) taking a screenshot of each of

Table 5.1: Mapping of locator parameters.

Similo	Selenium WebDriver	Selenium IDE	WATER	COLOR
Tag	tag	-	tagname	tag name
Class	class	-	class	class
Name	name	name	name	name
Id	id	id	id	id
HRef	-	-	-	href
Alt	-	-	-	alt
Absolute XPath	XPath	XPath	xpath	XPath
ID rel. XPath	XPath	XPath	xpath	XPath
IsButton	-	-	clickable	onclick
Location (x,y)	-	-	coord	X-axis+Y-axis
Area (w*h)	-	-	-	size
Shape (w/h)	-	-	-	-
Visible Text	text + partial link text	link text	linkText	link text+label
Neighbor Texts	-	-	-	-
-	-	-	hash	image
[all visible]	-	-	visible	-
[all in front]	-	-	zindex	-
-	-	-	-	type
-	-	-	-	src

the web elements (required for comparing all the possible candidate elements) would have taken a significant amount of time (a few tenths of a second per screenshot), reducing the time efficiency of Similo by orders of magnitude. The locator parameters and their corresponding comparison operators are visualized in Figure 5.4. We decided to use the Java String method equalsIgnoreCase (denoted equals) to compare some of the selected locator parameters (e.g., Tag, Id, Name, and IsButton) since they are only similar when the compared values are identical. While Tag, Id, and Name are commonly used attributes in a web element, the IsButton parameter (inspired by the clickable property in WATER and onclick in COLOR) was calculated to the value true or false based on the attributes Tag, Type, and Class.

View the replication package [10] for details on calculating the value of the IsButton locator parameter. Texts, links, and XPaths (e.g., Class, HRef, Alt, Absolute XPath, ID relative XPath, and Visible Text) were compared using Levenshtein distance (normalized and inverted to get the similarity) since they could be similar even if the compared locator parameters are not identical. Visible Text was constructed by extracting the first non-blank text from the Text, Value, and Placeholder (in that order) attributes of the web element. We did not include the type and scr properties from COLOR since they are only applicable to some types of elements. We used Euclidean distance (normalized and inverted to get the similarity) for comparing the area and shape of the web ele-

ments since width and height is likely to remain unchanged in between software releases according to the COLOR study by Kirinuki et al. [86]. Area was calculated by multiplying the width with the height and shape by dividing the width by the height. We decided to use the Euclidean distance (normalized and inverted) between the upper and left location of the compared web elements since a web element is likely to be close to its original position on the screen (again, based on the study by Kirinuki et al.). The Location comparison returns one when the web elements have the same location, zero when the distance exceeds 100 pixels, and a value between zero and one linear to the difference in distance. Web browsers interpret margins slightly differently, resulting in different coordinates for the same web element. The value of 100 pixels was chosen based on our experience with GUI-based test automation to give some flexibility in the browser layout of the web elements. Neighbor Texts contain a space-separated text of words collected from the visible text of nearby web elements (including the target or candidate web element). Instead of comparing Neighbor Texts using Levenshtein distance, we assumed to get a better result by comparing how many words the compared locator parameters have in common since the words gathered from the neighbor texts are unordered (all web elements have a different set of neighbors) making the Levenshtein distance less useful. For example, web element A has two neighbors with the texts: "OK" and "Cancel", resulting in the Neighbor Text: "OK Cancel". Web element B has three neighbors: "Cancel", "Help", and "OK," resulting in the Neighbor Text: "Cancel Help OK". Calculating the similarity using Levenshtein distance results in the value 0.21, while the similarity is 0.66 when using the word count since two out of three words are present. As an example, five common words out of 10 possible would result in a value of 0.5. We did not include WATER's visible and zindex properties since Similo only uses visible web elements. Note that all distance functions have been normalized (zero to one) and inverted (1 - normalized distance) when calculating the similarity. We refer to the replication package [10] for further details on implementing the comparison operators and extracting the locator parameters from the web elements. We want to stress that Similo can use any selection of locator parameters, comparison operators, and weights and that the choice (illustrated in Figure 5.4) will impact the results. For this paper, we did initial experiments and selected ones that give generally robust results. Future work should perform more systematic experiments to understand the effect of these choices.

Table 5.2: Locator parameters in newer and older version of the YouTube.com website.

	Newer YouTube.com version	**Older YouTube.com version**	**Simil.**	**Weight**
Tag:	SPAN	SPAN	1	1.5
Text:	History	History	1	1.5
XPath:	/html[1]/body[1]/ytd-app[1]/div[1]/ytd-mini-guide-renderer[1]/div[1]/ytd-mini-guide-entry-renderer[5]/a[1]/span[1]	/html[1]/body[1]/div[4]/div[4]/div[1]/div[1]/div[1]/div[1]/div[1]/ul[1]/li[1]/div[1]/ul[1]/li[3]/a[1]/span[1]/span[2]/span[1]	0.41	1
ID-based XPath:	id("content")/ytd-mini-guide-renderer[1]/div[1]/ytd-mini-guide-entry-renderer[5]/a[1]/span[1]	id("history-guide-item")/a[1]/span[1]/span[2]/span[1]	0.33	1
Class:	title style-scope ytd-mini-guide-entry-renderer		0	1

5.3.2 Selecting Weights for Similo

We initially assigned all the weights of the 14 locator parameters to the value one. Next, we divided the locator parameters into two groups. We placed the locator parameters that are (according to the COLOR study by Kirinuki et al. [86] more stable (i.e., less likely to break between software releases) in the first group and the remaining in the second group. The first group contains the locator parameters Name, Id, Visible Text, and Neighbor Texts since they got the highest weights (based on a combination of stability and uniqueness) in the COLOR study. The locator parameter Tag was also added to the first group since it has high stability, according to the COLOR study. All the other locators parameters were placed in the second group. Finally, we added 0.5 to the locator parameter weights in the first group and removed 0.5 from the locator parameter weights in the second group. The resulting locator property weights are illustrated in Figure 5.4. Bold connector lines represent a weight of 1.5, and the thinner lines represent a weight of 0.5. We realize that it would likely be possible to attain more optimal locator parameter weights. Still, we assumed a simple approach (motivated by prior work in the industry) would be sufficient for the experiment and so to evaluate the effectiveness of the Similo approach.

5.3.3 Example of Calculating a Similarity Score

As an example of how to create a similarity score, five locator parameters extracted from the History menu button (indicated by a black rectangle) in Figure 5.2 are listed in Table 5.2. We note that the Tag and Text parameters are identical in both website versions. XPath and ID-based XPath have, however, changed between versions, and the Class parameter was unassigned in the older version of the website. In this example, we assume using the Levenshtein distance as a comparison operator for all the locator parameters. We use the normalized version of the Levenshtein distance (GLD NED_2) in this paper as defined by Yujian et al. [169], and the similarity is calculated as the inverse (1 - distance) of the normalized Levenshtein distance that returns a value between zero and one. The comparison operator would return one when comparing the newer and older version of the Tag parameter since they are identical (SPAN). We get the same result when comparing the Text parameters in both versions since they are also identical (History). Comparing both versions of the XPath parameter would result in a value between zero and one since the XPaths in both versions begin and end in the same way, even though they are not identical. The comparison result is zero when comparing the Class parameters (both versions) since they have nothing in common (the older version is blank). Assuming that (1) the comparison operator returns the similarity specified in the Similarity column and (2) we use the weights from the Weight column in Table 5.2; the resulting similarity score, computed between the old target element and a possible candidate from the new page, would be 3.74 computed as (1 * 1.5 + 1 * 1.5 + 0.41 + 0.33 + 0).

5.4 Experimental study

This section presents the research design, the research questions, the research procedure of the performed empirical study. The first objective of the experiment is to evaluate the difference in robustness between Similo and the baseline approach by comparing the ratio of located and non-located web elements in two different releases of the same webpage. We used public webpages for the experiment where changes to the pages include addition, change, and removal of web element attributes and web elements. As a secondary objective, we evaluated the efficiency of Similo to make sure that its performance is viable for practical use.

5.4.1 Research Questions

The study aims to answer the following research questions:

- **RQ1:** What is the robustness (measured as the ratio between located and non-located web elements) of the Similo approach compared to the baseline LML approach?

- **RQ2:** How well does the Similo approach perform in terms of time efficiency?

The first research question (RQ1) is answered by looking at the ratio of located and non-located web elements on two different versions of 48 websites (801 web elements in total). We figured that 801 web elements in 48 websites would be enough since a previous study, performed by Leotta et al., evaluated the LML approach using six websites and a total of 675 web element locators [99]. Similo is more robust than the baseline if Similo can correctly locate more web elements than the baseline approach. Research question 2 (RQ2) is answered by measuring the execution time of locating web elements using Similo. The time efficiency of Similo would be acceptable if Similo could be of practical use for the industry. An order of magnitude lower average execution time (to locate one web element using Similo) than the expected execution time of a typical test step (as part of a test case) would likely be sufficient since the gain in robustness would likely outweigh the loss in time efficiency.

5.4.2 Selecting the Baseline Approaches

Our previous literature review revealed several different approaches and algorithms targeting the problem of robust localization of web elements [121]. You can find more information about some of the approaches and algorithms in Related Work (Section 4.2). Two approaches stood out as the most predominant (i.e., the most robust). The first one was the Multi-Locator approach proposed by Leotta et al. (LML), and the second one was the ATA approach proposed by Thummalapenta et al. [151], later refined by Yandrapally et al. [166] (now going by the name ATA-QV).

We decided to compare our proposed approach with LML, but we also planned to use the ATA-QV [166] algorithm as a baseline in our empirical evaluation since its contextual clues share some similarities with both the LML approach and our proposed approach. However, the ATA-QV system is not

openly available[1] and not trivial to implement based on the descriptions in the papers that presented it. For this reason, we contacted the authors of the ATA-QV paper and, with their helpful guidance, tried to re-implement its core elements. However, when trying our implementation, we could not prove (due to the lack of an oracle) that our version performed to a level consistent with the original experiments, and we had to exclude the re-implementation from our experiments.

5.4.3 Selecting single-Locators for LML

Leotta et al. used five XPath locators in a previous experiment [99]. The locators were: absolute XPath, relative ID-based XPath, Selenium IDE, Montoto, and Robula+. In consultation with two of the original authors, also co-authors of this paper, we decided to use the same selection of locators in our experiment.

We decided to use a similar, but not identical, implementation of the XPath locators as the ones used by Leotta et al. since we intended to automatically generate all the XPath locators using Java code instead of manually creating them from the browser to reduce some effort. Instead of relying on the (discontinued) FirePath browser plugin [98], we created the corresponding JavaScript code for generating both absolute and relative ID-based XPaths. The TypeScript code for ROBULA+, publicly available online[2], was manually translated into JavaScript code [100]. We created JavaScript code for generating the XPath locator proposed by Montoto et al. from the pseudocode presented in their work [117]. Since the algorithm proposed by Montoto et al. is not guaranteed to result in a unique XPath, we decided to ignore that locator when this instance occurred and instead focus on the four remaining locators for the LML approach. We constructed the Selenium IDE [12] locator based on the open-source code publicly available in GitHub [6]. The Javascript source code for all XPath generators was too large to include in this paper (about 400 lines of Javascript code) but is available in the replication package [10].

[1] It is under copyright of a commercial entity and it was not clear if we could get full access to and use the source code in a reasonable time (or ever). The full implementation is extensive, encompassing more than 10K lines of code and is no longer maintained; it is thus questionable if we would be able to compile and execute it without considerable investment of additional time.

[2] https://github.com/cyluxx/robula-plus/blob/master/README.md

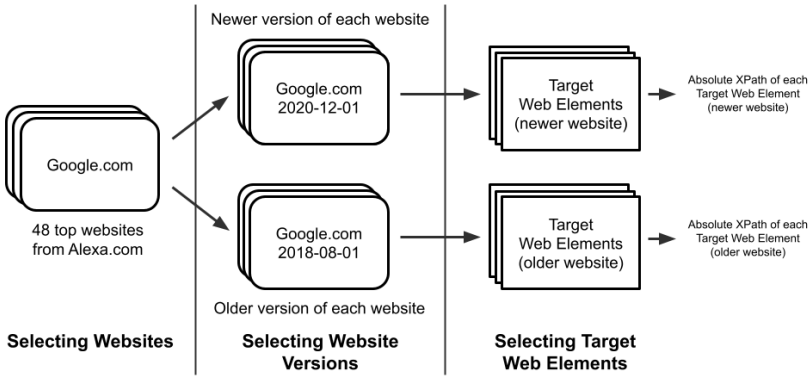

Figure 5.5: Selection of websites, website versions, and target web elements. The older version was selected as the closest version in the archive that was a random number of months, sampled in the range of 12 to 60 months old.

5.4.4 Selecting Websites

Figure 5.5 visualizes the procedure that we adopted to select websites, website versions, and target web elements for the experiment, further detailed in this and the following sections.

Alexa.com is a site that ranks websites based on global traffic [1]. The rank is calculated from unique visitors and page-views over the past three months. We selected the top-rated websites in the United States for our experiment to avoid websites that default to a language that we, the authors, could not fully understand (e.g., Chinese or Russian). The benefit of selecting websites from Alexa.com is that the top-rated websites are well known and that the selection is unbiased since we have no control over the ranked websites. Still, they represent commonly used websites that, most likely, have extensive testing to ensure consistent quality to its many users. Another benefit of selecting websites with heavy use and traffic is that older versions of these sites are more likely to be available on archiving sites; see further on this aspect below. We selected the top 50 websites (publicly listed on Alexa.com without a paid subscription) with two exceptions; (1) the website Force.com was excluded since the browser forwarded it to the included website Salesforce.com (both URLs

point to the same website), and (2) the website Chaturbate.com was excluded since the first page warned the visitor about adult content resulting in a total of 48 included websites. Chaturbate.com was excluded based on two motivations: (1) the URL did not point to the actual homepage, and (2) a website with adult content goes against the ethical guidelines prohibiting adult or discriminating content.

5.4.5 Selecting Website Versions

To determine the robustness of each approach, two versions of the 48 websites from Alexa's list were required for the experiment, i.e., to determine if the web elements in the newer version could be located using the locator parameters extracted from the web elements in the older version. The Internet Archive website [7] was used to acquire the versions since it stores previous versions of a large selection of websites. The later versions of the websites were acquired in December of 2020 within a span of a few days, primarily affected by the sampled web sites availability on the Internet Archive.

In the previous work by Leotta et al., the time difference between versions of the subject websites was 12 to 60 months and about 36 months on average. We decided to replicate this design and sampled the older websites using a random number, R, in the interval of 12 to 60 months backward in time for each website. Specifically, we sampled the version of the website available on the archive site and as close to R months older than the newer version. Versions that were exactly R months older could not always be acquired since the Internet Archive does not back-up the websites daily.

5.4.6 Selecting Target Web Elements

We manually selected target web elements from each of the 48 website homepages that: (1) were possible to perform actions on (e.g., anchors, buttons, menu items, input fields, text fields, check-boxes, and radio-buttons); (2) can be used for assertions or synchronization (e.g., top-level headlines); (3) belong to core functionality of the website homepage; (4) are present in both versions of the website homepage. A homepage is, in this context, the start webpage, in a website, loaded by the browser when using the URL extracted from Alexa.com. Figure 5.2 shows an example where target web elements in the newer (to the left) and older (to the right) versions of the YouTube.com website are indicated with rectangles of the same color. Each rectangle indicates one target web element. Note that these images were generated manually for the purpose of showing the

reader examples of changes that can occur on a website. Hence, the images are not outputs from Similo, nor essential to the approach in any way.

We generated an absolute XPath for each target web element in the older and newer versions of each website. The XPath from the older website version will be used when retrieving the web element used for generating single-locators and extracting locator parameters for Similo. We will use the XPath from the newer version as an oracle to verify that the correct web element was located.

The number of target web elements selected (801 in total) from the 48 websites are listed in Table 5.3 along with the date of the older and newer website versions and the number of randomly chosen months between the releases.

Note that the number of selected target web elements ranges between two and 45. Some homepages are very similar between versions, while others are completely redesigned with almost nothing in common between the older and newer versions. While the Internet Archive provides us with a convenient way of retrieving and comparing different website versions, a drawback with this service is that the websites are static (frozen in time). As such, there is no guarantee that the websites will respond to interaction (e.g., clicking a link) in the same way as a dynamic website (not frozen in time). To mitigate the risk of issues due to the static behavior, we used only web elements from the homepage (start page) of each website. This design choice poses a potential threat to our study since the web elements on the homepage might not contain the complete variety of tags as the entire website. We created a list of tag names that we expected to find in a good enough sample of websites to address this threat. The list included the following tags: input, button, select, a, h1, h2, h3, h4, h5, li, span, div, p, th, tr, td, label, svg. We gathered this list of commonly used tags from our previous experience of extracting web elements from websites [120]. Next, we extracted and counted the tag names of all the web elements for each application included in the study. We discovered that the only tag that is not represented by our sample of applications is the th tag, that five websites use the related td tag, and the tr tag is used by three. One possible explanation for the lack of th tags might be that the th tag is no longer needed since modern websites use style sheets when formatting the appearance of tables. As such, we concluded that only one out of 18 (5.6%) of the tags were unrepresented in the sample, which was considered reasonable for continued evaluation.

5.4.7 Locating Web Elements

Until this step, the preparations were performed manually. Still, this final step, to try to locate all target web elements in the newer version of the website, was

Table 5.3: The number of target web elements selected from the older and newer versions of each website.

Website	Months	Older version	Newer version	Target elements
Adobe.com	28	2018-07-02	2020-11-02	2
Aliexpress.com	44	2017-04-01	2020-12-01	39
Amazon.com	44	2017-04-02	2020-12-01	10
Apple.com	38	2017-10-02	2020-12-01	10
Bestbuy.com	32	2018-04-02	2020-12-01	40
Bing.com	12	2019-12-01	2020-12-01	5
Chase.com	24	2018-12-02	2020-12-02	31
Cnn.com	32	2018-04-02	2020-12-01	16
Craigslist.com	56	2016-03-31	2020-12-02	45
Dropbox.com	12	2019-12-02	2020-12-04	10
Ebay.com	30	2018-06-01	2020-12-02	25
Espn.com	12	2019-12-01	2020-12-02	23
Etsy.com	14	2019-10-02	2020-12-01	13
Facebook.com	49	2016-11-01	2020-12-01	25
Fidelity.com	35	2018-01-02	2020-12-02	19
Foxnews.com	35	2018-01-01	2020-12-01	29
Google.com	28	2018-08-01	2020-12-01	20
Hulu.com	12	2019-12-01	2020-12-02	2
Imdb.com	18	2019-06-02	2020-12-01	12
Indeed.com	60	2015-12-02	2020-12-01	13
Instagram.com	30	2018-06-02	2020-12-02	15
Instructure.com	37	2017-11-01	2020-12-02	2
Intuit.com	20	2019-04-01	2020-12-02	11
Linkedin.com	40	2017-08-02	2020-12-02	4
Live.com	40	2017-08-01	2020-12-01	3
Microsoft.com	54	2016-06-02	2020-12-01	4
Microsoftonline.com	16	2019-08-02	2020-12-01	6
Myshopify.com	59	2016-01-01	2020-12-01	2
Netflix.com	50	2016-10-03	2020-12-01	3
Nytimes.com	24	2018-12-01	2020-12-02	23
Office.com	21	2019-03-01	2020-12-01	12
Okta.com	37	2017-11-02	2020-12-04	10
Paypal.com	17	2019-07-02	2020-12-02	23
Reddit.com	45	2017-03-04	2020-12-01	30
Salesforce.com	22	2019-02-01	2020-12-01	17
Spotify.com	38	2017-10-01	2020-12-01	17
Target.com	42	2017-06-02	2020-12-01	17
Twitch.tv	57	2016-03-01	2020-12-02	5
Twitter.com	41	2017-07-02	2020-12-01	20
Ups.com	41	2017-06-29	2020-11-27	29
Usps.com	36	2017-12-02	2020-12-01	36
Walmart.com	39	2017-09-02	2020-12-01	7
Wellsfargo.com	14	2019-10-02	2020-12-01	35
Wikipedia.org	39	2017-09-01	2020-12-02	39
Yahoo.com	25	2018-11-01	2020-12-02	17
Youtube.com	16	2019-08-01	2020-12-01	8
Zillow.com	40	2017-08-01	2020-12-01	9
Zoom.us	55	2016-05-01	2020-12-02	8

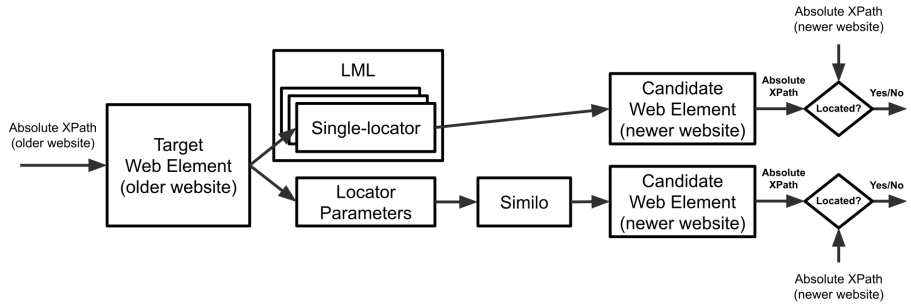

Figure 5.6: The process of locating a candidate web element from the absolute XPath of a target web element.

executed automatically using Java code to improve the accuracy and speed of the experiment. We initially intended to run all the 40 websites at once but decided to execute one website at a time since the Internet Archive website is slow and unreliable. This design choice makes it possible to rerun a website in case of a browser timeout.

Figure 5.6 shows the process of locating a candidate web element, in the newer version of a website, from the absolute XPath of each target web element, in the older version of the same website. First, the target web element, in the older version of the website, is located from its absolute XPath retrieved manually (described in Section 5.4.6). A collection of five single-locators (absolute XPath, relative ID-based XPath, Selenium IDE, Montoto, and Robula+) are then created from the identified target web element. The voting mechanism in LML uses this collection of single-locators. Next, each single-locator is executed in the newer version of the website, trying to locate the correct candidate web element. For each single-locator that identifies precisely one candidate web element, the absolute XPath of the identified candidate web element is compared to the correct absolute XPath (i.e., the oracle that is the absolute XPath of the web element in the newer version that actually corresponds to the target web element in the older version of the same website) previously retrieved in Section 5.4.6. The theoretical limit version of LML, which we compare to Similo, is successful (i.e., located) if any of the five single-locators can identify the correct web element.

Fourteen different locator parameters for Similo are also created from the target web element. The approach and the locator parameters for Similo are described in Section 5.3 and 5.3.1. Similo compares all fourteen locators pa-

Table 5.4: Description of the localization result.

Localization result	Description
Located	The localization approach is able to pick the correct candidate web element with an XPath matching the oracle.
Non-Located	The localization approach is unable to find a match among the candidate web elements, or it finds a match among the candidate web elements, but the XPath is not matching the oracle.

rameters of the target web element with the corresponding locator parameters in each candidate web element and returns the most similar candidate (the one with the highest similarity score). As with LML, the XPath of the most similar candidate web element is compared to the correct absolute XPath to evaluate if the target web element has been (correctly) located or not.

Table 5.4 contains a summary of the two possible outcomes after a localization attempt. The absolute XPath of the candidate web element is compared with the absolute XPath of the correct target web element (i.e., the oracle) using string comparison. Since a modified web element in a webpage can result in a slightly altered XPath, even though it is still the same visual GUI component, we decided to add some tolerance in the string comparison. Two identical XPaths are, of course, considered a match. We also decided that two XPaths match if only one element has been added (or removed) at the end of the XPath. In our case, the XPath: "/html[1]/body[1]/main[1]/section[1]/ul[1]/li[1]/a[1]" matches the XPath: "/html[1]/body[1]/ main[1]/section[1]/ul[1]/li[1]" but not the XPath:
"/html[1]/body[1]/main[1]/section[1]/ul[1]". Using a tolerance in the XPath comparison is not as reliable as a manual oracle. Still, it is faster and unbiased since the outcome of both approaches is validated in the same way.

To provide an example that highlights the complexity of the oracle and the choice of our comparison strategy, we examined the YouTube logo marked with a blue rectangle in Figure 5.2 in more detail. The HTML code from the YouTube logo in both the older and newer version of the YouTube.com homepage is listed in Listing 5.2. Let's assume that we manually selected the anchor tags (a) in the older and newer versions as the target web elements. A locator approach that can use information extracted from the older version of the HTML DOM to locate the anchor in the newer version is successful in locating the target web

element. But is it also correct if the locator approach identified the "div" element inside the anchor (in the newer version) as the best matching web element? Such a situation could happen with Similo since the "div" element has many locator parameters in common with the target anchor that we are trying to locate. We note that the "id" and "class" attributes share some similarities with the target anchor in the older version. Also, the location, size, and shape are likely to be very similar. It might be possible for the "div" element to get an even higher similarity score than the anchor (in the newer version) and would therefore be selected as the most similar candidate web element. By simply comparing the Absolute XPaths, the "div" element would not be correctly located unless we allow some tolerance in the comparison, as in our selected oracle. The tolerant comparison method was used when comparing an XPath with the correct XPath (the oracle) for all the single-locators, LML, and Similo.

```
1  <!-- YouTube logo in the older version of YouTube.com: -->
2  <a class="masthead-logo-renderer yt-uix-sessionlink" id="logo-container"
       title="YouTube Home" href="/web/20190802000022...">
3    <span title="YouTube Home" class="logo masthead-logo-renderer-logo yt-
         sprite"></span>
4  </a>

6  <!-- YouTube logo in the newer version of YouTube.com: -->
7  <a class="yt-simple-endpoint style-scope ytd-topbar-logo-renderer" id="
       logo" title="YouTube Home" href="/web/20201201235946mp...">
8    <div id="logo-icon-container" class="yt-icon-container style-scope ytd-
         topbar-logo-renderer">
9      <svg class="style-scope ytd-topbar-logo-renderer"...</svg>
10   </div>
11 </a>
```

Listing 5.2: HTML extracted from the YouTube logo in the older and newer version of YouTube.com.

5.5 Results

In this section we present the results of the experimental study we conducted by answering the two research questions.

5.5.1 RQ1 - Robustness

Table 5.5 contains the number of located and non-located web elements for all the single-locators, LML, and Similo. As can be seen from the Table, Similo

Table 5.5: The total number of located and non-located web elements for all websites.

Locator	Located	Non-Located	Non-Located %
Absolute XPath	136	665	83
Relative ID-based XPath	326	475	59
Selenium IDE	394	407	51
Montoto	422	379	47
Robula+	490	311	39
LML (theoretical limit)	587	214	27
Similo	710	91	11

failed to locate 11% of the web elements while the theoretical limit variant of LML failed to locate 27% out of 801 target web elements.

Robula+ was the most robust of the single-locators (39% non-located), while absolute XPath was the least robust (83% non-located). This result correlates well with the results gathered in the experiment performed by Leotta et al. that also concluded that Robula+ was the most robust single-locator and absolute XPath the least robust. The only notable deviation between the studies was that the Montoto locator was slightly more reliable (47% non-located) than the Selenium IDE locator (51% non-located) in our experiment. In contrast, the case was the opposite (i.e., Selenium IDE performed better) in the study performed by Leotta et al. Detailed results per website can be found in the replication package [10]. Similo performed better (more located web elements than LML) for 32 of the websites, LML worked better in four cases, and there was a tie when using the 12 remaining websites. LML was only able to locate one additional web element for all the websites where LML performed better.

Manual Analysis of one Failing Case

To better understand why the Similo approach can, in some cases, fail to locate the correct web element, we studied a simplified example from the Aliexpress.com website. Table 5.6 shows a comparison of the locator parameter values for the target web element, the selected candidate, and the correct candidate that we chose to use as an example. We decided to include four locator parameters (Tag, Visible Text, Absolute XPath, and ID-based XPath), compare all the locator parameters using Levenshtein distance, and use the same weight for all locator parameters. Note that we truncated the beginning of the Absolute

Table 5.6: Comparison of the locator values for the target, the selected candidate, and the correct candidate web elements. The locator values were extracted from the Aliexpress.com website.

Web element	Tag	Visible Text	XPath	ID-based Xpath
Target	A	Home & Garden	.../div[4]/div[1]/div[1]/div[2]/div[1]/div[2]/dl[7]/dt[1]/span[1]/a[1]	.../div[1]/div[2]/div[1]/div[2]/dl[7]/dt[1]/span[1]/a[1]
Selected candidate	A	Home Improvement	.../div[5]/div[1]/div[2]/div[1]/div[2]/div[1]/div[2]/dl[13]/dt[1]/span[1]/a[1]	.../div[2]/div[1]/div[2]/div[1]/div[2]/dl[13]/dt[1]/span[1]/a[1]
Correct candidate	A	Home	.../div[5]/div[1]/div[2]/div[1]/div[2]/div[1]/div[2]/dl[7]/dt[1]/span[1]/a[1]	.../div[2]/div[1]/div[2]/div[1]/div[2]/dl[7]/dt[1]/span[1]/a[1]

Table 5.7: The similarity (between 0 and 100) when comparing the target with the selected, and correct web element locator values.

Locator	Selected cand. similarity	Correct cand. similarity
Tag	1	1
Visible Text	0.43	0.30
XPath	0.89	0.91
ID-based XPath	0.89	0.91
Total similarity:	3.21	3.12

XPath and ID-based XPath values to save space in the Table since the leading path was identical in all cases.

In the simplified example, Similo located the web element on the second row in the table instead of the third row that is the correct one (according to the oracle). When comparing the XPath, ID-based XPath, and Text values, we note that none of the selected or correct candidates are identical to the target web element. The only locator parameter value they all have in common is the Tag name (the candidates are all anchors).

Table 5.7 presents the similarity when comparing the locator parameter values using Levenshtein distance with a weight of one. The Tag locator receives the value 1 since all the values are identical. Both the XPath and ID-based XPath values of the target locator parameters are slightly more similar to the correct candidate (0.91 vs. 0.89). Still, the difference was not big enough to compensate for the fact that the Levenshtein distance function evaluated that the Visible Text "Home & Garden" is more similar to "Home Improvement" than to "Home" (0.43 vs. 0.30).

In summary, the candidate selected by Similo got a similarity score of 3.21, while the correct candidate got only 3.12, resulting in an incorrect match.

To study how the distribution of similarity scores differs between candidate web elements, we picked a random application (Ups.com, using a digital dice). We extracted the calculated similarity scores between each target element and all the candidates. Figure 5.7 contains a scatter plot of similarity scores when

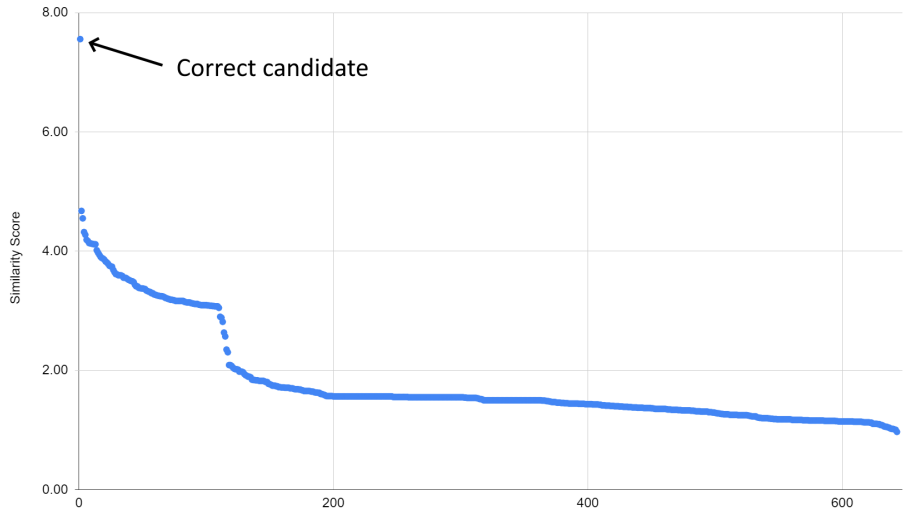

Figure 5.7: Similarity scores in a scatter plot containing all candidate web elements and the correctly located one on the Ups.com website.

comparing the first of the correctly located targets with all of the candidates. The scatter plot shows that the similarity score of the correctly located target (7.6) is separated from the remaining similarity scores (less than 4.7). All the correctly located targets follow the same pattern (clearly separated from the rest) with one exception where two similarity scores were close but separated from the remaining similarity scores. Next, we analyzed another target element and picked the first of the incorrectly located targets, visualized in Figure 5.8. In this case, the correct candidate web element (according to the human oracle) only got the fifth highest similarity score. We note that the correct candidate web element was not separated from the remaining candidates.

To summarize, for what concerns research question RQ1, we can say that, for the considered applications, the adoption of Similo results in a significant reduction (from 27% down to 11%) of the number of broken locators, which is expected to be associated with a corresponding reduction of the maintenance effort required to repair the test scripts using such broken locators.

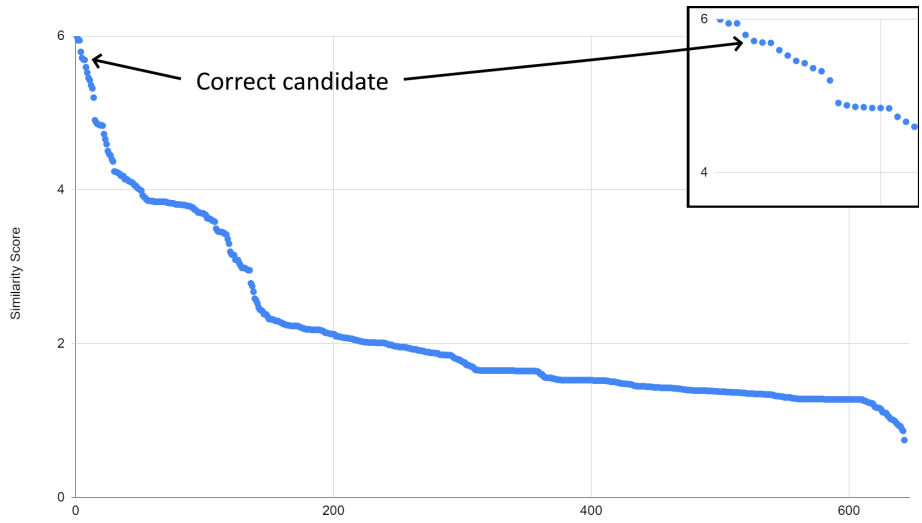

Figure 5.8: Similarity scores in a scatter plot containing all candidate web elements and the incorrectly located one on the Ups.com website.

5.5.2 RQ2 - Performance

Table 5.8 shows the average time to locate all the target web elements among the candidates using Similo on ten websites (randomly selected from the 48 included websites using a digital dice). The performance measurements were clocked on a Windows machine with an AMD Ryzen 9 3900X processor equipped with 12 cores running 20 simultaneous threads (out of 24) at 3.79 GHz. In this experiment, we used 20 threads only, leaving some threads available for other tasks, e.g., our development environment and the operating system, to avoid being interrupted. The "Locate all targets" column contains the total time (in milliseconds) to locate all the targets among the available candidates. Time per target is calculated by dividing "Locate all targets" by "Target count". We calculated the average time to locate one target web element using Similo to be four milliseconds on this machine (based on all the 48 included websites). As we can see in Table 5.8, localizing a target web element typically take more time when there are more candidate web elements. This result is natural since the Similo approach (described in Section 5.3) compares the target with each candidate to identify the best match resulting in a longer time to locate a target

Table 5.8: The average time (in milliseconds) to locate a target among the candidates on ten randomly selected websites.

Website	Locate all (ms)	Target count	Cand. count	Time/target (ms)
Apple.com	27	10	344	2.70
Netflix.com	12	3	157	4.00
Espn.com	110	23	1066	4.78
Amazon.com	53	10	1762	5.30
Fidelity.com	55	19	498	2.89
Paypal.com	62	23	192	2.70
Instagram.com	32	15	107	2.13
Reddit.com	225	30	1223	7.50
Usps.com	170	36	583	4.72
Twitter.com	57	20	68	2.85

when there are more candidates. Note that we have not included standard deviations in Table 5.8 since the Similo calculation is performed in memory isolated from other threads or network delays resulting in a predictable result and, therefore, a negligible standard deviation.

To summarise, with respect to the research question RQ2 we can say that the time required by Similo for selecting a web element is undoubtedly acceptable (in the order of milliseconds).

5.6 Discussion

The result that the Similo approach is more robust than the baseline approach is promising since it indicates that such novel approach can lower the maintenance cost of web-based test automation by reducing the manual effort to repair broken test automation scripts. The Similo approach can, for instance, be used as a foundation to create tools and frameworks for web-based test automation that can execute tests more reliably without sacrificing performance. With a more robust automated test execution, the human testers could focus on other tasks, e.g., test strategies or exploratory testing, instead of putting a lot of the effort into script maintenance. Tools or frameworks that rely on Similo could also aid the human tester by storing and automatically repairing all the web element locators, thus reducing the manual labor when application changes occur.

The average time to locate one web element in the evaluated websites is roughly four milliseconds using the Similo approach. This search time should be compared to the time for performing GUI interactions, which for automated GUI tests is typically measured in the order of hundreds of milliseconds or

even seconds. Also, for websites, network latency, etc., can be considerably larger than on the order of milliseconds. As an example, Mahmud et al. [108] report that in their study, automated tests are about six times faster to execute compared to a manual tester but that it still takes about ten minutes to run a test script with an average size of 45 test steps (13 seconds per test step in average). Therefore, we do not believe the Similo approach's performance will be a limitation for either future academic research or industrial application.

However, no solution comes without drawbacks. In the case of the Similo approach, see Section 5.3, one possible drawback is that it will always return a matching web element as long as there are candidates unless a threshold is used that rejects matches with a lower similarity score. Without a threshold, the Similo approach will return a matching but incorrect web element even if the target web element is not yet present or available on the webpage. This is a typical case of the synchronization problem that can occur on websites due to latency where parts of the webpage are loaded faster than other parts. The consequence is that not all elements are available for localization, increasing the chance of a faulty web element being matched with the target. Hence, using the Similo approach, a test script could attempt to perform the next action in a scenario on an incorrect web element instead of waiting for the correct target web element to appear/load. This problem, to synchronize the test execution with the webpage (or AUT), is also present for the LML approach and is a well-known challenge present in GUI-based testing in general [33, 36, 37, 57, 66, 76, 90, 115]. A possible solution to this challenge is to use a threshold, representing the lowest acceptable similarity score a target web element must have. For example, suppose that the threshold is not achieved with the current candidate web elements when expected to include a correct match. In this case, we can draw the likely conclusion that not all web elements have been loaded, triggering a rerun of the localization. If the rerun fails and the threshold still not obtained, we can continue to rerun the search or conclude that no matching elements are available, e.g., due to website failure. However, this solution presents some additional questions, for instance, how many times should the approach search and how much time should it wait between searches? We cover optimization of weights and other possible enhancements in Section 4.7.

Regardless, defining a suitable value for the threshold is non-trivial. If the threshold is set too high, that might eliminate valid matches, and if it is set too low, incorrect matches may be chosen due to the aforementioned synchronization challenge. The challenge is present during test execution of scripted test sequences, where dynamic waits are appropriate to minimize total test execution time, and therefore a common challenge that warrants more research.

Despite not resolving all challenges of effectiveness and efficiency, we claim that Similo shows great potential for improving web-based testing in industrial practice. The approach is currently implemented in Java and can, with minimal effort, be integrated into existing Java-based Selenium test suites. A possible solution would be to create a plugin that uses Similo to locate a Selenium WebElement given a set of locator parameters. Furthermore, the approach is agnostic to the AUT's programming language and implementation details, as long as there is an available GUI structure (e.g., the Android SDK or the Windows accessibility framework). While evaluating Similo on mobile and desktop applications would be interesting for future work, this research focused only on evaluating the effectiveness and efficiency of Similo on websites.

Similo can be used to improve the robustness of test execution (of websites) by making it tolerable to minor changes to the web elements and thereby mitigating locator maintenance costs. However, this additional robustness also presents an interesting but very relevant challenge. By increasing the robustness of the localization, the test scripts can identify web elements that have been, to some extent, or even significantly, modified. This tolerance helps the scripts carry out their purpose of testing the application on a scenario-level of abstraction. However, at the same time, this makes the scripts less sensitive to unintentional or faulty changes to the web elements that could, as an example, cause erroneous behaviors when the AUT is fully integrated with other applications or services. Hence, while Similo provides more robust scenario-based test execution, from a human perspective, it lowers the script's capabilities of finding technical issues such as incorrect tags, IDs, etc. However, this trade-off is considered acceptable since the purpose of most scenario-based GUI tests is to test the user scenarios and not the correctness of the GUIs architecture/implementation.

In summary, Similo utilizes the triangulation of multiple locator information to identify correct web elements. The approach is shown to be more effective at finding elements than the baseline solution and efficient enough for practical use. The approach thereby advances baseline but does not fully solve the problem of perfect element localization. Further research is warranted in the area, which should also investigate what locator parameters to use, how to weight locators, set suitable threshold values to evaluate if the approach can mitigate the synchronization challenge.

5.7 Threats to Validity

Selecting the target web elements to locate in our experiment, i.e., establishing the "ground truth" for our experiments, is a threat to the internal validity. We tried to minimize this threat by selecting all the web elements present on both versions of the website homepage that belonged to specified categories of web elements. However, some of the websites were redesigned or had almost nothing in common with the older version, resulting in few web elements in common between versions. We decided to include redesigned websites or websites containing few web elements since that occurred in public websites and is a realistic scenario. The choice of locator parameters included in our study is also a possible threat since we used 14 locator parameters with the Similo approach and only five localization algorithms with LML. We, however, consider this to be a realistic scenario since Similo supports a wide range of locator parameters while LML only works with locators that can identify unique matches. To reduce this threat to the internal validity, for LML, we decided to use the same selection of locators in our experiment as well as adopted in the paper [99] and in consultation with the original LML authors (also authors of this work).

The applications and versions selected for the study might also pose a threat to external validity. We decided to pick the top 48 sites from Alexa.com to reduce this threat since we have no control over the websites listed on that site. The website versions selected affect the number of failed localization attempts since a long time between two releases is likely to contain more changes. We reduced this threat by selecting the same interval (one to five years) between website versions as Leotta et al. [99] and picking a random number for each website that specifies the time in months between versions.

That we only selected web elements from the homepage (the start page) of each of the websites can also pose a threat since the homepage might not contain the same distribution of web elements as the entire website. To reduce this threat, we extracted and counted all the web element tags in all the homepages to check if any of the most commonly used tags were missing in our sample of homepages. We concluded that only one of the tags was unrepresented and that this was considered reasonable.

The choice of web element types to extract from the homepages could pose a threat to the validity of our study. Similo, and the LML approaches are, however, agnostic to web element types. For the approaches, the tag name and the attributes of a web element are just a collection of parameters that should be compared. Therefore, we argue that this design choice has a minor impact only on this study's external validity.

Since the similarity score is highly dependent on the weights chosen for the comparison, illustrated in Figure 5.4, they also affect the results. We decided to compare Similo against the theoretical limit variant of the LML approach and used only two different weight values (0.5 and 1.5) to mitigate this threat, even if we might have got an even better result by comparing with the weighted variant of LML and optimized the weights for Similo. Therefore, we perceive that the experiment was conducted non-optimally (from the Similo perspective) and, therefore, the results obtained are underestimations. This conclusion is logical as previous work on weight optimization has provided better results with optimized weights [86]. This actually highlights the high effectiveness and potential of the Similo approach.

5.8 Related Work

Although there are no established proposals, the problem of maintaining and evolving test scripts is well considered in both industry and academia. In practice, two categories of approaches, opposite but not mutually exclusive, exist: approaches that apply post-repair techniques when a locator fails to select the correct locator and others, more preventive, aiming to generate robust locators.

5.8.1 Automatic repair of broken locators

A category of approaches that shares the same goal of Similo, i.e., reducing the overall test suite maintenance effort, is the one based on automatic repair of test scripts and in particular of broken locators. This category has been investigated by various researchers (e.g., [49, 75, 86] and the contained approaches are often based on algorithms similar to those used to generate robust locators.

In this category we found WATER, proposed by Choudhary et al. [49], a tool-based approach able to repair web application test scripts. The authors of this paper claimed that test scripts mainly break for three reasons: structural changes (i.e., related to the DOM tree), content changes (i.e., attribute or web page changes), and blind changes (related to server-side changes). The approach is based on the concept of differential testing, i.e., comparing the execution of the test scripts on two different releases: one where test cases fail and one where they pass. Even though WATER is designed for script repair, the underlying algorithm contains an algorithm used to locate the most similar web element based on weighted locator parameters. Differently from us, the algorithm only considers six locator parameters (XPath, coord, clickable, visible, index, and

hash), where XPath's are compared using Levenshtein distance [8] and the rest of the locator parameters are considered correct only if their values are (exactly) equal. The similarity check is used when selecting the best web element among elements identified using id, XPath, class, linkText, or name when there is more than one candidate.

Another representative of this category is WATERFALL [75], built starting from WATER, but which improves its idea and effectiveness. The algorithm implemented in WATERFALL is based, similarly to WATER, on differential testing, and uses exactly the same heuristics to executes the repairs. However, it does take into account the intermediate minor versions occurring between two major releases of a web application. This modification to the original idea has improved its effectiveness, as shown by the experiments conducted (209% improvement of the number of correct repairs being suggested).

Recently, a novel tool, named COLOR, for repairing broken locators have been proposed by Kirinuki et al. [86]. The approach considers various properties such as attributes, positions, texts, and images to propose a repair. From an experiment conducted by the authors it can be seen that COLOR is more effective w.r.t. complex changes (e.g., page layout changes) than WATER, the state of the art tool in this context. Results shows that COLOR ranks the correct locator with a 77% - 93% accuracy.

Erratum is the name of another recent approach proposed by Brisset et al. [42] that utilizes a DOM tree matching algorithm to repair broken locators in a website. Their results indicate that Erratum has a 67% better accuracy of locator repair than WATER.

GUI DifferEntiator (GUIDE) is a non-intrusive, platform-, and language-independent tool proposed by Grechanik et al. [70, 163] that can identify changes in the GUI trees of two successive releases of a web application. Testers can use the tool to identify DOM changes to provide guidance or estimates for test planning and repair.

As already mentioned Similo does not belong to this category of approaches that carry out the repair of locators but approaches the same problem in a preventive way.

5.8.2 Generation of robust locators

The problem of generating robust locators is considered mainly in the context of information retrieval and data mining, for extracting information from semi-structured sources (e.g, XML and HTML pages). However, the same problem is also relevant in the context of automated browsing of web applications and in the

context of automated E2E testing for web application. In this section, we limit ourselves to this last context but the previous ones are also very important and often the techniques proposed in the testing field have been produced starting from the first ones.

Several algorithms for generating robust locators (we will call it single-locator generation algorithms to differentiate it from the concept of multi-locator) have been proposed in the literature.

Among these algorithms we find that of *Montoto et al.* [117]. This algorithm generates XPath change-resilient expressions iteratively, following a bottom-up strategy. It starts from a simple XPath expression and then extend it by concatenating sub-expressions until a target element is identified. First, the algorithm tries to identify the target element using text and the value of its attributes. Then, if the generated XPath is not a unique locator, its ancestors and the value of their attributes are considered one after the other until the root is reached.

Other algorithms for generating robust XPath's are ROBULA [98] and ROBULA+ [100], proposed by Leotta et al. The ROBULA+ algorithm [100] (ROBULA was its antecedent) is considered the state of the art algorithm for automatically generating robust XPath expressions. The intuition behind ROBULA+ is simple and effective: to combine XPath properties using ad-hoc heuristics in order to maintain the locators as short as possible and so robust. The algorithm, similarly to the one proposed by *Montoto et al.*, produces the locators iteratively starting from the most generic XPath locator that selects all nodes in the DOM tree (//*). Subsequently, it refines the generated XPath expression until only the element of interest is selected. In such iterative refinement, ROBULA+ applies a set of transformations, according to a set of specialisation steps, prioritisation and black listing techniques.

Another approach for increasing the robustness of a locator is to not only consider the attributes of the target web element, but also its neighboring web elements, as proposed by *Yandrapally et al.* [166]. Using neighbour information, a web element can be partly or entirely located based on the attributes of the neighboring web element through an approach similar to triangulation. As an example, assume that the web page contains a text field with a label above it. In this case, the text field can still be located even if it is replaced, given that the label above it can still be identified. The work by *Yandrapally et al.* is a suggested enhancement (called ATA-QV) to the technique and tool proposed by *Thummalapenta et al.* [151], simply called ATA. ATA is a commercial tool developed at IBM that aims to improve the robustness of locating web elements compared to using absolute XPath's by associating web elements with neigh-

boring labels. The idea underlying the tool is that robustness of locator can be pursued by relying more on labels (i.e., the visual landmarks) and less on page structure. When there is more than one web element with the same label, ATA uses an XPath complemented with additional attributes, such as index or class, and can, in many cases, relocate web elements even when they moved in the subsequent version or their attributes changed.

These last two approaches (ATA and ATA-QV) are promising for generating robust locators because they take advantage of multiple aspects of the representation of the application under test and eliminate almost entirely the usage of the web page structure. Although ATA is a promising approach, differently from Similo, it uses only one locator parameter (a text label) that will only result in a match when there is one unique label on the web page and when the label names match exactly. This drawback can be reduced by using contextual clues, as proposed by *Yandrapally et al.*, making the localization more tolerant to changes since it might be possible to locate the web element based on the labels in surrounding web elements.

Recently, some commercial state-of-art testing tools — such as e.g., *Testim*[3] and *Ranorex*[4] — apply and use locators generation algorithms based on Artificial Intelligence (AI) to improve robustness. This seems to be the new frontier in the context of E2E testing of Web applications and the results are promising, as evidenced also by some recently proposed academic papers (SIDEREAL tool [96] and the algorithm proposed by *Nguyen et al.* [128]). SIDEREAL [96] is a statistical adaptive algorithm able to learn the potential fragility of HTML properties from previous versions of the application under test and thus producing robust locators specific to a given web application. SIDEREAL, based on the property of adaptivity that distinguishes it, outperforms ROBULA+'s heuristics in terms of robustness. The other recent generation algorithm has been proposed by *Nguyen et al.* [128] and is based on a combination of two methods: a new XPath construction method and a rule-based selection method of the 'best XPath' for a target element. The former method uses the semantic structure of a Web page as starting point to build neighbor-based XPaths. Similarly to ATA, it also relies on textual presentation that is visible to users.

Similo is inspired by these related works, combining several technical solutions to improve locator robustness. In common with the LML approach, Similo tries to take advantage of multiple sources of information instead of just one as single-locator algorithms. Unique to the Similo approach is that it collects loca-

[3]https://www.testim.io/blog/why-testim/
[4]https://www.ranorex.com/blog/machine-trained-algorithm/

tor parameters from all visible web elements on a web page before making any comparisons. This information allows neighboring web element information to be used similarly to the the approach proposed by *Yandrapally et al.* Additionally, our approach allows all locator parameters to be compared, weighted, and tallied into a combined similarity score for each web element compared against all candidate web elements to find the best suitable match. Our approach also enables using a threshold value to filter how similar candidate web elements have to be considered a match. In contrast, for instance, the LML approach returns a set of candidate web elements that could all match the target web element. However, since the LML approach does not provide any additional information (other than the locator weight), it's more challenging to determine which of the candidates is the most probable match.

5.9 Conclusions and Future Work

Test script fragility, caused by unreliable localization of web elements, is one of the dominant challenges in GUI test automation [141]. We propose a novel approach, Similo, that identifies the web element, from a set of candidate web elements, with the highest similarity to the locator parameters of the target web element. We compared the robustness and performance of Similo against the baseline approach, identified in the multi-locator presented by Leotta et. [99]. Experimental results show that Similo only failed to correctly locate 91 out of a set of 801 web elements, while the baseline approach was unable to locate 214 of the web elements from the same set. The time needed to locate one web element was roughly 4 ms for Similo and should not be a major performance problem when executing GUI-based test scripts since the time to perform a test case is typically measured in the order of seconds.

A benefit of the Similo approach is that we can use any locator parameters regardless of the locator is able to uniquely identify a web element or not. We used fourteen locator parameters in this experiment, but we might have got an even better result with a more extensive set of locator parameters. However, more research is needed to identify the locator parameters that give the best contribution to the robustness.

The locator parameters, comparison operators, and weights (here referred to as properties) selected for the empirical study are all merely initial selections and values. We emphasize that we do not claim the properties to be optimal for any website. While the experiment shows that the properties selected resulted in less failed localization attempts, that does not mean that we cannot find an

even better set of properties. A more optimized set of properties would perhaps result in fewer failed localization attempts and a higher margin between the best matching web element and the second best, making the Similo approach even more robust. In future research, we aim to optimize the properties used in this study to improve the Similo approach results. This research includes looking at the possibility of dynamically adjusted weights and comparison methods using feedback-based optimization. Such techniques are considered suitable since the optimization of this problem is perceived to be context-dependent, i.e., the best combination of properties may be unique to each application.

In this paper, we have deliberately ignored the problem that it takes some time to transition from one application state to another after performing an action (e.g., a mouse click). The test execution needs to wait for the next application state to avoid the potential risk of fetching the candidate web elements before they are all available. Failing to fetch the complete set of candidate web elements might cause a script that relies on the Similo approach to fail if the correct web element is not present among the candidate web elements.

In conclusion, Similo, inspired by previous works, has been shown in this study to provide more robust web element localization with perceived suitable execution time to make the solution applicable in practice. Additionally, several possible improvements to the approach are discussed, and we outline future research based on these ideas. Hence, future research is warranted in this area to continue to address the fundamental challenges with GUI testing regarding robust web element localization and synchronization. In this study, the focus has been on websites only. Still, we see no reason why Similo cannot be used on any application with a GUI (e.g., desktop or mobile apps), not just websites, where locator parameters can be extracted from GUI widgets and used for localization.

5.10 Acknowledgements

This work was supported by the KKS foundation through the S.E.R.T. Research Profile project at Blekinge Institute of Technology. Robert Feldt has also been supported by the Swedish Scientific Council (No. 2015-04913, 'Basing Software Testing on Information Theory' and No. 2020-05272, 'Automated boundary testing for QUality of AI/ML modelS').

We want to sincerely thank Rahul Krishna Yandrapally and Saurabh Sinha, two of the authors of the ATA-QV paper [166], who went to considerable lengths to support us in trying to re-implement their approach.

Chapter 6

Robust Web Element Identification for Evolving Applications by Considering Visual Overlaps

Abstract

testFragile (i.e., non-robust) test execution is a common challenge for automated GUI-based testing of web applications as they evolve. Despite recent progress, there is still room for improvement since test execution failures caused by technical limitations result in unnecessary maintenance costs that limit its effectiveness and efficiency. One of the most reported technical challenges for web-based tests concerns how to reliably locate a web element used by a test script.

This paper proposes the novel concept of Visually Overlapping Nodes (VON) that reduces fragility by utilizing the phenomenon that visual web elements (observed by the user) are constructed from multiple web-elements in the Document Object Model (DOM) that overlaps visually.

We demonstrate the approach in a tool, VON Similo, which extends the state-of-the-art multi-locator approach (Similo) that is also used as the baseline for an experiment. In the experiment, a ground truth set of 1163 manually

collected web element pairs, from different releases of the 40 most popular web applications on the internet, are used to compare the approaches' precision, recall, and accuracy.

Our results show that VON Similo provides 94.7% accuracy in identifying a web element in a new release of the same SUT. In comparison, Similo provides 83.8% accuracy.

These results demonstrate the applicability of the visually overlapping nodes concept/tool for web element localization in evolving web applications and contribute a novel way of thinking about web element localization in future research on GUI-based testing.

Keywords: component, formatting, style, styling, insert

6.1 Introduction

In modern software engineering, test automation is a key activity, where automated tests are used to continuously monitor the software's quality and provide frequent feedback to developers [108]. However, much of this automation has been restricted to lower-level testing such as unit and integration tests [131]. Higher level testing, particularly with graphical user interface (GUI) tests, are still mostly manual and therefore a costly activity in practice [71, 73].

While GUI tests can be used to verify the correctness of the GUI's appearance, the focus of many GUI tests is on verifying functional correctness of the system under test (SUT) [30], i.e. system testing through the SUT's GUI [39]. However, despite continued research since the 1980s, several key challenges remain and limit the widespread adoption of automated GUI testing [121]. One of these challenges is the robust identification of GUI elements. This issue has been described for many domains, and several approaches have been proposed to increase the robustness of GUI element localization [98, 100, 117, 166]. Robustness is, in this instance, defined as the correct identification of a web element when it is available and reporting no match when a web element is unavailable. This property is particularly important as automated web application tests are typically used for regression testing as software systems, and their GUI elements, change and evolve [61].

Despite its importance, research has had marginal success in solving the challenge of robust GUI element localization [121]. Instead, much research has been focused on extending GUI testing technologies, only utilizing the already available web element localization solutions—example of such extensions are test generation and GUI ripping [127, 154]. Some research, however, has been

made investigating new types of locators, e.g., image recognition [168], or multi-locators [100]. However, we still consider web element localization an unsolved challenge [121], warranting more research to improve the general robustness and maintainability of available GUI testing techniques and tools.

Nass et al. proposed an approach called similarity-based web element localization (Similo) [122], which calculates a weighted similarity score between the target web element in a previous version, and all web elements (candidates), in a revised version, of a web application. The target web element contains the desired properties (e.g., attributes) that are compared with each candidate. They compared the Similo approach with the multi-locator approach proposed by Leotta et al. [99] and found that Similo can correctly locate more target web elements than the multi-locator when evaluating web elements extracted from 40 commonly used homepages.

This study presents the novel concept of Visually Overlapping Nodes (VON). The concept makes use of the structure of how modern web applications are constructed (i.e., as hierarchies of components with specific attributes and characteristics) and how this structure is formalized in the document object model (DOM) [160]. Our investigations show that multiple DOM nodes often point to the same visual element in the rendered GUI, implying that any of these nodes, if found to be a match to the sought visual element, will constitute a valid match if used for a graphic validation of the SUT. DOM nodes (typically arranged in a hierarchy) that represent the same visual element (e.g., a button or menu item) are, in this paper, referred to as visually overlapping nodes (VON). As an example, a menu item that is represented by an anchor tag (<a>) containing two span tags () in the DOM. All three DOM nodes visually appear as the same element (as observed by the user) even though they are three separate nodes. The benefit of this approach is that it increases the chance to find at least one good match for a provided target, thus leading to more robust web element identification.

While many other essential activities need to be in place for robust GUI-based test automation, we here focus on the vital aspect of GUI web element identification. If the identification is not robust, later steps of the chain cannot compensate for this. This focus is visualized in Figure 6.1.

As such, the main contributions of this work are:

- Insights into the relative power of different web element attributes for web element localization;
- A generally applicable, yet novel, concept called Visually Overlapping Nodes (VON);

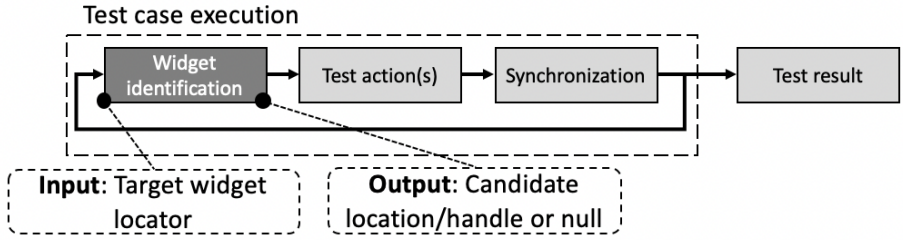

Figure 6.1: Graphical representation of the GUI test case execution process, highlighting the step (web element identification) that is studied in this work.

- An improved version of similarity-based web element localization (Similo) that implements VON (VON Similo).

The continuation of this paper is structured as follows. First, in Section 6.2, we present related work and give more detailed descriptions of the technologies evaluated in this study. Section 6.4 then explains the study's experimental setup. We present the results in Section 6.5.1, which describes the findings of our experiment. These findings are discussed in Section 6.6 before the paper is concluded in Section 6.7.

6.2 Background and Related Work

In the GUI-based web application testing discipline, a web-element locator is defined as a method, function, approach or algorithm that can locate a web element in a given web page according to a specific parameter. State-of-the-art techniques and tools typically make use of conditions on the attributes of the web elements in the HTML DOM tree (e.g., ids, attributes, or class names). Popular testing tools (e.g., Selenium) provide the possibility to use XPath or CSS expressions to locate elements on the web page [44]. We refer to these types of locators as *single-locators*.

Single-locators are a reported source of *fragility* for test suites. Fragility is defined as the lack of robustness of the locators to changes in the GUI definition of the SUT. Test script fragility typically manifests through test cases that fail not because of functional misbehaviors in the SUT but because of the inability of the testing engine to find the web elements needed during the test sequences

using existing locators [53]. When a locator cannot be found in the DOM tree of the current web page, it is typically referred to as a *Broken locator* [99].

Given the increased complexity and high pace of change of modern web applications, it is highly likely that the attributes or the XPaths of web elements are modified between two versions of the same web application. If single-locators are used, any such change would result in failed localization attempts and require additional effort by the testers to repair the broken test suites.

Leotta et al. proposed the multi-locator approach [99] to limit the fragility of single-locators in web application testing. The approach evaluates multiple locators and uses a voting procedure between the single-locators to improve the accuracy of locating the correct web element in a web-page, thereby improving the robustness and reducing the script maintenance cost.

Similo is another approach of multi-locators based on a weighted similarity score computed on the differences between the locator parameters of the web element to locate and each of the web elements on the current web page [122]. Unlike the multi-locator approach proposed by Leotta et al. [99], which uses five single-locators that uniquely identify precisely one (or none) web element each, Similo combine comparisons on multiple attributes in a single score. Such score can then be used to rank the possible candidate widgets for the current target, thereby finding multiple possible alternatives instead of a single result as Leotta's multilocator. The algorithm can then select as valid matches all the candidates for which the score is above a given threshold, or select the highest-rated candidate as the single match for the target.

Similo can take advantage of locators that pinpoint more than one web element. For example, Similo can use an absolute XPath that pinpoints exactly one web element in the DOM, but unlike the multi-locator proposed by Leotta et al., Similo can also use the CSS selector ".name" to find all web elements that contain a specific class name even if the query results in several matching web elements.

In the Similo study [122], currently available as a preprint at arXiv, the authors used 14 different locator parameters with corresponding comparison operators and weights summarized in Figure 6.2. The locator parameters Tag, Class, Name, Id, HRef, Alt, XPath and ID relative XPath, Location, Visible Text are collected directly from the DOM-tree. IsButton is a derivate boolean parameter, that was set to true or false according to the values of the attributes Tag, Type, and Class. Neighbor Texts contain a space-separated text of words collected from the visible text of nearby web elements. Specific comparison operators that return a value between zero and one (or exactly zero or one) were selected for each locator parameter. Some locator parameters were compared

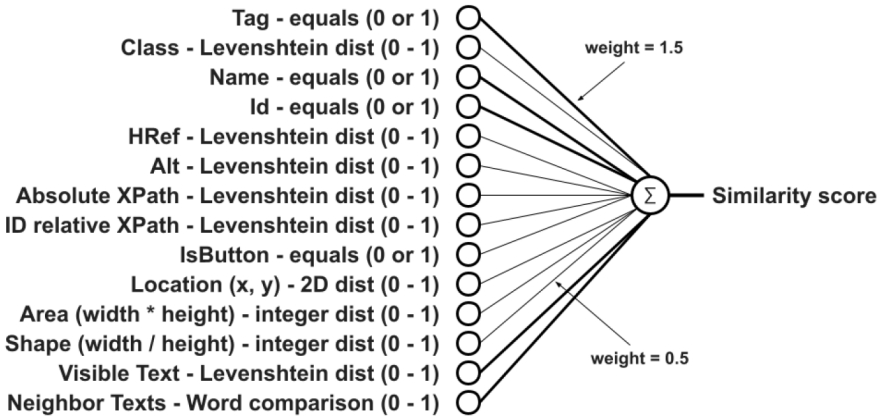

Figure 6.2: Graphical representation of the computation of similarity score between two different sets of locator parameters.

by the Java method equalsIgnoreCase (e.g., Tag, Name, and Id). Others were compared using Levenshtein distance, word comparison, or Euclidean distance. Detailed information about the comparison operators can be found in the Similo paper [122].

The weights for each locator parameter (and comparison operator) were assigned based on their respective stability and uniqueness found by the COLOR study by Kirinuki at al. that used a similar selection of locator parameters when evaluating four open-source web applications [86]. The COLOR approach considers various properties such as attributes, positions, texts, and images to propose a repair unlike Similo that approaches the same problem in a preventive way (i.e., before the script failure occurs). In Similo, the locator parameters were divided into two groups based on the corresponding weights from the COLOR study. Locator parameters with a higher stability and uniqueness were assigned the weight value 1.5 (bold lines in Figure 6.2) and the remaining were assigned the value 0.5 (thin lines in the graphical representation).

6.3 Visually overlapping nodes approach

In this section, we propose an approach to enhance the current state of the art in multi-locators for web application testing with an improved version of similarity-

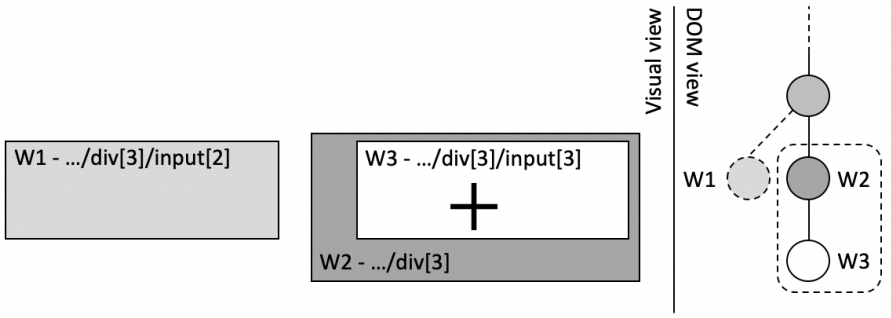

Figure 6.3: A visualization of a hierarchy of web elements represented both visually and from a DOM perspective. It shows that although W2 and W3 are unique entities, they appear to be the same visual component or, at least, overlap visually.

based web element localization (Similo) that implements visually overlapping nodes (VON Similo).

A characteristic of modern web applications is that they are built from elements that contain other elements, e.g., a text label can be contained in a button that is itself contained in a div tag, which in turn may be placed in another container, such as a frame. This hierarchical structure is modeled within the DOM, which is used by web browsers and GUI testing tools to render, identify or interact with elements. From a DOM perspective, each element is considered a unique entity, as it is described by a DOM node identifiable through a unique absolute XPath. However, from a visual perspective, multiple nodes — due to size, placement, and content — represent the same visual element, e.g., a button. This feature of modern web applications thereby implies the existence of a many-to-one connection between DOM nodes and visual elements. In this paper, all the DOM nodes that belong to the same visual element are referred to as visually overlapping nodes (VON).

Fig. 6.3 visualizes this many-to-one correspondence, showing how web elements can be contained in other web elements yet, from a visual perspective, occupy the same area of the screen. In this case, while web elements W2 and W3 are represented with different XPaths in the DOM, both visually point to the same (or nearby) screen area or web element. This phenomenon implies that both web elements (W2 and W3) are equally correct candidate elements

(i.e., elements available on the current web page) for a target element (i.e., the element that contains the properties that we are looking for) pointing to that screen area. We refer to this phenomenon as visually overlapping nodes (VON), where property-based localization approaches, like Similo, can utilize VON to increase the number of correctly located candidate web elements. Hence, rather than relying on finding one specific DOM node, or absolute XPath, any located DOM node that belongs to the same visual element is deemed a correct match. This approach mitigates test execution fragility by, as mentioned, increasing the number of candidate DOM nodes that can be matched when identifying one and the same web element. Essentially, the approach will be more robust as long as only one or a subset of these nodes change between two revisions.

Formally, we define the approach to identify equivalent web elements as:

Given a web element W1, we define the set of equivalent web elements e_0, ... e_N as the set of web elements that satisfy the following properties:

1. The ratio between the overlapping areas of the web elements on the screen, and the union of the areas of the two web elements, is higher than a set threshold value. Such ratio is computed as,

$$\frac{\cap(R_1, R_2)}{\cup(R_1, R_2)}$$

   where: R_1 and R_2 are the rectangles occupied on the screen by the two web elements; the set intersection symbol indicates the size (in pixels) of the common area occupied on screen by R_1 and R_2, and the set union symbol indicates the size (in pixels) of the union of R_1 and R_2.

   By experimenting with different threshold values to identify visually overlapping nodes, we finally selected 0.85 as a value for the threshold that allowed us to avoid a definition of visual overlap that was too loose (thereby considering visually separate GUI elements as overlapping) or too strict (thereby finding none or a few visually overlapping nodes).

2. The center of the web element W1 is contained in the rectangle R1.

The addition of the VON concept transforms the locator parameters (e.g., tag. id, or xpath) that are stored and utilized by the Similo algorithm. Each locator parameter value is no longer a single value but is instead substituted by a list of equivalent values collected from all the visually overlapping web elements of a DOM node.

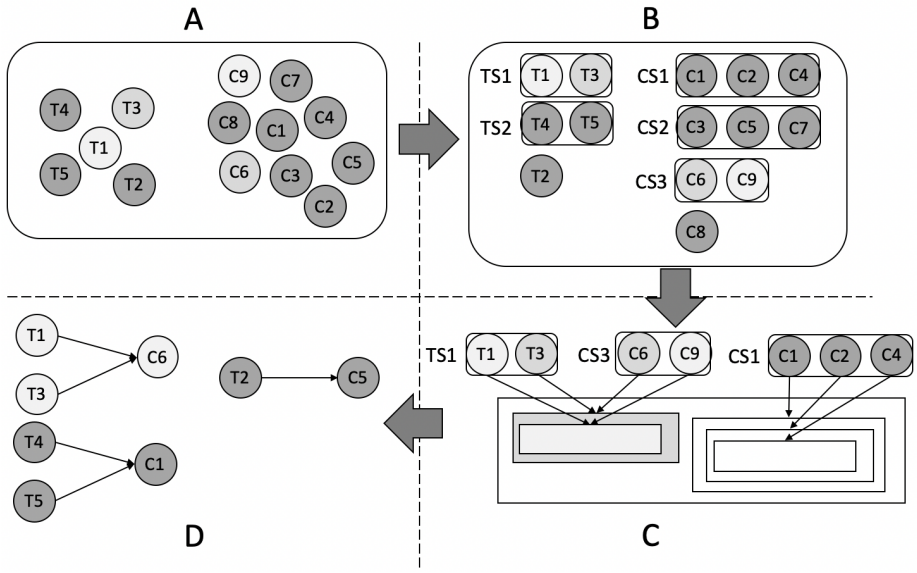

Figure 6.4: Visualization of how visually overlapping nodes are implemented in VON Similo.

The VON Similo score substitutes the comparison functions of the original Similo, with the following function:

Given the set ew_1 (set of web elements equivalent to w_1: $e_{1.1}$, $e_{1.2}$, ... $e_{1.N}$) and ew_2 (set of web elements equivalent to w_2: $e_{2.1}$, $e_{2.2}$, ... $e_{2.M}$), and the values for a specific attribute of the equivalent web elements, VON Similo computes the similarity between all the possible combinations of visually overlapping web elements for a specific attribute. The maximum of these similarity values is the VON Similo similarity score for that attribute in the comparison, representing the best possible match to the target web element. However, while the match may not be the target web element, this approach ensures that the coordinates of the match overlap with the intended target. As such, from a visual perspective, the target element is identified.

Figure 6.4 presents a visualization of the process of how visually overlapping nodes are utilized in VON Similo. In the first step, denoted A in the figure, a set of target web elements (denoted $T_x \in TS$) and candidate web elements (denoted $C_y \in CS$) are available, where $|TS| \leq |CS|$. In the second step, denoted B in

the figure, a pre-analysis of TS and CS is performed, clustering all target and candidate web elements according to the visual web elements they are associated with, using the formula presented previously in this section. The outcome of the pre-analysis are clusters TS_1-TS_i and CS_1-CS_j containing components with overlapping target areas on the screen but otherwise with an unknown overlap in terms of locator properties. In step 3, denoted C in the figure, each target web element $T_x \in TS_i$ is compared to every other candidate web element $C_y \in CS_j$, and a similarity score is calculated. After the comparison, the maximum similarity score of each cluster is kept and associated with all target web elements T_n-$T_m \in TS_i$, resulting in a mapping between T_i and C_j as visualized in the last step of the figure, denoted D. This mapping implies that (from a DOM perspective) a given target web element T_i may not be mapped to the candidate component C_i=T_i that was initially used when the target was set in the previous version of the SUT. In Figure 6.4, T_1 is equivalent to C_9 but is mapped to its parent component C_6. However, from a visual perspective, this is irrelevant since both C_6 and C_9 point to a visual area that is overlapping with the one containing T_1.

The benefits of this approach is that the number of valid matching candidates increases, implying that for clusters where a target web element can not be mapped to a suitable candidate, a candidate can still be associated. This increases test robustness by reducing the number of failed test actions caused by minor changes to components that would otherwise be considered false positive test results. One could argue that this approach would increase the number of false positives, or false negatives, by mapping targets to incorrect candidates. However, as we will show, no such trends were identified.

6.4 Empirical Evaluation

In this section, we report the goal, the research questions, the methodology, and the results of the empirical evaluation we performed when comparing VON Similo to the baseline. Similo was selected as the baseline since a previous study [122] (currently only available as a preprint) showed that it was able to locate more web elements when compared to the multi-locator approach proposed by Leotta et al. [99].

6.4.1 Goal

The study's high-level goal is to evaluate the efficiency of VON-based web element locators when applied to web applications. The results are interpreted from the perspective of software testing procedures needing methods to automatically locate web elements in evolving GUIs with high levels of accuracy.

To perform a comparison with the state-of-the-art, we performed an analysis of the accuracy and effectiveness for the base Similo algorithm, that was not performed in the original paper. We also complement the original study with an analysis of the aptness of DOM attributes for widget localization.

6.4.2 Research Questions and Metrics

- **RQ1**: Which are the most frequently used element attributes in web applications?
- **RQ2**: What is the effectiveness of similarity-based web element localization (Similo)?
- **RQ3**: What is the effectiveness of VON-based web element localization (VON Similo)?
- **RQ4**: Which type of multi-locator (Similo or VON Similo) performs better in terms of accuracy?

The objective of RQ1 is to corroborate the selection of an optimal set of attributes for the multi-locator approach employed in Similo (attributes shown in Figure 6.2). We chose these attributes based on related literature. Still, it is unknown how often the attributes are populated with data on a generic web page and, thereby, what value they give to the employed calculation of similarity. It is further unknown if any attribute, which could provide value, is not part of this set. To answer RQ1, we compute two different metrics: the *Relative Non-null* values for each attribute and the *Variability* of each attribute.

The objective of RQ2 is to extend and complement the original study performed on Similo [122]. Our purpose is to compute Accuracy, Precision, and Recall for the original similarity-based web element localization approach, to serve as a baseline for our further evaluations. Real-world web applications are used, where we can observe minor and significant changes to the site's appearance and functionality. The standard Accuracy, Precision, and Recall measures are calculated based on the use of a human oracle by mapping target web elements from the old version of the web application to the new web application.

The objective of RQ3 is to evaluate the accuracy with which VON Similo can locate the correct web element (from a visual perspective), given a target web element, on a new release of the same web application. To answer RQ3, we resort again to measuring three standard measures, Accuracy, Precision, and Recall.

Finally, the objective of RQ4, as a side-objective of the measurement of the Accuracy, Precision, and Recall provided by both types of locators, is to provide a comparison between them.

6.4.3 Methodology

This section presents the steps we took to evaluate the research questions in a controlled experiment.

A replication package is available as an open-source repository [1].

We will present threats to the validity of the study design in Section 6.6.1.

1. *Application Collection*: To gather a set of experimental subjects for our evaluations, we collected pairs of versions of the same webpage inside the same web application. The objective was to emulate the maintenance of web application by using target web elements from an old site version and identifying them on a new version.

   To avoid biased selections, we selected the 40 top-rated web application in the United States from the Alexa ranking web application[2][3]. Since the list contained one case of mirrored sites (different URLs leading to the same web application), we used only one of them. We also excluded one web application with adult content due to ethical reasons. The final list of 38 web applications can be found in the replication package.

2. *Application Version Selection*: The Internet Archive web application was used[4], to acquire the old version of the web application chosen in the previous step. To further reduce potential biases, we choose to replicate a design employed by Leotta et al., selecting as the newer version (*v2*) of each web application a version published in December 2020 and as the older version (*v1*) a version dated R months backward in time (with R

[1] https://figshare.com/s/e63c3679e925730397ac

[2] http://www.alexa.com

[3] Since the experiment, the web application (i.e., alexa.com) has been discontinued and is no longer publicly available.

[4] http://web.archive.org

randomly varying between 12 and 60 months, 36 months on average). This choice ensured enough time had elapsed between releases to see graphical and functional differences between the two versions of the web application, enabling us to evaluate the web element finding ability of the approach over time for both minor and significant changes. We perceive minor changes to have higher construct validity for regular operations in practice. Still, we do not want to exclude more significant changes, which can occur when companies re-brand or make more extensive technological updates to their web applications.

3. *Application scraping*: We developed a Selenium-based web scraper to analyze the distribution of the most commonly used web elements and attributes and their distribution among web applications. The scraper collects all attribute values for all web elements for a web page and stores them for further analysis. This scraping is achieved by a script that cycles all web elements in the DOM tree and extracts values defined in the W3C list of HTML attributes. The scraped information is stored in CSV files.

   Based on the scraped information, we were able to collect the following statistics to answer RQ1: (i) the number of occurrences of non-empty attributes, i.e., different than "" (empty) or *null*; (ii) the variability of the values, i.e., the ratio between the number of different values divided by the total number of non-null, non-empty values. We recognize that the latter metric has more or less inherent variability for specific attributes, e.g., labels. Still, since we did not look at the content for this evaluation, only the variation in content, we perceive this effect to be minor.

4. *Correspondent web element Selection*: To acquire target web elements and oracles for the automated evaluation, we manually selected web elements from each of the 40 web application homepages. We only selected elements from the start page since the Internet archive only stores static pages, meaning that javascript, databases, etc., do not always work. We were only interested in the web element finding ability of the approach and perceived that it is unlikely that web elements on other pages on a web application have a significantly different distribution than those in the home pages. The sample of web elements was also analyzed to verify that we had acquired a comprehensive set of different types of web elements.

   The web elements that were chosen for the evaluation had to comply with one or several of the following attributes: (i) were possible to perform actions on (e.g., anchors, buttons, menu items, input fields, text fields,

check-boxes, and radio-buttons); (ii) can be used for assertions or synchronization (e.g., top-level headlines); (iii) belong to the core functionality of the web application homepage; (iv) are present in both versions of the web application homepage. The rationale for (iv) is given by the study's objective to look at the approach's ability to find web elements over different versions of the web applications. Through this manual selection, we devised a set of 442 matches (pairs of corresponding web elements) for the experiment.

5. *Equivalent web elements Selection*: The number of manually-identified matches is then expanded by applying the equivalence definition according to the formula in Section III of the present paper. By considering the set of equivalents of both web elements in each of the 442 pairs, we came up with an extended set of 1163 matches.

   We finally define a balanced test set for applying the Similo and VON Similo algorithms. To build the test set, we include the aforementioned set of 1163 matches and 1163 randomly-selected non-matching web element pairs. We take the same number of matching and non-matching web element pairs for each considered web application.

6. *Match Definition*: In the experiment step, for each pair of web elements (either matching or non-matching), we compute the Similo similarity score and the VON Similo score. The scores are normalized in the range [0, 1].

   No normalization step was performed in the original formulation of the Similo locators. The algorithm, instead, computed the similarity scores for all candidate web elements and ranked them with no normalization steps. For our purposes, it was necessary to add a normalization step to compare the performance of Similo with that of VON-based locators.

   A web element of the new web application is considered a match to the target web element if the Similo (or VON Similo) score is higher than a threshold. The threshold is provided as a variable parameter to the algorithm, which is necessary since both algorithms would otherwise always return a web element (even with a meager similarity score).

7. *Analysis*: The results were then analyzed using formal and descriptive statistics to identify the metrics used to answer the study's research questions. For this study, we use the target web element (W_t) to describe the sought web element taken from the older web application version. In turn, the correct candidate web element (W_c) is sought in the new version

of the web application such that $W_c \in WS$, where WS is the set of all web elements in the newer version of the web application. Given these definitions, we get the following outcomes of a web element localization:

- **True positive (TP)** - When there is a found match such that $W_t = W_c$ when $W_c \in WS$.
- **False negative (FN)** - When there is no match found such that $W_t \neq W_c$ despite $W_c \in WS$.
- **True negative (TN)** - When there is no match found such that $W_t \neq W_c$ when $W_c \notin WS$.
- **False positive (FP)** - When there is a match found such that $W_t = W_c$ despite $W_c \notin WS$.

Counting the occurrences and distributions of the above measurements provides us with the information required to answer all the study's research questions. Based on such definitions we in fact compute the following derived metrics: *Precision*, as TP/(TP+FP); *Recall*, as TP/(TP+FN); *Accuracy*, as (TP+TN)/(TP+FP+TN+FN).

6.5 Results

In this section we report the results measured to answer the research questions defined for the study.

6.5.1 RQ1 - Ideal set of attributes for VON Similo

To analyze and justify the selection of attributes for the multi-locator-based algorithms, we computed statistics on the distribution of the attributes on all the included web applications. We computed statistics for all the 170 standard HTML attributes listed by W3C[5].

In the considered web applications, we utilized only 35 attributes (i.e., they were assigned at least a non-null value). In Figure 6.5 we report the percentage of present, empty ("") or absent (*null*) values for each attribute. For the sake of readability, we only report the attributes for which the percentage of times a value is present is above the median for all attributes (0.7%). We distinguish between *null* and *empty* values since for specific attributes (e.g., the textual

[5]https://www.w3.org/wiki/Html/Attributes/_Global

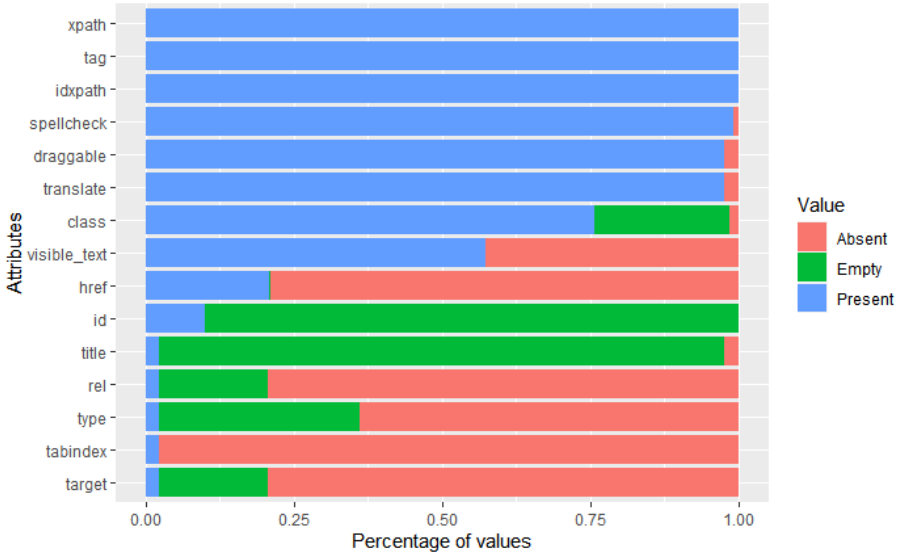

Figure 6.5: Distribution of absent, empty and valued attributes in the selected web pages

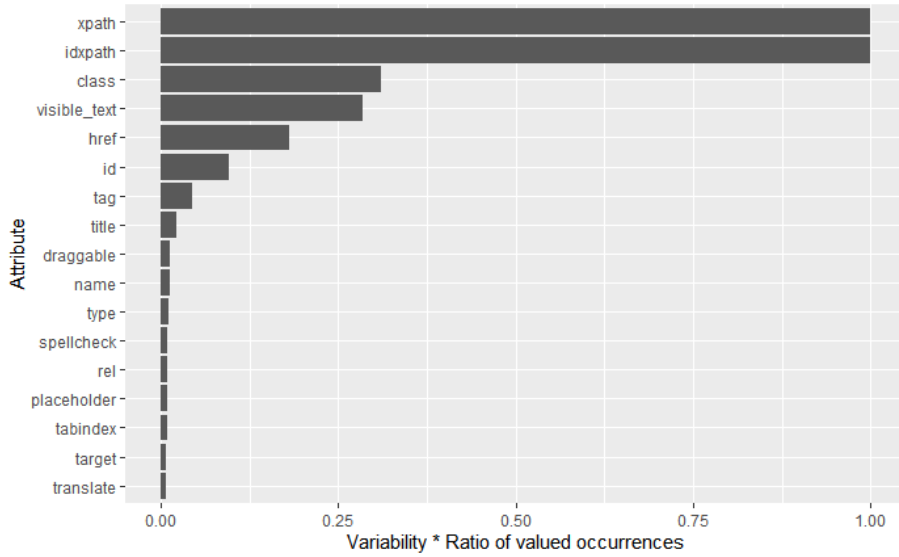

Figure 6.6: Top attributes for weighted variability in the selected web pages

Table 6.1: Mean (SD) values of Precision, Recall and Accuracy over the five test sets for Similo at varying thresholds

Threshold	Precision	Recall	Accuracy
0.20	0.688 (0.024)	0.965 (0.056)	0.766 (0.036)
0.28	0.796 (0.029)	0.898 (0.077)	0.823 (0.045)
0.40	0.892 (0.034)	0.615 (0.121)	0.770 (0.066)
0.60	1.000 (0.006)	0.265 (0.105)	0.632 (0.052)
0.80	1.000 (0.000)	0.008 (0.012)	0.503 (0.006)

content of a web element, identified by the attribute *visible_text*), the absence of a value is itself a piece of information that we can use to locate a web element.

We compute the variability of each attribute as the ratio between the number of different values present in the set of considered web elements for a given attribute and the number of times the attribute is valued (i.e., other than null or empty). To assign a weight to the variability with the relevance of the web elements in identifying web elements, we multiply it with the percentage of valued occurrences measured in the previous step. In Figure 6.6 we report the top attributes for variability in the considered web pages. This result validates the original selection of attributes performed in the original Similo paper since only the XPath, ID Xpath, class, text, href, id, and tag attributes were those with the highest variability.

It is worth noting that the XPath and ID XPath attributes have a value equal to 1 for the variability multiplied by the valued occurrence ratio, meaning that they are the only two attributes valued for every web element and for which no pair of web elements can have equal values.

Answer to RQ1: The analysis of Non-Null values and variability for attributes in web pages identify XPath, ID XPath, class, visible text, href, id, and tags as the attributes that are most likely to change when present with non-null values in elements of DOM models. The selection of attributes for the Similo algorithm is therefore confirmed as optimal for web elements multi-locators.

Table 6.2: Mean (SD) values of Precision, Recall and Accuracy over the five test sets for VON Similo at varying thresholds

Threshold	Precision	Recall	Accuracy
0.20	0.680 (0.011)	0.991 (0.008)	0.760 (0.011)
0.40	0.968 (0.007)	0.922 (0.110)	0.941 (0.052)
0.60	1.000 (0.004)	0.475 (0.210)	0.738 (0.105)
0.80	1.000 (0.000)	0.030 (0.042)	0.515 (0.021)

6.5.2 RQ2 - Similo Performance

Having finalized the selection of attributes to consider for comparing target and candidate web elements, we evaluated the original Similo over the set of 1163 matches as defined in section C.5.

We performed 5-fold cross validation and optimized the threshold value, used to detect a match, on a training set and then evaluated at that threshold on the hold-out test set. The optimal threshold values were very stable over all folds; for simplicity we thus report results for the full set of web elements.

Table 6.1 reports the mean and standard deviation of precision (P), recall (R), and accuracy (A) at varying thresholds. The optimal threshold, selected to maximize accuracy on the test sets, is shown in bold.

With minimal variations between different folds, the optimal threshold for the algorithm is a rather low value of 0.28. Raising the threshold of the algorithm, in fact, lowers the number of comparisons that are signalled as matches, which reduces the number of false positives. This low value for the threshold may suggest that few attributes with stable values (over the selected set of 14) are sufficient to identify a match between the original and the new web element. Raising the threshold over 0.40 guarantees a high precision for the algorithm (over 90%) at the expense of a rapidly increasing number of false negatives (i.e., number of candidates that are not matched while being correspondent to the target).

Answer to RQ2: applied on the test dataset, the original version of the Similo algorithm is capable of providing a mean 82.3% accuracy when a match threshold of 0.28 is used.

6.5.3 RQ3 - VON Similo performance

In Table 6.2, we report the results for VON Similo, in the same format as for Similo above. Also here, the optimal threshold value was stable over the five folds.

By comparing the results in the two tables, it is clear that the optimal threshold is higher for VON Similo. This is likely since the comparison of multiple attribute values — instead of the one-to-one comparison of the original Similo algorithm — increases the likelihood that a candidate web element is deemed as matching with the target one. In other words, the increase of the size of the candidate space, as described in section III, requires a larger threshold or the number of false positives increases.

Even in this case the number of true positives found decays rapidly with increasing threshold, however we can observe a precision of 100% at a threshold of 0.60, supported by a 47.5% recall. We note that practitioners can choose to select a different threshold level that favors either precision or recall depending on the risks and costs they judge a false positive or a false negative to have. However, it is clear that VON Similo can achieve higher accuracy scores than the original Similo algorithm.

Answer to RQ3: applied on the test dataset, the Visually Overlapping Nodes-based version of the Similo algorithm (VON Similo) is capable of providing 94.1% accuracy when a match threshold of 0.40 is used.

6.5.4 RQ4 - Comparison between Similo and VON Similo

In Figure 6.7 we report the comparison of the two ROC (receiver operating characteristic) curves drawn for Similo (blue) and VON Similo (red). The ROC curves show the trade-off between the true positive rate (i.e. sensitivity) and the false positive rate (i.e. specificity) of an algorithm, plotted at varying thresholds. For simplicity, we report the ROC curves computed over the whole set of matches without applying the 5-fold split on the data set.

We also report in the graph, as a baseline, a random classifier, which is expected to provide points along the diagonal (i.e., number of true positives equal to the number of false positives). An ideal classifier would instead provide a single point where TPR = 1.00 and FPR = 0.00. In a ROC curve, the closer the curve is to the main diagonal of the ROC space, the less accurate the test.

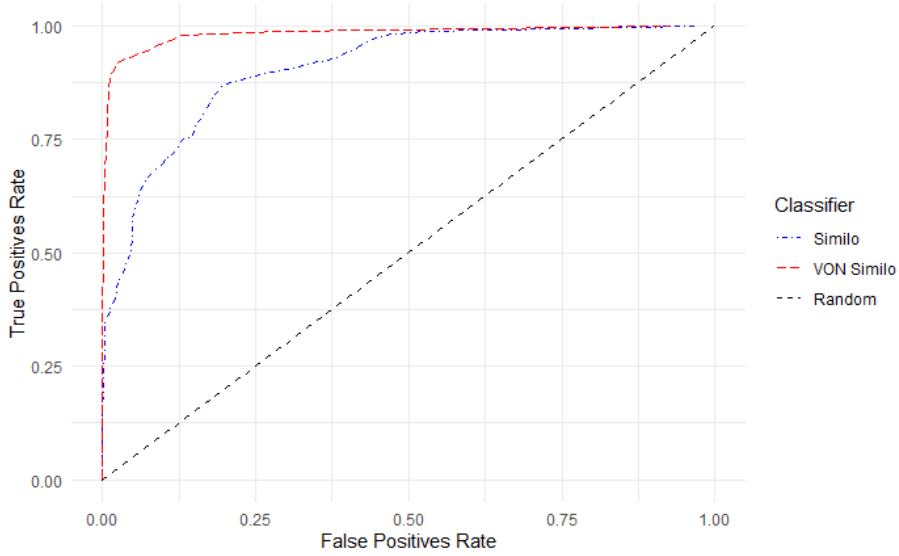

Figure 6.7: ROC comparison for Similo, VON Similo, and the baseline corresponding to a random classifier

Table 6.3: Comparison of the mean (std deviation) of precision, recall and accuracy of Similo vs. VON Similo, per subject application

Approach	Precision	Recall	Accuracy
Similo	0.799 (0.130)	0.772 (0.243)	0.783 (0.157)
VON Similo	0.965 (0.061)	0.790 (0.915)	0.879 (0.153)

For our purposes, there is no benefit in obtaining a Precision-Recall curve over a ROC curve since, by construction, the data are equally distributed between positives and negatives.

By visual comparison, it is evident how both the algorithms, Similo and VON Similo, provide ROC curves that are significantly distant from the main diagonal (corresponding to a Random classifier). At the same time, it is visually evident that the VON Similo algorithm is overall better than that of Similo. This advantage in using VON Similo can be quantified by computing the area below the ROC Curve (Area Under Curve, or AUC).

As a final comparison, in Table 6.3, we report a comparison of the performance of the two approaches in terms of average accuracy, precision and recall over the set of 33 web pages used in the experiment. From the comparison we notice that VON Similo has globally better values for all the measures on the set of apps, with 88% accuracy against 78.3% accuracy for Similo. We also observe rather high std. deviation values for the computed measures, caused by a variabile number of matches and performance obtained with the tool on individual applications.

One can also perform a Wilcoxon (paired) signed rank test [140] of the accuracy values per app which supports (p-value of 0.0001614) the hypothesis that VON Similo (mean accuracy, over the apps, of 88.0%) performs differently than Similo (78.3%). Since the argument has been made that Bayesian statistical analysis can have benefits [65] we also compared the two approaches with a Bayesian signed rank test [38] which gave a probability of 99.9% that VON Similo has a higher accuracy than Similo[6].

[6]There were clear differences also for precision and recall. We used Python 3.10 and the library baycomp for the Bayesian SignedRankTest and (base) R 4.2 for the Wilcoxon test.

Answer to RQ4: As demonstrated by the comparison of the respective ROC-curves and the statistical tests, VON Similo (AUC = 0.91, mean accuracy = 0.94, mean accuracy per app = 88.0) performs **significantly better** than Similo (AUC = 0.88, mean accuracy = 0.82, mean accuracy per app = 78.3).

6.6 Discussion

This study has shown that multi-locator-based approaches, using web element attributes, can provide robust web element localization, in the real world, for evolving web applications. When coupled with visually overlapping node based (VON-based) locators, the method is robust with a 94 percent success rate in finding the correct web element.

This result is significant, especially when put into context of how tests are maintained in industry. The study results are acquired from a random sample of web applications with both larger and smaller amounts of change between versions, controlled by the time in the past a web application was sampled (from 6 months to 5 years). In a real scenario, in industry, the probability of only more minor changes between versions is theoretically higher than the scenario presented here since test suites would be run daily or at least weekly. This leads us to the logical conclusion that; in an authentic setting, the average accuracy of the proposed approach would be higher than 94 percent. We base this discussion on the attribute-based locator technology, which precision is affected by change. For example, if only one attribute is changed, the accuracy will be higher than if two or more attributes have changed. For shorter iterations between test runs, the number of changes to the application will be less and, as such, the solution accuracy increases. However, although logically sound, future work is required to verify this conclusion, e.g., through empirical and industrial research.

Regardless, given the result of the study, an interesting question becomes what industry is willing to accept in terms of false test results, since, as stated in Section 6.1, robust web element localization is one of the core challenges with automated GUI testing and perceived a root-cause to its lack of general use in practice. Hence, although the presented results are significant, compared to, for instance, state of the art research by Leotta et al., which reported a success rate of 33-93 percent (93 percent being a theoretical best case limit) [99], a philosophical question remains if 94 percent success rate is good enough? A 94 percent success rate still implies that six web elements will not be found every 100 test actions, or one faulty test behavior every 17 steps of script code

run on modified parts of the SUT. For a test suite ordering in the hundreds of test cases, 94 percent success rate would still constitute a significant amount of root-cause analysis and test script maintenance. However, once more, the selected time elapsed between tests of the web application versions were in the experiment abnormal when compared to the time between executions of a GUI-based regression test suite in industrial practice.

However, as a speculative counterpoint, since any accuracy below 100 percent seems to be an issue for larger test suites, the question arises if 100 percent robustness is even achievable. Keeping in mind that web element localization implies searching for a target web element in a context where there is often not an exact match, only a partial match, due to changes in the attributes, appearance, or behavior of the sought web element. Under these circumstances, we also see humans having less than a 100 percent success rate, so the question is how far an automated approach can reach.

Additionally, the attribute-based approach shows that with impartial data, we can still identify web elements with high probability. When one of these web elements is identified, the attributes that are no longer valid can be automatically repaired—Repair, in this case, implies updating attributes that are no longer valid. Thus, mitigating much of the maintenance costs associated with test scripts. Consequently, keeping the delta between test script versions and the SUT low implies a larger probability of web element identification success. Thus, once more speaking towards the potential of the approach's ability to reach higher than 94 percent success rate in a real scenario in industrial practice.

Conceptually, VON-based locators can work for any DOM-based multi-locator approach, i.e., on other platforms not used in this paper, such as Android or desktop. However, VON-based locators only work with multi-locator approaches and would not work for single-locator (e.g., Selenium-based) scripts due to a lack of information. Consider a Selenium action based on an XPath locator, but said XPath is no longer available in a new version of the target web application. All candidate information can be acquired from the new site, but if additional information is not available for the target web element, there would not be a way to map the target to any of the candidates reliably. This is a limitation of the approach since, a priori, the amount of information to reliably locate a single-locator web element in a different version of the web application is not known. This is a subject of future work that entails ranking the attributes based on their relative power to be used as locators.

Furthermore, the analysis of the attributes used for this work shows that the 14 attributes that Similo used are the most frequently populated ones on

modern web pages. Thus, implying that these provide the most value for web element localization. This result is significant, as previous works have presented lists of suitable attributes [49, 75, 86], but not provided empirical evidence to support these claims. The list of attributes provided in this work, although only contemporary, thereby provides insights into how to design more robust GUI testing tools for the foreseeable future. Future work could also investigate if there are notable dependencies, or even domain aspects, to how these attributes are populated and how, for further improvements into web element localization and automated repair. Such analysis should also look at the relative power of different attributes. Looking at the list of attributes in Figures 6.5 and 6.6 we note some interesting observations. ID is an often mentioned attribute for web element localization [22, 99, 147]. However, as shown by the results, ID is seldom used in practice, implying that reliance on this attribute would have detrimental effects on web element localization. Hence, from a practical perspective, the attributes need to be weighed in terms of power versus availability. One or several commonly available attributes, e.g., XPath or tag, have a higher probability of being useful than a perceived more powerful attribute, which is seldom available, such as ID. Further research is, however, required to evaluate this relative power between attributes.

Another finding concerns the contents of the attributes of the web elements. In Similo, locator parameters are compared with various comparators such as equals, Levenshtein distance, and integer distance. These are utilized in this research based on expert judgment, but other comparators, e.g., euclidean distance, could also be used. In future work, a valuable analysis would investigate the types and variability of attribute data to assign the most suitable comparators. For instance, XPath strings are unique to each candidate and are represented as Strings of longer length. This paper uses Levenshtein distance for their comparison, but other approaches would likely work equally well or even better. This mapping between key web element attributes, the characteristics of the attribute data, and comparators could theoretically be done using machine learning, e.g., using learning to rank [162]. Similarly, we can also optimize the weighting of each comparator-attribute tuple. However, this is once more a subject of future research.

6.6.1 Threats to validity

In this section we discuss the threats to the study and its results pertaining to the internal, external and construct validity of the research.

Generalizability: The paper provides two contributions; (1) analysis of web element attributes used in industrial practice and (2) VON-based locators. For (1), we perceive the results to be generally valid, as they are gathered from commonly used web pages from larger companies that should be perceived as adopters of best state-of-practice approaches. However, for (2), the results are delimited to web element localization with multi-locators as discussed in Section 6.6. This diminishes the result's industrial applicability since most industrial frameworks, like Selenium, utilize a single-locator approach. As such, adopting VON-based multi-locators would also require a technological shift in approaches or tooling for developing such test cases. Although scripts that use multi-locators can be developed manually, we perceive recording such tests as beneficial from an efficiency perspective.

Subject selection: We took the subjects for this analysis from the Alexa.com web application (as of writing, the site has been discontinued), which reduces bias in the sampling as the web applications on the list are outside the researchers' control. However, we chose only a subset of the list, i.e., the top 40 pages (33 used after excluding, for instance, broken and duplicate sites). This set is still perceived as significant when compared to other studies in this area of research. However, it is unknown how representative the result is to any general web application on the internet. Analysis of the tags used in the web elements of these web pages does, however, provide us with confidence that all types of web elements that can, theoretically, be encountered have been identified. We stress, however, that this research only focused on web element identification, and therefore detrimental effects that may arise during test execution were not covered. An example of such an effect is the acquisition of web element attribute data during a state transition. A lack of synchronization between the test script and AUT could result in missing attributes if pre-analysis of web element information begins before the AUT is fully loaded.

Comprehensiveness: VON Similo was designed with flexibility in mind to optimize its potential use in practice. A drawback of this design is that the comparators and attributes used in this study are a subset of possible attributes and comparators, as discussed in Section 6.6. While not reducing the validity of the results presented in this work, this implies that we could achieve better results with other constellations of parameters. We defend this unorthodox statement by the presented results that indicate a considerable increase in precision and recall compared to state of the art. As such, although there is no threat to the validity of the results of the current implementation, given the set of analyzed web pages, we cannot state if this is the best implementation of the approach.

Further research is thereby required with other constellations and contexts to optimize the approach.

6.7 Conclusions and Future Work

In this paper, we proposed a novel location algorithm for web elements in web applications, i.e., Visually Overlapping Nodes (VON). We applied this approach by extending an existing multi-locator approach (Similo), thereby obtaining the VON Similo approach.

To investigate the accuracy of the algorithm, we performed an empirical investigation by manually identifying a set of matches between mutated widgets in different releases of the same application and then measuring the effectiveness of the original and extended approaches to determine correct matches. This analysis led us to measure a +9.9% increase in accuracy for VON Similo with respect to the original approach. It is worth underlining that the original Similo was already proven as better performing than state-of-the-art multi-locator algorithms (e.g., Leotta's ROBULA+ [100]).

This work first assessed the possible benefits introduced by the concept of Visually Overlapping Nodes when performing GUI testing of web applications. Therefore, fine-tuning the approach can be obtained by empirically finding the optimal values for several fixed coefficients and variables employed by the algorithm.

The algorithms evaluated and compared in this study used only static weights defined in the original Similo algorithm and were compatible with state-of-the-art tools like COLOR[86]. Selecting a different set of weights for the attributes used in the comparison can lead to significantly different results in terms of precision and accuracy. The same reasoning applies to selecting the attributes involved in the score computation.

In our immediate future work, we plan to employ machine learning-based approaches to identify the optimal set of attributes, and related weights, to maximize the accuracy of Similo. As a further improvement of the algorithm, it is possible to consider returning as output not only a single matching web element but a ranked list of the most matching candidates. This evolution of the algorithm can also benefit from the application of machine-learning-ranking (MLR or Learning to Rank) techniques.

Chapter 7

Improving Web Element Localization by Using a Large Language Model

Abstract

Context: Web-based test automation heavily relies on accurately finding web elements. Traditional methods compare attributes but don't grasp the context and meaning of elements and words. The emergence of Large Language Models (LLMs) like GPT-4, which can show human-like reasoning abilities on some tasks, offers new opportunities for software engineering and web element localization.

Objective: This paper introduces and evaluates VON Similo LLM, an enhanced web element localization approach. Using an LLM, it selects the most likely web element from the top-ranked ones identified by the existing VON Similo method, ideally aiming to get closer to human-like selection accuracy.

Method: An experimental study was conducted using 804 web element pairs from 48 real-world web applications. We measured the number of correctly identified elements as well as the execution times, comparing the effectiveness and efficiency of VON Similo LLM against the baseline algorithm. In addition, motivations from the LLM were recorded and analyzed for all instances where the original approach failed to find the right web element.

Results: VON Similo LLM demonstrated improved performance, reducing failed localizations from 70 to 40 (out of 804), a 43% reduction. Despite its slower execution time and additional costs of using the GPT-4 model, the LLM's human-like reasoning showed promise in enhancing web element localization.

Conclusion: LLM technology can enhance web element identification in GUI test automation, reducing false positives and potentially lowering maintenance costs. However, further research is necessary to fully understand LLMs' capabilities, limitations, and practical use in GUI testing.

Keywords: GUI Testing, Test Automation, Test Case Robustness, Web Element Locators, Large Language Models

7.1 Introduction

Software testing plays a vital role in ensuring the quality of software applications. However, testing is often a time-consuming and expensive process in practice [71, 73]. By leveraging automation, organizations can run tests more frequently, improve test coverage, and thereby identify more defects faster, with positive impacts on software lead times and software quality [17, 28, 131].

Automation is applied in various types of testing, but one of its primary uses in practice is in automated regression testing. Regression testing allows testers to evaluate the quality of each software release. Typically, at higher levels of system abstraction, such as the Graphical User Interface (GUI) level, testers create a suite of test scripts that simulate end-user scenarios and verify the application under test's (AUT) correct behavior by using automated oracles [103, 108]. However, it is common for new software releases to introduce changes that can break existing automated regression tests, which require maintenance efforts and costs to update and repair the test scripts. The maintenance cost is exceptionally high when testing an application through its GUI, as GUIs frequently change between releases [26, 58, 152]. In addition, GUI scripts are subject to breaking from changes to the underlying logic and architecture of the AUT that modifies its behavior.

Furthermore, GUIs are primarily designed for human interaction (i.e., not machine-to-machine communication), which presents additional challenges for automation, such as synchronization between the test scripts and the AUT. These challenges, although present, are not considered as prominent in lower-level testing techniques like unit testing [131].

Test script robustness is one of the most reported challenges in web test automation [121]. The challenge involves making tests resilient to smaller changes

to the AUT that should not affect the test execution while still allowing the tests to detect significant differences that could potentially be defects. Many solutions that increase the robustness of locating web elements have been proposed for mitigating this challenge [49, 98, 99, 100, 117, 151]. Some of the more recent approaches use similarity scores to identify the most similar web element to a target. This is done by using previously stored properties (i.e., extracted from the corresponding web element in a previous version of the web application) and comparing the stored properties to the updated web elements [122, 125]. The web element with the highest score is assumed to be the most likely web element to use in an interaction (e.g., a click or type action). While conventional algorithms (i.e., non-AI) can be used for finding similarities between web elements, they still typically lack knowledge about how web applications work and the semantic meaning of texts (i.e., skills possessed by a human tester). Being able to tell if different words or sentences have the same meaning or that two different web elements have contextual similarities (e.g., are closely located or are interchangeable solutions) could be a powerful feature in a testing tool. For example, assume a button in a web interface that changes the caption from 'Submit' to 'Send' in an updated version. A script that relies on the button caption to identify the next action would likely not find the new caption identical to the old caption without some form of semantic understanding, causing a false positive (i.e., a failed script execution). On the other hand, if a test tool could reason that the captions still have the same meaning (i.e., in that specific context), they could perceivably carry on without failing the test execution.

Large language models (LLMs) are trained on vast amounts of data and utilize deep learning techniques to capture linguistic patterns and dependencies [153]. We have only begun to explore the possibilities of using LLMs in test automation. One such example is SocraTest, a vision of a framework for conversational testing agents that could aid a human software tester by performing tasks autonomously [63]. Recent studies utilize natural language processing (NLP) with heuristic search and the DOM structure to identify web elements in web applications [85] or use LLMs to generate text inputs for GUI applications based on semantic understanding and GUI application context [105, 106, 161]. The proposed solution in this paper is based on the hypothesis that we can improve web element localization even further by combining an LLM with a traditional algorithm to take advantage of some of the benefits of the LLM, e.g., its assumed semantic understanding and contextual awareness, while utilizing the speed of the conventional algorithm.

The specific contributions of this paper are:

- A novel approach that can improve web element localization by utilizing a large language model.

- An empirical study that shows the effectiveness and efficiency of the proposed approach compared to the baseline approach.

- A qualitative content analysis on the motivations gathered from the LLM, explaining the main aspects used when comparing the similarity of two web elements.

This paper is structured as follows. Section 7.2 gives a short introduction to large language models. Section 7.3 covers the details of both previous versions and the proposed enhancement to the Similo algorithm. The design, research questions, and procedure of the empirical study are presented in Section 7.5, and the results in Section 7.6. We then discuss results in Section 7.7 and state conclusions and future work in Section 7.10. Section 7.9 presents related work.

A package for replicating the experiment is available for download from [16].

7.2 Large Language Models

Large language models (LLMs) like GPT-4 have revolutionized Natural Language Processing (NLP) by leveraging the Transformer architecture [153]. This groundbreaking approach replaced traditional recurrent neural networks (RNNs) with a self-attention mechanism, enabling the models to capture long-range dependencies efficiently. These models are pre-trained on vast amounts of data, allowing them to grasp the meaning of input prompts and generate text. Notable examples of recent LLMs include OpenAI's Generative Pre-trained Transformer (GPT-3, GPT-3.5, and GPT-4) [45] and Google's Pathways Language Model (PaLM) [50]. ChatGPT is a sibling model to the InstructGPT model, which is an improved version of GPT-3 that has been fine-tuned and trained with human feedback [51] to improve its ability to follow instructions [133]. We can also use ChatGPT as an interface to the newer GPT-4 model, which is several magnitudes larger (i.e., about 1000 times) than previous GPT models and performs close to human-level on some tasks [45]. We have included a more detailed comparison between the two latest versions of GPT in Section 7.5.4.

7.3 Similo

VON Similo is a web element localization algorithm that uses a multi-locator approach, similar to previous works, e.g., Leotta et al. [100]. In contrast to single-locator solutions, multi-locators use multiple properties of a web element, such as ID, XPath, label, and tag, to find a target. This is achieved by comparing the properties of each candidate web element on a web-page with the desired properties of a target element (i.e., the correct candidate), resulting in a similarity score. A heuristic is then applied that the web element with the highest similarity is the most likely candidate to be a match.

VON Similo also utilizes the concept of visually overlapping nodes (VON), which makes use of the hierarchical structure of web elements in modern web applications and their representation in a document object model (DOM) [160]. The VON concept considers that multiple DOM nodes (i.e., web elements) are often visually overlapping (i.e., displayed in the same visual area in the web browser) and conjointly represent the same visual web element to the user. These conjoint elements share or have similar properties, e.g., overlapping areas, coordinates, and similar XPaths. This implies that interactions (e.g., a click) on the area represented by any of these overlapping nodes will yield the same GUI state transition (i.e., event). As such, any of the nodes can be used to execute an automated test case, effectively increasing the number of valid web elements for an interaction from a single element to the number of overlapping elements in a visual area. This increase in targets improves the probability of finding a web element after changes to the tested application, thereby increasing the test execution robustness.

In this paper, we use the following nomenclature:

- **Properties**: attributes and other information (e.g., location, size, XPath, etc.) that can be extracted from a web element.

- **Candidate**: a web element containing properties that can be evaluated by VON Similo. Candidates are typically captured from the currently active (i.e., visible) web page.

- **Desired properties**: the properties we are looking for in a candidate. The desired properties are often captured or recorded from a target in a previous version of the SUT (i.e., when the test script was created or maintained).

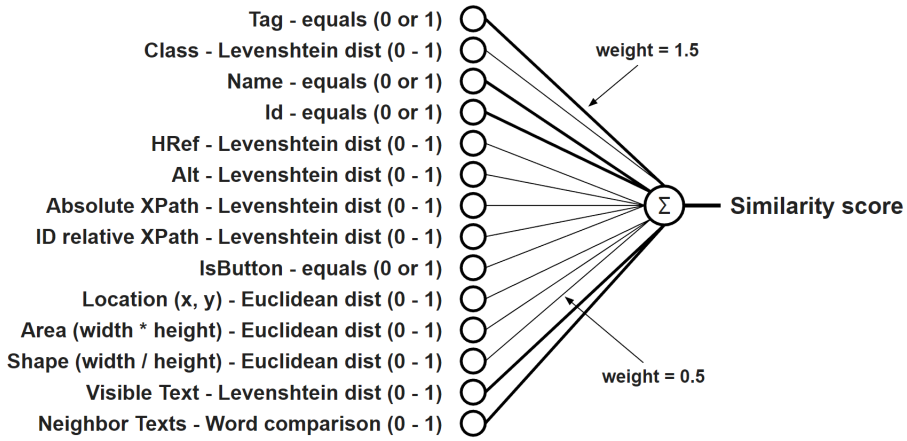

Figure 7.1: Graphical representation of the computation of similarity score between two different sets of web element properties.

- **Similarity score**: a score representing the distance in similarity between two sets of properties, where a higher score represents higher similarity between two web elements.
- **Visual web element**: one or many DOM nodes that overlap visually, according to the Visual Overlap heuristics defined by the VON-Similo algorithm (described in Section 7.3.2).

7.3.1 Standard Similo

VON Similo is based on the initial version of the Similo multi-locator algorithm proposed by Nass et al. [123]. Similo attempts to identify the web element among a set of candidates that is most similar to the desired properties. The desired properties are often gathered or recorded from a previous release of the same AUT but can be any set of properties. Candidate web elements are typically retrieved from the current (i.e., visible) web page. The standard version of Similo used 14 properties, listed in Figure 7.1.

Each property is associated with a comparison operator and a weight (also included in Figure 7.1). The comparison operator compares the property value of a candidate with the desired property value and returns an output value

Figure 7.2: The YouTube search bar.

between zero and one (or binary zero *or* one). Using the output values, a similarity score is then calculated for each candidate by summarizing the weight multiplied by the result from the comparison operator for all 14 properties. After comparison of all candidate element scores, Similo then returns the candidate with the highest similarity score, assumed to be the most similar web element to the target element with the desired properties. Optionally the algorithm can output a ranked list of candidates from higher to lower similarity scores.

7.3.2 VON Similo

The concept of visually overlapping nodes (VON) can be applied to Similo to increase the likelihood of locating the correct web element (i.e., according to the oracle) [124]. To illustrate the VON concept with an example, Figure 7.2 contains a picture of the search bar on YouTube.

The light-blue area of the image contains two DOM elements in a hierarchy. A simplified version of that DOM structure is shown in Listing 7.1.

```
<div class="sbib_b" id="sb_ifc50">
    <input id="search" name="search_query">
</div>
```

Listing 7.1: DOM hierarchy of the YouTube search bar.

As can be seen from this example, what visually appears to be only one element is actually represented by a div element containing an input element (i.e., two DOM elements in a hierarchy). This exemplifies how modern web pages are structured and presents a problem when selecting an oracle that represents the correctly located DOM element (i.e., web element) since there is more than one to choose from. The VON concept handles this problem by treating both of the DOM elements as equally correct by merging the properties of both elements together into one visual web element (i.e., a new virtual element). Listing 7.2 illustrates how such a visual web element could be represented where the double pipe (i.e., “OR” operator) denotes that an attribute could have more than one value.

```
<div || input class="sbib_b" id="sb_ifc50 || search" name="search_query"
    />
```

Listing 7.2: Example representation of a visual web element.

There are two benefits to the VON approach. First, it reduces the number of candidate web elements (i.e., since there are typically fewer visual web elements than DOM elements on a web page), resulting in a higher probability of locating the correct one. Secondly, merging the properties of all the DOM elements belonging to the same visual web element will result in a higher (or the same) similarity score than distributing the score on several DOM elements in the hierarchy (i.e., any contributions to the similarity score, when comparing the properties, is concentrated on the same visual web element instead of being distributed over several DOM elements).

Two web elements (W1 and W2) are considered to belong to the same visual web element if the following conditions are satisfied:

1. The ratio between the overlapping areas of the web elements on the web page, and the union of the areas of the two web elements, is higher than a set threshold value (0.85 was selected by Nass et al. [124]). The ratio can be computed as:

$$\frac{\cap(R_1, R_2)}{\cup(R_1, R_2)}$$

   where: R_1 and R_2 are the rectanglular areas—Calculated using the coordinates (i.e. x and y) and size (i.e. width and height)—where the elements are visible on the web page. The intersection of these areas thereby represents the size (in pixels) of the common area occupied by R_1 and R_2, and the union represents the size (in pixels) of the total area occupied by R_1 and R_2.

2. The center of the web element W2 is contained in the rectangle R1. Note that this condition is always true if the threshold is greater than 0.5.

VON Similo (i.e., the VON concept applied on Similo) uses the same set of properties as in Figure 7.1 with the difference that each property value can take multiple values instead of just one, as in the Similo case. A property that holds more than one value is compared several times (i.e., one time per value). Assuming we would like to compare the Tag property in the web elements W1 and W2, we would need to perform N*M comparisons assuming that the Tag

property in W1 contains N values and the Tag property in W2 contains M values. The highest (i.e., best) comparison outcome of the N*M comparisons is selected as the result and appended to the similarity score. As such, the final score can be comprised of the comparator outcomes of property values from multiple DOM elements joined in the new virtual element.

For example, assume that the Tag property values are 'div' and 'span' for W1 and that the corresponding property values are 'span' and 'button' for W2. Comparing all the combinations (i.e., four) will result in a match (i.e., 'span') and return the value one from the equals comparison operator (See Figure 7.1).

7.3.3 Limitations of Similo and VON Similo

While Similo and VON Similo increase the tolerance to changes (i.e., robustness), there are still situations where the algorithms fail to find the web element specified by the human oracle. Our hypothesis is that humans possess reasoning capabilities, e.g. semantic, logical or contextual, about language and web applications that the algorithms lack. For example, assume that a button changed the caption (i.e., visible text) from 'Save' to 'Store'. A human would likely consider them to be equivalent buttons since the semantic meaning (i.e., purpose) is still the same, while the algorithm would struggle since the calculated distance between the two captions, e.g. using Levenshtein distance, would be quite large, negatively impacting the similarity score. Another example is when a button changes from {tag: 'input', type: 'button'} to {tag: 'button'}. If the tags were compared using the equals comparator, or even a distance comparator, the algorithm would not spot any similarities, while a context-aware human might know that a 'button' is a common replacement for an input field of type 'button' (i.e., an older standard). The core hypothesis of this work is thereby that large language models (e.g., GPT-4), trained on a vast amount of texts and websites, possess some form of reasoning, akin to humans, which can complement conventional algorithms to improve their robustness.

7.4 VON Similo LLM

VON Similo LLM is an attempt to take advantage of the speed and determinism of a conventional algorithm, VON Similo, but improved by the language understanding/processing and assumed reasoning capabilities of a large language model (LLM). In VON Similo LLM, we begin by ranking all the candidates present on the current web page with VON Similo. This is done by comparing

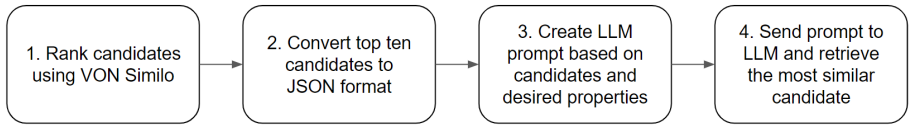

Figure 7.3: The VON Similo LLM process.

each element's properties to the desired properties (i.e., properties stored when creating or maintaining the test) of the target element. Next, we extract the top ten candidates from the ranked list of candidates provided by VON Similo. Each candidate in the top ten list and the desired properties of the target are then converted into a suitable format (i.e., we used JSON in the experiment since that should be a format familiar to an LLM). A prompt is then generated for the LLM (i.e., GPT-4 in our case) containing instructions for the comparison, the ten candidates, and the desired properties of the target. Figure 7.3 contains all the steps further detailed below.

1. The first step in the process is to extract all the candidate web elements from the currently visible web page and rank them, based on similarity, using VON Similo. VON Similo compares the desired properties with the properties of each of the candidates and produces a similarity score, as shown in Figure 7.1. The candidates are now sorted on similarity score, and the top ten continue to the next step. We decided to limit the number of candidates to ten for our experiments to prevent the prompt from exceeding the usage quota and also reduce the runtime cost of utilizing the LLM API. A usage quota is a limit of tokens spent over some time. Quotas prevent a user of GPT-4 from accidentally paying too much money on prompts.

2. The next step is to convert the ten candidates into a format that the LLM should be familiar with since that enables us to create a prompt without explaining the format. We decided to use JSON since that is a commonly used format when communicating over the Internet. Instead of creating an array, we decided to place each JSON structure on a separate line. Listing 7.5 shows an example of a prompt containing ten candidates encoded in JSON format.

3. The third step is to create a prompt that contains instructions on what we expect the LLM to do and what we would like as output. Listing 7.3 shows

the prompt structure we used. The first eleven rows of the prompt contain one line of instruction and ten lines of candidates in JSON format. We also provide a unique widget id (i.e., incremental count) with each candidate to simplify the output. Next, we add the instruction that we expect the LLM to return with the widget id to the candidate most similar to the desired properties, also converted into JSON format. Listing 7.4 shows an alternate prompt structure used when asking the LLM to provide us with motivations explaining why this candidate is considered the most similar. We have included a more complete example of the second prompt version in Listing 7.5.

4. The final step is to send the prompt to the LLM. The widget id of the most similar candidate and, optionally, the motivation of the choice, depending on the prompt used, are retrieved as output.

```
Given the following candidate web elements (|| means that an attribute
    can have multiple values):
<10 candidates in JSON format>

find the one that is most similar to the element:
<desired properties in JSON format>
Answer with the widget_id number(digits) only, no explanation or text
    characters
```

Listing 7.3: Prompt structure used in experiment when asking for widget id only.

```
Given the following candidate web elements (|| means that an attribute
    can have multiple values):
<10 candidates in JSON format>

find the one that is most similar (answer with the widget_id of the most
    similar and motivate why using a list) to the element:
<desired properties in JSON format>
```

Listing 7.4: Prompt structure used in experiment when asking for widget id and motivations.

7.5 Methodology

This section presents the research design, the research questions, and the research procedure of the empirical study performed to evaluate the benefits and

drawbacks of VON Similo LLM compared to VON Similo in terms of effectiveness and efficiency.

The first objective of the experiment is to evaluate the difference in effectiveness between VON Similo LLM and the VON Similo approaches (i.e. when finding web elements in two different releases of the same web application). The second objective is to compare the runtime performance (i.e., efficiency) of using the two approaches. Finally, the third objective is to evaluate the motivations returned from the LLM to explain why the LLM found the chosen candidate element to be the most similar match to the correct candfidate.

7.5.1 Research Questions

The study aims to answer the following research questions:

- **RQ1:** What is the effectiveness of VON Similo LLM compared to the VON Similo approach in terms of finding correct web elements?
- **RQ2:** What is the efficiency, measured as execution time, of VON Similo LLM compared to the VON Similo approach?
- **RQ3:** What main aspects does a large language model use to improve web element identification when the conventional approach fails?

The first research question (RQ1) was answered by running both approaches on a set of 804 web element pairs extracted from old and new versions of 48 real-world web applications. With a new version, we refer to a later iteration of a particular web application that has been subject to changes to its code or visual appearance that differentiates it from the older version (Further described in Section 7.5.2). Our hypothesis is that VON Similo LLM, using its reasoning capabilities, e.g., of semantic equivalence, logical patterns, or contextual information, would be able to correctly identify more correct candidates than VON Similo.

Next, research question 2 (RQ2) was answered by measuring the execution times of both approaches to determine the best matching web element. We measured the execution time as the time taken from calling an approach (i.e., by providing it with the desired and candidate properties) and returning the most similar candidate. Our hypothesis was that VON Similo would outperform VON Similo LLM in this aspect since VON Similo LLM utilizes the GPT-4 API (selected in Section 7.5), which, at the time of conducting the experiment, is relatively slow and restricted (i.e., in terms of requests per minute). In addition

to the actual overhead cost, this metric is assumed to give insights to allow us to discuss the current technology's industrial applicability.

Finally, we answered research question 3 (RQ3) by conducting a qualitative content analysis of the motivations gathered from the LLM, which aims to explain why the LLM found one candidate to be more similar to the correct candidate with some desired properties in more complex cases, i.e., in cases where the conventional approach (i.e., non-AI) failed to locate the correct candidate. This analysis was restricted to cases where the conventional solution failed out of cost constraints, i.e., the output from the GPT-4 API is associated with a monetary cost per output token.

7.5.2 Selecting Web Applications and Extracting Properties

The web applications chosen for this experiment are the same 50 websites used by Nass et al. to evaluate previous versions of Similo [122], taken from the Alexa top 50 list. One of the applications from the top 50 list was deemed inappropriate due to its adult content, and one was a duplicate (i.e., two URLs pointing to the same web application), resulting in a final set of 48 web applications. Additionally, we used the same web application versions, a new one and one 12 to 60 months older, as in the previous study [122], accessed through the Internet Archive website [1]. A scraping tool (developed in Java by the authors) was then applied to extract properties from all pairs of web elements that were perceived to be equivalent and available in both the old and new versions of each application. These elements were chosen manually through inspection of the applications and then used as oracles for the study. We manually included web elements for which the following criteria are met: (1) it is possible to perform an action on the web element, (2) the element can be used for assertions or synchronization by an automated testing tool, (3) the element belongs to the core features of the AUT, and (4) the element is present in both versions of the AUT's homepage (i.e., the page that the main URL points at). Criteria (4) was necessary since the Internet Archive only stores static pages, meaning that javascript, databases, etc., do not always work. Because the pages are static, they often, unintentionally, have diverse behaviors to newer versions of the AUTs. Whilst this design choice of only using the homepage may delimit the generalizability of the results, we perceive this to be a minor threat since most homepages contain the same elements as other pages of an AUT.

[1] http://web.archive.org

Furthermore, this selection process implies that if a human could identify the web element in both versions of the web application, it was likely included. This further implies that some web elements, which had been changed beyond recognition but which were still available, may have been overlooked during sampling. However, due to the size of the sample set and the efforts spent to capture all pairs in the extraction process, we find this threat to be negligible.

We wish to highlight that the experiment only concerns the web element finding ability of the approaches. We were not concerned with the types of interactions that can be performed on the elements nor how to utilize them for synchronization. This objective further justifies our delimitation of only choosing web elements on the homepage of each website since we perceive these to be representative of other pages as well. A perception that may not hold true for actions.

7.5.3 Applying the VON Concept on the Extracted Properties

In the next step of the research procedure, we applied the VON concept, described in Section 7.3.2, on each of the 804 web element pairs to add more values (i.e., from overlapping elements) to the target web element properties. Due to the VON concept, property values of visually overlapping web elements will be merged (i.e., using an "OR" operation) if the ratio between the intersection and the union of the areas exceeds the threshold value (i.e., 0.85 in our case). After applying the VON concept, many properties will contain several values (i.e., options) instead of just one, as when using the standard Similo approach.

7.5.4 Selecting the Large Language Model

Large Language Models (LLMs) are evolving quickly, and new versions are frequently released. For our experiment, we decided that effectiveness (i.e., in identifying the correct candidate) was the most important aspect to evaluate (i.e., before efficiency and cost) since we expect the performance, availability (i.e., allowed requests per minute), and price to change in time as the services mature. This design choice has a direct impact on RQ2, but we still perceive the results as valuable to get a snapshot of the currently available technology. We also expect the effectiveness of LLMs to improve, but evaluating the effectiveness today will still provide us with a baseline for the future. Therefore, we decided to select the most powerful LLM, in terms of effectiveness, available regardless of its efficiency, monetary cost (within reason), and limitations in

Table 7.1: Comparison between OpenAI GPT-versions.

GPT-version	Max tokens	RPM	TPM	Cost 1K tokens
GPT-3.5-turbo	4K	3500	90000	$0.002
GPT-4	8K	200	40000	$0.03

requests per minute. We also decided to go for an LLM provided by OpenAI due to its reputation and ease of access. Table 7.1 contains a comparison between the different versions currently provided by OpenAI (in April 2023, when we initiated the experiment). As seen from Table 7.1, GPT-3.5-turbo is better in all aspects (e.g., cheaper and more requests allowed per minute), except for max tokens (4K vs. 8K for GPT-4). The model size of GPT-4 is, however, 1000 times larger in size (170 trillion parameters vs. 175 billion parameters), hinting at enhanced capabilities and accuracy [89]. In our case, the additional number of tokens available for GPT-4 is welcome since the prompts of the solution are quite large since they include many web elements, encoded in JSON format, with the prompt. We expect each JSON representation of one web element to be close to 1K characters, meaning that each prompt, with ten web elements, would constitute around 10K characters or 2500 tokens (1 token $\sim$ 4 characters). This size is also feasible when using GPT-3.5-turbo since it is less than the allotted 4K tokens per prompt. However, since a JSON structure includes many special characters and digits, we expect a lower ratio than four characters per token (i.e., lower than the expected ratio for pure text). A ratio of two characters per token results in 5K tokens for the ten JSON representations alone, motivating our selection of GPT-4 that can receive 8K tokens in one prompt. In conclusion, we choose to use GPT-4 in our experiment even with the drawback of a higher cost, lower RPM (i.e., requests per minute), and lower TPM (i.e., tokens per minute) since increased accuracy and max number of tokens are more important for our evaluation.

7.5.5 Prompt Engineering

Prompt engineering is the intentional construction and refinement of prompts used in natural language processing tasks. It involves formulating precise instructions or queries to produce desired responses from large language models.

We experimented with larger and smaller prompts with or without examples to maximize the correctness of the output while trying to keep the prompt length

Table 7.2: The number of located (and not located) web elements when using one or zero examples included in the prompt.

Type	Total	Located	Not located	% Located
Zero-shot	70	37	33	52.9
One-shot	70	41	29	58.6

short enough to be of practical use (i.e., since the prompt size is limited and is associated with a cost).

Initially, the experiment was performed with a minimal prompt with no examples (zero-shot). Hence, each prompt only contained instructions, the ten web element candidates, and a target element in JSON format. Each JSON element contains the property names and values of one web element. We created the JSON elements from the following properties: Tag, Visible Text, Class, Id, Name, HRef, Location, Area, Shape, Alt, Is Button, XPath, and Neighbor Text. Each candidate is also given a unique id to make it possible to ask GPT-4 to return with the id instead of the entire JSON element. The prompt asks GPT-4 to return the id of the candidate that is most similar to the target web element (also provided in JSON format) and specify a list of reasons for the decision. Listing 7.5 shows an example of such a prompt, including the response from GPT-4.

Next, we reran the experiment with a more descriptive prompt that contained one set of example inputs (one-shot) and the corresponding output. The one-shot approach was hypothesized to help train the LLM in how to perform the comparison and thereby provide a better result.

Table 7.2 presents our findings from evaluating the zero- and one-shot approaches. These were calculated on a subset of 70 web element pairs where VON Similo failed to identify the correct target (i.e., by running VON Similo in all the 804 cases). These cases were chosen because they were perceived of higher complexity since the conventional algorithm failed to identify them. As can be seen from the last column in the Table, including one example improved the result from 37 to 41 (i.e., 52.9% to 58.6%), representing a 5.7 percent reduction in not located web elements. Based on this result, and since the additional data for the one-shot did not significantly extend the prompts' token size, we decided to include one example in all the prompts used in the full experiment, i.e., all 804 web element pairs.

To improve the results even further, we tried to increase the number of candidates sent to the LLM (i.e., a larger list of top candidates proposed by

VON Similo). We observed several drawbacks with increasing the number of candidates: (1) increased cost due to a larger prompt, (2) failure to identify the most similar web element due to many candidates, and (3) GPT-4 needed more detailed examples sticking to the instructed output format (i.e., got confused by the increased prompt size and did not return with the widget id and motivations in the format specified by the prompt). Instead of exhaustively exploring (i.e., with a different number of candidates), we decided that ten candidates and one set of examples (one-shot) would be sufficient for our experiment. Thus, concluding that finding an optimal balance of the number of elements is out of scope for this study. The impact of this design choice results in 13 cases where the correct web element (i.e., according to our oracle) was not part of the top ten candidates sent to the LLM. Hence, making it impossible for the LLM to select the correct web element. As a result, by increasing the prompt size, VON Similo LLM could, theoretically, have reported 13 more identified web elements in this study. However, even doubling the number of candidates from VON Similo (i.e., from ten to 20) would have only resulted in five more instances where the correct element would have been part of the list of widgets sent to the LLM. As such, we concluded that the additional results would not outweigh the additional prompt size and cost of using the GPT-4 API.

```
Given the following candidate web elements (|| means that an attribute
    can have multiple values):
{widget_id:"202",tag:"a",text:"Beauty, Health",href:"https://www.
    aliexpress.com/category/66/health-beauty.html",location:"277,541",
    area:"1584",shape:"488",is_button:"no",xpath:"/html/body/div/div[5]/
    div/div[2]/div/div[2]/div/div[2]/dl[11]/dt/span/a",neighbor_text:"
    toys kids & babies outdoor fun sports beauty health hair automobiles
    motorcycles home improvement tools"}
{widget_id:"200",tag:"span || a",text:"Outdoor Fun & Sports",href:"https
    ://www.aliexpress.com/category/18/sports-entertainment.html",location
    :"236,497",area:"8400",shape:"685",is_button:"no",xpath:"/html/body/
    div/div[5]/div/div[2]/div/div[2]/div/div[2]/dl[10]/dt/span",
    neighbor_text:"bags & shoes toys kids babies outdoor fun sports
    beauty health hair automobiles motorcycles"}
{widget_id:"201",tag:"span",text:"Beauty, Health & Hair",location:"
    236,532",area:"8400",shape:"685",is_button:"no",xpath:"/html/body/div
    /div[5]/div/div[2]/div/div[2]/div/div[2]/dl[11]/dt/span",
    neighbor_text:"toys kids & babies outdoor fun sports beauty health
    hair automobiles motorcycles home improvement tools"}
{widget_id:"204",tag:"span || a",text:"Automobiles & Motorcycles",href:"
    https://www.aliexpress.com/category/34/automobiles-motorcycles.html",
    location:"236,567",area:"8400",shape:"685",is_button:"no",xpath:"/
    html/body/div/div[5]/div/div[2]/div/div[2]/div/div[2]/dl[12]/dt/span"
    ,neighbor_text:"outdoor fun & sports beauty health hair automobiles
    motorcycles home improvement tools"}
{widget_id:"197",tag:"span",text:"Toys , Kids & Babies",location:"236,462
    ",area:"8400",shape:"685",is_button:"no",xpath:"/html/body/div/div
    [5]/div/div[2]/div/div[2]/div/div[2]/dl[9]/dt/span",neighbor_text:"
```

```
    home pet & appliances home bags shoes toys kids babies outdoor fun
    sports beauty health hair"}
{widget_id:"199",tag:"a",text:"Kids & Babies",href:"https://www.
    aliexpress.com/category/1501/mother-kids.html",location:"313,471",
    area:"1476",shape:"455",is_button:"no",xpath:"/html/body/div/div[5]/
    div/div[2]/div/div[2]/div/div[2]/dl[9]/dt/span/a[2]",neighbor_text:"
    home pet & appliances bags shoes toys kids babies outdoor fun sports
    beauty  health hair"}
{widget_id:"194",tag:"span",text:"Bags & Shoes",location:"236,427",area:"
    8400",shape:"685",is_button:"no",xpath:"/html/body/div/div[5]/div/div
    [2]/div/div[2]/div/div[2]/dl[8]/dt/span",neighbor_text:"jewelry &
    watches home pet appliances home bags shoes toys kids babies outdoor
    fun sports"}
{widget_id:"206",tag:"a",text:"Home Improvement",href:"https://www.
    aliexpress.com/category/13/home-improvement.html",location:"277,611",
    area:"2196",shape:"677",is_button:"no",xpath:"/html/body/div/div[5]/
    div/div[2]/div/div[2]/div/div[2]/dl[13]/dt/span/a",neighbor_text:"
    beauty health & hair automobiles motorcycles home improvement tools"}
{widget_id:"205",tag:"span",text:"Home Improvement & Tools",location:"
    236,602",area:"11040",shape:"521",is_button:"no",xpath:"/html/body/
    div/div[5]/div/div[2]/div/div[2]/div/div[2]/dl[13]/dt/span",
    neighbor_text:"beauty health & hair automobiles motorcycles home
    improvement tools flash deals"}
{widget_id:"203",tag:"a",text:"Hair",href:"https://www.aliexpress.com/
    category/200002489/hair-extensions-wigs.html",location:"381,541",area
    :"450",shape:"138",is_button:"no",xpath:"/html/body/div/div[5]/div/
    div[2]/div/div[2]/div/div[2]/dl[11]/dt/span/a[2]",neighbor_text:"toys
     kids & babies outdoor fun sports beauty  health hair automobiles
    motorcycles home improvement tools"}

find the one that is most similar (answer with the widget_id of the most
    similar and motivate why using a list) to the element:
{tag:"span || a",text:"Health & Beauty",href:"https://www.aliexpress.com/
    category/66/health-beauty.html",location:"237,465",area:"7140",shape:
    "793",is_button:"no",xpath:"/html/body/div/div[4]/div/div/div[2]/div/
    div[2]/dl[11]/dt/span",neighbor_text:"toys, kids & baby toys sports
    outdoors health beauty automobiles motorcycles home improvement"}

The most similar element is the one with widget_id "201". The reasons for
     this choice are:

1. Both elements have "span" as one of their 'tag' attribute.
2. The text "Beauty, Health & Hair" in the element with widget_id "201"
    is closely related to the text "Health & Beauty" in the given element
    .
3. The 'location', 'shape', 'is_button', and 'neighbor_text' attributes
    in both elements have similar values, indicating that they might be
    close to each other on the layout of the website and have a similar
    structure.
4. Although the given element has an 'href' attribute and the element
    with widget_id "201" does not, this could be a minor change during
    the evolution of the web application, and the overall similarity of
    other attributes makes it the best candidate.
```

Listing 7.5: Sample GPT-4 prompt with response.

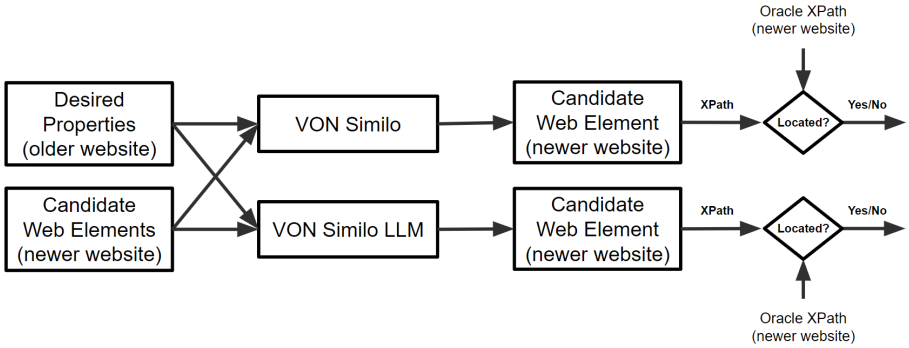

Figure 7.4: The process of locating a candidate web element from desired properties using the two approaches.

7.5.6 Locating Web Elements

We created a tool (implemented in Java and included in the replication package) that uses the extracted web element properties (see Section 7.5.2) to compare the effectiveness and efficiency of the two approaches.

Figure 7.4 shows the process of locating a candidate web element among the candidates extracted from the newer web application version based on the target's desired properties extracted from an older version of the same application. For each of the 804 web elements that were previously extracted from the older versions of 48 web applications, the desired properties and all the available candidates for the web application homepage were submitted as input to both approaches. VON Similo and VON Similo LLM then identify the candidate that holds properties most similar to the desired properties by comparing the properties of each candidate. Next, the XPaths of an identified candidate are compared with the Oracle XPath. Note that each candidate can have multiple XPaths due to the VON concept since a visual web element may consist of several DOM elements. The candidate is considered located if any of the candidate XPaths are identical to the Oracle XPath (and not located otherwise). Table 7.3 contains a summary of the two possible outcomes after a localization attempt.

We decided to divide the experiment into three phases. Figure 7.5 contains an overview of the phases further explained below. The first and last phases target research questions RQ1 and RQ2, while we aim to answer RQ3 with results from the second phase.

Table 7.3: Description of the localization result.

Localization result	Description
Located	The approach is able to identify the correct candidate web element where one of the XPaths is identical to the oracle.
Not located	The approach finds a match among the candidate web elements, but none of the XPaths are identical to the oracle.

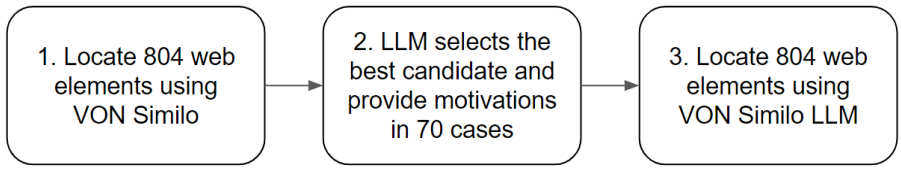

Figure 7.5: Overview of the three phases of the experiment.

1. Initially, we attempted to locate all the 804 web elements using VON Similo, which resulted in 70 not being found (see Section 4.4) in the newer web application versions based on the properties extracted from older versions.

2. Next, we asked the LLM (i.e., GPT-4) to identify the correct web element and motivate that choice, given the ten top-ranked elements provided by VON Similo, for the 70 cases where VON Similo failed. We analyzed the motivations given by the LLM to tell if the motivations were based on semantic understanding, context awareness, or using a standard comparison operator (i.e., like VON Similo). See definitions in Section 4.4. The three categories were coded based on literature that utilizes abilities in traditional algorithms, NLP, or LLMs when comparing GUI elements or creating input for testing [105, 106, 122, 125, 161].

   We decided to use a subset of the 804 cases to lower the cost of using the LLM API and to reduce the number of motivations to categorize. Selecting the cases where VON Similo failed has several benefits: (1) it is a significantly smaller sample (i.e., less costly), (2) the correct alternative is never the first candidate (i.e., since VON Similo failed), making the choice less evident, and (3) it is more valuable if the LLM can find the

Table 7.4: The total number of located (and not located) web elements for the two approaches.

Approach	Total	Located	Not located	% Located	Cost ($)	Time / localization (ms)
VON Similo	804	734	70	91.3	0	29
VON Similo LLM (one-shot)	804	764	40	95.0	35.86	1934 (STD 537)

correct web element when the conventional approach (i.e., VON Similo) fails.

3. Finally, to evaluate VON Similo LLM, we extracted the top ten elements that best match all of the 804 oracles (i.e., correct targets) using VON Similo and asked the LLM to select the candidate that is most similar to the oracle (i.e., the properties extracted from the older version) for all oracles. To optimize (i.e., reduce) the cost and time of the experiment, we did not ask the LLM to provide us with motivations, instead only to return with the id of the best candidate. This design greatly reduced the output prompt from the LLM and, thereby, the execution time since each output character increases the execution time and cost of using the LLM API.

7.6 Results

In this section, we present the results of the experiment study. We present the results according to the order of the study's three research questions.

7.6.1 RQ1 - Effectiveness

Table 7.4 contains the result when comparing the effectiveness, in terms of being able to locate the correct candidate based on desired properties, of the two approaches. When attempting to identify the correct candidate in 804 cases extracted from 48 web applications, VON Similo failed to locate the correct candidate in 70 cases (i.e., 91.3% correctly located). In comparison, the VON Similo LLM approach (i.e., use an LLM to identify the best candidate among the ten provided by VON Similo) only failed in 40 cases (i.e., 95.0% correctly located). Thus, resulting in a 42.9 percent reduction of not-located web elements when using VON Similo LLM.

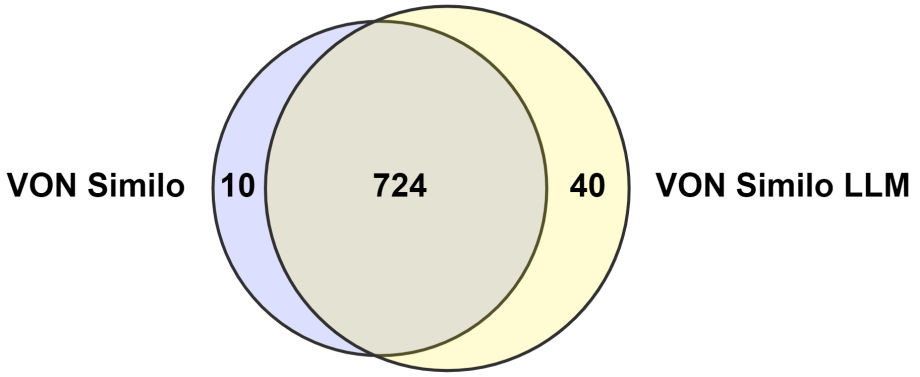

Figure 7.6: Venn diagram containing the number of correctly located candidates (i.e., web elements) for each approach.

The Venn diagram in Figure 7.6 shows the number of located web elements by VON Similo and VON Similo LLM. Both approaches located 724 of the correct candidates. The VON Similo LLM approach located 40 candidates that VON Similo did not locate, and VON Similo located ten candidates that VON Similo LLM failed to locate.

Because we instructed the LLM to only provide a widget id as output and no motivation, in this experiment, it is impossible to analyze why VON Similo LLM did not find the ten cases VON Similo found (further elaborated on in Section 4.6).

To summarize, for what concerns research question RQ1, using the VON Similo LLM approach instead of the conventional VON Similo algorithm, we reduced the number of not located candidates from 70 to 40 cases, i.e., 42.9 percent.

7.6.2 RQ2 - Efficiency

The Time/localization column in Table 7.4 shows the average time in milliseconds to locate one candidate using both approaches (29 vs. 1934 ms). Also, within parentheses, the standard deviation is included for the VON Similo LLM approach (537 ms). We were unable to measure the standard deviation of the VON Similo approach due to the lack of precision (i.e., we could only measure

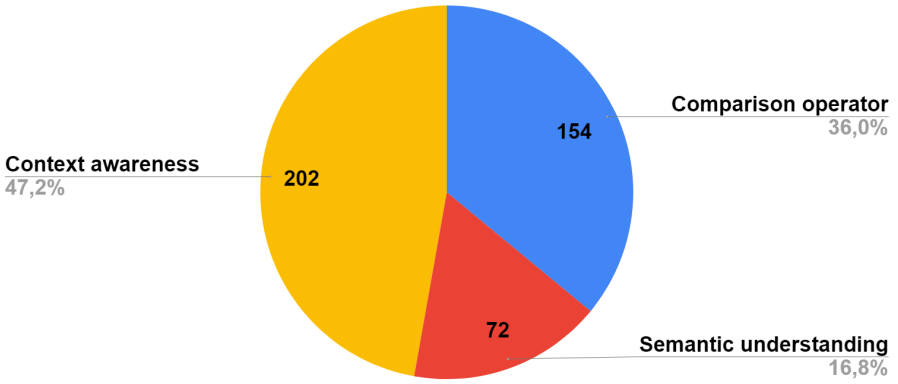

Figure 7.7: Motivations from the LLM classified as codes.

whole milliseconds). As expected, the performance of the VON Similo algorithm is much higher (i.e., almost two magnitudes lower execution time) than the VON Similo LLM approach due to the slow response time of the GPT-4 API.

To summarize research question RQ2, the performance of the LLM approach at the time of writing is almost two magnitudes slower than the conventional algorithm due to the long response time from the GPT-4 API (i.e., around 2 seconds on average). While we cannot generalize this result to all LLM solutions, it gives insights into a snapshot of the order of magnitude of time required when we conducted this study.

7.6.3 RQ3 - What main aspects does a large language model use when comparing similar web elements when a conventional approach struggles?

Figure 7.7 shows the results from our qualitative content analysis of the 428 motivations provided by GPT-4 for all the 70 cases (i.e., six motivations per case, on average) when VON Similo could not identify the correct candidate. We provided motivations for selecting the cases where VON Similo failed in Section 7.5.6.

We defined three categories of motivations before the analysis (see Section 7.5.6) to be able to evaluate how frequently GPT-4 incorporates either of the

aspects; comparison operator, semantic understanding, or context awareness, in its motivations:

- **Comparison operator:** Motivation based on conventional comparison operators (e.g., equals, Euclidean distance, Levenshtein distance). Hence, the only category that the VON Similo approach uses to identify elements.
- **Semantic understanding:** Motivation based on semantic understanding. Semantic understanding refers to interpreting the meaning of information within its context. It involves understanding the relationships between words, sentences, and concepts and the intended or implied meaning behind them.
- **Context awareness:** Motivation based on context awareness. Context awareness refers to the capability to perceive and understand the situational context (e.g. layout and positioning of elements in web applications, in our case).

We categorized 202 motivations (i.e., 47%) to be associated with context awareness, 72 motivations (i.e., 17%) to be associated with semantic understanding, and 154 motivations (i.e., 36%) to be associated with the use of some form of conventional comparison operation (e.g., equals). Some motivations could belong to more than one category. In those cases, we sorted the motivation into the nearest category (i.e., the most appropriate according to the authors). We did not encounter any motivations leading us to refine existing, or add new, categories while performing the analysis. Table 7.5 contains examples of motivations returned from GPT-4, categorized as associated with the comparison operator, semantic understanding, or context awareness. Half of the motivations were gathered from the first prompt responses, while the remaining examples were manually selected to show some alternate or interesting motivations. When comparing motivations from GPT-4 when it was correct or incorrect (i.e., selected the correct candidate according to the oracle), we did not find any pattern in the motivations that would indicate when it was more or less confident of the selection of the most similar candidate.

To summarize research question RQ3, the result indicates that the LLM mainly uses context awareness or semantic understanding (64% of the time) rather than relying on some form of comparison operation (i.e., like a conventional, non-AI algorithm) in more complex cases where a conventional approach failed.

Table 7.5: Example motivations from GPT-4 classified as comparison operator, semantic understanding, or context awareness.

Examples that indicate that Comparison operator is used to find the target	**Examples that indicate that Semantic understanding is used to find the target**	**Examples that indicate that Context awareness is used to find the target**
The 'location' attribute in both elements is the same: "20,20".	The text "Upgrade to premium" in the element with widget_id "8817" is closely related to the text "Get premium" in the given element.	The 'location' attribute indicates that they might be far apart in the layout of the website, but the 'neighbor_text' attribute has some overlapping words (e.g., "spotify", "support", "download", "premium").
Both elements have "span" as one of their 'tag' attribute.	The text "Beauty, Health & Hair" in the element with widget_id "201" is closely related to the text "Health & Beauty" in the given element.	The 'location', 'shape', 'is_button', and 'neighbor_text' attributes in both elements have similar values, indicating that they might be close to each other on the layout of the website and have a similar structure.
Both elements have "a" or "span" as their 'tag' attribute.	The text "Sign up" in the element with widget_id "8065" is closely related to the text "Log in" in the given element, as both texts are related to account actions.	Although the given element has an 'href' attribute and the element with widget_id "201" does not, this could be a minor change during the evolution of the web application, and the overall similarity of other attributes makes it the best candidate.
The 'class' attribute values in both elements are very similar, containing "nav-logo-base" and "nav-sprite".	The text "Order Status" in the element with widget_id "1823" is not exactly the same as the text "Shopping History" in the given element, but both texts are related to account and order information, which leads to the assumption that they are similar in purpose.	Both elements have a similar 'location' attribute, indicating that they might be close to each other on the layout of the website.

Table 7.6: Example motivations continued.

Examples that indicate that Comparison operator is used to find the target	**Examples that indicate that Semantic understanding is used to find the target**	**Examples that indicate that Context awareness is used to find the target**
The 'href' attribute in both elements is the same, as they both point to the same URL ("https://www.cnn.com/us").	The text "Account" in the element with widget_id "1815" is not exactly the same as "Store Locator" in the given element, but there's no other candidates with the text "Store Locator". In this case, "Account" may represent a location-related functionality.	Both elements have relatively large 'area' and 'shape' attributes, suggesting that they are both prominent elements on the webpage.
The text "Enterprise" is exactly the same in both elements.	The text "Start your free trial" in the element with widget_id "3214" is closely related to the text "Try free for 30 days" in the given element.	Both elements have a similar 'location' attribute with only a minor difference in the x coordinate, indicating that they are situated near each other on the layout of the website.
The 'id' attribute in both elements is the same: "hero-banner-get-office-link".	The text "Support" in the element with widget_id "10880" is closely related to the text "Help" in the given element. Both serve the same purpose of assisting users with issues or questions.	Despite some differences in 'xpath', both elements seem to be part of the navigation menu, as suggested by the 'neighbor_text' attribute.
The text "Find jobs" in the element with widget_id "7973" is identical to the text "Find Jobs" in the given element.	The 'neighbor_text' attribute is similar in both elements, with both mentioning social platforms like "twitter", "instagram", "snapchat", "youtube", and "the espn daily podcast".	The text "Items in cart" in the given element is related to the functionality of a shopping cart, and the element with widget_id "12341" also has a cart-related functionality, although the text is not present.
Both elements share the same 'href' attribute, which points to "https://www.instructure.com/".	The text "Claims Support" in the element with widget_id "11882" is closely related to the text "Delivery Issues" in the given element, as both deal with issues regarding deliveries.	The 'xpath' and 'neighbor_text' attributes also show similarities, suggesting that they are part of the same group of links within the footer of the website.
The text "Cart" is present in both elements.	The text "Plans & Pricing" in the element with widget_id "13858" is closely related to the text "PLANS" in the given element.	Although the 'href' attribute is different, the change could be due to the updated web application using a different method to handle account sign-in functionality.

7.7 Discussion

LLMs with human-like abilities such as semantic understanding and context awareness have the potential to increase the effectiveness of identifying web elements. Instead of just comparing attributes and other properties (i.e., like a conventional algorithm), LLMs can relate to the meaning of neighbor texts, understand the purpose of an element, and evaluate the structure (i.e., both the DOM and visually in terms of layout and element placement) to make more informed decisions when comparing and identifying web elements. One example is the following motivation from the LLM: "The 'location' attribute indicates that they might be far apart in the layout of the website, but the 'neighbor_text' attribute has some overlapping words (e.g., 'spotify', 'support', 'download', 'premium'.)". This and the following examples can be found in Table 7.5. LLMs recognize common patterns such as menus, forms, footers, or groups and use this contextual information to refine the identification process. For example, the LLM motivated one decision with the text: "The 'xpath' and 'neighbor_text' attributes also show similarities, suggesting that they are part of the same group of links within the footer of the website.". Another example is: "Despite some differences in 'xpath', both elements seem to be part of the navigation menu, as suggested by the 'neighbor_text' attribute.". With almost human-like abilities when identifying web elements, LLMs can reduce the need for manual intervention and script maintenance in tools and frameworks for web-based test automation. More reliable test scripts save time for the human testers, who can focus on more meaningful tasks like test strategies and exploratory testing.

There is also a downside to utilizing GPT-4 (i.e., the LLM used in our experiment) for web element identification. API requests are very slow today compared to a conventional algorithm like VON Similo. We measured the average API request to be around two seconds, which would result in a noticeable delay even in an automated GUI script (i.e., that, in turn, is very slow compared to Unit tests). Although we expect future advancements of GPT and other LLMs to become faster, there might always be some delay that would affect the execution time of the automated test script in a noticeable way.

Using GPT-4 also comes with a cost in terms of a fee charged by OpenAI for utilizing the API. The cost is not easy to grasp since it is based on the number of tokens sent between the client and server. According to our measurements, see Table 7.4, the cost is not negligible ($36 for 804 prompts, i.e., $0.045 per prompt) and needs to be taken into consideration when evaluating if the price of using the API (i.e., runtime cost) is lower than the expected reduction in maintenance

cost. Such a calculation is complicated due to the many variables that affect the maintenance cost (e.g., software maturity, time between releases, number of test cases, size of the test cases, and the salary of developers). However, assuming a test suite with an average maintenance time of 110 minutes per test case between two major versions, 47 localizations on average per test case, and an estimated cost of 100 dollars per hour for an employee (as reported in Alégroth et al. [27]), the LLM approach would likely provide a positive return on investment in just one software release cycle. The cost of test case maintenance would be: 100 * (110 / 60) = $183, and the cost of using the GPT-4 API would be: 47 * 0.045 = $2. Since the cost of utilizing the GPT-4 API is negligible compared to the manual maintenance cost, the additional robustness gained by the LLM approach would likely be more valuable. Assuming the same increase in robustness as in our results (40 vs. 70 not located), the maintenance cost would be reduced to: 183 * (40 / 70) = $105. Even though the calculations are based on industrial data, they only provide an indication since salaries for engineers differ globally.

However, the cost and payment plans will likely also change over time. There may even be adequate LLMs that are entirely free or that you can install locally, eliminating, at least, the cost aspect. A locally installed LLM would probably also impact the performance but may not eliminate the problem that the API request delay has a noticeable effect on the test execution. That GPT-3.5-turbo is considerably faster than GPT-3 is also an indication that we might expect to see a turbo version of GPT-4 in the future.

Since LLMs are based on artificial neural networks (ANNs) [79], we can only assume that the motivations provided by the LLM have anything to do with the candidate selected since a large ANN can be seen as a black box model (i.e., with inputs and outputs) that we cannot fully comprehend due to the complexity of the network. Ongoing discussion exists about whether LLMs are probabilistic models or if they truly learn to understand the world [116, 139, 165].

In summary, LLMs can be used to further improve web element localization due to their assumed semantic understanding and context awareness with the drawbacks of slower test execution and the cost of using the API. However, more research is needed to fully grasp the potential and shortcomings of using LLMs for web element identification and if the models actually possess knowledge about context and semantics.

7.8 Threats to Validity

To reduce the internal validity threat, we limited the influence of the selection of web elements on our experiment by selecting specific categories of web elements that could be used for actions, assertions, or synchronization and that were available on both versions of the website's homepage. As we focused on investigating the web element finding ability only, we do not believe that the possibility that elements on the homepage differ significantly from other web elements is a substantial threat.

The choice of applications and versions analyzed in the study may compromise its external validity. To address this issue, we opted to focus on the top 48 sites based on Alexa.com rankings, as we have no control over the websites listed on that site. Additionally, the version of a website can impact the number of failed localization attempts, mainly since longer intervals between releases often result in more changes. To mitigate this concern, we selected the same interval (one to five years) for website versions as previous studies conducted by Leotta et al. [99] and Nass et al. [122].

The construct validity is low since the time between releases (i.e., between 12 to 60 months) should be compared with a typical case in the industry. However, industrial cases differ a lot. Some test suites are run every time the source code is updated (i.e., several times per day), while some test suites are run with months in between. We decided to prioritize getting some changes (i.e., both larger and smaller) by picking a greater period between releases to reduce the risk of not finding any changes at all.

That we selected to use GPT-4 from OpenAI in favor of some other LLM can impact the construct validity since choosing a different LLM would likely give a different result. We tried to mitigate this threat by motivating our selection of LLM when focusing on effectiveness before efficiency and cost. Also, future LLM versions will likely be even more capable, making this study merely a baseline for the future.

Limiting the number of candidates to ten was made to prevent the prompt from exceeding the quota and reduce the API's runtime cost. However, this design choice leads to 13 cases where none of the top ten candidates are correct, resulting in inevitable failure for the LLM. This threat to validity arises because the chosen constraint on the number of candidates potentially restricts the LLM's ability to provide accurate responses. By limiting the available options, the experiment does not fully assess the LLM's ability to generate appropriate and correct responses. This limitation could lead to an underestimation of the LLM's performance, as it may have the potential to generate correct responses

beyond the limited set of candidates. An alternative approach to mitigating this threat would have been to increase the number of candidates to 20, which would have included additional correct candidates in the top 10. However, concerns about the impact on prompt size and cost led to the decision against this option.

7.9 Related Work

In practical terms, two categories of methodologies have emerged, each possessing contrasting yet non-contradictory characteristics: post-repair approaches that address locator failures by employing remedial measures and more preventive strategies that focus on generating resilient locators. Only a few of the current algorithms and approaches utilize natural-language processing (NLM) or large language models (LLMs). This Section covers them both, emphasizing the ones taking advantage of LLM or NLM.

7.9.1 Post-repair approaches

This category of approaches aims to automatically repair the automated test execution or script after a failure has occurred (i.e., post-execution). Automatic repair reduces the costly manual labor of repairing test cases or scripts and has been researched by many, e.g., [49, 75, 86].

Khaliq et al. [84] proposed a novel automated GUI testing approach using a sequence-to-sequence transformer model in GPT-2, which perceives the application state through element classification and generates test flows in English. Their model aims to repair flaky tests when the GUI is modified and automatically generate new test flows for regression without manual intervention. They showed that abstract English test flows could be converted into executable test scripts using a simple parser.

A more conventional approach (i.e., non-AI), named WATER, proposed by Choudhary et al. [49], compares the test execution on two software versions, one where the test succeeds and one when it fails. In common with Similo (and VON Similo), WATER uses weighted locator parameters when repairing a broken locator. The WATER approach is, however, a post-repair technique and utilizes an entirely different set of locator parameters than Similo (i.e., XPath, coord, clickable, visible, index, and hash) that are compared using equality or Levenshtein distance [8].

Another post-repair tool is WATERFALL [75]. WATERFALL is an advancement on WATER and uses the same heuristics for repairs but can improve the effectiveness of script repair (by 209%) by taking advantage of the knowledge that minor versions occur between major versions in software releases.

COLOR, proposed by Kirinuki et al. [86], is another approach that uses several attributes, positions, images, and other properties to suggest a repair. Their experiments show that COLOR can be more effective than WATER (especially concerning more complex changes, like switching from one web page to another) and that the algorithm can identify the repair with 77% to 93% accuracy.

Repairing broken locators utilizing a DOM tree comparing algorithm is an approach presented by Brisset et al. [42]. They compared their tool, Erratum, with WATER and found that it has 67% higher accuracy.

Grechanik et al. [70, 163] proposed GUIDE, a tool for a non-intrusive, platform-, and language-independent repair algorithm for web applications by identifying changes occurring between two released software versions. The tool can be used for suggesting repairs or providing guidance for test planning.

7.9.2 Resilient locators

Resilient (i.e., robust) locators in GUI test automation refer to the challenge of reliably identifying and interacting with GUI elements during automated testing. Changes in GUI layout and dynamic content can cause locators to fail, leading to test script failures. Researchers aim to develop techniques for generating robust locators tolerant to GUI changes, ensuring efficient and reliable test automation. Many approaches have been proposed seeking to mitigate this problem in the literature.

A study by Kirinuki et al. attempts to solve the locator maintenance problem by not relying on attributes and the structure of the DOM and instead leverages NLP with heuristic search to identify web elements in web pages from natural-language-like test cases [85]. An example of such a test step could be: enter 'admin' in 'username'. Evaluation of three open-source web applications showed a success rate of 94% in identifying web elements and correct identification in 68% of the test cases.

Another interesting approach that takes advantage of GPT (i.e., GPT-3 in this case) while avoiding the shortcomings of a traditional test script is GPT-Droid, proposed by Zhe Liu et al. [106]. Utilizing the strengths of ChatGPT (i.e., understanding human knowledge), they formulate test steps in plain English and pass the GUI page content to the LLM. Next, the LLM responds

with an instruction about what step to do next when asked: 'What operation is required?'.

Zhe Liu et al. also proposed to use the power of an LLM to automatically generate more realistic test scenarios that can interact with a GUI application more similar to a human tester, for example, fill out forms with suitable content that makes it possible to progress to the next step. Their tool QTypist, can generate text input related to the GUI context and semantic requirement, thereby enabling better test coverage [105].

CrawLabel is a test-generation tool (a plugin for Crawljax) that utilizes grammar learning (i.e., NLP) to perform unsupervised end-to-end testing of web applications [104].

Among the more traditional algorithms (i.e., not utilizing some form of AI), we need to mention the approaches (i.e., Similo and VON Similo) we aim to advance in this paper. The different Similo approaches are covered, in detail, in Section 7.3.

Several approaches attempt to create robust XPath locators. The algorithm proposed by Montoto et al. [117] is one of them and uses a bottom-up strategy to generate a change-resilient XPath locator iteratively. Starting from a simple XPath expression, the algorithm concatenates sub-expressions until the resulting XPath can uniquely identify the target element. If the resulting XPath is not unique, the attribute values of the ancestors are considered until the root is reached.

Other approaches that generate robust XPaths are ROBULA [98] and ROBULA+ [100], proposed by Leotta et al. ROBULA+ improves upon the earlier ROBULA algorithm and is often considered state-of-the-art in generating resilient XPath locators for web applications. The idea behind ROBULA+ is to generate a short but robust locator as possible, given the content of the web page and heuristics about the robustness of various attributes. ROBULA and ROBULA+ begin with a generic XPath that selects all the nodes in the DOM (i.e., similar to the Montoto approach). Next, the algorithms refine the XPath, using a set of transformations or prioritizations until only one element is selected.

While some solutions aim to increase the resilience of XPath locators (e.g., ROBULA+ and Montoto), other approaches increase the number of information sources (e.g., attributes and other properties), thereby introducing voting mechanisms or triangulation when identifying the target web element. The multi-locator, proposed by Leotta et al., is an example that takes advantage of several locators (i.e., with diverse strengths and weaknesses) and uses a voting procedure to select the best candidate web element (i.e., the top-voted one) [99].

Another interesting approach, ATA-QV, proposed by Yandrapally et al. [166], is to take advantage of neighboring web elements instead of only relying on attributes and properties of each web element. We can use the information extracted from neighbor web elements to triangulate the location of the target web element. For example, assume we have a text field with a label describing the text field on the left and a button on the right side. Even if the attributes and properties of the text field change entirely from one version to the other, it might still be possible to find it by utilizing the label on the left side and the button's caption on the right side. ATA-QV is an improvement to the technique and tool called ATA proposed by Thummalapenta et al. [151]. ATA is a commercial tool that was developed in collaboration with IBM that aims to increase the resilience of locators by relying more on labels (i.e., visual attributes) than the DOM structure.

Nguyen et al. recently suggested an approach that can generate resilient locators by using a new way of constructing XPaths that relies on semantic structures and neighbor web elements and a rule-based method for selecting the best (i.e., most robust) one [129].

SIDEREAL is a tool for automated end-to-end (E2E) testing of web applications [97]. It addresses the problem of broken locators by using a statistical adaptive algorithm that learns the potential fragility of web element properties to generate robust XPath locators. Compared to the baselines (i.e., ROBULA+ and Montoto), SIDEREAL significantly reduces the number of broken locators, resulting in more reliable E2E testing for web applications.

There are also some commercial products that can learn and adapt their web element localization from existing applications or application versions, like Testitm [2] and Ranorex [3].

The Similo approach combines many of the techniques of these related works. For example, Similo utilizes multiple sources of information like the multi-locator approach by Leotta et al. and triangulating using neighbor web elements like the ATA-QV approach by Yandrapally et al. [166].

VON Similo LLM enhances standard Similo by adding a semantic understanding of attributes (e.g., the caption) in web elements like the approaches proposed by Kirinuki et al. and Zhe Liu et al. [106]. However, VON Similo LLM goes beyond the semantic understanding of web elements since GPT-4 displays some form of context awareness by relating to the possible use of web

[2] https://www.testim.io/blog/why-testim/
[3] https://www.ranorex.com/blog/machine-trained-algorithm/

elements in a web page or application, taking it even further than the ATA-QV approach by Yandrapally et al. [166].

7.10 Conclusions

Accurate web element localization is crucial for robust automated scripts in web-based test automation. Traditional approaches lack semantic understanding and context awareness. The emergence of Large Language Models (LLMs) like GPT-4 offers human-like abilities that can enhance web element identification. This study highlights the potential benefits (but also challenges) of using LLMs for web element localization in an automated GUI test case. Our results show that LLMs can be employed to understand the purpose of elements, analyze neighboring text, and evaluate web page structures, enabling more accurate localizations. They can reduce manual intervention and script maintenance, freeing human testers' time for more meaningful tasks. However, using LLMs through APIs like GPT-4 introduces delays in test execution due to long response times. The cost of utilizing the API is another factor to consider, as it can be significant and needs to be weighed against the expected reduction in maintenance costs. Future advancements and alternatives, such as locally installed LLMs, may address these concerns. Overall, further research is necessary to fully understand the potential and limitations of using LLMs for web element identification.

7.11 Future Work

Even though the VON Similo LLM approach exceeds a 95% success rate when locating the correct candidate, there are still almost 5% to a perfect result. Still, we do not know how the approach compares to humans since they might not reach 100% success either. However, we expect LLMs to become even more capable in the future. They will also likely support more extensive prompt length (i.e., more tokens), become faster (i.e., lower response times), and the cost of using the APIs will decrease.

As a next step, we envision an approach that only relies on an LLM without needing a conventional algorithm to narrow down the number of candidates (e.g., VON Similo) that have the potential to enhance the effectiveness of web element localization further. Such an approach could employ tournament selection [41] where all the visual web element candidates extracted from a web page attend,

and the tournament winner is the selected candidate. For example, assume 200 visual web elements extracted from a web page. First, we divide the 200 candidates into ten groups of 20 candidates each. The winner of each group will attend the final that selects the most similar candidate on the web page. Our reasons for not trying such an approach today are: (1) a tournament would take a long time to complete since it requires many API requests, and (2) the cost would be high since the prompts will contain information gathered from all the web elements on the web page. However, as advancements in LLM technology continue and API efficiency improves, the viability of such an approach may increase, making it promising for future exploration.

Another possible improvement is to provide the LLM with more information about the candidates to compare. One such example could be a representation of the pictorial user interface (i.e., pixels) since that type of information is available to the human eye. We decided to leave that out of our experiments since gathering and processing images from all visible images is likely time-consuming. Also, there are many ways of processing and analyzing images, and exploring the alternatives would take lots of resources and time.

Instead of just asking the LLM once (i.e., one input returns one output) as in our experiment, we could employ other frameworks such as Chain of Thoughts (CoT) or Three Of Thoughts (ToT) that try to improve the results using a process of exploration of thoughts and self-evaluation [167]. The drawback is, again, that more prompts increase the time and cost of using the API.

A possible way of increasing the efficiency and reducing the cost is to use VON Similo in cases when we expect it to be correct (i.e., a high probability) and only take advantage of the LLM in other cases. This approach involves comparing the similarity score of the highest-ranked candidate with the remaining candidates to determine if it stands out as an outlier (i.e., clearly separated from the rest). If a clear separation is detected, the top-ranked candidate from VON Similo is chosen as the result. However, if no outlier is identified, the LLM is employed to decide among the top ten (or more) candidates. This approach optimizes efficiency and cost by using the most appropriate model based on the probability of correctness and the distinctiveness of the top-ranked candidate. The challenge with this approach is that imperfect detection of the outlier has a negative impact on the effectiveness since the LLM will not get the opportunity to find a better candidate.

The GPT API (all versions) is today provided as a cloud service. One potential drawback of utilizing a cloud service is the inherent security risks associated with transmitting sensitive data to remote servers outside the company domain. Relying on a third-party cloud provider might be a reason for not taking ad-

vantage of the benefits an LLM can provide regarding script robustness due to the possible security risk. We might be able to solve this risk in the future by using an LLM that is powerful enough, and that can be locally installed, thus avoiding a cloud service.

7.12 Acknowledgements

This work was supported by the KKS foundation through the S.E.R.T. Research Profile project at Blekinge Institute of Technology. Robert Feldt has also been supported by the Swedish Scientific Council (No. 2015-04913, 'Basing Software Testing on Information Theory').

References

[1] Alexa. https://www.alexa.com/topsites/countries/US.

[2] Angular. https://angular.io.

[3] Css selectors. https://en.wikipedia.org/wiki/CSS.

[4] Dom. https://www.w3.org/TR/WD-DOM/introduction.html.

[5] Eggplant. https://www.eggplantsoftware.com.

[6] Github. https://github.com/SeleniumHQ/selenium-ide.

[7] Internet archive. https://web.archive.org/.

[8] Levenshtein. http://levenshtein.net.

[9] Protractor. https://www.protractortest.org.

[10] Replication package. http://www.michelnass.com/resources/WidgetLocator.zip.

[11] Selenium. https://www.seleniumhq.org.

[12] Selenium ide. https://www.selenium.dev/selenium-ide.

[13] The v-model. https://en.wikipedia.org/wiki/V-Model_(software_development).

[14] Wikipedia. https://www.wikipedia.org.

[15] Xpath. https://en.wikipedia.org/wiki/XPath.

[16] Replication package, 2023. https://github.com/michelnass/SimiloLLM.

[17] Adamoli, A., Zaparanuks, D., Jovic, M., and Hauswirth, M. Automated gui performance testing. *Software Quality Journal 19*, 4 (2011), 801–839.

[18] Afzal, W., Ghazi, A. N., Itkonen, J., Torkar, R., Andrews, A., and Bhatti, K. An experiment on the effectiveness and efficiency of exploratory testing. *Empirical Software Engineering 20*, 3 (2015), 844–878.

[19] Aho, P., Suarez, M., Kanstrén, T., and Memon, A. M. Murphy tools: Utilizing extracted gui models for industrial software testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (2014), IEEE, pp. 343–348.

[20] Aho, P., and Vos, T. Challenges in automated testing through graphical user interface. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2018), IEEE, pp. 118–121.

[21] Aldalur, I., and Diaz, O. Addressing web locator fragility: a case for browser extensions. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (2017), pp. 45–50.

[22] Aldalur, I., Larrinaga, F., and Perez, A. Abla: An algorithm for repairing structure-based locators through attribute annotations. In *International Conference on Web Information Systems Engineering* (2020), Springer, pp. 101–113.

[23] Alégroth, E. *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice*. Chalmers University of Technology, 2015.

[24] Alégroth, E. *Visual gui testing: Automating high-level software testing in industrial practice*. Chalmers University of Technology, 2015.

[25] Alégroth, E., Bache, G., and Bache, E. On the industrial applicability of texttest: An empirical case study. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (2015), IEEE, pp. 1–10.

[26] Alégroth, E., and Feldt, R. On the long-term use of visual gui testing in industrial practice: a case study. *Empirical Software Engineering 22*, 6 (2017), 2937–2971.

[27] Alégroth, E., Feldt, R., and Kolström, P. Maintenance of automated test suites in industry: An empirical study on visual gui testing. *Information and Software Technology 73* (2016), 66–80.

[28] Alégroth, E., Feldt, R., and Olsson, H. H. Transitioning manual system test suites to automated testing: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* (2013), IEEE, pp. 56–65.

[29] Alégroth, E., Gao, Z., Oliveira, R., and Memon, A. Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (2015), IEEE, pp. 1–10.

[30] Alégroth, E., Karlsson, A., and Radway, A. Continuous integration and visual gui testing: Benefits and drawbacks in industrial practice. In *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on* (2018), IEEE, pp. 172–181.

[31] Alégroth, E., Nass, M., and Olsson, H. H. Jautomate: A tool for system-and acceptance-test automation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* (2013), IEEE, pp. 439–446.

[32] Ali, M., and Elish, M. O. A comparative literature survey of design patterns impact on software quality. In *2013 international conference on information science and applications (ICISA)* (2013), IEEE, pp. 1–7.

[33] Amalfitano, D., Riccio, V., Amatucci, N., De Simone, V., and Fasolino, A. R. Combining automated gui exploration of android apps with capture and replay through machine learning. *Information and Software Technology 105* (2019), 95–116.

[34] Anand, T., Reddy, C., and Mani, V. System testing optimization in a globally distributed software engineering team. In *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)* (2016), IEEE, pp. 99–103.

[35] Arlt, S., Bertolini, C., and Schäf, M. Behind the scenes: an approach to incorporate context in gui test case generation. In *2011 IEEE*

Fourth International Conference on Software Testing, Verification and Validation Workshops (2011), IEEE, pp. 222–231.

[36] Arlt, S., Podelski, A., and Wehrle, M. Reducing gui test suites via program slicing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014), ACM, pp. 270–281.

[37] Bauersfeld, S., Vos, T. E., Condori-Fernandez, N., Bagnato, A., and Brosse, E. Evaluating the testar tool in an industrial case study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2014), ACM, p. 4.

[38] Benavoli, A., Corani, G., Demšar, J., and Zaffalon, M. Time for a change: a tutorial for comparing multiple classifiers through bayesian analysis. *The Journal of Machine Learning Research 18*, 1 (2017), 2653–2688.

[39] Berner, S., Weber, R., and Keller, R. K. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering* (2005), ACM, pp. 571–579.

[40] Bertram, D. Likert scales. *Retrieved November 2* (2007), 2013.

[41] Blickle, T., and Thiele, L. A mathematical analysis of tournament selection. In *ICGA* (1995), vol. 95, Citeseer, pp. 9–15.

[42] Brisset, S., Rouvoy, R., Seinturier, L., and Pawlak, R. Erratum: Leveraging flexible tree matching to repair broken locators in web automation scripts. *Information and Software Technology 144* (2022), 106754.

[43] Brooks, F., and Kugler, H. *No silver bullet.* April, 1987.

[44] Bruns, A., Kornstadt, A., and Wichmann, D. Web application tests with selenium. *IEEE software 26*, 5 (2009), 88–91.

[45] Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[46] Campos, J. C., Fayollas, C., Gonçalves, M., Martinie, C., Navarre, D., Palanque, P., and Pinto, M. A more intelligent test case generation approach through task models manipulation. *Proceedings of the ACM on Human-Computer Interaction 1*, EICS (2017), 9.

[47] Chang, T.-H., Yeh, T., and Miller, R. C. Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010), ACM, pp. 1535–1544.

[48] Cheng, Y.-P., Li, C.-W., and Chen, Y.-C. Apply computer vision in gui automation for industrial applications. *Mathematical biosciences and engineering: MBE 16*, 6 (2019), 7526–7545.

[49] Choudhary, S. R., Zhao, D., Versee, H., and Orso, A. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering* (2011), pp. 24–29.

[50] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[51] Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., and Amodei, D. Deep reinforcement learning from human preferences. *Advances in neural information processing systems 30* (2017).

[52] Cohn, M. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[53] Coppola, R., Morisio, M., and Torchiano, M. Mobile gui testing fragility: a study on open-source android applications. *IEEE Transactions on Reliability 68*, 1 (2018), 67–90.

[54] Cruzes, D. S., and Dyba, T. Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement* (2011), IEEE, pp. 275–284.

[55] de Cleva Farto, G., and Endo, A. T. Reuse of model-based tests in mobile apps. In *Proceedings of the 31st Brazilian Symposium on Software Engineering* (2017), ACM, pp. 184–193.

[56] de Gier, F., Kager, D., de Gouw, S., and Vos, E. T. Offline oracles for accessibility evaluation with the testar tool. In *2019 13th International Conference on Research Challenges in Information Science (RCIS)* (2019), IEEE, pp. 1–12.

[57] Debroy, V., Brimble, L., Yost, M., and Erry, A. Automating web application testing from the ground up: Experiences and lessons learned in an industrial setting. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)* (2018), IEEE, pp. 354–362.

[58] Dobslaw, F., Feldt, R., Michaëlsson, D., Haar, P., de Oliveira Neto, F. G., and Torkar, R. Estimating return on investment for gui test automation frameworks. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)* (2019), IEEE, pp. 271–282.

[59] Ekman, F., Johannesson, S., Peber, E., Sandberg, C., et al. Test-driven development: Drawbacks, benefits, industrial usage and complementary methods.

[60] Eladawy, H. M., Mohamed, A. E., and Salem, S. A. A new algorithm for repairing web-locators using optimization techniques. In *2018 13th International Conference on Computer Engineering and Systems (ICCES)* (2018), IEEE, pp. 327–331.

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

[62] Entin, V., Winder, M., Zhang, B., and Christmann, S. Introducing model-based testing in an industrial scrum project. In *Proceedings of the 7th International Workshop on Automation of Software Test* (2012), IEEE Press, pp. 43–49.

[63] Feldt, R., Kang, S., Yoon, J., and Yoo, S. Towards autonomous testing agents via conversational large language models. *arXiv preprint arXiv:2306.05152* (2023).

[64] Fleiss, J. L., and Cohen, J. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and psychological measurement 33*, 3 (1973), 613–619.

[65] Furia, C. A., Feldt, R., and Torkar, R. Bayesian data analysis in empirical software engineering research. *IEEE Transactions on Software Engineering 47*, 9 (2019), 1786–1810.

[66] Garousi, V., Afzal, W., Çağlar, A., Işik, İ. B., Baydan, B., Çaylak, S., Boyraz, A. Z., Yolaçan, B., and Herkiloğlu, K. Comparing automated visual gui testing tools: an industrial case study. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing* (2017), ACM, pp. 21–28.

[67] Garousi, V., and Yildirim, E. Introducing automated gui testing and observing its benefits: an industrial case study in the context of law-practice management software. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2018), IEEE, pp. 138–145.

[68] Gill, K. S. *Human machine symbiosis: The foundations of human-centred systems design.* Springer Science & Business Media, 2012.

[69] Gorschek, T., Garre, P., Larsson, S., and Wohlin, C. A model for technology transfer in practice. *IEEE software 23*, 6 (2006), 88–95.

[70] Grechanik, M., Mao, C. W., Baisal, A., Rosenblum, D., and Hossain, B. M. Differencing graphical user interfaces. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (2018), IEEE, pp. 203–214.

[71] Grechanik, M., Xie, Q., and Fu, C. Creating gui testing tools using accessibility technologies. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on* (2009), IEEE, pp. 243–250.

[72] Grechanik, M., Xie, Q., and Fu, C. Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. In *2009 IEEE International Conference on Software Maintenance* (2009), IEEE, pp. 9–18.

[73] Grechanik, M., Xie, Q., and Fu, C. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st international conference on software engineering* (2009), IEEE Computer Society, pp. 408–418.

[74] Gupta, P., and Surve, P. Model based approach to assist test case creation, execution, and maintenance for test automation. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering* (2011), ACM, pp. 1–7.

[75] Hammoudi, M., Rothermel, G., and Stocco, A. 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* (New York, NY, USA, 2016), FSE 2016, Association for Computing Machinery, p. 751–762.

[76] Heiskanen, H., Jääskeläinen, A., and Katara, M. Debug support for model-based gui testing. In *2010 Third International Conference on Software Testing, Verification and Validation* (2010), IEEE, pp. 25–34.

[77] Hou, Y., Chen, R., and Du, Z. Automated gui testing for j2me software based on fsm. In *2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing* (2009), IEEE, pp. 341–346.

[78] Huang, S., Cohen, M. B., and Memon, A. M. Repairing gui test suites using a genetic algorithm. In *2010 Third International Conference on Software Testing, Verification and Validation* (2010), IEEE, pp. 245–254.

[79] Jain, A. K., Mao, J., and Mohiuddin, K. M. Artificial neural networks: A tutorial. *Computer 29*, 3 (1996), 31–44.

[80] Janicki, M., Katara, M., and Pääkkönen, T. Obstacles and opportunities in deploying model-based gui testing of mobile software: a survey. *Software Testing, Verification and Reliability 22*, 5 (2012), 313–341.

[81] Jiang, W., Li, X., and Wang, X. A black-box based script repair method for gui regression test. In *2018 7th International Conference on Digital Home (ICDH)* (2018), IEEE, pp. 148–153.

[82] Jorgensen, P. C., and Erickson, C. Object-oriented integration testing. *Communications of the ACM 37*, 9 (1994), 30–38.

[83] Kaner, C. Exploratory testing. In *Quality assurance institute worldwide annual software testing conference* (2006), pp. 1–14.

[84] Khaliq, Z., Farooq, S. U., and Khan, D. A. Transformers for gui testing: A plausible solution to automated test case generation and flaky tests. *Computer 55*, 3 (2022), 64–73.

[85] Kirinuki, H., Matsumoto, S., Higo, Y., and Kusumoto, S. Web element identification by combining nlp and heuristic search for web testing. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2022), IEEE, pp. 1055–1065.

[86] Kirinuki, H., Tanno, H., and Natsukawa, K. Color: Correct locator recommender for broken test scripts using various clues in web application. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), 310–320.

[87] Kitchenham, B., Pretorius, R., Budgen, D., Brereton, O. P., Turner, M., Niazi, M., and Linkman, S. Systematic literature reviews in software engineering–a tertiary study. *Information and software technology 52*, 8 (2010), 792–805.

[88] Kitzinger, J., and Barbour, R. *Developing focus group research: politics, theory and practice*. Sage, 1999.

[89] Koubaa, A. Gpt-4 vs. gpt-3.5: A concise showdown.

[90] Kresse, A., and Kruse, P. M. Development and maintenance efforts testing graphical user interfaces: a comparison. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation* (2016), ACM, pp. 52–58.

[91] Kumar, Y., et al. Comparative study of automated testing tools: selenium, soapui, hp unified functional testing and test complete. *Journal of Emerging Tech-nologies and Innovative Research 2*, 9 (2015), 42–48.

[92] Landis, J. R., and Koch, G. G. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[93] Larivière, V., Pontille, D., and Sugimoto, C. R. Investigating the division of scientific labor using the contributor roles taxonomy (credit). *Quantitative Science Studies 2*, 1 (2021), 111–128.

[94] Laţiu, G. I., Creţ, O., and Văcariu, L. Graphical user interface testing using evolutionary algorithms. In *2013 8th Iberian Conference on Information Systems and Technologies (CISTI)* (2013), IEEE, pp. 1–6.

[95] Leotta, M., Clerissi, D., Ricca, F., and Spadaro, C. Comparing the maintainability of selenium webdriver test suites employing different locators: A case study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation* (New York, NY, USA, 2013), JAMAICA 2013, Association for Computing Machinery, p. 53–58.

[96] Leotta, M., Ricca, F., and Tonella, P. Sidereal: Statistical adaptive generation of robust locators for web testing. *Software Testing, Verification and Reliability 31*, 3 (2021), e1767. e1767 stvr.1767.

[97] Leotta, M., Ricca, F., and Tonella, P. Sidereal: Statistical adaptive generation of robust locators for web testing. *Software Testing, Verification and Reliability 31*, 3 (2021), e1767.

[98] Leotta, M., Stocco, A., Ricca, F., and Tonella, P. Reducing web test cases aging by means of robust xpath locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops* (2014), IEEE, pp. 449–454.

[99] Leotta, M., Stocco, A., Ricca, F., and Tonella, P. Using multi-locators to increase the robustness of web test cases. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on* (2015), IEEE, pp. 1–10.

[100] Leotta, M., Stocco, A., Ricca, F., and Tonella, P. Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process 28*, 3 (2016), 177–204.

[101] Leotta, M., Stocco, A., Ricca, F., and Tonella, P. Pesto: Automated migration of dom-based web tests towards the visual approach. *Software Testing, Verification and Reliability 28*, 4 (2018), e1665.

[102] Li, X., Chang, N., Wang, Y., Huang, H., Pei, Y., Wang, L., and Li, X. Atom: Automatic maintenance of gui test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2017), IEEE, pp. 161–171.

[103] Liebel, G., Alégroth, E., and Feldt, R. 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* (2013), IEEE, pp. 17–24.

[104] Liu, Y., Yandrapally, R., Kalia, A. K., Sinha, S., Tzoref-Brill, R., and Mesbah, A. Crawlabel: computing natural-language labels for ui test cases. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test* (2022), pp. 103–114.

[105] Liu, Z., Chen, C., Wang, J., Che, X., Huang, Y., Hu, J., and Wang, Q. Fill in the blank: Context-aware automated text input generation for mobile gui testing. *arXiv preprint arXiv:2212.04732* (2022).

[106] Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., and Wang, Q. Chatting with gpt-3 for zero-shot human-like mobile automated gui testing. *arXiv preprint arXiv:2305.09434* (2023).

[107] Luo, L. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA 15232*, 1-19 (2001), 19.

[108] Mahmud, J., Cypher, A., Haber, E., and Lau, T. Design and industrial evaluation of a tool supporting semi-automated website testing. *Software Testing, Verification and Reliability 24*, 1 (2014), 61–82.

[109] Mariani, L., Pezze, M., Riganelli, O., and Santoro, M. Autoblacktest: Automatic black-box testing of interactive applications. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (2012), IEEE, pp. 81–90.

[110] Mariani, L., Pezzè, M., Riganelli, O., and Santoro, M. Automatic testing of gui-based applications. *Software Testing, Verification and Reliability 24*, 5 (2014), 341–366.

[111] Mariani, L., Pezzè, M., and Zuddas, D. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), IEEE, pp. 280–290.

[112] Maxwell, J. A. *Qualitative research design: An interactive approach*. Sage publications, 2012.

[113] McMaster, S., and Memon, A. Call-stack coverage for gui test suite reduction. *IEEE Transactions on Software Engineering 34*, 1 (2008), 99–115.

[114] Memon, A. M. An event-flow model of gui-based applications for testing. *Software testing, verification and reliability 17*, 3 (2007), 137–157.

[115] Memon, A. M., Pollack, M. E., and Soffa, M. L. Hierarchical gui test case generation using automated planning. *IEEE transactions on software engineering 27*, 2 (2001), 144–155.

[116] Min, S., Lyu, X., Holtzman, A., Artetxe, M., Lewis, M., Hajishirzi, H., and Zettlemoyer, L. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837* (2022).

[117] Montoto, P., Pan, A., Raposo, J., Bellas, F., and López, J. Automated browsing in ajax websites. *Data & Knowledge Engineering 70*, 3 (2011), 269–283.

[118] Moreira, R. M., Paiva, A. C., Nabuco, M., and Memon, A. Pattern-based gui testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability 27*, 3 (2017), e1629.

[119] Nass, M., Alégroth, E., and Feldt, R. Augmented testing: Industry feedback to shape a new testing technology. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2019), IEEE, pp. 176–183.

[120] Nass, M., Alégroth, E., and Feldt, R. On the industrial applicability of augmented testing: An empirical study. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2020), IEEE, pp. 364–371.

[121] Nass, M., Alégroth, E., and Feldt, R. Why many challenges with gui test automation (will) remain. *Information and Software Technology 138* (2021), 106625.

[122] Nass, M., Alégroth, E., Feldt, R., Leotta, M., and Ricca, F. Similarity-based web element localization for robust test automation. *arXiv preprint arXiv:2208.00677* (2022).

[123] Nass, M., Alégroth, E., Feldt, R., Leotta, M., and Ricca, F. Similarity-based web element localization for robust test automation. *ACM Transactions on Software Engineering and Methodology 32*, 3 (2023), 1–30.

[124] Nass, M., Alégroth, E., Feldt, R., and Coppola, R. Robust web element identification for evolving applications by considering visual overlaps. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (2023), pp. 258–268.

[125] Nass, M., Coppola, R., Alégroth, E., and Feldt, R. Robust web element identification for evolving applications by considering visual overlaps. *arXiv preprint arXiv:2301.03863* (2023).

[126] Nass, M., Olsson, H., and Alégroth, E. Jautomate: a tool for system-and acceptance-test automation. In *IEEE Sixth International Conference on Software Testing, Verification and Validation* (2013), Citeseer.

[127] Nguyen, B. N., Robbins, B., Banerjee, I., and Memon, A. Guitar: an innovative tool for automated testing of gui-driven software. *Automated software engineering 21*, 1 (2014), 65–105.

[128] Nguyen, V., To, T., and Diep, G.-H. Generating and selecting resilient and maintainable locators for web automated testing. *Software Testing, Verification and Reliability 31*, 3 (2021), e1760. e1760 stvr.1760.

[129] Nguyen, V., To, T., and Diep, G.-H. Generating and selecting resilient and maintainable locators for web automated testing. *Software Testing, Verification and Reliability 31*, 3 (2021), e1760.

[130] Ohba, M. Software quality= test accuracy× test coverage. In *Proceedings of the 6th international conference on Software engineering* (1982), pp. 287–293.

[131] Olan, M. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges 19*, 2 (2003), 319–328.

[132] Onoma, A. K., Tsai, W.-T., Poonawala, M., and Suganuma, H. Regression testing in an industrial environment. *Communications of the ACM 41*, 5 (1998), 81–86.

[133] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems 35* (2022), 27730–27744.

[134] Öztürk, M. M., and Zengin, A. Improved gui testing using task parallel library. *ACM SIGSOFT Software Engineering Notes 41*, 1 (2016), 1–8.

[135] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442* (2023).

[136] Patel, S., and Shah, V. Automated testing of software-as-a-service configurations using a variability language. In *Proceedings of the 19th International Conference on Software Product Line* (2015), ACM, pp. 253–262.

[137] Pham, R., Holzmann, H., Schneider, K., and Brüggemann, C. Beyond plain video recording of gui tests: linking test case instructions with visual response documentation. In *Proceedings of the 7th International Workshop on Automation of Software Test* (2012), IEEE Press, pp. 103–109.

[138] Rafi, D. M., Moses, K. R. K., Petersen, K., and Mäntylä, M. V. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test* (2012), IEEE Press, pp. 36–42.

[139] Razeghi, Y., Logan IV, R. L., Gardner, M., and Singh, S. Impact of pretraining term frequencies on few-shot reasoning. *arXiv preprint arXiv:2202.07206* (2022).

[140] Rey, D., and Neuhäuser, M. Wilcoxon-signed-rank test. In *International encyclopedia of statistical science*. Springer, 2011, pp. 1658–1659.

[141] Ricca, F., Leotta, M., and Stocco, A. Chapter three - three open problems in the context of E2E web testing and a vision. vol. 113 of *Advances in Computers*. Elsevier, 2019, pp. 89–133.

[142] Runeson, P., Engström, E., and Storey, M.-A. The design science paradigm as a frame for empirical software engineering. *Contemporary empirical methods in software engineering* (2020), 127–147.

[143] Runeson, P., and Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering 14*, 2 (2009), 131–164.

[144] Sackett, D. L. Bias in analytic research. In *The Case-Control Study Consensus and Controversy*. Elsevier, 1979, pp. 51–63.

[145] Silva, D. B., Endo, A. T., Eler, M. M., and Durelli, V. H. An analysis of automated tests for mobile android applications. In *2016 XLII Latin American Computing Conference (CLEI)* (2016), IEEE, pp. 1–9.

[146] Staron, M. *Action research in software engineering*. Springer, 2020.

[147] Stocco, A., Yandrapally, R., and Mesbah, A. 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* (2018), pp. 503–514.

[148] Takahashi, J., and Kakuda, Y. Effective automated testing: a solution of graphical object verification. In *Proceedings of the 11th Asian Test Symposium, 2002.(ATS'02).* (2002), IEEE, pp. 284–291.

[149] Takala, T., Katara, M., and Harty, J. Experiences of system-level model-based gui testing of an android application. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (2011), IEEE, pp. 377–386.

[150] Thummalapenta, S., Devaki, P., Sinha, S., Chandra, S., Gnanasundaram, S., Nagaraj, D. D., and Sathishkumar, S. Efficient and change-resilient test automation: An industrial case study. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 1002–1011.

[151] Thummalapenta, S., Sinha, S., Singhania, N., and Chandra, S. Automating test automation. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 881–891.

[152] Tonella, P., Ricca, F., and Marchetto, A. Recent advances in web testing. In *Advances in Computers*, vol. 93. Elsevier, 2014, pp. 1–51.

[153] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems 30* (2017).

[154] Vos, T. E., Kruse, P. M., Bauersfeld, S., Wegener, J., et al. Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD) 6*, 3 (2015), 46–83.

[155] Wang, W., Sampath, S., Lei, Y., Kacker, R., Kuhn, R., and Lawrence, J. Using combinatorial testing to build navigation graphs for dynamic web applications. *Software Testing, Verification and Reliability 26*, 4 (2016), 318–346.

[156] Wiklund, K., Eldh, S., Sundmark, D., and Lundqvist, K. Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability 27*, 8 (2017), e1639.

[157] Wohlin, C. 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* (2014), Citeseer, p. 38.

[158] Wohlin, C., and Runeson, P. Guiding the selection of research methodology in industry–academia collaboration in software engineering. *Information and software technology 140* (2021), 106678.

[159] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[160] Wood, L., Le Hors, A., Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Nicol, G., Robie, J., Sutor, R., et al. Document object model (dom) level 1 specification. *W3C recommendation 1* (1998).

[161] Wu, J., Swearngin, A., Zhang, X., Nichols, J., and Bigham, J. P. Screen correspondence: Mapping interchangeable elements between uis. *arXiv preprint arXiv:2301.08372* (2023).

[162] Xia, F., Liu, T.-Y., Wang, J., Zhang, W., and Li, H. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning* (2008), pp. 1192–1199.

[163] Xie, Q., Grechanik, M., Fu, C., and Cumby, C. Guide: A gui differentiator. In *2009 IEEE International Conference on Software Maintenance* (2009), IEEE, pp. 395–396.

[164] Xie, Q., and Memon, A. M. Using a pilot study to derive a gui model for automated testing. *ACM Transactions on Software Engineering and Methodology (TOSEM) 18*, 2 (2008), 7.

[165] y Arcas, B. A. Do large language models understand us? *Daedalus 151*, 2 (2022), 183–197.

[166] Yandrapally, R., Thummalapenta, S., Sinha, S., and Chandra, S. Robust test automation using contextual clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014), pp. 304–314.

[167] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601* (2023).

[168] Yeh, T., Chang, T.-H., and Miller, R. C. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology* (2009), pp. 183–192.

[169] Yujian, L., and Bo, L. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence 29*, 6 (2007), 1091–1095.

[170] Zaugg, H., West, R. E., Tateishi, I., and Randall, D. L. Mendeley: Creating communities of scholarly inquiry through research collaboration. *TechTrends 55*, 1 (2011), 32–36.

[171] Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W., and Xie, T. Automated test input generation for android: Are we really there yet in an industrial case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016), ACM, pp. 987–992.

[172] Zheng, Y., Huang, S., Hui, Z.-w., and Wu, Y.-N. A method of optimizing multi-locators based on machine learning. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (2018), IEEE, pp. 172–174.

[173] Zhu, H., Hall, P. A., and May, J. H. Software unit test coverage and adequacy. *Acm computing surveys (csur) 29*, 4 (1997), 366–427.