Testing without Scripts: An Approach to Smart GUI Exploration

Olivia Rodríguez Valdés

The work in this thesis has been carried out at the Open Universiteit, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). Parts of the research were conducted within the European research project IVVES (Industrial-grade Verification and Validation of Evolving Systems, project number 18022).

ISBN: 978-94-6522-322-3

Typeset using LaTeX
Printing: Ridderprint | www.ridderprint.nl

Testing without Scripts: An Approach to Smart GUI Exploration

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. dr. Th.J. Bastiaens
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op donderdag 19 juni 2025 te Heerlen
om 16.00 uur precies

door
Olivia Rodríguez Valdés
geboren op 12 november 1994 te Havana, Cuba

Promotores:

Prof. dr. T.E.J. Vos	Open Universiteit
Dr. B. Marín	Technical University of Valencia

Leden beoordelingscommissie:

Prof. dr. M.F. Genero	Universidad de Castilla-La Mancha
Prof. dr. O. Pastor	Universitat Politècnica de València
Prof. dr. N. Alechina	Open Universiteit
Prof. dr. S. Bromuri	Open Universiteit

Abstract

Software testing through graphical user interfaces (GUIs) remains a critical challenge in quality assurance, particularly as software systems grow in complexity and evolve rapidly. Traditional script-based testing approaches, which rely on predefined test cases, are widely used in industry but often struggle with high maintenance costs, limited adaptability to GUI changes, and restricted coverage of unforeseen user behaviours.

Scriptless GUI testing has emerged as a powerful alternative, dynamically exploring applications without the need for predefined test scripts. This approach introduces randomness, allowing the execution of unexpected sequences of actions and the discovery of faults that scripted tests often miss. This thesis investigates the effectiveness of scriptless testing, examining how its exploratory nature complements existing testing practices and reduces manual efforts.

To establish a strong foundation, this research first analyses thirty years of GUI testing literature, tracing the evolution of the field. The findings reveal a growing transition from manual and script-based testing to scriptless approaches. With this motivation, this thesis investigates the effectiveness of scriptless GUI testing through the lens of TESTAR, an open-source tool that serves as this study's primary research vehicle. A generalisation study of the tool allowed the intro-

duction of an architectural analogy for scriptless testing deployment, built upon the existing industrial case studies with TESTAR.

This thesis examines the role of state models in guiding scriptless testing by evaluating how different levels of state abstraction can influence model inference and test coverage. The results provide guidelines for balancing model complexity with exploration effectiveness. Additionally, this thesis explores the impact of reinforcement learning-driven reward mechanisms in balancing pure randomness with targeted exploration to enhance test effectiveness.

The thesis further evaluates the industrial applicability of scriptless testing through empirical studies in collaboration with companies participating in the European IVVES project (Industrial-grade Verification and Validation of Evolving Systems). The research aims to bridge the gap between traditional testing adequacy criteria and quality-oriented metrics by investigating whether code smell could serve as a complementary adequacy criterion when evaluating scriptless testing effectiveness. Findings reveal that while increasing traditional code coverage leads to broader exploration, it does not necessarily translate into covering code with deeper structural or maintainability issues.

This research extends TESTAR's scriptless testing into the mobile domain, adapting it for mobile platforms and introducing MINTestar, a specialised Android testing tool. Developed as part of the industry collaboration within the IVVES project, these efforts explore the feasibility of scriptless testing in real-world mobile environments, and the integration of mobile-specific oracles and probabilistic exploration strategies. The results highlight the adaptability of scriptless approaches across platforms and their potential for adoption in industrial mobile testing workflows.

This thesis integrates insights from literature reviews, empirical evaluations, and industrial case studies to provide both theoretical and practical contributions to the field of scriptless GUI testing. By improving state models, leveraging reward-based exploration, refining test adequacy metrics, and extending automation to mobile platforms, this thesis lays the foundation for future advancements in smart testing, domain-specific oracles, and distributed testing architectures.

Abstract in het Nederlands

Software testen via grafische gebruikersinterfaces (GUIs) blijft een kritische uitdaging in kwaliteitsborging, vooral omdat softwaresystemen steeds complexer worden en zich snel ontwikkelen. Traditionele scriptgebaseerde testmethode, die afhankelijk zijn van vooraf gedefinieerde testgevallen, worden veel gebruikt in de industrie, maar hebben vaak te maken met hoge onderhoudskosten, beperkte aanpasbaarheid aan GUI-wijzigingen en beperkte dekking van onvoorziene gebruikersgedragingen.

Scriptloos GUI-testen is een krachtig alternatief, waarbij applicaties dynamisch worden verkend zonder vooraf gedefinieerde testscripts. Deze aanpak introduceert willekeur, waardoor onverwachte sequenties van acties kunnen worden uitgevoerd en fouten kunnen worden ontdekt die gescripte tests vaak missen. Deze thesis onderzoekt de effectiviteit van scriptloos testen en bestudeert hoe de verkennende aard ervan bestaande testmethoden aanvult en handmatige inspanningen vermindert.

Om een stevige basis te leggen, analyseert dit onderzoek eerst dertig jaar aan literatuur over GUI-testen en volgt de evolutie van het vakgebied. De bevindingen tonen een groeiende verschuiving van handmatig en scriptgebaseerd testen naar scriptloze benaderingen. Met deze motivatie onderzoekt deze thesis de ef-

fectiviteit van scriptloos GUI-testen door middel van TESTAR, een open-source tool die als een primair onderzoeksinstrument in deze studie dient. Een generalisatiestudie van de tool heeft geleid tot de introductie van een architecturale analogie voor de implementatie van scriptloos testen, gebaseerd op bestaande industriële casestudies met TESTAR.

Deze thesis onderzoekt de rol van toestandsmodellen bij het sturen van scriptloos testen door te evalueren hoe verschillende niveaus van toestandsabstractie de modelinferentie en testdekking kunnen beïnvloeden. De resultaten bieden richtlijnen voor het balanceren van modelcomplexiteit met de effectiviteit van exploratie. Daarnaast verkent deze thesis de invloed van beloningsmechanismen op basis van reinforcement learning om pure willekeur te balanceren met gerichte exploratie, met als doel de testeffectiviteit te verbeteren.

Verder beoordeelt deze thesis de industriële toepasbaarheid van scriptloos testen via empirische studies in samenwerking met bedrijven binnen het Europese IVVES-project (Industrial-grade Verification and Validation of Evolving Systems). Het onderzoek richt zich op het overbruggen van de kloof tussen traditionele testadequaatheidscriteria en kwaliteitsgerichte metrieken, door te onderzoeken of *code smells* als een aanvullende adequaatheidsmaatstaf kunnen dienen bij de evaluatie van scriptloos testen. De bevindingen tonen aan dat een hogere traditionele *code coverage* weliswaar leidt tot een bredere exploratie, maar niet per se resulteert in het dekken van code met diepere structurele of onderhoudsproblemen.

Dit onderzoek breidt het scriptloze testen met TESTAR uit naar het mobiele domein, door de tool aan te passen voor mobiele platforms en MINTestar te introduceren, een gespecialiseerde Android-testtool. Ontwikkeld als onderdeel van de industriële samenwerking binnen het IVVES-project, verkennen deze inspanningen de haalbaarheid van scriptloos testen in realistische mobiele omgevingen, evenals de integratie van mobiele-specifieke orakels en probabilistische exploratiestrategieën. De resultaten benadrukken de aanpasbaarheid van scriptloze benaderingen over verschillende platformen en hun potentieel voor adoptie in industriële mobiele testworkflows.

Deze thesis combineert inzichten uit literatuurstudies, empirische evaluaties

en industriële casestudies om zowel theoretische als praktische bijdragen te leveren aan het vakgebied van scriptloos GUI-testen. Door verbeteringen aan toestandsmodellen, het benutten van beloningsgestuurde exploratie, het verfijnen van testadequaatheidscriteria en het uitbreiden van automatisering naar mobiele platforms, legt deze thesis de basis voor toekomstige ontwikkelingen op het gebied van slim testen, domeinspecifieke orakels en gedistribueerde testarchitecturen.

Acknowledgments

This thesis was developed at the Open Universiteit of the Netherlands and funded by the European research project ITEA3 Industrial-grade Verification and Validation of Evolving Systems (IVVES 18022). This thesis has also been supported by research carried out within the European research projects TESTOMAT (16032), H2020 Intelligent Verification/Validation for Extended Reality Based Systems (IV4XR 856716), and the NWO OTP project Automated Unobtrusive Techniques for LINKing requirements and testing in agile software development (AUTOLINK 19521).

I would like to express my gratitude to my supervisors, prof. dr. Tanja Vos and prof. dr. Beatriz Marín, for their patience and support throughout this long journey. Tanja, thank you for changing my life. Bea, thank you for your warmth, even when I least deserved it. I would also like to thank Dr. Pekka Aho for his guidance in the early stages of this work. My gratitude also goes to the TESTAR team for their support and dedication at every stage of this research. In particular, I want to thank Ramon de Vries, who has been a support since literally day one after I moved to the Netherlands; Fernando Pastor Ricos and Lianne Hufkens, who are also my paranymphs; and Niels Doorn, for his kindness and support.

My sincere thanks go to the co-authors of my publications for their invaluable help and contributions, and those who actively participated in the research behind these publications, even if their names are not listed in the authorship. I would also like to thank the master's and bachelor's students whose work contributed directly to this thesis.

I am also grateful to my former professors and mentors, Fernando Rodríguez Flores and Oscar Luis Vera Pérez, from the University of Havana, for laying the first stone on this long path.

I would like to thank my parents, family and friends. To my mother, who has been a steady light at the harbour while I sail through this journey. To my cousin Ivonne, who has supported me in every stage of my life. To Mick, who has been by my side every day, sharing every challenge. I also extend my sincere thanks to the Verhagen-Geelen family for welcoming me so warmly. To the friends who have walked this road by my side in the Netherlands, making the process lighter with their presence, and to my old Cuban friends, now scattered around the world but as close as ever.

A special thanks to the Open Universiteit for giving me a home as a researcher, and for the patience and continuous support, which made this journey possible.

Contents

List of Figures

List of Tables

1 Introduction

"Still round the corner there may wait,
A new road or a secret gate"

J.R.R. Tolkien, The Road Goes Ever On

In The Lord of the Rings [1], by J.R.R. Tolkien, the land of Mordor is guarded by the formidable Black Gate, a symbol of impenetrable defence. Tall walls, countless sentinels, and Sauron's ever-watchful eye ensured that no army could pass undetected. The Dark Lord's resources were concentrated there, confident that no intruder would attempt to cross this main entry point. Yet, in an unexpected twist, two humble hobbits, Frodo and Sam, completely bypassed this seemingly insuperable barrier.

Instead, they ventured through Cirith Ungol, a risky and neglected mountain pass. This *unexpected path*, both unlikely and poorly defended, became the route through which the One Ring was destroyed, leading to Sauron's downfall. Despite his exhaustive preparations, Sauron's obsession with securing the most obvious threat led him to ignore the unconventional approach that ultimately caused his defeat.

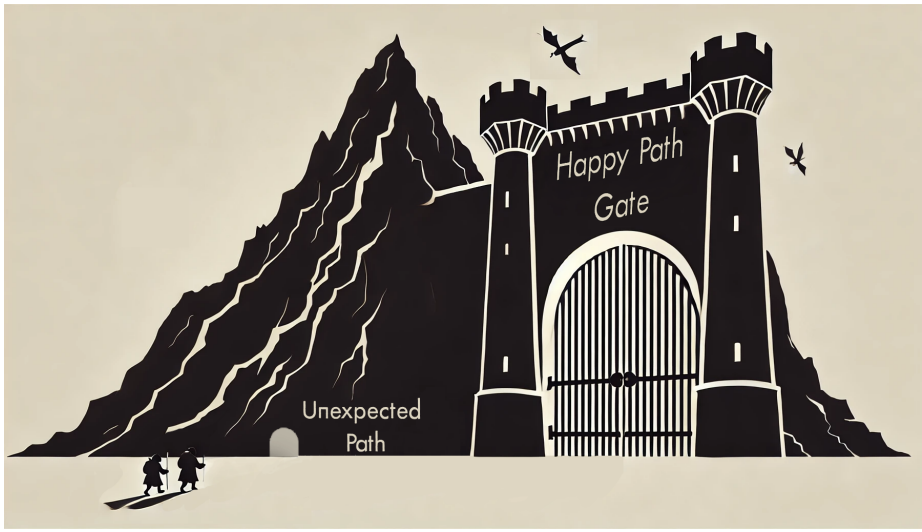


Figure 1.1: The "Happy Path" vs. the "Unexpected Path"

This story illustrates a timeless lesson beyond fantasy: even the most robust preparations can crumble when the unexpected is overlooked.

Beyond the Happy Path: Lessons for Software Testing

Much like Sauron's focus on the Black Gate (as illustrated in Figure 1.1), software testing often prioritises the main flows: the "happy paths" that users are expected to follow. These flows (e.g., logging in, adding products to a cart, or completing a purchase) receive detailed attention during testing. Test scripts are developed to ensure these paths are error-free, with significant resources dedicated to validating their functionality.

However, just like the overlooked mountain pass that led to the collapse of Mordor's defences, neglecting less-travelled paths in software testing can leave critical flaws undiscovered and exposed. For Frodo and Sam, the unexpected route led to a happy ending; for software, it could uncover catastrophic failures. Uncommon user interactions, unexpected sequences, or rare edge cases can reveal flaws that traditional approaches fail to detect.

This thesis's work on scriptless testing draws inspiration from these ne-

glected routes, advocating for broader exploration through graphical user interfaces (GUIs) beyond the expected.

1.1 Motivation and problem statement

Software systems have seamlessly become integral to our daily lives, supporting activities across personal, professional, and industrial domains. From the apps on our smartphones to the enterprise systems running businesses, their reliability and quality directly impact productivity, safety, and user satisfaction across all sectors of society. However, as these systems grow in complexity and scale, ensuring their reliability and functionality becomes more critical than ever.

Significant real-world incidents illustrate the consequences of insufficient or ineffective testing. In 2018, a flawed interface at the Hawaii Emergency Management Agency allowed an employee to send a false ballistic missile alert [2], causing widespread public panic and disruptions across the state. Investigations revealed that the drop-down menu for sending a real alert was nearly indistinguishable from the rehearsal option, and no secondary confirmation step existed. This same year, a disastrous core banking system migration resulted in the loss of internet and mobile banking services for TSB Bank customers for at least a week. Thousands of customers found themselves locked out, seeing incorrect balances or detailed account information of other customers [3]. The fiasco reportedly cost millions of pounds in remediation expenses, fines, and compensations, and the brand suffered a significant blow in customer confidence. In 2022, Southwest Airlines cancelled thousands of flights over the holiday season, attributing much of the disruption to outdated mobile scheduling software [4]. Southwest received the largest fine in history for consumer protection violations, reporting losses of over $1 billion as a result of the events.

More recently, in 2024, a flawed content update in CrowdStrike's Falcon threat detection software caused catastrophic disruptions, leading to system crashes and blue screens of death on millions of devices worldwide. The root cause was traced back to a bug that the company's testing process failed to identify. The resulting outage impacted critical infrastructure, including airlines and emergency

operators, with estimated damages exceeding $5 billion. Experts highlighted the industry-wide underappreciation of the importance of testing [5]. That same year, a customer of ING Bank unexpectedly gained access to a stranger's account after logging into the mobile app via facial recognition, raising serious concerns about the robustness of the bank's system safeguards [6].

These examples demonstrate that testing inadequacies, whether due to overlooked edge cases, outdated processes, or insufficient validation, can have devastating consequences. They underscore the need for robust, holistic testing approaches that address the growing complexities of modern systems to avoid similar failures in the future.

Yet, software testing remains one of the most demanding and resource-intensive aspects. Designing and executing comprehensive test suites requires significant expertise, creativity, and effort, often with time constraints and limited resources. Testers must balance the trade-off between coverage, efficiency, and cost. The challenge is compounded by rapid development cycles, diverse user behaviours, and the need to test across multiple platforms and configurations. For end-users, effective testing is the invisible assurance that software will meet their expectations and avoid disruptive faults.

The advent of Graphical User Interfaces (GUIs) in the early 1970s revolutionised software interaction, replacing complex command-line interfaces with intuitive visual elements. GUIs allowed users to interact with systems through buttons, menus, and icons, significantly enhancing accessibility and usability. However, this evolution introduced new challenges in software testing as GUIs frequently change throughout a system's lifecycle. The repetitive task of manually executing tests necessitates automation to ensure efficiency and reliability.

This thesis explores the challenges of automating test execution at the GUI level of software systems, commonly referred to as GUI testing. To set the stage, the following section first provides an overview of this field's current state of the art.

1.2 GUI Testing: State of the Art

Since the introduction of GUIs, testing at the interface level has become a crucial aspect of software quality assurance. As desktop applications transitioned to web and mobile platforms, testing faced a continuously evolving landscape of distributed systems, smaller screens, and increasingly complex interactions. Automating the execution of GUI tests, with the earliest papers dating back to the late 1980s [7], helps manage these challenges by improving efficiency and repeatability. However, frequent GUI modifications and intricate user interactions continue to pose significant hurdles, as automated tests often require maintenance and may struggle to replicate actual user behaviour fully.

Furthermore, regression testing, which involves re-executing existing test cases to ensure that recent changes or updates have not introduced new defects, plays a vital role in software maintenance. Because GUIs are frequently modified, regression testing must be performed repeatedly, making manual execution impractical. Automation is essential to efficiently conduct regression tests, ensuring software stability, reliability, and consistency across different versions and platforms.

GUI testing has been classified in various ways. In [8], a classification is defined based on how the test automation tool interacts with the System Under Test (SUT). This results in a classification of three generations: the first is based on the mouse coordinates, the second is based on technical APIs, and the third is based on image recognition.

A subsequent classification was described on [9] that extends these three generations with another axis addressing the level of automation. This section will follow the classification from [9] because it was also used in one of the key papers [10] on scriptless testing.

1.2.1 Script-based GUI Testing

Within automated approaches, traditional scripted methods coexist with emerging scriptless techniques, each addressing unique aspects of software assurance.

Traditional *scripted testing* relies on manually crafted test cases following predefined application paths. Once written, these scripts can be executed re-

peatedly to confirm whether the application continues to respond correctly to the same set of inputs across different builds or releases. Over time, multiple sub-approaches to script-based GUI testing have emerged, each offering distinct levels of flexibility, maintainability, and reliance on tooling or coding expertise.

One of the earliest and most common methods is *capture-and-replay (C&R)*. The testing tool "records" interactions as the tester manually navigates through the GUI. The tool then generates scripts that can be "replayed" automatically to repeat the exact steps. Memon et al. [11] described this technique as a straightforward entry point for test automation due to its easy initial setup, minimal coding required, and fast test creation. However, C&R suffers high sensitivity to minor GUI changes, often breaking recorded scripts and limiting the adaptability when the GUI evolves.

An alternative to C&R is scripting through frameworks like Selenium[1] or Cypress[2], where testers manually code interactions. However, programming skills are required, and maintainability remains an issue.

While each approach differs in how scripts are created and managed, all share certain challenges. Frequent GUI changes can break test sequences, leading to high maintenance costs [12]. Research has long noted that script maintenance can overshadow original creation effort [13]. Complicated workflows may require detailed scripting to cover different branches or states. Tests can also easily break if they depend on precise timing or exact element positioning. Furthermore, both approaches typically focus on "happy paths" at the expense of unconventional or edge-case scenarios. Consequently, significant risks remain when unexpected user behaviour triggers untested application states.

Several academic approaches are proposed to mitigate the high maintenance overhead. One technique consists of heuristic locator strategies [14] by generating more stable element locators in web applications. Another approach is automated script repair mechanisms that adapt test scripts when interface elements change [15–18]. Yet, despite promising results in controlled studies, full adoption remains challenging due to tool integration complexities, variability of real-world

[1] SeleniumHQ Browser Automation, https://www.selenium.dev/
[2] Cypress.io: JavaScript End to End Testing Framework, https://www.cypress.io/

GUIs, and the need for ongoing maintenance of the repair heuristics. Another research direction [19–21] has explored Visual GUI Testing (VGT) techniques with image recognition to automate user interactions with the GUI, providing a robust alternative to traditional locator-based approaches. These tools mitigate maintenance challenges by focusing on the GUI's appearance rather than its internal structure, although they remain sensitive to visual inconsistencies and changes.

Ultimately, script-based methods remain a foundational element of many QA strategies, particularly where critical user paths are well-defined and must be repeatedly validated. However, these methods can become less effective in environments with rapidly changing interfaces or unpredictable user behaviour, leading to interest in more flexible testing methods, such as scriptless and model-based approaches.

1.2.2 Model-based GUI Testing

Model-based GUI testing (MBGT) addresses some limitations of script-based approaches by representing the application's states and transitions as a model, often a state machine or a graph [22–26]. Systematically, test sequences are generated based on these models. In MBGT, testers specify or infer the application's possible states and transitions. Automated tools then produce test cases that explore these states, aiming for more systematic coverage. This structured approach can reduce duplicated effort, ensure broad exploration, and provide better traceability.

Several approaches to MBGT exist, each with distinct advantages and challenges. *Static Analysis* techniques [27, 28] infer GUI models by analysing source code but often overlook runtime behaviour. On the other hand, *Dynamic Analysis* approaches [29–32] observe the GUI while the SUT is running, enabling the capture of runtime interactions. Hybrid approaches [33–35] combine static and dynamic methods, striving to balance the strength of both.

However, the effectiveness of MBGT is highly dependent on the quality and completeness of its underlying model. Constructing and maintaining an accurate model requires formal expertise and can be labour-intensive. Any divergence from the actual GUI may lead to undetected defects or irrelevant test cases [36]. Automated model inference methods, including GUI ripping [37] and reverse engi-

neering [38], offer automation of the modelling process. However, these methods face challenges, such as selecting an appropriate level of abstraction to ensure model usefulness [31, 34].

Furthermore, frequent GUI updates can quickly render existing models obsolete, mirroring the maintenance challenges seen in script-based testing methods. Despite these difficulties, model-based approaches remain an important stepping stone toward more adaptive and intelligent testing strategies.

1.2.3 Scriptless GUI Testing

Scriptless GUI testing aims to overcome some long-standing problems of rigid scripts and extensive modelling by dynamically exploring the application without a fixed pre-written set of instructions. This dynamic exploration can lead to broader coverage, as the exploration is not strictly limited to known scenarios. Like Frodo and Sam's journey through their unexpected path, scriptless testing navigates beyond the "happy paths" traditionally prioritised in software testing. Instead of executing only predefined steps, scriptless testing tools generate test sequences in real time, opening up the possibility of uncovering neglected and unexpected interaction paths and increasing the likelihood of exposing critical flaws that rigid scripts or model-based approaches might miss.

Scriptless GUI testing is based on agents implementing various action selection mechanisms and test oracles. The underlying principles are simple: generate test sequences of (state, action)-pairs by starting up the SUT in its initial state and continuously selecting an action to bring the SUT into another state. The action selection characterises the fundamental challenge of intelligent systems: what to do next. The difficult part is optimising the action selection [39] to find faults and recognising a faulty state when it is found [40–42]. Faulty states are not restricted to errors in functionality; violations of other quality characteristics, like accessibility or security, can also be detected by inspecting the state. This approach shifts the paradigm of GUI testing: from developing scripts to developing intelligent AI-enabled agents.

Figure 1.2 presents an overview of the continuous cycle of the three core components of scriptless GUI testing:

- **Interaction** (Obtain the state and derive actions): The testing tool analyses how to interact with the SUT and obtains a representation of the current state of the GUI. This step involves gathering information about the interface, such as available widgets and interactions, and deriving a set of possible (inter)actions that can be done in the observed state.
- **Exploration** (Select and execute an action): From the derived set of actions, the tool selects an action based on some predefined exploration strategy (e.g. random) and executes it on the GUI.
- **Test oracles**: After executing an action, the tool uses predefined test oracles to validate the resulting application state or behaviour, revealing potential faults.

This loop iteratively continues until some defined stopping condition is met, such as achieving sufficient coverage or reaching a certain test length.

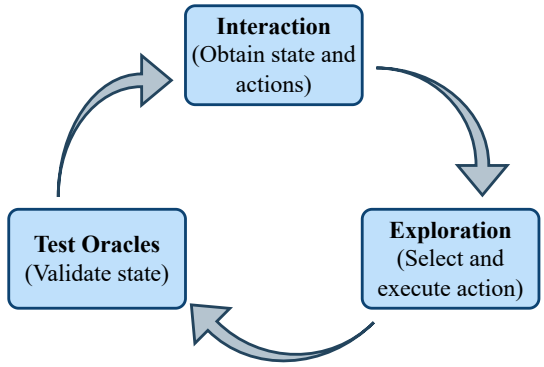


Figure 1.2: Overview of scriptless GUI testing

The most simple Action Selection Mechanism (ASM) for exploration consists of randomly selecting actions and navigating the interface. This approach has shown to be surprisingly effective [43, 44], reducing maintenance overhead, as testers do not need to update scripts or models continually. Nonetheless, random

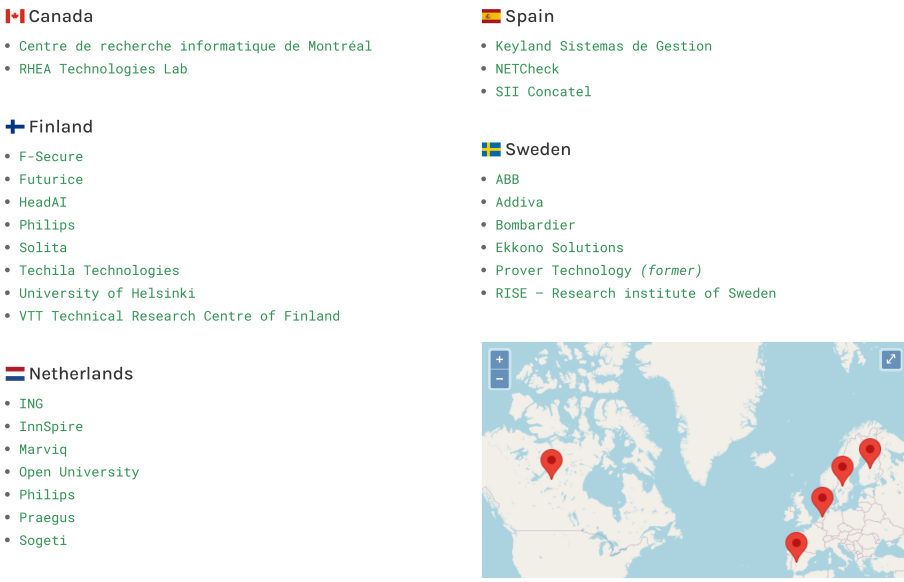


Figure 1.3: IVVES: 26 partners from 5 countries

testing requires a lot of execution time, and challenges arise in choosing the right exploration strategies, ensuring comprehensive coverage, and formulating effective oracles—mechanisms to decide whether a detected application state is correct or erroneous.

1.3 Context and goal of the thesis: the IVVES project

This thesis is embedded in the scope of the European IVVES[3] project [45], funded by the ITEA Framework (Project Number 18022), running 3 years during 2019-2022. IVVES is a project with 26 partners from 5 countries (see Figure 1.3). Its goal is to address the challenges of quality assurance posed by modern, complex, and evolving systems, which are increasingly being used in Banking & Finance, Healthcare, and Cybersecurity, among others. These systems require robust and

[3]Industrial-Grade Verification and Validation of Evolving Systems, https://ivves.eu/

innovative verification and validation methodologies to ensure trustworthiness, safety, and compliance in mission-critical applications.

One of the primary objectives of IVVES is to advance software testing automation through intelligent methods. As a project partner, the Open University focused on researching how to achieve this using scriptless testing, giving rise to the general research goal of this thesis: **to advance the effectiveness and efficiency of scriptless GUI testing**. Additionally, industrial partners played a key role by identifying challenges in their existing testing practices and defining their specific needs for scriptless testing solutions.

Two IVVES partners, Marviq and ING, outlined specific requirements for integrating scriptless testing into their workflows, which directly contributed to the formulation of two of the six research questions explored in this thesis. Section 1.4 will discuss these research questions. First, however, we introduce the two companies and provide an overview of the baseline scriptless testing tool TESTAR [10] used in this study.

1.3.1 Marviq

Marviq[4] is a software development company specialised in Team as a Service, Software Development as a Service, and IoT development. It operates with 35 professionals managing eight concurrent agile development projects while serving 25 clients. Given the tailored nature of these projects, Marviq follows a customised Quality Assurance (QA) process, including business alignment workshops, Minimum Viable Product (MVP) development, agile-based implementation using SCRUM [46], and ongoing client support. However, as a small company, Marviq faces several QA challenges [47, 48], such as unclear requirements, prototype misconceptions, business process mismatches, and limited testing time.

The SUT Marviq proposes for IVVES studies is Yoho[5], a digital platform developed by Marviq to improve operations and communication in industrial environments. Yoho provides functionalities such as alert and notification management, task handling, work instructions, and communication tools.

[4] Official website: https://marviq.com/

[5] Yoho showcase: https://marviq.com/our-showcases/yoho-factory-management-platform/

Initially, Yoho started as an MVP but underwent continuous scope changes due to shifting market demands, making it more of a prototype than a functional product. Marviq took over development to transform it into a market-ready platform, stabilising its focus as the first customers emerged. The platform features high customizability and role-based access, meaning test execution for a specific role or customer may result in low code coverage, as not all functionalities are accessible to every user.

Yoho presents key testing challenges, particularly in dealing with the dynamic nature of modern web applications, such as GUI elements with dynamic identifiers. These characteristics make Yoho an ideal candidate for evaluating scriptless GUI testing techniques.

1.3.2 ING

In today's digital economy, banks are no longer just financial institutions; they are software-driven companies that specialise in money management. As a leading financial institution, ING[6] has embraced this transformation, developing cutting-edge digital banking solutions that serve a diverse customer base, including individuals, small businesses, and large corporations. ING's mission is to empower customers to achieve their goals, whether launching a new business, buying a home, or managing daily transactions. However, seamless and reliable digital services are critical to fulfilling this mission. Customers do not think about banking itself. Instead, they expect it to work effortlessly in the background.

As ING increasingly relies on mobile and online banking platforms, software reliability has become a key differentiator. The bank is committed to delivering full-time availability, ensuring customers can access their accounts, make payments, and manage their finances instantly, securely, and without disruption. ING's strategy focuses on providing an easy, personal, and relevant experience at every customer touchpoint. However, the complexity of modern financial systems and strict regulatory requirements (e.g., the Dutch National Bank's (DNB) 99.88% uptime mandate for payment infrastructure) sets a high bar for quality assurance

[6]Official website: www.ing.nl

and software testing.

Despite best efforts, technical failures and disruptions can have severe consequences, as seen in past incidents of transaction failures [49], service outages [50], and even security vulnerabilities [51], ING recognises that every disruption is costly, not just in financial terms but in trust and reputation. The bank continuously invests in robust software testing, automation, and exploratory testing strategies to prevent issues before they impact customers. As part of the IVVES project, ING is actively exploring scriptless testing approaches for mobile banking applications, aiming to improve test automation, exploration strategies, and mobile-specific testing oracles. By strengthening its quality assurance processes, ING is not just keeping up with the demands of modern banking: it is setting a new standard for software-driven financial reliability.

1.3.3 TESTAR

The research described in this thesis was conducted using TESTAR[7] [10] as the primary vehicle for advancing scriptless GUI testing. TESTAR is an open-source tool that has been co-developed by the Open Universiteit (OU) and the Technical University of Valencia (UPV) for over a decade, offering the flexibility and extensibility necessary for this study.

While TESTAR was chosen due to its long-standing academic development and our familiarity with the tool, it was not the only available tool. Several academic tools exist for scriptless testing (described in detail in Chapter 3). However, compared to these alternatives, TESTAR still emerged as the most suitable choice for the research objectives.

One of TESTAR's key strengths is its ability to interact with a diverse range of SUTs, including Windows, Web, and Java applications. Additionally, it seamlessly integrates with widely used GUI libraries such as UIAutomation [52], WebDriver [53], and Java Access Bridge [54], making it particularly well-suited for testing complex systems. Furthermore, TESTAR's active development community and its proven deployment in industrial settings reinforce its reliability and prac-

[7] Official website: www.testar.org

tical relevance. With more than 10 peer-reviewed and published industrial case studies [55–64], TESTAR has demonstrated its effectiveness in real-world applications. These factors further justify its selection as an optimal tool for advancing research in GUI testing.

1.4 The research methodology and questions

This thesis was conducted in three distinct phases. Given that most IVVES project partners were unfamiliar with scriptless testing and TESTAR, the *initial phase* focused on investigating the state of the art in scriptless testing while also conducting a meta-analysis of industrial case studies specifically involving TESTAR. This dual approach ensured a broad understanding of scriptless testing methodologies in general and a deeper insight into TESTAR's practical applications in specific.

To build a strong foundation, a **systematic bibliometric study** [65] was first conducted to gain a comprehensive understanding of the research landscape in GUI testing. A bibliometric analysis provides a macro-level quantitative view of a research domain. Given the large volume of publications in this field, this approach was chosen over a systematic literature review to explore the evolution of automated GUI testing efficiently. The visualisations and quantitative insights from bibliometric analysis serve as a foundation for identifying promising research directions. This study addressed the first research question:

> **RQ1**: How has automated GUI testing evolved over time regarding size, research trends, collaboration, authors and publication patterns?

Subsequently, a **generalisation study** grounded in architectural analogy, as described by Wieringa et al. [66], was conducted. This study involved a comprehensive analysis of all TESTAR-related case studies performed over the years, aiming to identify recurring components and key insights into its application in various industrial contexts. This study directly addressed the second research question:

RQ2: What general insights do industrial case studies provide about using TESTAR for GUI testing in industry?

This initial phase resulted in a comprehensive overview of the GUI testing field (RQ1, Chapter 2), which informed the creation of an architectural analogy [66] derived from meta-analysing all the existing case studies on TESTAR (RQ2, Chapter 3). This analogy captures the core components, interactions, and dependencies that define scriptless GUI testing with TESTAR, facilitating a more systematic understanding of its applicability and potential future research directions. Although the analogy is described in detail in Chapter 3, a simplified version is repeated here in Figure 1.4 to position the second phase of the research, whose goal was to improve some of the components of TESTAR.

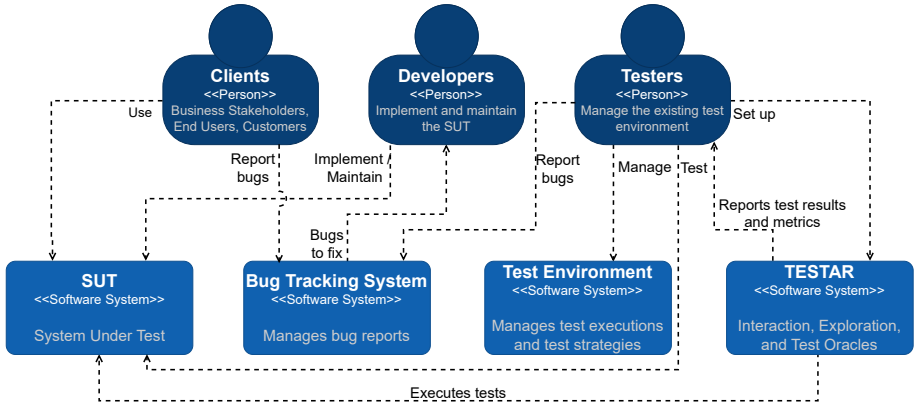


Figure 1.4: Simplified architectural analogy showing the most important components

At the core of the architecture is the SUT as it is the primary focus, representing the software being tested. Testing the SUT is influenced by multiple actors: the work of **Developers**, the needs of the **Clients** and the **Testers**. The scriptless testing tool TESTAR contains three parts (as described in Section (1.2.3)) needed to create test sequences to test the SUT:

1. through *interaction*, it engages with the user interface elements, such as

buttons, menus, and dialogues, without requiring predefined scripts and automatically exploring the SUT.

2. *exploration* is performed using an ASM, enabling TESTAR to navigate through different application states and uncover unexpected behaviours.

3. The key component consists of the *oracles*, which validates whether the observed software behaviour aligns with expected outcomes.

Setting up TESTAR requires an initial configuration process to ensure the tool is correctly adapted to the Test Environment. Testing the SUT generates test results that are evaluated using various metrics. Bugs that are found are added and managed in the Bug Tracking System.

The architectural analogy laid the groundwork to describe the *second phase* of this research, in which specific components of TESTAR were aimed to be enhanced. Two principal research objectives were defined, both focusing on improving the *exploration* part of the TESTAR-component. The first objective focused on the abstraction of the states in the models that can be learned during the on-the-fly exploration. State space explosion is still an open challenge for learning state-based models through a GUI. Most SUTs with a GUI exhibit an enormous number of possible states, necessitating some degree of state abstraction to ensure that the resulting models remain tractable. The key challenge lies in determining an appropriate level of abstraction that balances expressiveness and computational complexity. If the level of abstraction is too low, the resulting state model may become overly detailed, leading to an impractically large number of states. Conversely, if the level of abstraction is too high, the model may become overgeneralised and non-deterministic, making it unreliable. Understanding this trade-off is essential for improving model accuracy and usability in scriptless testing, leading to the following research question:

RQ3: How does state abstraction in TESTAR influence the inference of state models during on-the-fly exploration with scriptless testing?

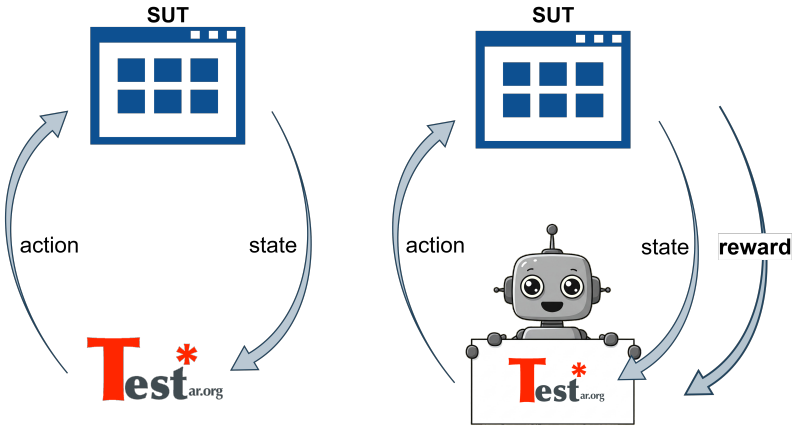


Figure 1.5: Reinforcement learning, a natural fit for scriptless testing

The second objective of the second phase also focused on enhancing the *exploration* component of TESTAR, specifically in relation to its action selection mechanism. As previously discussed, action selection plays a crucial role in determining how the system is explored, directly influencing the quality and effectiveness of the generated tests. Reinforcement learning (RL) [67] aligns naturally with the scriptless testing loop, as it follows the iterative process of retrieving the state, selecting an action, and executing it, as illustrated in Figure 1.5. At its most basic level, reinforcement learning requires defining a *reward function* for each state-action pair, guiding the learning process toward more effective exploratory testing. This led to the formulation of the fourth research question:

> **RQ4**: Which reward mechanism is most effective for exploratory testing with reinforcement learning in TESTAR?

The project's third phase consisted of industrial collaboration with the two identified industrial partners of IVVES: Marviq and ING. This process aligns with the **design science research** methodology [68], where real-world challenges drive the development and evaluation of practical solutions.

As part of the IVVES project, Marviq actively explored the integration of code smell detection with SonarQube [69] to enhance the quality and effectiveness of its software testing processes. Code smells [70] refer to structural weaknesses in software that, while not necessarily defects, indicate potential design flaws that can compromise maintainability and increase the risk of hidden defects if left unaddressed. Despite their recognised importance in software engineering, traditional test adequacy metrics such as line, branch, and complexity coverage often fail to account for these. To bridge this gap, Marviq desired to investigate whether code smell coverage could serve as a complementary adequacy criterion for evaluating the scriptless testing effectiveness of their industrial web application, Yoho. If successful, this approach could enhance the industrial applicability of scriptless testing by incorporating maintainability-focused test evaluation criteria, potentially improving defect detection and maintainability assessment. The research question for this research reads as follows:

RQ5: To what extent can scriptless GUI testing with TESTAR provide meaningful coverage of code smells, and how does this relate to traditional test adequacy metrics?

As a financial institution, ING Bank increasingly relies on mobile applications to provide essential services such as account management, payments, and investment tools. However, mobile application testing presents unique challenges, including a highly diverse device ecosystem, rapid update cycles, and stringent platform-specific guidelines. Ensuring high-quality, reliable mobile applications requires robust testing strategies that can adapt to these dynamic conditions. Consequently, within the IVVES project, ING proposed adopting a scriptless testing approach for mobile Android platforms to explore how automated exploratory testing can be enhanced for mobile environments. In addition to automating test execution, ING identified several key features and improvements necessary for effective scriptless testing in mobile applications, including: (1) a more advanced exploration strategies to guide the automated navigation of the SUT; (2) enhanced mobile-specific testing oracles to improve failure detection and validation accu-

racy. These needs led to the formulation of the following research question:

RQ6: How can scriptless GUI testing be adapted for mobile applications by improving exploration strategies and integrating mobile-specific testing oracles?

1.5 Publications

The findings of this thesis are underpinned by a series of peer-reviewed research papers developed through engagement in academic projects and industry collaborations. Each article targets one or more of the research questions defined previously. This section details my contributions to these works and the projects I participated in during the research period.

- *Rodríguez-Valdés, O., Vos, T. E. J., Aho, P., & Marín, B. (2021). 30 years of automated GUI testing: a bibliometric analysis.* published in the proceedings of the International Conference on the Quality of Information and Communications Technology (QUATIC). As the first author, I took primary responsibility for conducting the bibliometric study, including data collection, analysis, and visualisation. While all authors collaborated on identifying aspects such as keywords or other elements requiring general agreement to minimise threats to validity, the study's execution was my primary responsibility. This article targets research question RQ1 of this thesis, presenting a quantitative evaluation of key themes and collaborations in automated GUI testing over three decades. The paper is contained in Chapter 2.

- *Vos, T. E. J., Aho, P., Pastor Ricos, F., Rodriguez-Valdes, O., & Mulders, A. (2021). TESTAR–scriptless testing through graphical user interface.* published in Software Testing, Verification, and Reliability. This publication is the most updated key paper of TESTAR, describing its advances from 2010 till 2021 and paving the way for an international research agenda in GUI testing that can be built upon stable and open-source infrastructure. My contribution to this paper is the following:

  - conducting the comparative study of TESTAR with existing academic scriptless tools (mentioned in Section 1.3.3)
  - conducting the generalisation study grounded in architectural analogy to target research question RQ2 of this thesis.

  These contributions and a description of TESTAR are contained in Chapter 3.

- *Mulders, A., Rodriguez-Valdes, O., Ricós, F. P., Aho, P., Marín, B., & Vos, T. E. J. (2022). State model inference through the GUI using runtime test generation* presented at the International Conference on Research Challenges in Information Science (RCIS). My contributions to this research were critical in empirically validating the proposed state model inference mechanisms. I was responsible for running the experiments and analysing the results, ensuring the findings were backed by rigorous empirical evaluation. The experiments systematically assessed the impact of different abstraction techniques on the inferred state model. The results provided insights into the trade-offs associated with different levels of abstraction, contributing to a better understanding of how scriptless GUI testing can effectively infer state models while maintaining scalability and practical applicability in industrial contexts. This article targets research question RQ3 of this thesis regarding state abstraction and is contained in Chapter 4.

- *Rodríguez-Valdés, O., Vos, T. E. J., Marín, B., & Aho, P. (2023). Reinforcement learning for scriptless testing: An empirical investigation of reward functions* presented at the International Conference on Research Challenges in Information Science (RCIS). As the first author, I was responsible for the overall coordination of the research, including designing experiments, implementing reinforcement learning frameworks, conducting statistical analysis and discussing the results. This article targets research question RQ4 of this thesis and is contained in Chapter 5.

- *Rodríguez-Valdés, O., Amalfitano, D., Sybrandi, O., Marín, B., Vos, T. E. J. (2025). The Scent of Test Effectiveness: Can Scriptless Testing Reveal Code*

Smells?, presented at the International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). I adapted TESTAR to the company's SUT, maintained continuous feedback with industry practitioners to identify their needs, and discussed the benefits of using TESTAR in their software development lifecycle. I also implemented and executed the experiments and statistical analyses, and discussed the results, highlighting the practical applicability of the proposed solutions. This article targets the use of quality-oriented metrics as an indicator of test effectiveness, addressing question RQ5 of this thesis. The paper is contained in Chapter 6.

- *Rodríguez-Valdés, O., van der Vlist, K., van Dalen, R., Marín, B., & Vos, T. E. J. (2024). Scriptless and Seamless: Leveraging Probabilistic Models for Enhanced GUI Testing in Native Android Applications*, presented at the International Conference on Research Challenges in Information Science (RCIS). As the first author, I led the integration of the framework developed in collaboration with ING into TESTAR, performed a state-of-the-art review, and conducted the experiments. This article examines the use of a probabilistic heuristic in scriptless exploration for mobile testing, addressing question RQ6 of this thesis. The paper is contained in Chapter 7.

1.6 Supervision, academic service, and professional engagement

During this period, I was actively engaged in various academic conferences, symposia, and workshops to disseminate my research and the TESTAR approach:

- Published the short paper *Finding the shortest path to reproduce a failure found by TESTAR* [71] at the 2019 ESEC/FSE (European Software Engineering Conference and Symposium on the Foundations of Software Engineering). This work served as the basis for my initial learning about the scriptless GUI testing approach.

- Prepared and presented a poster at the 2020 SEN Symposium[8] (National Symposium Software Engineering) titled *ITEA3 IVVES project: Industrial-grade verification and validation of evolving systems (In Finance).*

- Published the Doctoral Symposium paper *Towards a testing tool that learns to test* [72] at the 2021 ICSE (International Conference on Software Engineering) Companion Proceedings, and later presented this paper again at an OUrsi (Open Universiteit Research Seminar Informatica) event for further peer engagement and collaborative refinement.

- Co-authored and disseminated a poster on the IVVES project [73] at RCIS Workshops in 2022, bridging theoretical advances with industry needs

- Delivered the tutorial *Getting started with scriptless test automation through the graphical user interface, a hands-on tutorial* [74], at RCIS 2023. The tutorial covered key topics such as GUI testing fundamentals, the scriptless approach, and practical exercises to analyse test results. This engagement strengthened academic and industry connections by disseminating knowledge on scriptless GUI testing and fostering potential collaborations in research and industry.

- Organised and presented the workshop *Inspiratie sessie: TESTAR* at Topicus in Deventer, The Netherlands, to give practitioners hands-on exposure to the scriptless GUI testing tool TESTAR. The session focused on demonstrating how TESTAR can streamline testing processes and improve software quality in real-world projects. By bridging academia and industry, this session helped participants understand the immediate practical impact of automated GUI testing while fostering a local community for collaboration and shared learning.

- Presented *Empowering Students with Modern Skills and Connections Through Open Source GUI Testing with TESTAR*, at 10th ACM Celebration of Women in Computing: womENcourage 2023. The conference context highlighted

[8] National Symposium Software Engineering, https://www.sen-symposium.nl/

how inclusive communities and open-source collaboration can help women in technology overcome challenges and gain international visibility. By demonstrating effective strategies for integrating TESTAR into student projects, the workshop also illustrated how students from diverse backgrounds can develop essential skills, access broader professional networks, and prepare for successful careers in computer science.

- Provided a hands-on session at the Universitat Politècnica de València (UPV) Master Class: *TESTAR – An Open Source Tool for Scriptless Testing Through Graphical User Interface (GUI)*, thereby bridging research insights with practical student engagement.
- Presented the work *Reinforcement Learning for Scriptless Testing* at the 2023 PROMIS-ES symposium organised by the Faculty of Science of the Open University, where ongoing PhD research was showcased.

In parallel, I supervised and advised various Bachelor and Master theses, including:

- Supervising the Bachelor's thesis of Mark Dourlein at the Open Universiteit: "TESTAR and reinforcement learning" (2020).
- Advising the Master's final project of Borja Davo Gelardo at Universitat Politècnica de València: "Improving action selection in TESTAR with artificial intelligence techniques" (2020–2021).
- Supervising the Master's project of Sven Ordelman at the Open Universiteit: "Q-learning for action selection" (June 2022).
- Supervising the Master's final project of Moujib Chorfi at Universitat Politècnica de València: "An empirical investigation comparing different GUI testing tools for Android" (2022–2023).

Additionally, serving as Web Chair for the 2022 International Conference on Software Testing (ICST), widely recognised as one of the most important conferences in the field, enabled me to establish important connections with international experts, helping to refine the direction of this research. Participation in

IPA events[9], and TestDag[10], further expanded my professional network, provided new practical insights, and influenced the overall scope of this work.

1.7 Thesis Structure

The thesis contains seven chapters in addition to this introduction. Chapter 2 to Chapter 7 correspond to research questions RQ1 till RQ6 respectively. Chapter 8 concludes with the answers to the research questions and offers a synthesis of key findings, implications for the field, and directions for future research.

[9] Institute for Programming research and Algorithmics (IPA), https://ipa.win.tue.nl/
[10] Dutch Testing Day (Nederlandse Testdag), https://www.testdag.nl/

2

Thirthy years of automated GUI testing

"We can only see a short distance ahead, but we can see plenty there that needs to be done"

Alan Turing, Computing Machinery and Intelligence

A bibliometric study is presented in this chapter to gain insight into the community, publication patterns, and trends in automated GUI testing. A bibliometric analysis enables visualisation of the main topics in the literature, their evolution, and their interrelationships. Furthermore, this analysis helps to objectively identify the most impactful works based on the number of citations received [75].

To provide an overview of the state of the art of GUI testing, Bao et al. [76] conducted a mapping study spanning the years 1991 to 2011, which included 136 publications. Since then, the field of GUI testing has experienced significant growth, with the number of papers on the topic reaching 52 in 2020 alone.

The rising interest in GUI testing has led to specialised workshops focused on this topic. In 2009, the first edition of the International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS) was held, co-located with IEEE's International Conference on Software Testing, Verification, and Validation (ICST). This event was followed by the first edition of

INTUITEST (International Workshop on User Interface Test Automation) in 2015, co-located with STV and UCAAT. In 2018, these two workshops merged to form INTUITESTBEDS[1], focusing on user interface test automation and testing techniques for event-based software.

To the extent of available knowledge, this is the first bibliometric analysis of automated GUI testing over the past 30 years. The main contributions are to:

1. Provide facts about the size and growth of the field.
2. Indicate the type of publications and their rankings, including most cited papers, prolific authors, and influential journals and conferences.
3. Show the distribution of the publications among the available sources and over the years using the Referenced Publication Year Spectroscopy.
4. Present and discuss the productivity and the level of collaboration among researchers in the literature.
5. Use the bibliometric laws of Bradford [77] to know the most influencing journals, and of Lotka [78] to evaluate scientific productivity of authors.
6. Show the evolution of the major research topics in the field by analysing the keywords used by the authors.

The chapter is structured as follows. Section 2.1 presents the scope of automated GUI testing, establishing the criteria and definitions used to select relevant studies for the bibliometric analysis. Section 2.2 details the methodology employed in this study, including data retrieval, pre-processing steps and the techniques applied to interpret the data. Section 2.3 presents the analysis results, encompassing the growth and size of the field, influential authors, relevant publications and keyword trends. Section 2.4 addresses potential threats to the study's validity. Finally, Section 2.5 concludes the chapter by summarising the key insights derived from the bibliometric analysis.

[1] International Workshop on User Interface Test Automation and Testing Techniques for Event Based Software, https://www.intuitestbeds.org/

2.1 Scope: automated GUI testing

This section outlines the definition of *automated GUI testing* used to determine which papers should be included in the bibliometric analysis. As explained in Chapter 1, GUI testing involves executing sequences of events on the GUI widgets of a System Under Test (SUT) and checking test oracles, intending to identify failures, reduce risks, and improve the quality of the SUT.

It is possible to automate the execution of these test sequences, known as *automated GUI testing*. However, other activities related to GUI testing can also be automated. Therefore, the definition of automated GUI testing was refined to encompass these additional activities. Thus, automated GUI testing in the context of this study includes the automated generation of test cases, test oracles, and test execution, as well as other related activities such as test selection, prioritisation, and debugging.

Automating the creation of test sequences: Test sequences in GUI testing consist of sequences of GUI actions/events on widgets together with input values. Test sequences are made to cover some test goal of the SUT (e.g., checking some specific functionality or finding a failure). Test sequence defines which path through the SUT should be taken (which *states* should be visited), i.e., *what* actions will be executed, and in which *order*.

Automating the definition or checking of the oracles: Oracles [40] are procedures that distinguish between the correct and incorrect behaviour of the SUT. Since test cases in GUI testing are sequential, oracles can be checked after each action (test step) during execution (online oracle), just once at the end of each sequence, or analyse the results after the execution (offline oracle). Test oracle automation is essential for removing the current bottleneck that inhibits greater overall test automation [40]. Without test oracle automation, a human has to determine whether the observed behaviour is correct.

Automating the analysis of test results: This consists of analysing, for example, the failures that were found in a specific SUT or evaluating the quality of the test cases that were executed, using a set of defined metrics.

If any of these activities are automated, the study will consider it "automated GUI testing" (even if the test execution is conducted manually), and the papers related to such automation will be included in this bibliometric analysis.

2.2 Methodology

This study adopts the workflow for bibliometric analysis defined in [79], which consists of the following steps: data retrieval, pre-processing, analysis, and visualisation.

2.2.1 Data retrieval

Scopus was used for the search process as it is the largest abstract and citation database of peer-reviewed literature, providing broader coverage compared to other scientific repositories such as WoS [80]. Scopus gives a comprehensive overview of research output across various fields, including science and technology, and also includes valuable tools for research analysis and visualisation. To ensure no relevant papers were missed, the initial search term "Automated GUI testing" was expanded by generating a set of related terms for each keyword (refer to Table 2.1).

Term	Family
automated	automated, automatic, automatically, automation, automating, automate, generation, generate, generating, generator
GUI	GUI, UI, "graphical user interface"
testing	testing, test, tested

Table 2.1: Family of words for the search string

Figure 2.1 presents the complete search query. To ensure the relevance of the results, the search terms had to appear in the article's title, abstract, or keywords, which was achieved using the Scopus operator TITLE-ABS-KEY (refer to lines 1-3). To further refine the search results, a minimum distance between terms

was established using the W/ operator. After several tests, the minimum distance was set to 5. In Figure 2.1, each family of words is represented by its primary term, and the search query was adjusted to include the entire family of words using the OR operator, allowing for the appearance of at least one term within each family. At this stage, the search query was designed to return all indexed papers that contained at least one term from each family in the title, abstract, or keywords (TITLE-ABS-KEY), with at least one pair of terms from different families within the minimum distance.

```
1  TITLE-ABS-KEY((Automated W/5 Testing) AND GUI)
2    OR TITLE-ABS-KEY((Automated W/5 GUI) AND Testing)
3    OR TITLE-ABS-KEY((GUI W/5 Testing) AND Automated)
4
5  AND LIMIT-TO(LANGUAGE, "English")
6
7  AND PUBYEAR>1989 AND PUBYEAR<2021
8
9  AND (LIMIT-TO(DOCTYPE, "cp") OR LIMIT-TO(DOCTYPE, "ar")
10      OR LIMIT-TO(DOCTYPE, "ch") OR LIMIT-TO(DOCTYPE, "Undefined"))
11
12 AND (LIMIT-TO(SUBJAREA, "COMP")
13      OR LIMIT-TO(SUBJAREA, "ENGI")
14      OR LIMIT-TO(SUBJAREA, "MATH"))
```

Figure 2.1: Boolean search query for systematic review

Using the Scopus facilities, papers were also excluded according to their type, language and publication date, excluding works that:

exC1: are not written in English (on line 5, using the Scopus Document field code: LANGUAGE and limiting it to "English")

exC2: are published before the year 1990 and after 2020 (on line 7 using the Scopus Publication field code: PUBYEAR)

exC3: are not conference, workshop, journal publications or book chapters (in lines 9 and 10) using the Scopus Document field code: DOCTYPE and limiting it to types Conference Paper-"cp", Article-"ar", Book Chapter-"ch" and "Undefined"). The last one was included because some documents have been

accepted for publication but have not yet been assigned to a journal or conference, so they are temporarily indexed as "Undefined".

exC4: do not belong computer science area (in lines 12-14) using the Scopus subject areas: COMP [2], ENGI [3] and MATH[4].

The search was performed in January 2021. The total amount of papers retrieved was 2240.

2.2.2 Pre-processing

Pre-processing the data retrieved is necessary since references may be duplicated, the authors' names may appear in different formats, and papers that contain the terms can be unrelated to automated GUI testing, among others.

Initially, papers from unrelated fields in Scopus were manually excluded, reducing the total number of papers to 1233. This step was necessary because some papers may be classified under multiple fields, such as Computer Science and Social Science, if they describe a social science study using a computational system. As a result, such papers are retrieved by the search query even if they do not belong to the Computer Science, Engineering, or Mathematics (COMP, ENG or MATH) fields. Any papers that were unrelated to the topic of automated GUI testing were manually excluded from the study.

In addition to the goal of conducting a bibliometric analysis on automated GUI testing, this study aimed to establish a repository of GUI testing research.

[2]classifying: Computer Science(miscellaneous), Artificial Intelligence, Computational Theory and Mathematics, Computer Graphics and Computer-Aided Design, Computer Networks and Communications, Computer Science Applications, Computer Vision and Pattern Recognition, Hardware and Architecture, Human Computer Interaction, Information Systems, Signal Processing, Software

[3]classifying: Engineering(miscellaneous), Aerospace Engineering, Automotive Engineering, Biomedical Engineering, Civil and Structural Engineering, Computational Mechanics, Control and Systems Engineering, Electrical and Electronic Engineering, Industrial and Manufacturing Engineering, Mechanical Engineering, Mechanics of Materials, Ocean Engineering, Safety, Risk, Reliability, and Quality, Media Technology, Building and Construction, Architecture

[4]classifying: Mathematics (miscellaneous), Algebra and Number Theory, Analysis, Applied Mathematics, Computational Mathematics, Control and Optimisation, Discrete Mathematics and Combinatorics, Geometry and Topology, Logic, Mathematical Physics, Modelling and Simulation, Numerical Analysis, Statistics and Probability, Theoretical Computer Science

A simple and flexible environment was sought to support the work and enable future interactions with the extracted papers. Consequently, BUHOS [81], an open-source, web-based paper management system, was used. The 1233 papers were uploaded to BUHOS, and additional exclusion criteria (exC5 and exC6) were defined and manually applied by carefully reviewing the title and abstract of each paper.

exC5: clearly off-topic, i.e. not at all related to the scope (Section 2.1)

exC6: not a primary study

The papers were uploaded to BUHOS in BibTeX format, including all available information, such as authors, citation counts, and venue. Any missing information not present in the BibTeX file was automatically extracted from Crossref[5].

The 1233 papers were divided among the authors, who, after reading the title and abstract, marked them as included, excluded or undecided. Next, a collective analysis was carried out involving all authors to make a final decision on the undecided papers, resulting 720 papers.

Additionally, BUHOS provides a backward snowballing [82] feature that scans the references of each paper and includes any papers referenced by a minimum number of papers already included in the initial pool. This feature was used to identify interesting works that Scopus did not initially retrieve. This added 24 papers, resulting in the total of 744 included publications.

2.2.3 Analysis and Visualization

CRExplorer [83] and Biblioshiny [84] were used to analyse and visualise the data. Both tools were chosen for their specific capabilities in generating bibliometric maps. Moreover, Biblioshiny[6] is a free tool that provides a broader range of analysis possibilities compared to other bibliometric tools [85]. In addition, Scopus analysis functionalities were used in conjunction with Microsoft Excel to generate charts.

[5]Crossref is a digital citation and linking service that provides metadata for scholarly content, https://www.crossref.org

[6]Official website: https://www.bibliometrix.org/home/index.php/layout/biblioshiny

To ensure data consistency, normalisation was applied to the keywords using a thesaurus of synonyms[7], and on the author names by taking accent marks and different formatting into account. For the conference information, the description was split to accurately extract the name of the conference separately from the publisher and year of publication.

2.3 Results

This section presents the results of this study and a brief discussion related to the size and growth of the area. It also classifies the type of publication (journals, conferences, or workshops), the most influential journals and conferences, the most influential papers, and the most influential authors.

2.3.1 Size of the area and growth

The number of publications in a field over time is a central piece of information for investigating its growth and development.

Table 2.2 presents, on a year-to-year basis, the total amount of publications (column #), the percentage of the cumulative amount by each year (column %), and the growth of the number of publications against the previous year (column ↗). Figure 2.2 depicts the evolution of the growth per year along with the trend.

The first decade covered by this study only included 18 papers related to the field. Two years (1992 and 1993) passed without papers. In the second decade, this number increased to 170 works. Finally, in the third decade, 556 works were found. Given that 41.4% of all documents have been published in the last five years, it is likely that the automated GUI testing field will continue to grow at a similar pace to the last decade.

[7] Available at: https://gui-testing-repository.testar.org/keywords

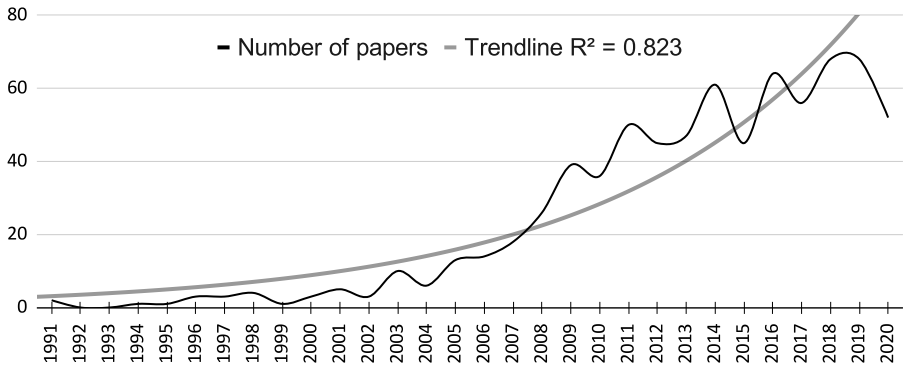


Figure 2.2: Evolution of the number of publications

Year	#	%	Growth (↗)	Year	#	%	Growth (↗)
1991	2	0.27%	-	2006	14	1.88%	7.69%
1992	0	0.00%	-	2007	18	2.42%	28.57%
1993	0	0.00%	-	2008	26	3.49%	44.44%
1994	1	0.13%	-	2009	39	5.24%	50.00%
1995	1	0.13%	0.00%	2010	36	4.84%	-7.69%
1996	3	0.40%	200.00%	2011	50	6.72%	38.89%
1997	3	0.40%	0.00%	2012	45	6.05%	-10.00%
1998	4	0.54%	33.33%	2013	47	6.32%	4.44%
1999	1	0.13%	-75.00%	2014	61	8.20%	29.79%
2000	3	0.40%	200.00%	2015	45	6.05%	-26.23%
2001	5	0.67%	66.67%	2016	64	8.60%	42.22%
2002	3	0.40%	-40.00%	2017	56	7.53%	-12.50%
2003	10	1.34%	233.33%	2018	68	9.14%	21.43%
2004	6	0.81%	-40.00%	2019	68	9.14%	0.00%
2005	13	1.75%	116.67%	2020	52	6.99%	-23.53%

Table 2.2: Publications by Year: Number (#), Percentage (%), and Growth (↗).

Between 2009 and 2013, the number of papers increased, deviating from the overall trend. There could be various reasons for this. The first ICST conference, held in 2008, was the first international conference entirely dedicated to software testing. Moreover, the first TESTBEDS workshop was celebrated at ICST in 2009. There was also an increase in papers related to web testing. This can be related to the fact that in 2009, it was decided to merge Selenium RC and Webdriver and called the new project Selenium WebDriver [53], or Selenium 2.0. A third reason might be that Sikuli started in 2009 [86]. Sikuli is a visual approach to searching GUIs using screenshots, allowing users to take a screenshot of a GUI element (such as a toolbar button, icon, or dialogue box) and query a help system using the screenshot instead of the element's name. Finally, in 2009, there was an increase in papers related to mobile testing. This is probably related to the fact that Apple's App Store went live in July 2008, and the Android Market went live in August.

During 2020, a decrease in the number of publications was observed, which could be attributed to the Covid-19 pandemic. This situation likely impacted research outcomes due to the cancellation of several conferences, reduced mobility, and other disruptions [87].

2.3.2 Types of publications and their ranking

Publications were found across various types, including journals, conferences, workshops, and book chapters. Figure 2.3 shows the number of papers of each kind.

The majority of papers have been published in conference proceedings. This makes sense since conferences provide feedback to researchers more quickly than journals. Moreover, in many cases, papers describing part of a more extensive solution are presented at conferences to obtain feedback and validate each piece of work. Later, the entire proposal is presented in a journal. This is also the behaviour in the whole Computer Science field [88].

Table 2.3 shows journals with the highest number of publications in the field, highlighting IEEE Transactions on Software Engineering (TSE) as the top journal with 12 published articles. Even though the automated GUI testing field has been

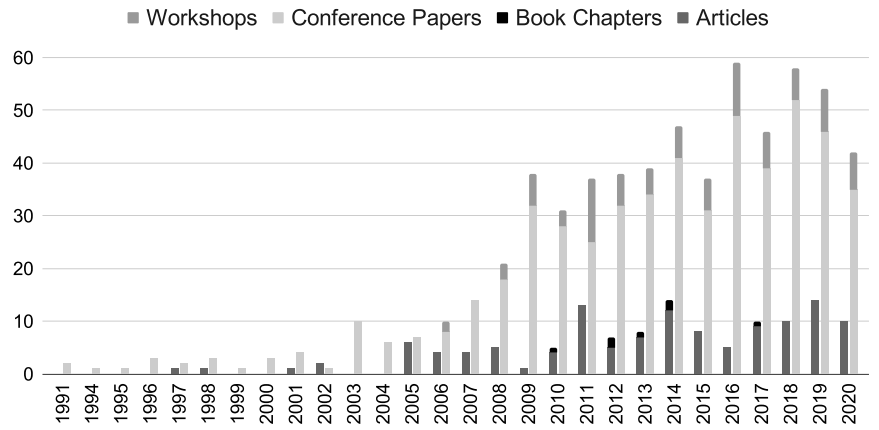


Figure 2.3: Number of papers published in (journals + books) vs (conferences + workshops)

steadily growing during the last 3 decades, STVR is the first journal to launch a special issue entirely dedicated to this field in only 2020. Papers included in that special issue were not counted for this study because they were not published yet.

By examining the data in Table 2.3, Bradford's Law [77] can be applied. Bradford's law is related to the distribution of papers among journals of a specific discipline. This law establishes that the total number of journals in a research field can be divided into three categories or zones, each containing approximately one-third of the total number of papers in the field. The first zone includes a small number of highly influential journals that publish a disproportionately large number of papers in the field. These journals are considered the core of the field. The second zone corresponds to the journals with an average number of papers. The last zone corresponds to several journals that publish fewer papers.

The zones are characterised by a growth factor n, which describes how the number of journals increases between the zones. This progression leads to a characteristic ratio of journals across the zones: $1{:}n{:}n^2$, with the Second Zone containing n times as many journals as the Core Zone and the Third Zone con-

Journal Name	Publications	Proportion	Impact Factor (SJR)
Transactions On Software Engineering (TSE)	12	9.83%	1.19
Information And Software Technology (IST)	8	6.55%	0.78
Software Quality Journal (SQJ)	7	5.73%	0.36
IEEE Software	6	4.92%	0.81
Transactions On Software Engineering And Methodology (TOSEM)	5	4.10%	0.76
Software Testing Verification And Reliability (STVR)	5	4.10%	0.31
Empirical Software Engineering (ESE)	4	3.28%	1.08
Information Technology Journal	4	3.28%	0.11
ACM SIGPLAN Notices	3	2.46%	4.90
IEEE Access	3	2.46%	3.90
Innovations In Systems And Software Engineering	3	2.46%	1.90
Remaining 54 from the total of 65 journals	62	50.82%	-
Total number of papers	**122**	**100%**	-

Table 2.3: Top 11 contributing Journals

taining n^2 times as many journals as the Core Zone. The Leimkuhler model [89] provides a mathematical formalization of Bradford's Law, employing a logarithmic function to define the cumulative number of articles as a function of journal rank. This model enables precise computation of rank boundaries for the zones, offering a quantitative approach to analyse journal productivity.

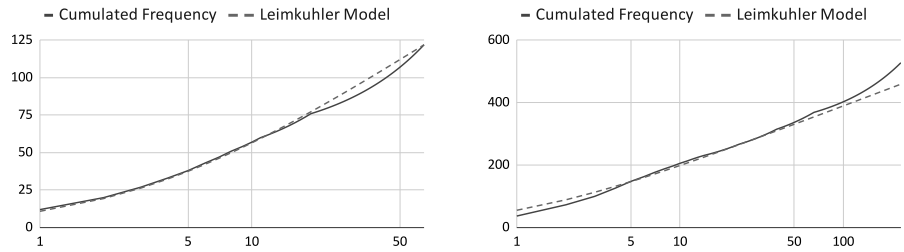
From Table 2.3, the Leimkuhler model was used to compute the rank boundaries for the zones. The top 5 journals are the core journals since they correspond to 38 articles, which is 31.1% of all 122 journal papers. The next group is found in the following 15 journals (39 articles or 32%). In order to represent the last articles, the 45 remaining journals are necessary. The Bradford relation for journals is $1{:}3{:}3^2$, reflecting the characteristic progression described by Bradford's Law, and the details per zone can be found in Table 2.4.

Zones	Journals	Publications	Bradford multiplier
Core	5	38	1
Zone 1	15	39	3
Zone 2	45	45	9
Total	65	122	

Table 2.4: Bradford's Law zones for journal articles

Zones	Conferences	Publications	Bradford multiplier
Core	8	188	1
Zone 1	38	141	4.75
Zone 2	179	199	22.38
Total	225	528	

Table 2.5: Bradford's Law zones for conference publications



(a) Applied to journals. **(b)** Applied to conferences.

Figure 2.4: Leimkhuler models.

The same model was applied to the conferences among the papers published in conference editions, obtaining a Bradford relation for conferences of approximately $1{:}5{:}5^2$ for which the details per zone can be found in Table 2.5. Given that Leimkuhler's model describes properly both journals and conferences distributions, as shown in Figures 2.4a and 2.4b respectively, Bradford's law fits this data set very well.

The 87 workshop papers were presented at 56 workshops, of which 37.50% was co-located at a CORE A* conference, 28.57% at a CORE A conference, 3.57% at a

CORE B conference and 12.50% at a CORE C conference, 5.36% at conferences with no CORE ranking, and 5.36% at workshops not co-located with any conference. The remaining 7.15% workshops were in years when no CORE ranking was given (yet).

The 528 papers were presented at 386 conference proceedings, of which 4.15% has CORE ranking A*, 16.32% CORE A, 16.84% CORE B, 10.36% CORE C and 37.31% has no CORE ranking. The remaining 14.51% conferences were in years when no CORE ranking was given (yet).

Conference name	Publications	Proportion
International Conference on Software Engineering (ICSE)	37	7,01%
International Conference on Software Testing, Verification and Validation (ICST)	36	6,81%
International Conference on Automated Software Engineering (ASE)	27	5,11%
International Symposium on Software Testing and Analysis (ISSTA)	26	4,92%
Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)	22	4,17%
IEEE International Symposium on Software Reliability Engineering (ISSRE)	15	2,84%
International Conference on Software Maintenance (ICSM)	14	2,65%
International Computer Software and Applications Conference (COMPSAC)	11	2,08%
International Conference on Software Engineering and Knowledge Engineering (SEKE)	9	1,70%
International Conference on Software Quality, Reliability and Security (QRS)	9	1,70%
Remaining 215 conferences from the total of 225 conferences	322	60,98%
Total number of papers	**528**	**100%**

Table 2.6: Top 10 of most influential Conferences

Table 2.6 shows the number of papers published in the most contributing conferences. ICSE and ICST are nearly equal at the top, although by 2020, ICSE had celebrated 42 editions, compared to ICST's 13 editions.

2.3.3 Citations and Reference Publication Year Spectroscopy

Table 2.7 presents the top 10 papers with the highest number of citations in Scopus, along with the year of publication, the complete reference, the number of citations retrieved by Scopus (Sc), and the number retrieved by Google Scholar (GS). The cites from Scopus and Scholar differ because Scholar has a much higher count. According to [90], Scholar citation data is essentially a superset of Scopus, offering substantially broader coverage.

Ref	Title	Author(s)	Year	Sc	GS
[91]	*Dynodroid: An input generation system for android apps*	Machiry, A., Tahiliani, R., Naik, M.	2013	397	672
[92]	*Using GUI ripping for automated testing of android applications*	Amalfitano, D., Fasolino, A., Tramontana, P., De Carmine, S., Memon, A.	2012	343	563
[93]	*Automated test input generation for android: Are we there yet?*	Choudhary S.R., Gorla A., Orso A.	2016	245	401
[94]	*Automated concolic testing of smartphone apps*	Anand, S., Naik, M., Harrold, M., Yang, H.	2012	231	428
[95]	*Testing Web applications by modeling with FSMs*	Andrews A.A., Offutt J., Alexander R.T.	2005	227	477
[96]	*Sikuli: Using GUI screenshots for search and automation*	Yeh T., Chang T.-H., Miller R.C.	2009	217	400
[97]	*Sapienz: Multi-objective automated testing for android applications*	Mao K., Harman M., Jia Y.	2016	207	336
[98]	*RERAN: Timing- and touch-sensitive record and replay for Android*	Gomez L., Neamtiu I., Azim T., Millstein T. Total	2013	202	341
[99]	*An event-flow model of GUI-based applications for testing*	Memon A.M.	2007	193	364
[100]	*PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps*	Hao S., Liu B., Nath S., Halfond W.G.J., Govindan R.	2014	192	321

Table 2.7: Top 10 papers with most cites in Scopus (includes cites in Google Scholar)

Most of the top 10 most cited papers focus on Android testing, with 7 out of 10 dedicated to this area. The remaining three frequently cited papers are related

to models (event-flow or state models) and widget detection (Sikuli).

The technique of Reference Publication Year Spectroscopy (RPYS) [101] is a quantitative method to identify the historical origins or turning points of research fields. This method analyses the publication years of the references cited by all the papers in a specific field. A Reference Publication Year (RPY) is reflected in the spectrogram as a pronounced peak, usually corresponding to a frequently referenced publication. These publications are of significant importance, as they may represent the origins of the research field in question.

An RPYS chart was obtained using CRExplorer and is shown in Figure 2.5, from 1960, although there are references up to 1901. The most influential year seems to be 2001 when Atif M. Memon finished his PhD entitled *A comprehensive framework for testing graphical user interfaces* [102]. He gave a big impulse to the field, as demonstrated by the RPYS.

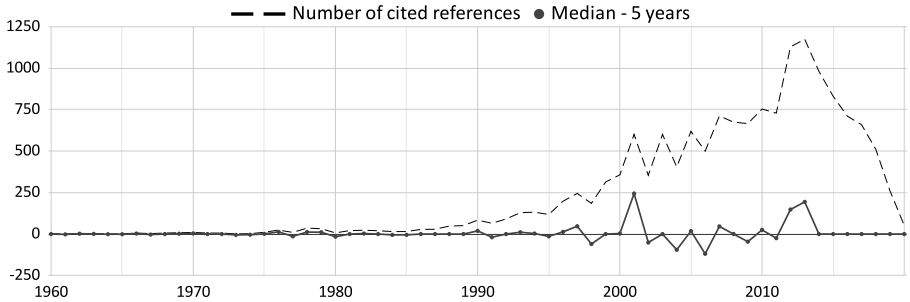


Figure 2.5: Reference Publication Year Spectroscopy

In that year, Memon published two final papers for his thesis. The first paper [103] presents a new test case generation technique based on Artificial Intelligence Planning and using a model based on a GUI structure. Artificial Intelligence and Model-based Testing are trends that will guide the research field in the posterior years to this publication, as it will be explained later in Section 2.3.7. In the second paper, Memon et al. [104] introduce different coverage criteria for GUI testing and evaluate them through a case study for the first time.

In addition, the years 2012 and 2013 appear as peaks in the Spectroscopy chart. Five publications [91, 105–108] appear among the most cited within the

field. All of them have one common topic: Android testing.

2.3.4 Most influential authors

The 744 documents that integrate this study have been written by a total of 1488 authors. Table 2.8 shows the 11 most prolific authors, among them contributing 203 publications (27.28 %). For this ranking, all authors of each paper are counted, not just the first author.

Name	Total	Journals	Conferences	Workshops	Book Chapters	Year of first publication
Memon, A.M.	53	18	29	5	1	1999
Paiva, A.C.R.	31	6	20	5	0	2005
Alégroth, E.	17	3	8	5	1	2013
Vos, T.E.J.	16	2	11	3	0	2012
Xie, Q.	15	4	10	1	0	2004
Fasolino, A.R.	13	4	5	4	0	2010
Zeller, A.	13	1	10	2	0	2012
Aho, P.	12	0	7	4	1	2011
Amalfitano, D.	11	3	4	4	0	2010
Coppola, R.	11	4	3	4	0	2016
Ramler, R.	11	1	8	2	0	2008

Table 2.8: Ranking of authors by number of publications

There is a remarkable difference between the 1st and 2nd position, as well as between the 2nd and the rest, from which a smooth distribution among the authors is observed. One notable fact is that 7 of the 11 authors published their first paper in the field since 2010, and only one published before 2000.

The distribution of the number of publications among authors is presented in Table 2.9. The largest group comprises authors who published a single paper, representing 75.81%. As shown in the table, the number of authors tends to decrease as the number of publications increases. Lotka's law describes this behaviour and states that the number of authors y publishing a certain amount of papers x is inversely proportional to x, as $y = \frac{c}{x^n}$, where n and c are two constants to be estimated for every data set. The software Lotka [109] was used to

apply the Maximal Likelihood method and estimate the parameters for this study, resulting in $n \approx 2.59$ and $c \approx 0.77$, i.e., this data set follows Lotka's general law as $y = \frac{0.77}{x^{2.59}}$. The Kolmogorov-Smirnov statistical test was applied to assess the fitness between this hypothesised Lotka model and the actual distribution of the data. Even for a significance level of 0.2, the results support the hypothesis: the calculated Lotka model fits the observed distribution.

Publications	1	2	3	4	5	6	7	8	9	10	11	12	13	15	16	17	31	53
Authors	1128	198	60	36	21	14	8	3	3	6	3	1	2	1	1	1	1	1

Table 2.9: Distributions of number of authors per number of publications

2.3.5 Productivity and funding

Figure 2.6 shows the distribution of publications per country of origin. There is a large gap between the most contributing country, the United States, and the rest. China published its first papers in 2006 and has contributed 108 publications since then, keeping a rate of 7.2 publications per year, similar to that of the United States, with 7.5 annual papers since 1991.

Although China and the US are the main contributors to the field, the European region has had a boost in the last decade and has occupied first place with 308 publications since 2015. The Asian continent has contributed 242 publications, closely followed by North America, with 245 publications so far.

A 21% of the papers included funding information. Of all the mentions, 9,7% came from private funding by big companies, such as Google, Microsoft, Amazon Web Services, and Boeing. Asia is the continent that provides the most funding resources for the majority of sponsored works (33,7%), followed by Europe (28,6%) and North America (27,1%).

The leading funding agency in Asia is the National Natural Science Foundation of China. Likewise, the leading funding agencies in Europe and North America are the European Commission and the National Science Foundation, respectively. It is worth mentioning that the only South American country that has funding is Brazil.

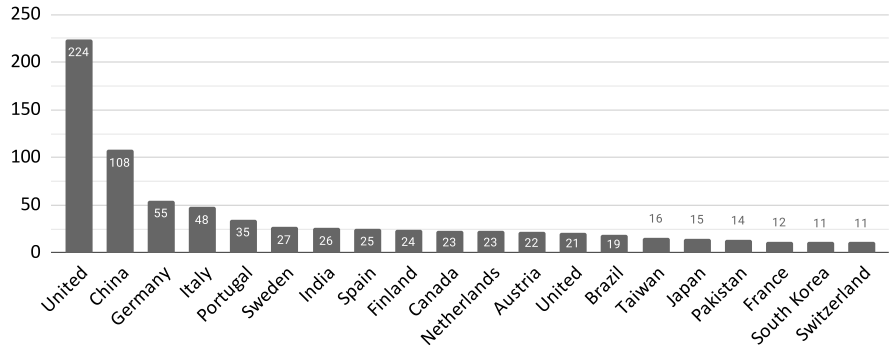


Figure 2.6: Most contributing countries

2.3.6 Collaboration

Figure 2.7 depicts the collaboration between the most prolific authors in the field from Table 2.8. Six authors have co-authored with Atif M. Memon, who can also be related to another two authors through those six. Only 2 of the 11 authors do not have co-authorship with any of the most contributing authors.

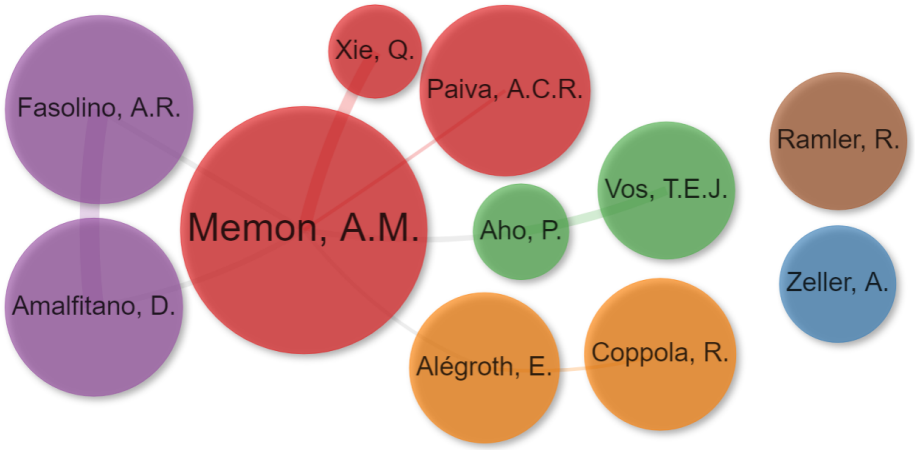


Figure 2.7: Collaboration network of authors

Regarding the co-authorship between the authors, Figure 2.8 shows the evolution of the author's collaboration over 30 years. Single-author publications have historically remained low, while publications of more than 4 authors have been increasing. However, only 18.95% of the papers have resulted from a collaboration among affiliations from different countries.

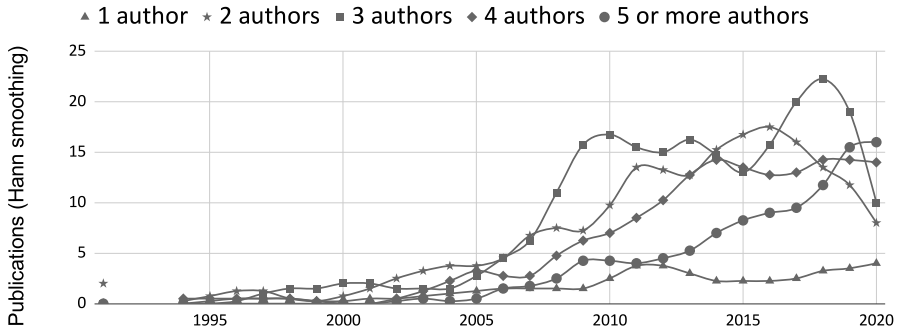


Figure 2.8: Authorship evolution

2.3.7 Trends in keywords

By analysing the keywords provided by the authors, the goal is to reveal the significant research topics within the domain and their introduction to the field. This is not as easy as counting the most used keywords [110,111]. Many keywords do not give specific information on the details of the field because they are inherent to it (e.g., software testing, GUI testing, tools, regression testing, oracle, coverage, test case). In addition, different terms are often used to describe the same concept, requiring them to be grouped.

Plural forms were standardised into their singular form using NLTK [112]. The available keywords were analysed to group the keywords, and the authors performed individual classifications, as detailed in [113]. Two brainstorming sessions were organised to develop the following classification, which represents relevant research themes in the domain under study:

mobile, **web**, model-based testing (MBT), search-based testing (**SBT**), visual-based testing (**VBT**), Artificial Intelligence and Machine Learning (**AI&ML**), Capture and Replay (**C&R**) and Automated **Exploration**

The objective is to study: **mobile** and **web** to distil the trend in the types of SUTs that are tested; **MBT**, **SBT**, **VBT**, **AI&ML** to visualise the timeline of the pick-up of different technologies into automated GUI testing; **C&R** to investigate the evolution of the trend where the focus was on these tools; and **Automated Exploration** for the shift from scripted to scriptless testing using random testing, traversal techniques and crawling. Table 2.10 shows the specific classification of each group of keywords.

Group	Keywords
mobile	mobile-device, mobile-application, smartphone-application, smartphone, android, android-testing, google-android, android-application, android-phone, mobile-testing, mobile-application-testing, mobile-application-gui-testing, mobile-application-gui-testing, smartphone-application-testing, mobile-application-test-case-generation, mobile-development, mobile-application-development, mobile-cross-platform-development, mobile-software-development, android-testing, android-application-testing, android-gui-testing, android-ui-testing, android-compatibility-testing, android-test, android-testing-automation, automated-android-testing, automated-mobile-application-testing, android-security, android-malware-detection, android-security, android-permission, secure-virtual-mobile-platform, ios, ios-application, ios-testing, ios-ui-testing, android-crawler, android-system-webview, android-gui-model, android-mobile-accessibility, mobile-communication, mobile-crawler, mobile-environment, mobile-interaction, web-based-android-emulator, mobile-browser-security, mobile-computing, mobiguitar, droidbot, humanoid-robot, espresso, robotium, selendroid, tema-tool, appium
web	web-application, webbased, web-user-interface, dynamic-web-page, web-testing, web-application-testing, testing-of-web-application, testing-web-application, web-application-test, web-interface-testing, web-system-testing, dom-based-testing, cross-browser-compatibility, cross-browser-

Group	Keywords
web (cont.)	testing, web-element-locator, web-object-identification, xpath, xpath-locator, dom, dom-selector, web-of-data, web-scraping, web-usability-evaluation, web-page-visual-representation, web-requirement, semantics-web-service-test-generation, web-accessibility, web-automation, selenium, selenium-webdriver, selenium-web-driver, webdriver, selenium-ide, selenium-testing-tool, ajax, ajax-application, asp-net, web-cat, webaii, xml-injection, rich-internet-application-testing
SBT	genetic-programming, genetic-combination, evolutionary-testing, evolutionary-algorithm, genetic-algorithm, search-based, search-based-software-engineering, search-based-testing, ant-colony-optimisation, particle-swarm-optimisation, ant-colony-optimisation-(aco), multi-objective-optimisation, multi-objective-pso, metaheuristics, traversal-algorithm, best-first-search, depth-first-search, depth-first-traversal
VBT	image-recognition, sikuli, image-analysis, image-processing, image-recognition-testing, image-search, image-similarity, image-storage, opencv, visual-gui-testing, visual-testing, object-detection, visual-gui-testing, visual-testing, jautomate, eyeautomate, pixel-comparator, ocr, element-recognition
C&R	capture/replay, record-and-replay, capture-and-replay-tool, test-recording-and-playback, gui-capture/replay, record-and-playback-problem, record/replay, test-recording, capture-replay, capture-replay-testing, capture/playback, gui-regression-testing, visual-regression-testing, selenium, selenium-webdriver, selenium-web-driver, webdriver, selenium-ide, selenium-testing-tool, testcomplete, record
exploration	systematic-exploration, automatic-exploration, random-testing, monkey-testing, monkey, monkey-test, testar, automated-traversal-tool, gui-exploration, gui-traversal, systematic-gui-exploration, automated-gui-exploration, crawling-efficiency
AI&ML	ai-algorithm, ai-planning, machine-intelligence-quotient, intelligent-planning, computer-vision, automation-computer-vision-gui, active-learning, active-learning-testing, machine-learning, learning, reinforcement-learning, q-learning, deep-reinforcement-learning, deep-q-network, multi-armed-bandit-problem, neural-network, deep-learning,

Group	Keywords
AI&ML (cont.)	deep-neural-network, convolutional-neural-network, computational-intelligence, unsupervised-learning, support-vector-machine, natural-language-processing, clustering, cluster-algorithm, cluster-analysis, multi-agent-collaboration, agent-based-testing, teaching-learning-based-optimization
MBT	model-based-gui-testing, mbt, mbgt, modelbased-gui-testing, modelbased-gui-testing, model-based-testing-mbt, model-testing, model-based, model-based-test-generation, model-based-testing-uml, multi-model-testing, test-model, test-model-development, model-based-input-generation, model-based-test-input-generation, model-generation, automatic-model-generation, automatic-gui-map-generator, automatic-gui-model-generation, gui-model-gen-eration, generative-model, gui-model-gen-eration, automatic-gui-model-generation, gui-modeling, gui-modelling, gui-model, visual-gui-modelling, gui-map, ui-model, user-interface-model, visual-gui-modelling, user-interface-model, gui-state-model, modeling, graphical-modeling-dsl, model-extraction, model-inference, model-analysis, model-mining, model-transformation, model-validation-and-analysis, directed-graph-model, software-modeling, dynamic-modeling, dynamic-gui-model, event-flow, event-interactive-graph, event-pattern, event-sequence, event-flow-graph, event-flow-model, context-event, event, event-sequence-graph, gui-event, model-based-exploration, activity-diagram, activity-flow-graph, uml-activity-diagram, gui-specification, user-interface-specification, spec-explorer, guitar-testing-system, guitar, mobiguitar, domain-specific-language, fsm, finite-state-machine, finite-state-machine-testing-framework, state-machine, finite-state-automaton, specification-based-testing, test-specification-language, ontology, ontological-modeling, gui-ripping, sequence-based-specification, uml, uml-profile, concur-task-tree, concurtasktrees, gui-call-graph, gui-control-flow-graph, gui-controls-graph, augmented-model, component-tree-graph, dsl, knowledge-graph, labeled-transition-system, petri-net

Table 2.10: Grouping the keywords

Figure 2.9 shows the cumulative frequency values per each group of keywords annually. MBT and C&R made their first appearance in 1998. Since then, MBT has been the main topic of the field until Mobile reached a greater number of papers in 2019. Since 2007, in just 13 years, Mobile has become the most frequent keyword.

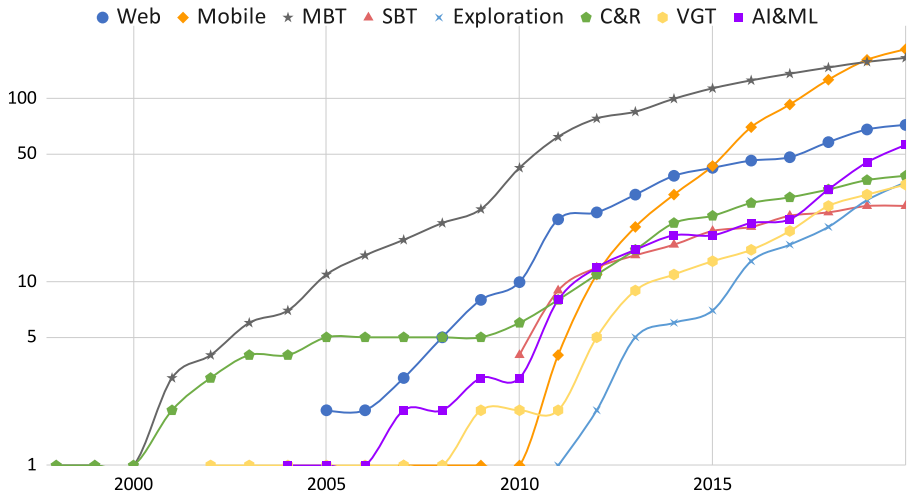


Figure 2.9: Cumulative frequency of keywords

As of 2010, two topics were introduced: SBT, which includes genetic algorithms and swarm intelligence, and Automated Exploration, with algorithms for traversing or randomly exploring the GUI. Exploration has grown in recent years, as seen in Figure 2.9. AI&ML is the technology that has had the greatest increase in the last 5 years, only being surpassed by MBT towards the conclusion of this study.

The evolution of the eight groups of keywords was analysed using the information presented in Figure 2.10, generated with Biblioshiny. The size of the points classifies each keyword according to the number of papers in which it has been used (i.e., its frequency), while the position of the points indicates the year in which each keyword has reached 50% of its frequency. The horizontal lines begin and end in the year in which a keyword reaches 25% and 75% of publications,

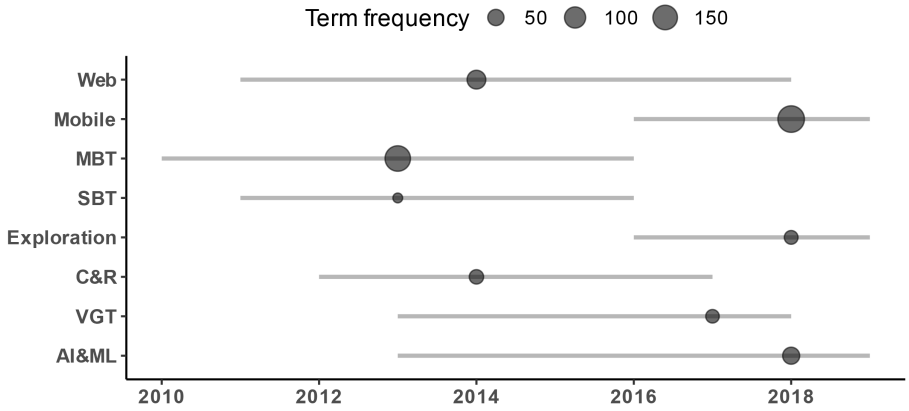


Figure 2.10: Keywords trends

respectively. All keywords reached 25% of their frequency in the last decade, i.e., 75% of the papers that mention these keywords were published in the last 10 years.

Publications mentioning Web-based SUT have remained constant. Remarkably, 50% of MBT papers have been published as of 2014, given that MBT is one of the first topics in the field. This coincides with the increase in Exploration techniques. Conversely, SBT techniques have a lower frequency.

C&R has decreased in frequency, coinciding with the considerable increase in VBT. This might indicate that CR is being replaced by Image Recognition or Image Comparison techniques.

AI&ML has appeared in 56 papers: by 2013, it had appeared in 48 papers (26.79%), and it took 5 years to reach 50% of its total frequency. However, just one year was needed for AI&ML to reach 75%. In the last two years, AI&ML appeared in as many papers as in the entire previous history of the field.

Considering their frequency and accelerated growth in recent years, Mobile-based SUT and AIML are the trending topics in the field. In addition, exploration techniques have accelerated in the last five years, although they have not yet reached a large number of papers.

2.3.8 Discussion

This bibliometric analysis of 30 years of automated GUI testing provides valuable insights into the evolution and dynamics of this research field. The study highlights significant publication growth, with the field gaining traction in the last decade due to technological advancements and the increasing complexity of software systems. The introduction of specialised workshops and the steady rise of contributions from mobile and web testing contexts underscore the field's responsiveness to industrial and technological shifts. Notably, the emergence of mobile testing and Artificial Intelligence techniques in the past decade indicates a transition toward more sophisticated and adaptive testing methodologies.

The keyword analysis reveals shifts in research focus over time, with model-based testing maintaining a consistent presence while mobile testing and AI-based approaches have surged in popularity. Traditional capture-and-replay methods have declined, potentially supplanted by visual-based testing methods leveraging image recognition. The rapid adoption of AI and novel exploration strategies signals a fundamental shift in testing approaches, highlighting the growing demand for more adaptable, efficient, and intelligent quality assurance methods. These findings not only chart the historical trajectory of automated GUI testing but also provide a roadmap for future research directions.

2.4 Threats to Validity

This section discusses the potential threats to the validity of the bibliometric analysis conducted in this study, following the guidelines proposed by [114].

2.4.1 Internal Validity

Scopus, the largest database of peer-reviewed scientific literature, was used to mitigate internal validity threats. A search string was defined to retrieve relevant publications, and the results were validated with a small set of known relevant works to ensure accuracy. However, some relevant studies might not have been

captured due to potential misclassification within the database or inherent limitations of the search string.

Additionally, the keyword extraction process threatens internal validity, as keyword grouping and classification inaccuracies could skew trend analyses. To mitigate this, keyword grouping was conducted collaboratively by multiple authors to enhance objectivity and consistency.

2.4.2 External Validity

This study focuses exclusively on automated GUI testing within the Scopus-indexed literature, predominantly in English. Consequently, publications in other languages or those indexed in different databases may not be represented, potentially limiting the applicability of the findings to the global research landscape. Future studies could incorporate additional databases and languages to provide a more comprehensive view of the field.

Regarding the replicability of the study, a protocol has been clearly defined, and the entire process has been documented to mitigate this threat. The metadata of the works was used to perform this analysis, mitigating the threat that results may be biased by researchers' judgment. To gain a deeper understanding of the techniques used for automated GUI testing, this work proposes conducting a mapping review to establish trends in the area.

2.4.3 Construct Validity

This analysis used a combination of established bibliometric tools, CRExplorer and Biblioshiny, to analyse and visualise the data. These tools are widely recognised and validated within the bibliometrics community, ensuring that the measured constructs (e.g., publication trends, author influence) are accurately captured.

2.4.4 Conclusion Validity

Using validated bibliometric tools strengthens the confidence in the study's findings. Ensuring transparent documentation of the methodology and analysis processes reinforces the validity of this study's conclusion.

2.5 Conclusions

The bibliometric study of automated GUI testing over the past 30 years reveals a dynamic and evolving field that has grown significantly in response to the increasing complexity and variety of software systems. Publications have increased continuously, with exponential growth observed in the last decade, suggesting that this trend is likely to continue.

The analysis exposes the transformation in automated GUI testing methodologies. The decline in capture-and-replay techniques has paralleled the rise of more sophisticated approaches using visual recognition, exploration, and AI capabilities. This evolution reflects broader technological shifts, particularly evident in the expanding focus on mobile and web testing domains. These changes demonstrate how testing strategies have adapted to meet modern software development needs.

Collaborative efforts, particularly in Europe, and funding from prominent agencies such as the European Commission and the National Natural Science Foundation of China have driven much of the recent progress. However, limited cross-border collaboration and a concentration of highly productive authors suggest opportunities to broaden engagement within the global research community.

Finally, a repository[8] was developed, listing all the 744 referenced papers and further bibliometric results. This repository provides access to the curated dataset, further enhancing the study's replicability. This study demonstrates the evolution of automated GUI testing over three decades, revealing key research trends and highlighting promising directions for AI integration in future work.

[8]Available at: https://gui-testing-repository.testar.org

3 TESTAR

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald E. Knuth, Correspondence to Dr. P. van Emde Boas

TESTAR[1] is an open-source tool that carries out automated testing without the need for scripts, falling into the scriptless GUI testing tools category. It implements a scriptless approach, meaning the test cases do not have to be defined prior to test execution. Instead, each test step is generated during the test execution based on the actions available at that specific time and state of the GUI.

The underlying principle of TESTAR is straightforward: generate test sequences of *(state, action)*-pairs by starting up the SUT in its initial state and continuously selecting an action to bring the SUT to another state. The action selection characterises the most fundamental problem of intelligent systems: *what to do next*. The challenging parts are optimising the **action selection** to find faults and recognising a faulty state when it is encountered with an **oracle**.

A testing session with TESTAR is illustrated in Figure 3.1. After starting up the

[1] Official website: https://testar.org/

SUT, the tool goes into the loop of continuously selecting and executing an **action** to bring the SUT from one **state** to another state until some stopping criterion has been met, after which the SUT is closed. In the following sections, each of the basic steps of the approach will be described:

- Obtaining the GUI state (Section 3.1).
- Deriving the set of actions that a potential user can execute in that specific state (Section 3.2).
- Selecting and executing one of these actions (Section 3.3).
- Evaluating the new state to find failures (oracles) (Section 3.5).

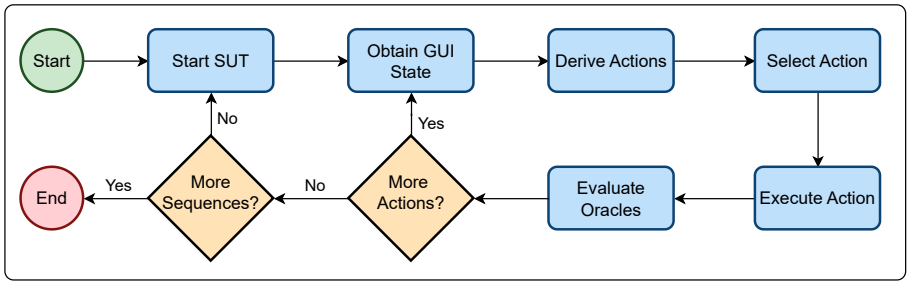


Figure 3.1: TESTAR testing cycle

These steps are implemented with a modular design, as depicted in Figure 3.2, allowing testers to enhance and customise them as needed. This design offers several advantages, such as scalability, ease of maintenance, and more flexibility.

Section 3.4 introduces the representation of states and actions, explaining how concrete and abstract identifiers are constructed to identify GUI states and actions effectively. Subsequently, Section 3.6 describes the runtime execution of TESTAR and the execution of the test sequence loop (as shown in Figure 3.1). This Section also explains how new Action Selection Mechanisms (ASMs) can be added to enhance the exploration process by prioritising or selecting actions strategically. Section 3.7 discusses a test run's outputs, including logs, screenshots, and HTML reports that support fault analysis and reproducibility.

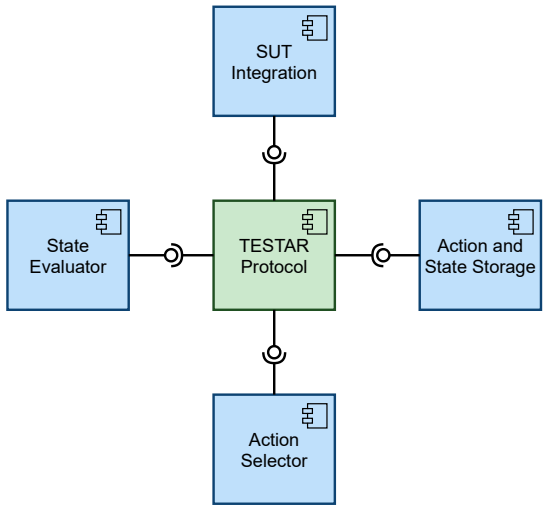


Figure 3.2: TESTAR modular architecture

The chapter further elaborates on advanced functionalities. Section 3.9 explores action filtering techniques, such as regular expressions, click-based filtering, and programmatic customisation, to optimise the testing process. The chapter concludes with Section 3.10, where industrial case studies demonstrate the practical application and effectiveness of TESTAR, outlining the phases for setting up TESTAR for a SUT, providing a structured, systematic approach to deployment.

3.1 Obtaining the GUI State

A GUI can consist of a wide range of **widgets**. Examples of these are in Table 3.1. These widgets are structured in a hierarchy called the **widget tree**. Figure 3.3 displays an example of a widget tree. Each node corresponds to a visible widget and contains widget properties like its type, position, size, title, and whether it is enabled.

These trees and properties can be defined as plugins for the interaction with the SUT as shown in Figure 3.2, in various ways:

Windows	Menus	Controls
• Main window • Child windows • Popup windows • Dialog windows	• Menu bars • Dropdown menu • Context-aware menu	• Buttons • Textboxes • Links • Radio buttons • Checkboxes • Dropdown select boxes • Sliders • Tabs • Scrollbars

Table 3.1: Examples of widgets of which a GUI can be composed.

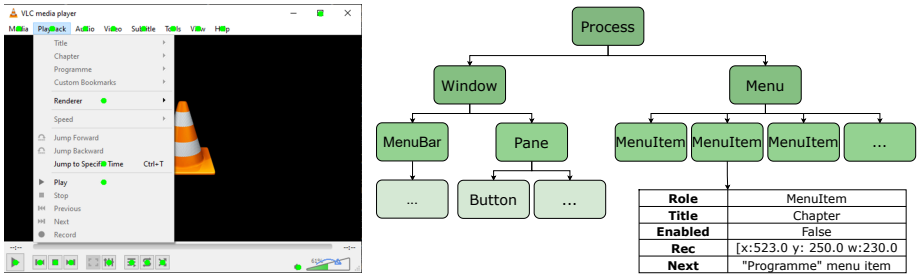


Figure 3.3: The state of a GUI as a widget tree.

- **accessibility APIs**, which allow computer usage for people with disabilities at the Operating System level (i.e. UIA Automation for Windows, ATK/SPI for Linux, NSAccesibilty for MacOS). These accessibility APIs allow us to gather information about the visible widgets of an application and give TESTAR the means to query their property values.

- programmatic APIs, or **automation frameworks**, like the Selenium Webdriver [53] at the browser level, or Appium [115] for iOS, Android, and Windows.

- **language specific APIs** like the Java Access Bridge for existing Java objects at the Java Virtual Machine level.

- **SUT specific APIs** are an option: a first experience with it has been done by testing a smart home through a RESTful API with TESTAR [116].

- **Image recognition** [117] can be used as a platform-independent way to obtain the state of the GUI from the screenshots. However, image recognition algorithms are less accurate and give less information than the technical APIs.

In the current[2] implementation of TESTAR there are plugins for detecting the state using the UIA Automation for Windows, Selenium Webdriver for web applications and Appium for Android. For example, the UIA API gives access to around 170 attributes or properties [52], enabling the retrieval of detailed information such as:

- The **role** of a widget: whether it is a button, checkbox, dropdown, etc.
- The **path** that the widget has in the stack of widgets on the screen, i.e. the widget tree.
- The **size** which describes a widget's rectangle (necessary for clicks and other interactions).
- Whether a widget is **enabled**, as interacting with disabled widgets may not be meaningful.
- Whether a widget is **focused** (has keyboard focus) so that the tool knows when to type into text fields.
- Attributes such as **title, help** and other descriptive attributes are essential to distinguish widgets from each other and give them an identity.

All these properties and their values are stored in the *widget tree*. In this way, these trees capture the current *state s* of the GUI like the example from Figure 3.3.

Consider a widget tree that represents a specific state s. The nodes of this *widget tree* are the widgets visible on the GUI in that particular state s. This set of nodes is denoted as $W(s) = \{w_1, w_2, \ldots, w_k\}$, where each w_i represents a widget such as a button, slider, text field, or menu. The edges of the tree are

[2]Plugins for Linux using ATK/SPI, and for macOS using NSAccesibilty exist in older versions.

defined by the parent-child relationships: each child widget is displayed within the screen area occupied by its parent widget. The set of edges is denoted by $E(s)$. A directed edge $(w_i, w_j) \in E(s)$ exists when $w_i \in W(s)$ is the parent widget of $w_j \in W(s)$ in state s.

The state is further defined by the values of all properties associated with the widgets. For a widget $w \in W(s)$, $P(w, s)$ denotes the set of all properties $\{w.p_1, w.p_2, \ldots, w.p_m\}$ (e.g. **role**, **title**, **position**, **enabled**)

All the properties $P(w, s)$ obtained by TESTAR in state s for the widgets in $W(s)$ through a plugin for interacting with the SUT are associated with the TESTAR representation of States, Widgets and Actions. This is done through Tags and is depicted in Figure 3.4.

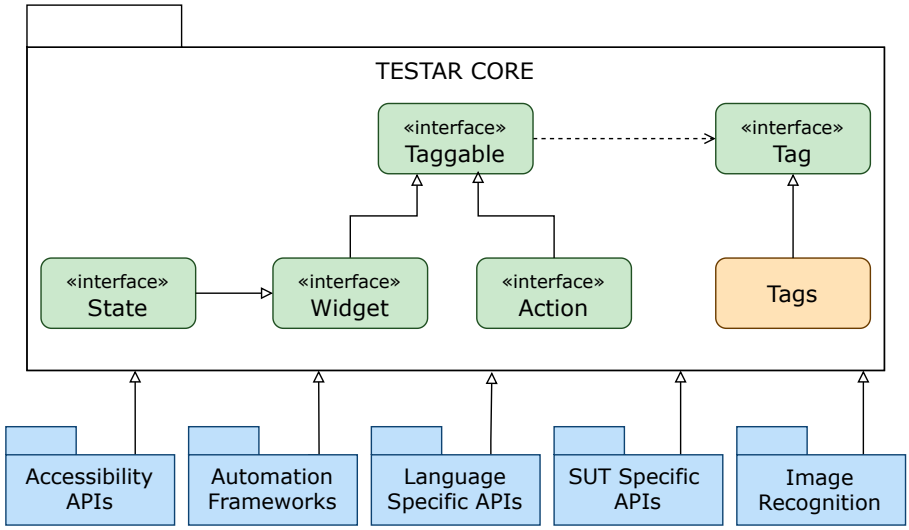


Figure 3.4: Taggable classes: `State`, `Widget` and `Action`.

Taggable classes implement the Taggable interface, which means that Tags can be added to their instances. In TESTAR, the interfaces State, Widget and Action are taggable, and the **`Tags`** are pairs of: (property name, value). Properties that are common to all widgets are defined in a final class **`Tags`**. The properties specific to an implemented API technology or automation framework (such

as Windows UIAutomation, Selenium Webdriver, ATK/SPI) are defined in specific API-Taggable final classes (UIATags, WebTags, AtSpiTags). We can use the get method to read the properties of taggable objects (i.e. an instance of the classes **State**, **Widget** and **Action**) as follows:

```
taggableObjectName.get(Tags.PropertyName)
```

In Example 3.1 there is an if-statement in line 1 whose guard checks whether some **action**'s **role** tag equals **LeftClick**. Similarly, in line 3, the **href** tag is checked for some **example-text** that we want to act upon in the if-statement.

```
1 if (action.get(Tags.Role).equals("LeftClick"))
2     {.....}
3 if (widget.get(WebTags.Href).contains("example-text"))
4     {.....}
```

Example 3.1: Obtaining property values

Obtaining the state, i.e. extracting the properties of the widgets and building the widget tree, is done automatically after each executed action. For Windows desktop applications, TESTAR monitors the CPU usage of the SUT process to figure out when the SUT has finished executing a GUI action. However, the widget tree is sometimes extracted before the GUI has finished updating, resulting in a partial widget tree. If the partial tree contains interactive widgets, actions are derived for them. If not, a default action (such as executing an NOP action or pressing the ESC key) will be executed, and testing will continue by deriving the state again. TESTAR can be configured to change the waiting time between the executed action and the next widget tree construction.

For Web applications and the Selenium Webdriver framework, TESTAR offers the possibility to use a JavaScript command document.readyState to wait until the web page has been loaded. However, this has the disadvantage of waiting for web pages to load their ads. Moreover, collaboration with partners has revealed that this functionality is insufficient in some cases, as the web document may indicate it is ready while the internal server is still processing data.

3.2 Deriving a set of actions

Once the GUI's current state s is obtained, a set of available actions that a user can choose from in that specific state can be derived, which is suitable for most applications. To achieve this, a set of **actionable widgets** is first identified (see Figure 3.5). Actionable widgets are defined as widgets on which actions can be performed because they:

- are enabled
- are unblocked
- are not blocklisted or filtered by a tester (see Section 3.9)
- expect user interaction, i.e.:
  - widgets that are clickable (left or right mouse button);
  - widgets that are typable;
  - widgets that are draggable or slidable.

For example, considering a scenario in which state s includes a clickable button widget $\mathtt{b} \in W(s)$ that is enabled and unblocked[3]: if a tester did not blocklist or filter this widget, then this means there exists a possible action that can click on that button (click(b)). Likewise, for an actionable typeable text field widget $\mathtt{t} \in W(s)$, it means there exists a possible action that can click to focus and type into that text-field (type_into(t)).

To derive the actions that can be executed in a certain state s, TESTAR loops through the widget tree and collects those actionable widgets. To create executable actions from these actionable widgets, TESTAR converts them into implementations of the Action interface (see Figure 3.4). An execution scheme for button $\mathtt{b} \in W(s)$ from above is:

1. Determine the `position` on the screen that falls inside the widget;
2. Move the mouse cursor to that point;
3. Press the mouse down;
4. Release the mouse.

The movement of the cursor and pressing and releasing the mouse button each have their own implementation of Action, called MouseMove, MouseDown

[3]More specifically: `b.(Tags.Enabled) == true` and `b.(Tags.Blocked) == false`

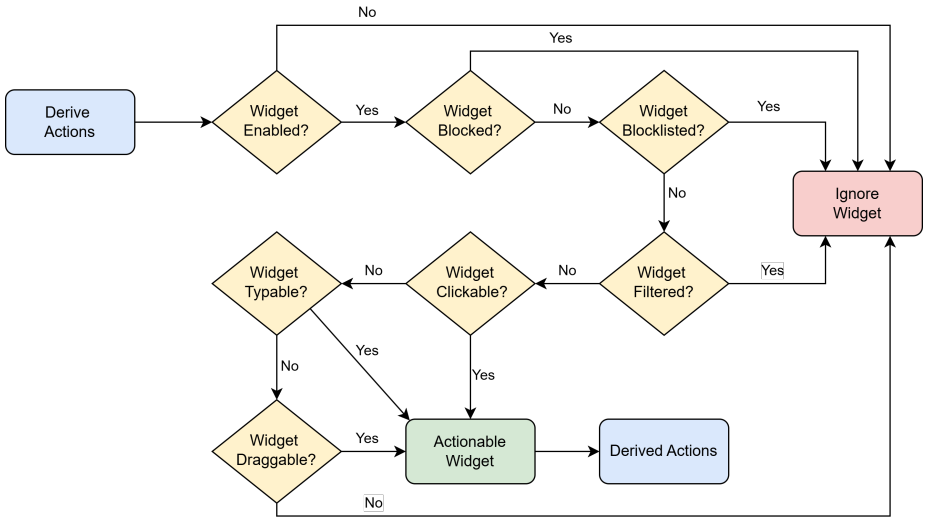


Figure 3.5: Deriving actions from actionable widgets.

and MouseUp, respectively. A fourth implementation of Action is introduced in the form of the CompoundAction class. This class aggregates sequences of actions into an Action. Example 3.2 shows how to create an action to click a button.

```
1 public Action leftClickAt(Position position) {
2     return new CompoundAction.Builder()
3         .add(new MouseMove(position), 1)  // Move mouse to position
4         .add(MouseDown, 0)                // Press mouse button
5         .add(MouseUp, 0)                  // Release mouse button
6         .build();
7 }
```

Example 3.2: Create action for button $\mathtt{b} \in W(s)$

When the current state is obtained using the Selenium WebDriver implementation, in addition to interacting with the browser through the actions described above, a set of actions representing JavaScript commands executable

through the Selenium WebDriver interface can also be derived (i.e., calling WebDriver.executeScript(**JScommand**)).

These JavaScript commands allow us to interact with the web elements that exist in the current web document using DOM API web methods and existing element attributes to find them or interact with the browser window tabs themselves. TESTAR predefines a couple of JavaScript commands to define useful actions during testing. These are defined internally as calls to WebDriver.executeScript:

- **WdCloseTabAction** to close a tab.
- **WdHistoryBackAction** to simulate a click on the history back button in a browser
- **WdSubmitAction** to simulate a click on a submit button in a detected web-form
- **WdAttributeAction** to find a web element by its unique identifier and write a value in the desired attribute using a pair (key, value).

It is possible to change or add new actions on the tester's need. As an example, consider **WdAttributeAction**. A web element can be searched and retrieved within a web document using one of its web attributes. With the focus on the desired web element, certain DOM API web methods enable reading or writing a value to one of the multiple attributes of the web element. This is defined in **WdAttributeAction** as follows:

```
1 public WdAttributeAction(String elementId, String key, String value) {
2     WebDriver.executeScript(
3         String.format(
4             "document.getElementById('%s').setAttribute('%s','%s');",
5             elementId, key, value));
6 }
```

Example 3.3: Create a custom WebDriver action using a JavaScript command and the **executeScript** interface

Another type of action being derived includes bringing the SUT to the foreground or terminating undesired processes. To achieve the first objective, native calls invoke the main window to the foreground. If this is not possible for any reason, keyboard commands such as **`Alt + Tab`** are utilised. To terminate undesired processes, the existing processes in the SUT's environment are constantly monitored after each action. Moreover, for web applications, it is necessary to ensure not only that the desktop browser remains in the foreground but also that the URL domain of the SUT being tested retains focus, preventing the exploration of undesired web pages.

The set of actions derived in state s is denoted as $A(s)$.

3.3 Select and execute one of these actions

In state s, a set of actions $A(s)$ is derived and made available for execution. One action, denoted as a, is then selected and executed. In TESTAR's default mode, this selection is performed randomly. Upon executing the action a, the system transitions to a new state s'. In this manner, test sequences are generated as follows:

$$s \rightarrow_a s' \rightarrow_{a'} s'' \rightarrow \dots$$

until some stopping condition holds. Such stopping conditions can be, for example, when a failure was found or when a configured number of actions have been selected and the test sequence has reached its predefined length. To achieve this, the number of sequences to generate (**`number of sequences`**) and the number of actions to select for creating each sequence (**`number of actions`**) can be defined during the startup of TESTAR.

3.4 Representation of States and Actions

A unique and stable identifier must be assigned to each state and action to facilitate their recognition and comparison. This can be achieved by using the

attribute or property values associated with each widget in the widget tree of a specific state s. A *concrete* identifier is obtained if all properties are used. However, it is not necessary to use all properties; instead, a subset can be selected to create *abstract* identifiers.

A *concrete state* encompasses all widgets and their properties, capturing the precise status of the SUT. In contrast, an *abstract state* refers to a high-level representation that simplifies this information by focusing on a relevant subset of properties.

To illustrate this concept of abstraction, consider that concrete actions can be *Press key 'q'* or *Press key 'w'* while both actions are represented abstractly as *Press key*. Hence, certain actions may be considered equivalent and can be executed interchangeably. In the case of pressing a key, the specific key that is pressed may not be important at a high level of abstraction. Similarly, an abstract state depends on the attributes selected from each widget.

When selecting properties for the identifier, it is important to ensure they are relatively stable. For example, a window's **title** is quite often not a stable value (opening new documents in a text editor will change the title of the main window), whereas its help text is less likely to change. However, the **role** is a more stable property.

To identify a GUI state s, all widgets $w \in W(s)$ are considered, and a subset ABS_PROP of stable properties is selected from the complete set of properties of all widgets on the screen. This subset ABS_PROP defines what is referred to as an *abstraction function* $P_{\text{ABS\_PROP}}(w, s) \subseteq P(w, s)$, such that:

$$P_{\text{ABS\_PROP}}(w, s) = \{w.p \mid w \in W(s) \wedge p \in \text{ABS\_PROP}\}$$

The abstraction function is configurable in the test settings of TESTAR. By default, ABS_PROP is defined as **role**, **title**, **position**, **enabled**.

A hash value generated from these properties is stored instead to manage the potentially large number of property values. TESTAR recursively calculates a unique hash for each widget based on the concatenation of the mentioned attributes. It then combines the hashes for the widgets and uses them to calculate the unique hash for the state. Of course, this could lead to collisions. However,

for the sake of simplicity, it is assumed that this scenario is unlikely and does not significantly impact the optimisation process.

The same approach can be applied to represent actions. However, each action type may have parameters. For example, a click action has two parameters: the button (e.g., left or right) and the clicking position (x and y coordinates). Action identifiers need to also take these parameters into account. The method of calculation is as follows: for an action identifier, for an action identifier, TESTAR uses the identifier of the current state and concatenates it with a hash generated from the details of the action. These details include the mouse cursor position, the key typed, and other relevant information. A unique hash is then computed based on this concatenation.

For example, to create a unique identifier for a button click, a combination of the button's property values can be used, such as its **role**, **title**, **help text**, or its **path** within the widget hierarchy. To create a unique identifier for a text field, if the action identifier incorporates the entered text, the action that types the text *foo* will have a different identifier from the action that types *boo*.

3.5 Evaluate the new states to find failures (oracles)

A test oracle is a mechanism that distinguishes between a passed or failed test case. As explained in the previous section, scriptless testing generates the test sequence one step at a time during execution. The test oracles verify each state visited. This means that TESTAR oracles assign verdicts to states, referred to as *online* or *on-the-fly state oracles*. Without specifying anything, TESTAR can detect the violation of general-purpose system requirements or implicit oracles, like those stating that the system should not:

- crash, i.e. an *unexpected close*,
- freeze, i.e. get in an *unresponsive state*,
- contain any *suspicious titles* in any of the GUI widgets. [4]

Suspicious titles can be easily specified using regular expressions, as shown in Example 3.4.

[4] In newer versions, these are referred to as "suspicious tags."

```
SuspiciousTitles = .*[eE]rror.*
                 |.*[eE]xception.*
```

Example 3.4: Regular expression for suspicious titles

When this oracle is active, each state s visited during the generation of the test sequence is checked to determine whether the patterns defined by the regular expression of the suspicious titles appear in the widgets composing $W(s)$. A good example from web testing could be defining the HTML error codes in suspicious titles to detect dead links that throw *404 Not Found* error.

TESTAR also allows the user to define more sophisticated application-specific test oracles programmatically in the SUT-specific TESTAR protocol in Java code. Considering an example that checks for a security vulnerability, the OWASP[5] lists a vulnerability for *Information exposure through query strings in url*. When sensitive data is passed to parameters in the URL, attackers can easily obtain sensitive information such as usernames, passwords, tokens (authX), database details, and other confidential data. This vulnerability cannot be resolved simply by using HTTPS; instead, sensitive data should be prevented from appearing in the URL. Example 3.5 shows an oracle capable of detecting these vulnerabilities.

In line 1, a variable **inputTextData** is defined to store all text entered into text fields during executing actions (lines 3-6) to create test sequences. The oracle (lines 10-14) checks in each state whether elements from **inputTextData** are exposed in the current URL of the SUT.

Besides the online state oracles, TESTAR can also interact with the process of desktop applications, listening to the buffers of its process in the *System output* and *Error output* of the operating system. This enables the tester also to define buffer oracles enable to find *suspicious output* coming from the processes, similar in the way that it checks *suspicious titles*. Moreover, the output of the processes is stored in logs for subsequent offline manual inspection to identify anomalies.

[5]Open Web Application Security Project, https://www.owasp.org

```
Set<String> inputTextData = new HashSet<>();

method executeAction(Action action){
    if (action.get(Tags.Role).equals("clickTypeInto")){
         // Save the inputted text into the set inputTextData
        inputTextData.add(action.get(Tags.Desc));
    }
}

method getVerdict(State state){
    for(String dataText : inputTextData){
        if(state.get(WebTags.Href).contains(dataText)){
            return new Verdict(Verdict.SEVERITY_WARNING,
            "Be careful with sensitive information and HTTP GET method");
        }
    }
}
```

Example 3.5: Programmatic Java oracle

3.6 Runtime execution and modes

The entry point of the TESTAR Java runtime process is the Main class. This class has access to the **`test.settings`** configurations file, defined by the tester. Besides settings like **`number of sequences`**, **`number of actions`** and **`Suspicious Titles`**, the testers can define their own specific TESTAR **protocol** class that needs to be used for testing. This can be specific for a SUT, a kind of test or just for the tester.

A TESTAR protocol is a Java class that is responsible for executing the different parts of the test sequence loop as depicted in Figure 3.1. The code in the protocol class gets compiled at runtime. The SUT-, test- or tester- specific TESTAR protocols are at the bottom of an inheritance tree as shown in Figure 3.6.

The Desktop and the WebdriverProtocol add a default implementation for specific platforms. Action filtering as it will be explained in Section 3.9 is done

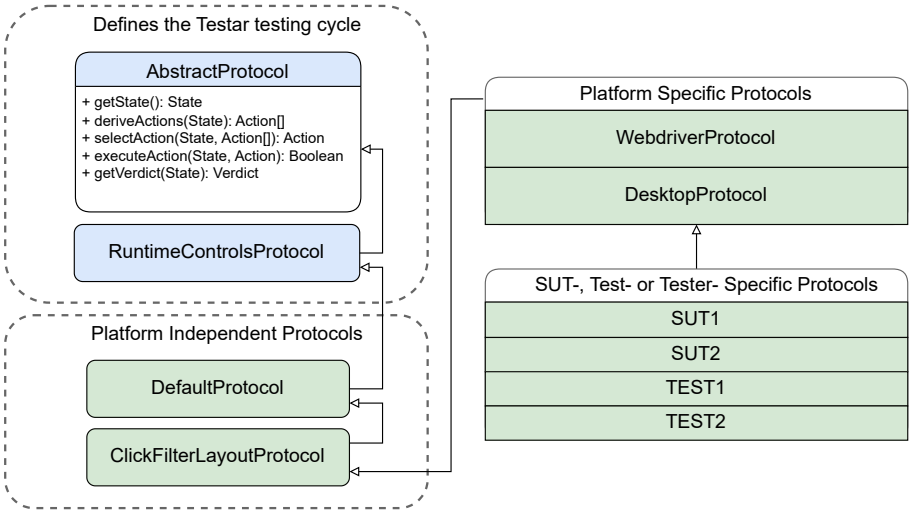


Figure 3.6: Layers of the different TESTAR protocols

by the The ClickFilterLayerProtocol class.

The DefaultProtocol class is the class that contains all the code that actually executes the test sequences. It implements the interface as defined in the AbstractProtocol class that contains methods for executing the different parts of the test sequence loop (conform Figure 3.1 and the previous four sections):

- `getState()` (from Section 3.1),
- `deriveActions()` (from Section 3.2),
- `selectAction()` and `executeAction()` (from Section 3.3),
- `getVerdict()` (from Section 3.5).

Finally, the RuntimeControlsProtocol class offers controls that allow for the manipulation of TESTAR's runtime modes during execution. There are currently four modes of runtime execution:

- The SPY mode can be used to inspect the widgets of the SUT and see all

the information that TESTAR can extract. In this mode, actions can be filtered (see Section 3.9).

- In GENERATE mode the test cycle depicted in Figure 3.1 is executed.
- The RECORD mode can be used to manually interact with the SUT and store the actions into test sequences.
- The REPLAY mode permits replaying an existing test sequence.

This flexible architecture allows the addition of new Action Selection Mechanisms (ASM) by implementing the ActionSelector interface (see Figure 3.7). The most used ASM is Random (*RND*) [118,119]: arbitrarily selecting one action out of all possible actions in the current state. Another known ASM is *Least executed actions* (LEA) [120], or the frequency-based algorithm [121] using Q-learning, selecting the least explored actions from the current state.

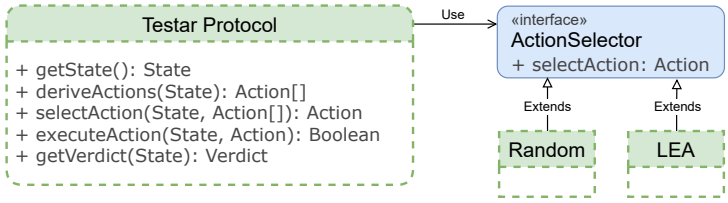


Figure 3.7: Extending TESTAR with different ASMs.

3.7 Test Results

As explained in previous sections, a TESTAR run results in a specified number of test sequences with a specified number of actions that have been executed. For each of the resulting sequences, the following information is saved in a directory with a name composed of a timestamp and the name of the SUT:

- Logs that include all the executed actions, the target widget and the different states of the test sequence, as well as a timestamp that can help synchronise results with other applications.

- Screenshot images that capture the GUI state after each action in a sequence. For this, the coordinates of the states and widgets obtained through the API are used.
- HTML reports to help users follow the flow of executed actions. They combine the API textual information and the visual screenshots to display the different sequences.
- Sequences replayable by TESTAR in REPLAY mode (`.testar` format). These sequences are classified in directories according to the final verdict obtained from the defined oracles (i.e. *unexpected close*, *unresponsive*, *suspicious titles*). These sequences consist of a Java object stream that saves the object information of states, actions and widgets.

All the results of a TESTAR run are saved in a directory with a name composed of a timestamp and the name of the SUT. An index log is created during the first TESTAR run and is updated with each sequence execution. This index is particularly useful for supporting the integration and synchronisation of TESTAR with other applications. Timestamps can be used to locate all TESTAR sequences by navigating to the directory with the corresponding timestamp (see Figure 3.8).

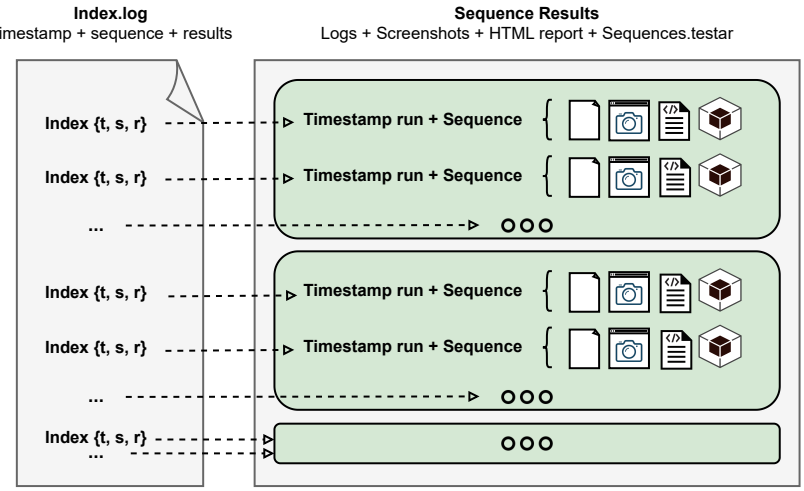


Figure 3.8: Output Structure for Test Results

3.8 Advanced Derive Actions

As explained in Section 3.2, after obtaining the GUI's current state, a set of actions can be derived, from which one will be selected. It is important to note that the larger the set of available actions, the greater the sequence space, which can increase the time required to search for crashes.

Ideally, the selection should be limited to a small set of actions most likely to expose faults. Therefore, the challenge is to keep the search space as small as possible while ensuring it is sufficiently large to find faults.

Deriving sensible actions: This strategy involves generating a set of *sensible* actions, ensuring that the actions are appropriate for the widgets on which they are executed: buttons should be clicked, scrollbars should be dragged and text boxes should be filled with text. Furthermore, the focus is on exercising only those widgets that are *enabled* and not *blocked*. For example, in a window that is blocked by a message box, it would not make sense to click on any widget behind the message box. Since the box blocks the input, it is unlikely that any event handling code (with potential faults) will be invoked. Putting more intelligence into action derivation will reduce the likelihood of selecting uninteresting actions during the action selection process.

Deriving top-level actions: Widgets at the top of the layout hierarchy are more likely to lead to actions that trigger state transitions. Elements such as menus or emerging windows within the SUT typically contain and are designed to facilitate the functional flow of the application. To favour top actions, TESTAR implements a prioritisation approach based on an internally defined property called *z-index*. The z-index of widgets presents their position in the stack of windows. The window with the highest z-index is on top. This gives the possibility of deriving actions from top-level widgets.

Deriving new actions: Another prioritisation approach for faster GUI exploration is comparing the available actions in the current state and the previous state to detect which are new. Suppose an action requires multiple steps, such as opening the File menu and selecting a menu item. In that case, this prioritisation increases the likelihood of triggering new actions after opening the File menu.

3.9 Filter Actions

Besides telling TESTAR the actions it can do, it is also important to tell what it should *not* do. Action filters can be defined for this purpose. Letting TESTAR randomly interact with widgets on a GUI could trigger *hazardous* operations like deleting or overwriting files, possibly damaging the operating system. A way to safeguard this is simply running test monkeys in a safe environment, like a virtual machine that can be easily recovered or a sandbox. Filtering actionable widgets remains helpful to ensure the focus is on actions contributing to testing the SUT. For example, actions that minimise/maximise a window, close the SUT or open a Help menu that goes to a website outside the SUT are not interesting for testing. Filtering those will reduce the search space and save time during the test execution.

Action filters can also define specific areas for scriptless testing, such as filtering different menu options or directing the tool to test a particular area of the SUT. Action filtering with TESTAR can be performed in three different ways, each of which will be discussed next.

Filtering with regular expressions. Similarly to using regular expressions for defining oracles to detect suspicious titles (see Example 3.4), regular expressions can also be used to filter widgets whose titles match a specified pattern. For example:

```
WidgetTitleFilter = .*[cC]lose.*
                  |.*[mM]inimi[zs]e.*
                  |.*[sS]ave.*
                  |.*[pP]rint.*
```

Example 3.6: Regular expression to filter widgets

When this filter is active, in each state s visited while generating the test sequence, the title property w.`title` of all widgets $w \in W(s)$ are matched against the regular expression. Actions on that widget will not be considered in case of a match. The `WidgetTitleFilter` mentioned above serves as a general-purpose

filter pattern that applies to almost all SUTs. It prevents actions such as closing or minimising the application under test, saving files that could lead to hazardous outcomes, and accessing the system's print menu for printing documents.

Using the click-filter. TESTAR click-filter functionality allows testers to filter widgets just by clicking on them through the GUI of the SUT during SPY-mode. The filtered widgets are stored in a blocklist, rendering them non-actionable, meaning no actions will be derived for them. Filtering of widgets can be undone in the same manner, even if the filtering was performed using a regular expression match.

Accurate filtering relies on the uniqueness of the abstract identifiers, as detailed in Section 3.2. Selecting the right level of abstraction and precision is important to guarantee uniqueness. For example, consider an OK button: if the **role** (i.e. button) and **title** (i.e. OK) properties are used for the abstract identifier, it would result in filtering all OK buttons across all the states of the SUT. Including the **path** property would make the filtering more precise, and the path of the button in the widget tree will most likely differ across different states.

Programmatic filtering. The third and most flexible way to filter actions is by programming the desired behaviour in the SUT-, test- or tester- specific TESTAR Java protocol. In some SUTs, the configured set of properties may not be sufficient for proper filtering of the existing widgets, for example, because they use a different set of accessibility properties. Then, programmatic filtering is the best option.

TESTAR allows customising all methods within its execution flow, enabling users to define specific action filtering based on the appropriate properties.

For example, in web applications, specific properties, such as **href**, **helpText**, or **class**, can be particularly useful for filtering widgets. In the *deriveActions()* function of a specific TESTAR protocol, actions can be filtered based on custom conditions. For instance, as demonstrated in Example 3.7, widgets with undesired URLs in their **href** tag property can be excluded from action generation.

```
for (widget: state)
    if (widget.get(WebTags.Href).contains("Undesired-URL"))
        continue; // skip this widget
    else
        // derive actions as defined
```

Example 3.7: Filtering widgets programmatically

3.9.1 Comparison of Scriptless GUI Testing Tools

This section provides a comparison between existing scriptless GUI testing tools, highlighting the main differences. The tools included in the comparison were selected based on the following criterion: they must be scriptless testing tools using dynamic analysis during automated GUI exploration.

Table 3.2 presents a detailed comparison of scriptless testing tools, summarising key aspects such as the implementation language, license, types of SUTs that can be tested with the tools, methods of ASMs during test sequence generation, models used or inferred during testing, oracles employed for fault detection, and actions considered as components for the sequences.

Most of the compared tools are implemented in Java. This predominance might be due to the well-established Java libraries for GUI handling and their ability to run across multiple platforms. Similarly, the majority of the tools are currently open source. The tools differ substantially in the types of SUT they target. Murphy, for instance, is specific to Windows applications, whereas GUI Driver and Augusto focus primarily on Java-based GUIs. By comparison, TESTAR provide broader support (Windows, Web and Java) with respect to the other tools. This flexibility better accommodates organisations and researchers working with heterogeneous technology stacks, as it removes the need to switch tools when testing different types of software.

Table 3.2: TESTAR comparison with other tools

Tool	Impl.	License	SUT Types	GUI Libraries	Action Selection	GUI Actions	Model Inference	Oracles
TESTAR	Java	OS	Windows, Web, Java (AWT, SWT, Swing, FX)	UIAutomation, WebDriver, Java Access Bridge	Random, programmable, model-based, Q-learning	Mouse clicks, text input, drag-and-drop, keyboard events	State graphs, configurable widget properties	Crashes, freezes, suspicious widget properties, programmable
Murphy [122]	Python	OS	Windows	UIAutomation, image recognition	Random, state model-based, programmable triggers	Mouse clicks, text input	State graphs, data values abstracted from states	Crashes, freezes, programmable
GUI Driver [123]	Java	NA	Java (AWT, SWT, Swing)	Jemmy Java library	Random, state model-based, user actions captured into model	Mouse clicks, text input	State graphs, actions define state abstraction, data in transitions	Crashes, freezes, exceptions, errors in system outputs
Crawljax/ ATUSA [124]	Java	OS	Web	WebDriver	K shortest paths algorithm	Click actions only	State-flow graph of DOM states	Client/server-side errors, dead clickables, inconsistent back-button
Webmate [125]	NA	Commercial	Web	WebDriver	Inferring state model, Dijkstra's shortest path	Mouse clicks, hovering, text input heuristics	Finite state automaton, multiple state abstraction options	Cross-browser differences, crashes, HTTP/JS errors, programmable
GUITAR [29]	Java	OS	Java JFC, Java SWT, Web, UNO (Open Office)	Java Accessibility, WebDriver, UNO Accessibility	Graph model-based, configurable triggers	Invoke GUI API-library events	GUI event flow graph	Crashes, StateVerifier, programmable
Augusto [126]	Java	OS (IBM Functional Tester)	Java (AWT, SWT, Swing)	IBM Functional Tester	Depth-first and model-based	Mouse clicks, text input, complex actions	GUI event flow graph, widget properties	Functional oracles, non-crashing faults
Auto-BlackTest [127]	Java, .NET	OS (IBM Functional Tester)	Java (AWT, SWT, Swing), .NET, Windows, Linux	IBM Functional Tester, Selenium	Random, RL, state model	Mouse clicks, text input, complex actions	GUI event flow graph, widget properties	Crashes, hangs, uncaught exceptions, assertion violations

Random exploration constitutes a baseline method, augmented in various tools by state-based models (GUI Driver, Murphy, Crawljax) and advanced algorithms (K shortest path in Crawljax and Dijkstra's in Webmate). TESTAR and Murphy further support programmable triggers for higher customizability. Moreover, TESTAR and AutoBlackTest use Reinforcement Learning for action selection, with a reward function designed to encourage exploration of less-frequent actions. In practice, this means each action gains a higher reward the less often it has been executed, aiming to steer the exploration towards rarely visited states.

While all tools include basic mouse clicks and text input capabilities, there is a variation in the complexity of supported actions. TESTAR, Murphy and AutoBlackTest incorporate dragging, dropping or keyboard events in addition to standard interactions. Furthermore, most tools construct some representation of the state, typically in the form of a state or event-flow graph. These representations capture transitions triggered by GUI events. While abstraction methods differ (for instance, Murphy and GUI Driver encode data values or abstract properties, whereas Webmate and TESTAR offer multiple levels of state abstraction), the consensus is clear: some form of model inference is incorporated to the scriptless GUI testing approach.

Finally, the oracle mechanisms (to detect faults) consistently check for critical issues such as crashes, exceptions and freezes. TESTAR also allow for programmable extensions, enabling verification of specific domain properties or suspicious GUI elements. Although every solution supports fundamental crash/error detection, the exact scope and sophistication of these oracles vary, leaving room for further research into more comprehensive, automated, or domain-tailored correctness checks.

3.10 Industrial case studies involving TESTAR

The successful transfer of academic results into industry is important. On the one hand, academic research activities should be guided more towards the challenges of industry and solutions to their immediate problems. On the other hand, industry practitioners should help academics validate their research results within a real

industrial context. Technology transfer has always been on the top priority list of the TESTAR project and remains to be. This section summarises collaboration projects that have been successfully executed over the years.

All studies are case-driven and executed following the Methodological Framework for Evaluating Testing Techniques and Tools (MFEST^3) described in [128]. The need for this framework emerged during the execution of the EU-funded project EvoTest (IST-33472, 2007-2009, [129]) and continued emerging during the EU-funded project FITTEST (ICT-257574, 2010-2013, [130]). The framework conforms to the well-known and general guidelines and checklist from case study research [131–134], but has been made specific for evaluating software testing treatments.

MFEST^3 outlines different scenarios for conducting and evaluating studies, with increasing dependence on the available information for comparison. **Scenario 1** consists of only a qualitative assessment, with insufficient information for direct comparison. In this scenario, the number of faults is unknown, error injection is not feasible, and no documentation exists to compare it with other testing techniques or establish a company baseline. Despite these limitations, measurements of effectiveness, efficiency, and subjective satisfaction are collected through semi-structured interviews. **Scenario 2** builds on Scenario 1 by incorporating some quantitative analysis, made possible through the availability of a company baseline for comparison. **Scenario 3** extends Scenario 1 or 2 by including a quantitative analysis of the **Fault Detection Rate (FDR)**, leveraging access to a known set of faults, whether injected or naturally occurring. **Scenario 4** includes the elements of Scenario 1 or 2 and adds a quantitative comparison between the test cases generated by the evaluated approach and those in an existing test suite. **Scenario 5** builds on Scenario 4 by incorporating the FDR of the different test suites. Two additional scenarios are described in [128], but these types of studies have not yet been conducted with TESTAR.

In [128], numerous metrics are defined to address research questions related to test effectiveness, efficiency, and subjective satisfaction. The metrics employed in TESTAR studies are listed and numbered in Table 3.3, facilitating cross-referencing with Table 3.4, which provides a summary of the executed case studies.

Table 3.3: Metrics from [128] used in the TESTAR studies.

Effectiveness	Efficiency	Subjective Satisfaction
1. Number of failures	1. Time needed to design the test suites.	1. Reaction cards
2. Code coverage	2. Time needed to run	2. Informal interview
3. Functional test coverage	3. Lines Of Code (LOC) for setup	3. Face questionnaires
4. Number of false positives	4. Time needed for post analysis	
5. Reproducibility		
6. Impact or severity of faults		

In Table 3.4, the *GUI testing*-column describes how GUI testing was done before the case started (M meaning manual, and CR meaning using Capture & Replay). The *scenario*-column refers to the scenarios from MFEST[3] described above. The *context/subject*-column mentions the project in which the study was carried out and indicates how many academics (**aca**) and how many industrialists (**ind**) participated. The numbers in the "effectiveness, efficiency, and subjective satisfaction"- columns correspond to those in Table 3.3. These industrial case studies covered different contexts, providing a comprehensive understanding of TESTAR's capabilities.

TESTAR's **learnability** varied depending on the context and prior expertise of the testers. Across studies, the early stages of using TESTAR, such as configuring its basic settings and employing predefined oracles (e.g., regular expressions for fault detection), were straightforward. However, challenges emerged when testers attempted to delve deeper into TESTAR's advanced functionalities. For instance, in the SOFTEAM study [55], testers required significant training to configure more sophisticated oracles and effective setup action sets.

Interestingly, the enthusiasm for learning TESTAR seemed to grow as testers gained more confidence in its capabilities. During the SOFTEAM study, testers reported a deeper understanding of TESTAR's potential after one month of hands-on training. However, they emphasised the need for more detailed manuals tailored to industrial testers without advanced programming skills. Similarly, the iterative development process in the Clave study [57] allowed testers to progressively improve their understanding of TESTAR's customisation options, creating more powerful and context-specific oracles.

Table 3.4: TESTAR Case Studies

SUT				Evaluation Study			Metrics			Results
Company	Platform	Language/LOC	GUI testing	Scenario	Context/Subjects	Publication	Effectiveness	Efficiency	Satisfaction	
Softeam (Large)	Web	PHP (2k)	M	5	FITTEST (2 aca, 2 ind)	[55]	1, 2, 5	1, 2	1, 2, 3	Automated tests generated by TESTAR were competitive with manual tests. With customization of action selection and oracles, subjects believed TESTAR could achieve better coverage and fault detection, saving manual testing time.
Pro-develop (SME)	Web	Java	M	1	TESTOMAT (1 aca, 3 ind)	[56]	1, 5	1, 2	2	Integration into CI required adjustments. TESTAR ran for 12 hours over 4 nightly builds (with random selection protocol and a configuration of 30 sequences of 200 actions), detecting 21 failure sequences caused by two faults.
Clavei (SME)	Windows Desktop	VB	M	1	SHIP (1 aca, 2 ind)	[57]	1, 5, 4	1, 2, 3	2	Setup took 26 hours, with 100 minutes needed for manual log inspection, reproduction and comprehension of errors. TESTAR detected 10 critical faults previously unknown.
Cap Gemini (Large) and Prorail (Large)	Web	Java (12k)	M	5	OU, UPV (1 aca, 3 ind)	[58]	1, 3, 6	1, 2	1, 2	TESTAR ran for 71 hours (192 sequences and 98.081 actions), finding 4 failures undetected by manual testing, with 80% functional coverage. Failures included: null pointer exception, functional fault, and concurrent modification error, all rated high severity. One failure was not reproducible.
Kuveyt Türk Bank (Large)	Web	Many (562k)	M	1	TESTOMAT (2 aca, 3 ind)	[59]	-	-	2	TESTAR's Webdriver protocol was extended to exclude extensions of resources, such as PDFs, and to restrict testing to whitelisted domains and URLs, avoiding external pages. The goal was to evaluate scriptless testing and reduce maintenance costs while improving test coverage.

SUT				Evaluation Study			Metrics			Results
Company	Platform	Language/LOC	GUI testing	Scenario	Context/Subjects	Publication	Effectiveness	Efficiency	Satisfaction	
Ponsse (Large)	Embedded Windows	VB (>1M)	M	1	TESTOMAT (1 aca, 3 ind)	[60]	1, 6	-	2	TESTAR integrated into CI, running tests against nightly builds, finding faults missed by manual GUI testing and scripted test automation.
Indenova (SME)	Web	-	M	1	SHIP, PERTEST (1 aca, 1 ind)	[61]	1	1, 2, 3	2	Setup required 35 lines of code and 10 minutes. Oracles did not require any lines of code, but just a regular expression with the list of unwanted localised words. TESTAR detected two issues during a one-hour test session.
B&M (SME)	Windows Desktop	Java (240k)	CR	1	FITTEST (1 aca, 2 ind)	[62]	1	1, 2	2	TESTAR revealed null pointer exceptions when minimizing the main editor, which was not functionally specified. The company saw TESTAR as complementary to their current practices for improving fault detection.
E-Dynamics	Web	JavaScript/React (616k)	M	3	iv4XR, IVVES (2 aca, 4 ind)	[63]	1, 2, 3, 6	1, 2, 4	1, 2	Compared scriptless (TESTAR) and scripted (Selenium) testing approaches. Both tools were complementary, with TESTAR excelling in GUI fault detection abd high event coverage, and Selenium in process faults. Selenium needs more time for creating test cases and TESTAR needs more time in evaluating test reports.
ING	Mobile (Android, iOS)	-	M	4	IVVES (4 aca, 3 ind)	[64]	2, 3	2	2	Outperformed other scriptless tools in coverage and matched scripted test automation, complementing existing scripts and covering additional code.

Regarding **effectiveness**, TESTAR consistently demonstrated its ability to explore diverse GUI paths and detect unique and valuable faults that other approaches, including manual testing and script testing tools like Selenium, did not detect. For example, in the E-Dynamics study [63], TESTAR uncovered issues that stemmed from unexpected application states and that were often less visible in traditional testing scenarios. These faults reflected the tool's strength in exploring paths that were less explored and identifying areas of the application that may not receive sufficient attention during manual or scripted tests.

Furthermore, TESTAR's complementary nature to other techniques was emphasised. While scripted approaches (e.g. Selenium or Espresso tests) excelled in verifying common use cases and process flows, TESTAR's exploratory nature enabled it to cover broader, less predictable aspects of the SUT. This complementarity underscores TESTAR's value in enhancing overall testing effectiveness when integrated into a diverse test suite.

The **efficiency** of TESTAR is closely tied to its ability to automate test execution with minimal manual intervention over time. In several studies, the initial setup and configuration required significant manual effort, particularly in defining oracles and refining action definitions. The Clave study [57] showed that this effort was comparable to the time required for crafting manual test cases. Importantly, once configured, TESTAR required minimal manual intervention during execution, making it particularly resource-efficient in identifying critical issues with minimal human effort.

The autonomous nature of TESTAR of executing tests unattended for extended periods of time makes it especially suitable for integration into continuous integration pipelines, where it can complement other testing tools and methods. These findings suggest that while the setup phase is labour-intensive, the long-term benefits of TESTAR outweigh the initial costs, especially in environments with frequent updates or large-scale GUIs.

Another insight of TESTAR's impact was its **subjective satisfaction** among testers. Tester feedback across studies revealed mixed but generally positive perceptions of TESTAR. Participants in the studies generally appreciated TESTAR's flexibility, with many recognising its potential to uncover faults that traditional methods

might overlook. However, concerns about its user-friendliness were recurrent, particularly regarding the complexity of its configuration process and the technical expertise required to utilise its advanced features fully. Feedback suggested that while TESTAR's exploratory approach was refreshing and effective, enhancements in its reporting mechanisms and usability could significantly improve tester satisfaction.

The industrial case studies collectively underscore the importance of **iterative learning** in adopting scriptless GUI testing tools like TESTAR. The studies also highlight the **complementary** role of TESTAR in augmenting traditional testing practices, particularly in autonomously detecting state failures and exploring untested paths without further manual intervention once finished the configuration phase.

To try to generalise the results of these case-based studies, an architectural analogy [66] can be used. This requires describing the architecture of the cases, i.e. components with interactions, such as the systems, the people and their roles. The architectural model for TESTAR (depicted in Figure 3.9) was developed based on the insights gained from the industrial case studies, ensuring that it reflects the practical realities of integrating TESTAR into diverse testing environments.

At the highest level, the testing environment with TESTAR can be viewed as an interconnected system of actors and components. The system context shows four key actors: **Clients**, **Developers**, and **Testers**. In some case studies, researchers can also fill the tester role.

Clients interact directly with the **SUT** by using it or providing requirements and feedback. If they encounter issues, they can report them to be processed by the **Bug Tracking System**. Developers implement and maintain the SUT, relying on bug reports from the Bug Tracking System to identify and address defects. Testers manage the **Test Environment** by planning, executing, and evaluating tests. They also configure and provide domain knowledge to TESTAR, interpret outputs, and report any defects discovered to the Bug Tracking System. This high-level view illustrates how TESTAR integrates into existing testing workflows, particularly the need for collaboration among these actors to continuously improve the SUT.

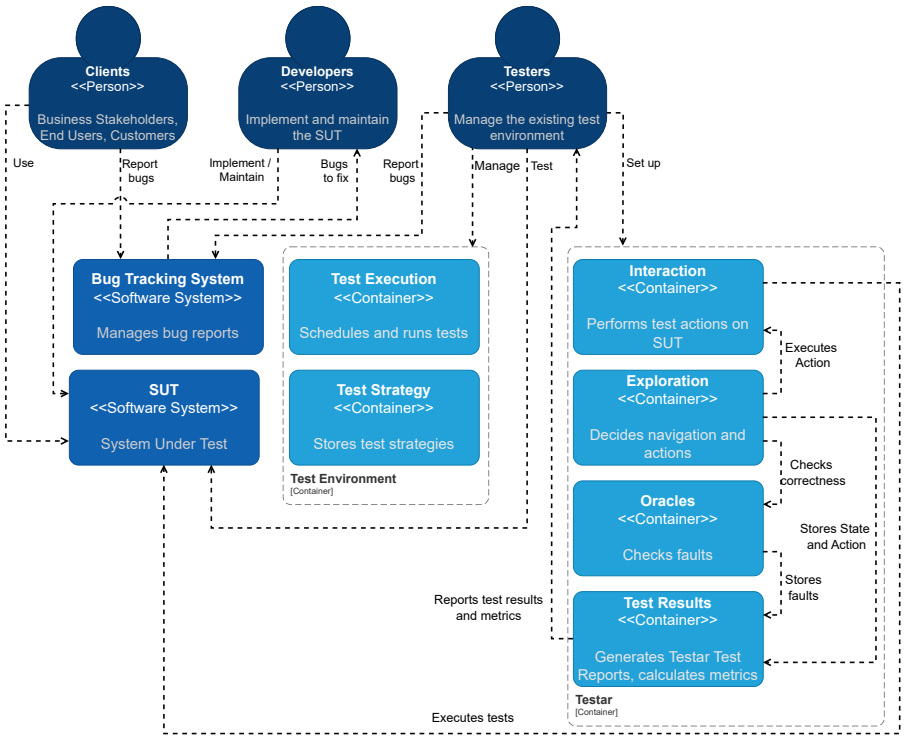


Figure 3.9: Similar components and interactions of the cases for generalisation through architectural analogy

Examining the architecture at a more detailed level reveals the specific containers comprising the Test Environment and TESTAR. The existing Test Environment is composed of two main containers that structure how tests are defined and executed. **Test Strategy** defines testing methodologies and strategies, guiding manual and automated tests. **Test Execution** schedules and runs tests.

Within TESTAR, four main containers handle different aspects of testing. The **Interaction** container manages how TESTAR performs test actions on the SUT. The **Exploration** container decides the navigation paths and actions taken during testing. The **Oracles** evaluates the SUT's state against fault-detection rules, helping identify unexpected behaviour. The **Test Results** container compiles reports and

calculates metrics. These findings are shared with the testers, who will inform the Bug Tracker System if any new bug is encountered. Testers might also use the test results to optimise TESTAR in future iterations, ensuring continuous improvement. These components enabled TESTAR to detect faults related to unexpected application states. This container view shows how the test artefacts flow between components and how TESTAR's automated testing capabilities complement the existing test infrastructure.

These industrial case studies highlight the practical applicability and effectiveness of TESTAR in diverse real-world settings. However, a crucial aspect of these studies was the process of setting up TESTAR for the SUT. Establishing this process in a structured and systematic manner ensures that the tool can be effectively integrated into existing testing workflows while minimising overhead.

Figure 3.10 captures TESTAR's **iterative learning** process by breaking it down into four key phases: Planning, Implementation, Testing, and Evaluation. These phases provide a generic framework that can be tailored to suit the unique requirements of each industrial case study. This iterative approach underscores the gradual learning curve where testers require ongoing refinement and feedback to achieve optimal results. The connection between TESTAR's test artefacts and the test environment, as shown in the container diagram, supports the iterative learning process, allowing testers to integrate lessons learned into future test sessions.

Planning Phase. The first phase involves setting up the technical environment, including the installation and configuration of TESTAR and the necessary components to interact with the SUT. Key configurations included defining the SUT by specifying the execution method (e.g., a URL for web applications or an executable path for desktop applications), setting up any login or initialisation procedures, and configuring the environment such as browser settings or operating system details (e.g., Windows). Additionally, during this phase, the tester must anticipate and define potential faults that the system might encounter.

Implementation Phase. In the second phase, the focus shifts towards implementing the configurations and customisations necessary for TESTAR to carry out the tests effectively.

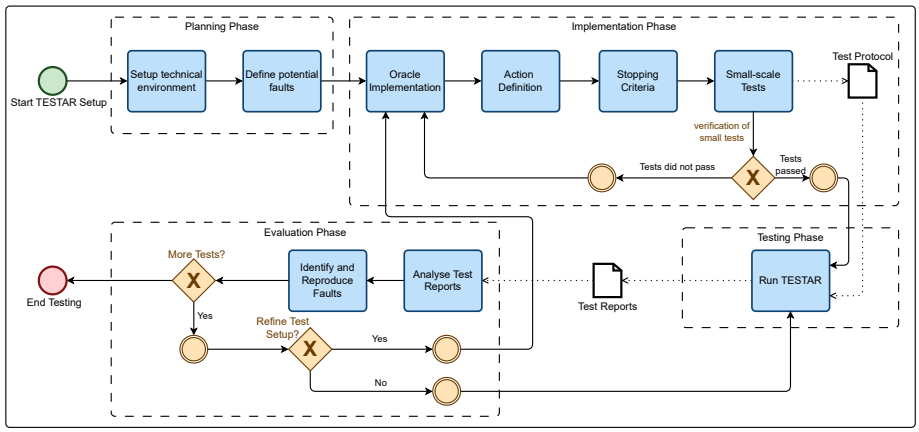


Figure 3.10: Generic Process for setting up TESTAR for automated scriptless testing.

First, *Oracle Implementation* is performed to ensure TESTAR can detect the errors defined in the planning phase. This involves setting up oracles that monitor the application's behaviour and identify when a failure occurs. For example, TESTAR can be configured to check for faults in the browser's console log or to monitor the response time of the web pages.

Next, the *Action Definition Implementation* involves refining the actions that TESTAR could execute. Defining these actions helps TESTAR explore the application and simulate user behaviour during testing. Additionally, the *Stopping Criteria* are set to manage the limit of each test run, like the maximum number of interactions.

Finally, the *Testing Configuration Sub-phase* is performed. Before proceeding to full test execution, small-scale tests should be run to validate that the configurations made during the implementation phase behave as expected. These tests ensure that the oracles, actions, and stopping criteria are functioning correctly.

The tester can ensure the basic setup is correct by running these small tests before proceeding to a more comprehensive test run. If any configuration issues are detected, they can be resolved during this sub-phase, saving time and reducing the risk of faulty test executions during the full-scale run.

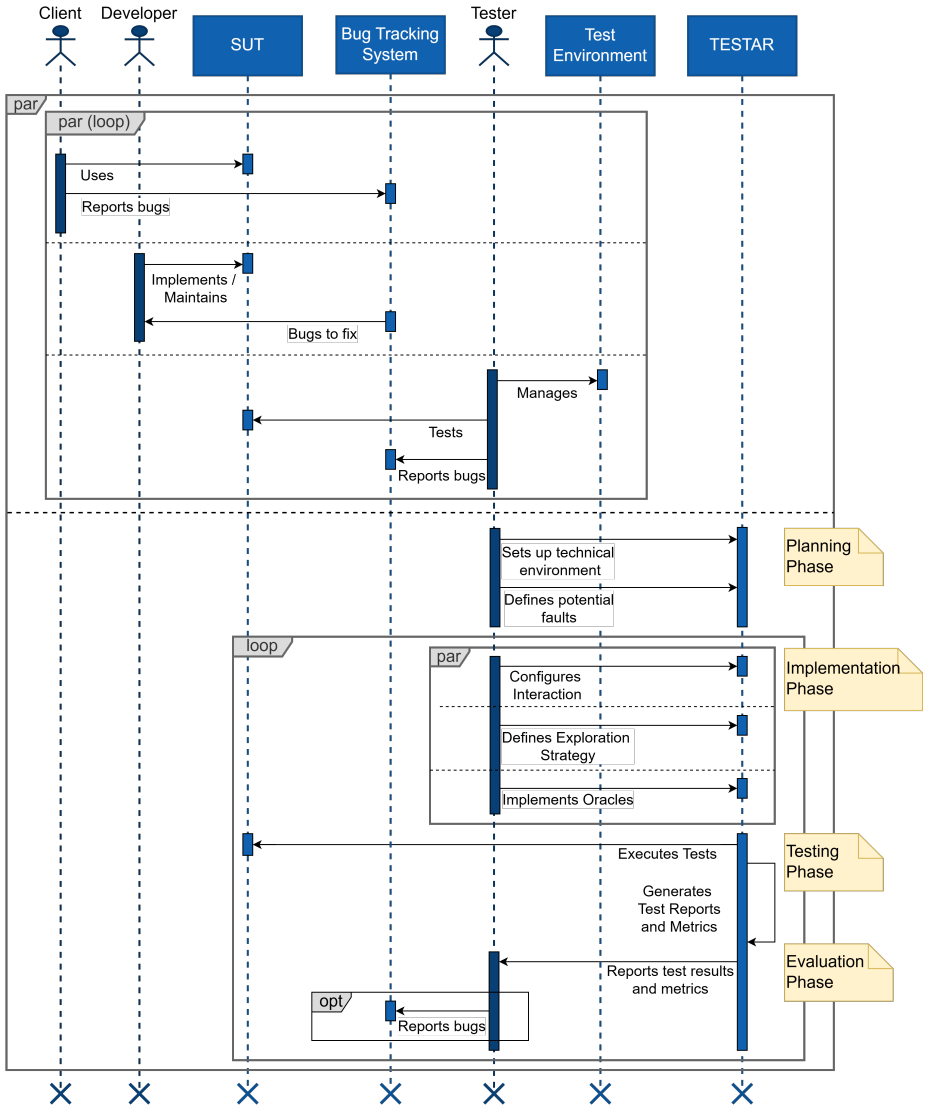


Figure 3.11: Iterative use of TESTAR in an industrial testing workflow.

Testing Phase. With the configurations complete, the third phase runs the test process. In this phase, TESTAR autonomously executes the defined actions and monitors the web application for faults. This phase may be repeated several times with different configurations to cover various test cases.

Evaluation Phase. After executing the test process, the Evaluation phase focuses on analysing TESTAR's reports. These reports provide a detailed breakdown of the actions performed, the system's responses, and any detected faults.

The tester then investigates the most severe issues, reproduces problematic sequences, and refines the test configuration based on the findings. This iterative process ensures the test environment evolves with each run, improving its coverage and fault detection capabilities in subsequent iterations.

The sequence diagram (Figure 3.11) provides a structured, time-ordered view of how TESTAR integrates into typical industrial workflows, encapsulating its interaction with the SUT, the Bug Tracking System, and the Test Environment. This representation builds on the architectural analogy and the generic process for setting up TESTAR. All these systems interact in parallel and over repeated cycles as part of modern development practices.

Several key insights from the industrial case studies are highlighted in the sequence diagram. It emphasises the iterative learning process where TESTAR undergoes continuous refinement with each execution cycle. These industrial studies demonstrated how repeated iterations improved oracles and exploratory behaviour, making TESTAR progressively more effective.

The diagram also showcases how TESTAR automatically executes tests by continuously interacting with the SUT, allowing bug tracking and software improvements to occur in parallel. This aligns with modern software development methodologies, particularly in continuous integration environments where automated testing runs alongside development activities.

These industrial case studies collectively illustrate the potential of TESTAR as a scriptless GUI testing tool capable of enhancing traditional testing approaches. While initial setup and configuration require some effort, the tool's ability to autonomously explore diverse application states, detect critical faults, and complement scripted testing approaches demonstrates its long-term value. The insights

gained from these studies highlight not only the challenges of adopting TESTAR but also its strengths in scalability, fault detection, and integration into iterative and continuous testing workflows.

3.11 Conclusions

This chapter provides a comprehensive overview of TESTAR, an automated scriptless testing tool designed to explore the graphical user interface (GUI) of a system under test (SUT). The tool's key components and workflow were described, including state identification, action derivation, action selection, and the application of oracles for fault detection. With its modular architecture and flexible configuration, TESTAR delivers an effective solution for GUI-based software testing across diverse environments.

The practical relevance of TESTAR was demonstrated through industrial case studies, highlighting its ability to complement existing testing practices and uncover edge failures. Additionally, the structured setup process presented in Section 3.10 emphasises how TESTAR can be systematically integrated into various testing workflows, making it suitable for continuous integration and regression testing scenarios.

However, the industrial case studies also reveal challenges associated with adopting TESTAR. These challenges served as motivation for creating the architectural analogy and the iterative process for setting up TESTAR, ensuring that the tool can be more effectively deployed in different industrial contexts.

4

Inferring state models with TESTAR

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

Edsger W. Dijkstra, "The Humble Programmer"

This chapter presents an extension to TESTAR that enables state model inference during GUI exploration. This feature allows the creation of a map of the SUT navigation and actions during testing. Initial steps toward this functionality were described in previous work [10, 135], where models were used for offline oracles (i.e., after testing) that consisted of querying the model for accessibility information. However, the current work expands these capabilities for multiple purposes:

- Enhancing TESTAR's Action Selection Mechanisms (ASMs) during and after the model inference
- Defining offline oracles to compare models between different releases or versions of a SUT
- Providing visual reference models for testers and users

- Supporting model-based GUI testing (MBGT) tools [136].

The integration of model inference capabilities is particularly valuable for MBGT, which has seen limited adoption due to the expertise and effort required for model creation [137]. If even initial models can be inferred, these problems may be (partially) solved.

While several approaches exist for inferring models during automated GUI exploration (described in Section 4.1), most are academic prototypes or abandoned open-source projects. State space explosion is still an open challenge for the inference of state-based models through GUI. Most programs with a GUI have a large number of possible states, and to make the size of the models manageable, some information has to be abstracted away. It is challenging to define a suitable level of abstraction and find an equilibrium between the necessary expressiveness of the extracted models and the computational complexity [138].

Abstracting away too much information might make a model unsuitable for its purpose (e.g., ASM, MBGT, oracles) and lose opportunities to discover faults and changes between versions. Abstracting away too little information might result in state space explosion, making the model less suitable for its purpose. Most of the existing related work does not explain in sufficient detail how they deal with the abstraction, raising questions about whether their solutions are generally applicable or simply tailored for the applications used in validation.

The main contributions of the research that is contained in this chapter are:

- A description of the model inference functionality implemented into TESTAR.
- A novel algorithm for ASM based on the inferred model.
- An initial validation of the test effectiveness of the new ASM regarding code coverage and reached states.
- An initial validation of the approach by experimenting with how various abstraction mechanisms affect the inferred models.

The rest of the chapter is organised as follows. Section 4.1 presents related work. Section 4.2 presents the approach used to infer state models, and their

application for action selection and experiments on the test effectiveness of the implemented ASM. Section 4.4.2 describes the experimentation to find out how various abstraction mechanisms affect the inferred models. Section 4.5 analyses the findings and challenges. Finally, Section 4.6 presents the main conclusions.

4.1 Related work on Model-based GUI testing

As mentioned in Chapter 1, Model-based GUI testing (MBGT) [136,139,140] generates test cases from a model. MBGT approaches require modelling the GUI and its expected behaviour on a higher level of abstraction than the GUI itself. The modelling language should be understandable by a tool that uses it to generate tests automatically.

An advantage of this type of testing is that it is possible to precisely specify the exact test specifications that a GUI should conform to. Another advantage is that when the GUI changes, the test scripts do not have to be manually updated. Instead, the model is updated, and the scripts/tests are generated again.

However, the main disadvantages are that MBGT approaches require a deep knowledge of the application domain and expert knowledge of formal modelling methods and languages to manually create a model of the GUI. Modelling also requires quite a lot of time and effort.

To address these limitations, several approaches for automated GUI model inference, also referred to as GUI ripping [37] or GUI reverse engineering [38], have emerged. These can be categorised into three main types:

1. **Static Analysis** uses the program's source code to infer a model of the GUI [27,28]. Static techniques concentrate only on the structure of the GUI, not taking the runtime behaviour of the GUI into consideration in the model.

2. **Dynamic Analysis** approaches analyse the GUI while the system is running [31]. APIs or libraries are used to automatically explore the GUI and get access to all the GUI elements in a specific state of the application. To create a model, these tools can recognise whether the application is in a state that the tool has already visited before or whether the state is being

visited for the first time. Examples of tools using model inference through dynamic analysis are GUITAR [29], GUI Driver [30], Crawljax [32], Extended Ripper [141], GuiTam [142] and Murphy Tools [31].

3. **Hybrid Approaches** combines static and dynamic techniques [33–35], leveraging benefits of both techniques.

A critical challenge across all approaches is determining a suitable level of abstraction for the model inference that ensures that the model is useful for its purpose (e.g., ASMs, offline oracles, or visualisation of testing). Most related work does not sufficiently explain how they deal with abstraction. This chapter presents the first attempt to research how the abstraction level affects the results when using the models.

4.2 State model inference for TESTAR

TESTAR's operational flow was described in Chapter 3. When the SUT has *started*, TESTAR captures the current *state of the GUI* using APIs like Windows Automation API (WUIA) (for desktop), Selenium Web Driver (for web), or the Java access bridge (for Swing). This (concrete) state consists of *all* the properties (that are available through the API) of all the widgets that are part of the GUI.

Subsequently, to *derive* the actions that it is able to perform in that state, it cycles through all the widgets and adds all possible actions associated with the widgets to a pool. Sometimes, if the SUT includes custom widgets and the API does not detect all the widget attributes, the user must provide TESTAR with some extra configuration to correctly detect all the available actions.

From this action pool, one is *selected* by the ASM of TESTAR. After the action has been *executed* and the GUI has reached a new state, TESTAR will again capture the new state and derive, select, and execute an action. This process repeats until the specified stop criterion is reached (e.g., the test sequence length or the occurrence of an error condition).

State abstraction[1] is an important facet of scriptless GUI testing. TESTAR has

[1] "Abstraction is the elimination of the irrelevant and the amplification of the essential." by Robert

an implementation to calculate state identifiers based on hashes over a selected set of widget attributes. This selected set defines the abstraction level, which determines the number of different states TESTAR will distinguish. This can evidently influence test effectiveness and is related to the equilibrium explained previously. Experiments were conducted to gather evidence about the suitable set of widget attributes for state abstraction as described in Section 4.4.2.

TESTAR uses dynamic analysis techniques to infer a model. The flow for capturing the state model is depicted in Figure 4.1, which extends Figure 3.1 from the previous chapter. The state of the SUT is constantly saved in the OrientDB **graph database**, together with available actions and the executed action. As explained later in this section, the state model can be **queried** by an ASM, but also by a human, an offline oracle, or other MBGT approaches.

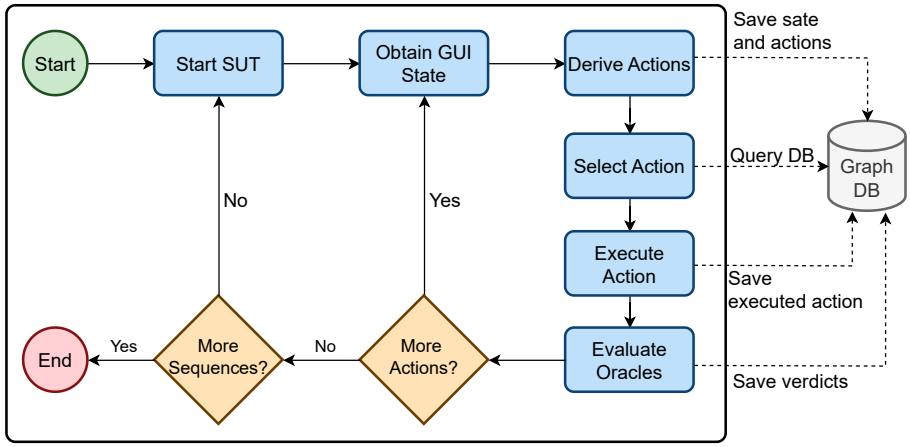


Figure 4.1: TESTAR testing cycle including model inference

As indicated, the model will be built incrementally with subsequent TESTAR runs. All states (concrete and abstract) visited during a run are stored in the database. For analysis and reporting, the structure of the inferred model is divided into three layers, as shown in Figure 4.2.

The **top layer** is an *abstract state model*. It allows ASMs to use the model

C. Martin (Agile Principles, Patterns, and Practices in C#)

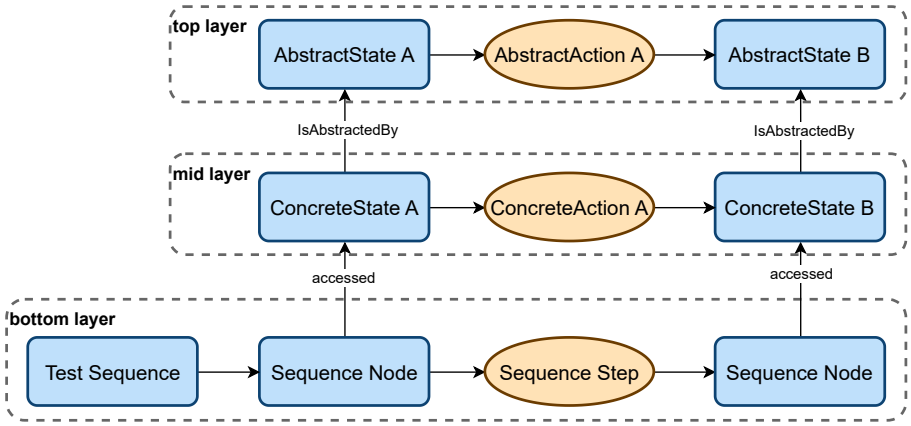


Figure 4.2: Layered design of the state model

for action selection or end users to analyse the behaviour of the SUT. Creating the abstract model requires identifying unique states at a *suitable* abstraction level. As indicated, this means trying to avoid state space explosion while, simultaneously, not losing the purposefulness of the model. Overly abstract states can introduce non-determinism in the inferred model.

The **mid layer** is the *concrete state model*. This model contains all the information that can be extracted through the APIs used by TESTAR. The concrete state model will contain too many states to drive the execution of TESTAR or serve as a visual model for humans. It will serve as information storage, e.g., when a specific part of the abstract model requires deeper analysis. Each concrete state of this layer will be linked to an abstract one in the top layer, and each action will be linked to an abstract transition.

The **bottom layer** is the *management layer*, whose purpose will be to record meta-information about the executed test sequences. Where the abstract and concrete layers describe the SUT, the management layer represents the execution of the tests in TESTAR. The individual test sequences will be linked to the concrete states and actions of the middle layer.

Figure 4.3 shows an example of the layered model, where the SUT was extremely simple (only three abstract and three concrete states). The management

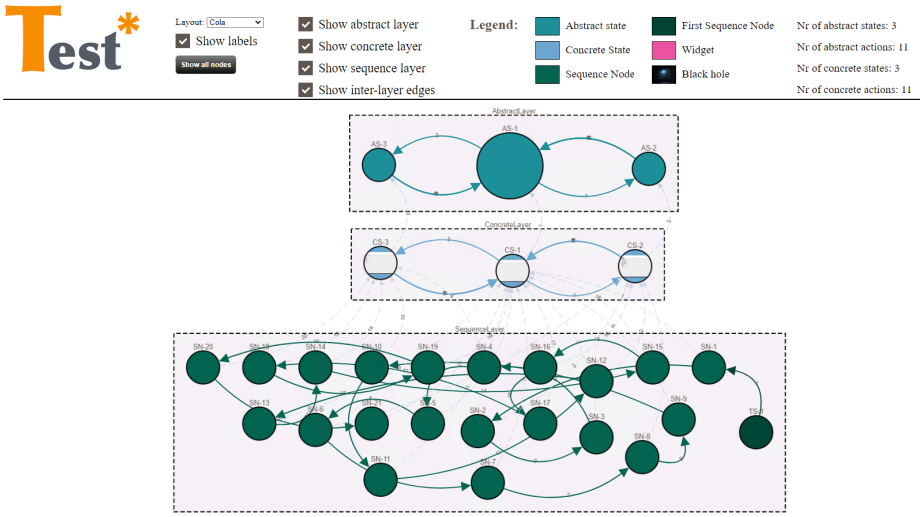


Figure 4.3: Visualization of an example model inferred by TESTAR

layer has information about the exact sequence generated by TESTAR. During the model inference, when TESTAR arrives at a new state and discovers actions that have not been executed before, *"BlackHole"* state is used as their destination to mark unvisited actions. When a previously unvisited action is visited and TESTAR observes the SUT behaviour, the destination of the executed abstract action is updated with the observed abstract state.

TESTAR was extended with a new ASM (ASM_*statemodel*). The algorithm prioritises actions that have not yet been visited and can be found in Algorithm 1. The goal is to select a new action when in state *s*. It uses the *State_Model* and maintains a *path* of actions that leads to a specific unvisited action it wants to prioritise. If a *path* has been previously identified (i.e., *path* is not empty line 1), then the ASM selects the next action on that path. If the *path* is empty[2], the ASM will try to find an unvisited action. It does so by searching (in BFS[3] order) for *unvisitedActions* (line 8) from all the states that are reachable from *s* in the state model (line 4). Since *s* is reachable from *s* in 0 steps, *s* itself is the first state

[2]Because TESTAR has initialised the algorithm or the previous path was already completed.
[3]Breadth-First Search (BFS) explores all neighbours of a node before proceeding to the next level

Algorithm 1 ASM_*statemodel*: Select an Unvisited Action

```
Input: s                                                    ▷ Current state of the SUT
Input: State_Model                                          ▷ The inferred state model
Input: path                                                 ▷ Path to an unvisited action (if any)
 1: if path ≠ ∅ then                                        ▷ Follow the existing path
 2:     a ← path.pop()                                      ▷ Next action in the path
 3: else                                                    ▷ Create a new path to an unvisited action
 4:     reachableStates ← getReachableStatesWithBFS(s, State_Model)
 5:     unvisitedActions ← ∅
 6:     while unvisitedActions = ∅ and reachableStates ≠ ∅ do
 7:         s' ← reachableStates.pop()
 8:         unvisitedActions ← getActions(State_Model, s', unvisited)
 9:     if unvisitedActions ≠ ∅ then                       ▷ Found unvisited actions with BFS
10:         ua ← selectRandom(unvisitedActions)            ▷ Select a random unvisited action
11:         path ← pathToAction(ua)                        ▷ Determine path from s to ua
12:         a ← path.pop()                                 ▷ Next action in s towards ua
13:     else                                               ▷ No unvisited actions found
14:         availableActions ← getActions(State_Model, s, all)   ▷ All available actions in s
15:         a ← selectRandom(availableActions)
16: return a
```

checked for unvisited actions (line 7). If unvisited actions are found, it randomly selects one (*ua*, line 10) and updates the *path* to the state where that action can be found (line 11). Then, it selects the first action that leads towards that action (line 12). If no unvisited actions are found, the ASM just randomly selects an action from those available in state *s* (line 15).

To integrate this new ASM, TESTAR's flexible architecture (Figure 3.7) discussed in Section 3.6 was extended by implementing the ActionSelector interface, as depicted in Figure 4.4.

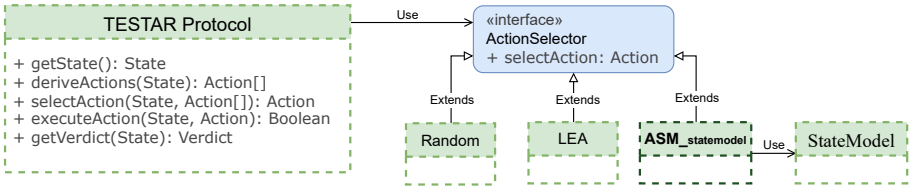


Figure 4.4: Extending TESTAR with ASM_*statemodel*

4.3 Experimental Design

Building on the discussed challenges of state model abstraction, two key aspects of automated model inference are researched:

RQ1: How do different levels of abstraction affect automated GUI exploration of ASM_*statemodel* compared to random selection?

This question examines the practical impact of model abstraction on testing effectiveness.

RQ2: Which widget attributes contribute to generating deterministic models in state abstraction?

This question addresses the fundamental challenge of creating reliable models through automated inference.

The experimental design encompasses two distinct studies aligned with the research questions. Each study uses a different subject system to investigate specific aspects of model inference and the effectiveness of GUI testing. Figure 4.5 presents an overview of the experimental design.

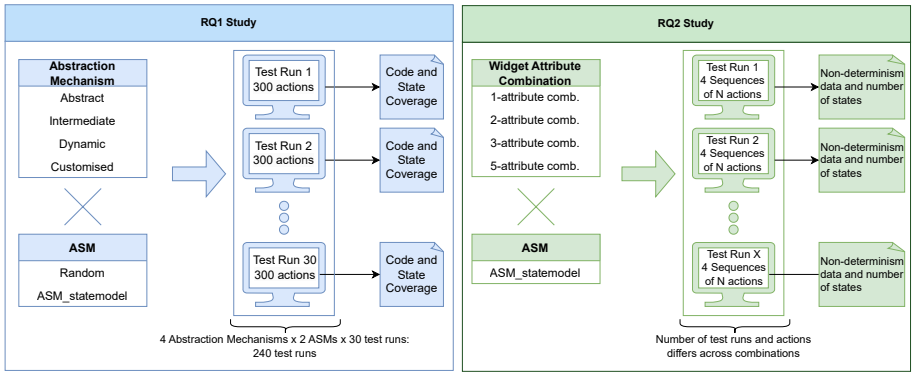


Figure 4.5: Overall experimental design showing separate studies for RQ1 and RQ2

4.3.1 Subject SUTs

Rachota and Windows Notepad were selected as the subject systems, each serving different experimental purposes. Rachota, an open-source Java Swing time-

tracking application, provides an ideal platform for evaluating testing effectiveness due to its accessible source code and measurable coverage metrics. Table 4.1 presents an overview of Rachota's characteristics.

Metrics	Rachota
Java Classes	52
Methods	934
LLOC	2722
Classes incl. Inner classes	327

Table 4.1: Overview of Rachota

For the second study, Windows Notepad (version 1909, OS Build 18363.535) was explicitly selected due to its rich set of widget attributes accessible through the Windows Automation API, providing over 140 attributes to choose from. That offers more choice compared to the six attributes of the Java Bridge from Table 4.2. Notepad presents common GUI patterns, including menus, dialogues, and text input areas, while offering both static and dynamic interface elements. This combination makes it particularly suitable for investigating model determinism and attribute selection impacts.

4.3.2 Independent and Dependent Variables

The independent and dependent variables were defined as follows to address each research question.

4.3.2.1 RQ1 Study

The first study examines how different abstraction levels affect GUI exploration effectiveness. The variables used in this study are summarised below.
The **Independent Variables** are defined as follows:

- *Abstraction Level for State Identification:* Different abstraction levels are defined by selecting attributes from those available in the Java Access Bridge API (see Table 4.2). Four abstraction levels were investigated:

1. **`Abstract`**: ControlType (cf. was defined in [135])
2. **`Intermediate`**: ControlType, Path
3. **`Dynamic`**: ControlType, Path, Title (including the dynamic attribute Title)
4. **`Customised Abstraction`**: ControlType, Path, HelpText, IsEnabled (this one was customised for Rachota following the impacts described in Table 4.2)

- *Action Selection Method*: The new ASM_*statemodel* algorithm is compared to the baseline ASM_*random* approach.
- *Test Run Parameters:* Each test run contains one sequence of 300 actions, which is enough [143] to show the differences between ASMs. Each test run was repeated 30 times to account for randomness [144].

Attribute	API	Impact on Abstract Representation
Title	**`name`**	Visual name of the widget. In Rachota, this is dynamic because widgets update the current time.
HelpText	**`description`**	Tooltip or help text of the widget. Static attribute in Rachota.
ControlType	**`role`**	Role of the widget. It may fail to distinguish elements, causing non-determinism.
IsEnabled	**`states`**	Indicates whether the widget is enabled or disabled.
Boundary	**`rect`**	Pixel coordinates of the widget's position. Too concrete; one-pixel changes result in a new state.
Path	**`childrenCount`** + **`parentIndex`**	Position in the widget tree. Useful for distinguishing states by widget tree structure.

Table 4.2: Java Access Bridge properties and their impact on state abstraction in Rachota.

The **Dependent Variables** are defined as follows:

- *Code Coverage:* Measured as both instruction and branch coverage using Jacoco [145].

- *Abstract State Coverage:* The number of abstract states visited in the state model during testing.

- *Concrete State Coverage:* The number of concrete states visited in the state model during testing.

Code coverage and state discovery metrics are collected after each executed action. The open-source application Rachota [146] is used as the SUT to measure these metrics effectively for the experiments.

4.3.2.2 RQ2 Study

This study aims to select a suitable subset of widget attributes for state abstraction that generates deterministic models without causing state explosion. The following independent and dependent variables were defined.
The **Independent Variables** are defined as follows:

- *Widget Attribute Combination:* The primary independent variable is the combination of widget attributes used for state abstraction. A systematic evaluation is conducted on single-attribute configurations from the available Windows Automation API properties, followed by two-attribute combinations, three-attribute combinations using the top performers, five-attribute combinations, and extended pattern combinations.

- *Action Selection Mechanism:* The ASM_$statemodel$ algorithm (from Algorithm 1).

- *Test Sequence Length:* Different test sequence lengths were employed to evaluate model behaviour across different temporal scales. These configurations comprised two lengths (50 and 100 per test sequence). To analyse the effect of abstraction levels on the number of states created, longer test runs of 5000 actions are also performed.

The **Dependent Variables** are defined as follows:

- *Steps Before Non-Determinism;* Number of actions executed before encountering the first non-deterministic state.
- *Model Characteristics:* Includes the count of abstract states and concrete states generated during testing.
- *Sources of Non-Determinism:* Analyses dynamic content changes and history-dependent behaviours contributing to non-determinism.

4.4 Results

This section presents the results of the experimental evaluation of TESTAR's state model inference capabilities. The results encompass quantitative metrics, including code coverage and state counts, and qualitative analysis of model characteristics and non-determinism sources.

4.4.1 RQ1: Impact of abstraction on GUI exploration

Figure 4.6 shows the results of the code coverage measurements across different abstraction levels. The ASM_*statemodel* consistently outperformed ASM_*random*, even with a less suitable abstraction. This means that model-based ASMs are a promising way to improve the effectiveness of scriptless testing.

The coverage data shows that the level of abstraction affects the GUI exploration performance of the ASM_*statemodel*. Having too high or too low level of abstraction negatively impacts the performance. The customised abstraction level (ControlType, Path, HelpText, IsEnabled) achieved the highest code coverage, followed by the intermediate configuration. However, the best random sequences can achieve coverage similar to the average coverage provided by abstract, dynamic, and intermediate abstract mechanisms. This highlights the significance of customising a suitable abstraction level to enhance effective exploration compared to random but also underscores the inherent capabilities of random exploration itself.

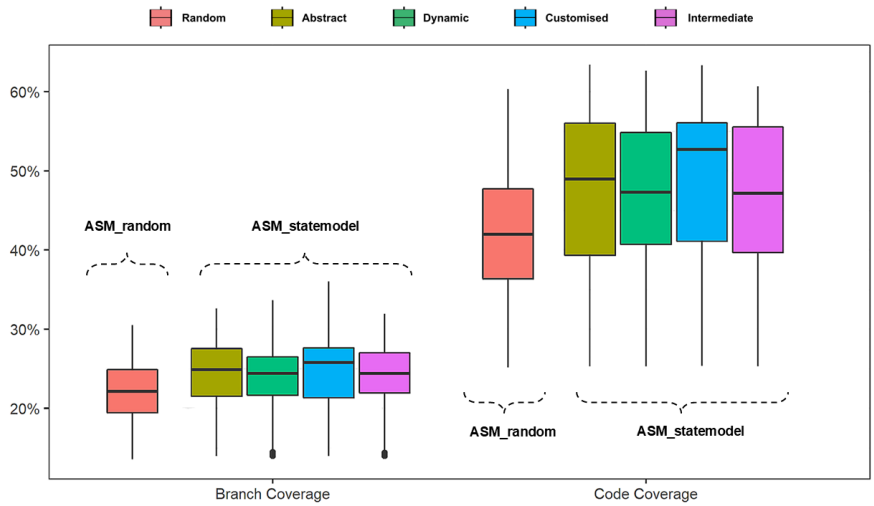


Figure 4.6: The code coverage that was reached when comparing ASM_*random* with 4 different abstraction levels of the ASM_*statemodel*

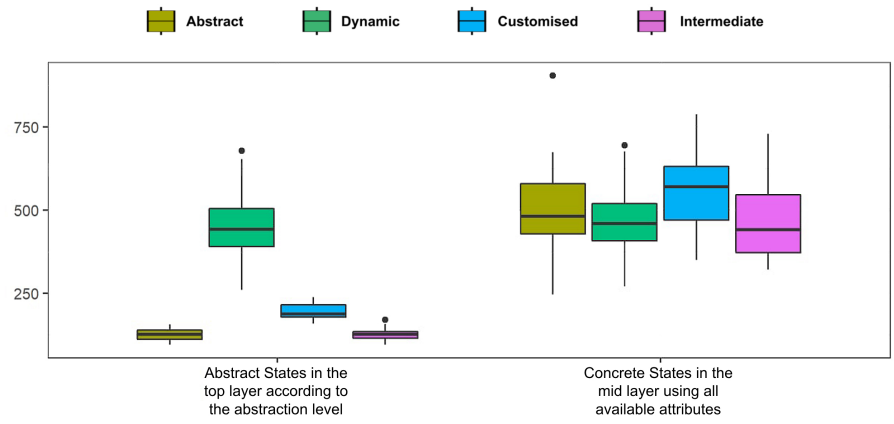


Figure 4.7: The number of abstract states (top layer) and concrete states (mid layer)

The results to analyse state discovery are shown as a box plot in Figure 4.7. Results show that too concrete level of abstraction creates almost as many abstract states as concrete states. As expected, the customised level creates more abstract states in comparison with abstract and intermediate configurations, but significantly less than the dynamic one. The customised level finds more concrete states, which indicates slightly better GUI exploration capability and matches the code coverage results.

The dynamic mechanism includes the dynamic widget title as part of the abstraction level, and as a consequence, new abstract states are constantly being discovered. However, this similar number of abstract and concrete states hinders exploration effectiveness, as the abstract mechanism fails to adequately track which states are newly discovered or were already visited during exploration.

RQ1 answer: *Different abstraction levels significantly affect GUI exploration effectiveness. A customised abstraction level, tailored to the application's characteristics, provided the best balance between state abstraction and exploration effectiveness. An ASM based on the state model consistently outperformed random selection regardless of abstraction level.*

4.4.2 RQ2: Defining a suitable level of abstraction

As observed in the previous section (and results from Figure 4.7), widget attributes used for abstraction should not be dynamic because they lead to state space explosion. Dynamic attributes are not stable because they can change their value during or in between runs without a detectable reason. Potentially stable attributes selected for the experiment are in the first column of Table 4.3.

Various attribute combinations were systematically evaluated to examine which widget attributes contribute to the generation of deterministic models. As illustrated in Figure 4.8, each combination (from single-attribute to multi-attribute sets) was applied to the subject SUT, and the impact on model determinism and state abstraction was measured.

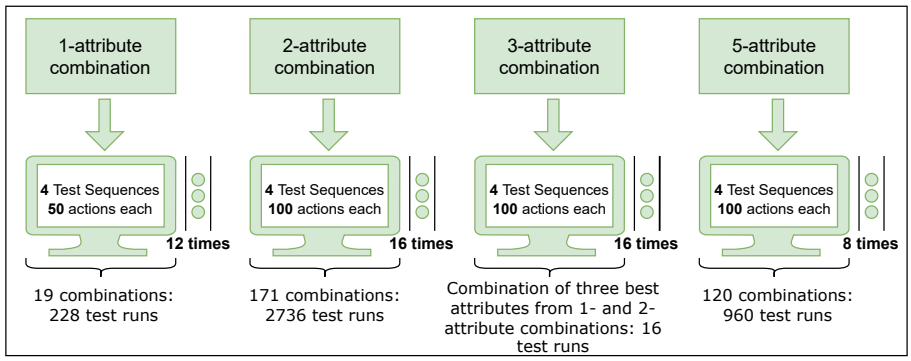


Figure 4.8: Overview of RQ2's attribute-combination experiment setup

4.4.2.1 Single Attribute Analysis

First, an experiment was conducted with only one attribute in the state abstraction. A test run consisted of four sequences, with a maximum of 50 actions per sequence. After some initial tests, these values were enough to detect non-determinism. Twelve consecutive test runs were executed for each widget attribute using the 12 available Virtual Machines (VMs).

Table 4.3 shows the results. Displayed are the widget attributes used, the average number of generated test steps executed in the test for each attribute, and the total number of steps taken in each test run before non-determinism was encountered. The results are ordered by the total number of steps executed over all 12 tests, starting with the widget attributes that "lasted the longest" before the model became non-deterministic. Although none of the generated models were deterministic, the WidgetTitle, WidgetBoundary, and WidgetHasKeyboardFocus attributes noticeably stand out from the others regarding the average number of steps executed.

4.4.2.2 Multi-Attribute Analysis

A second experiment was conducted with two attributes in the state abstraction. Combining two widget attributes gives 171 possibilities. Instead of four test sequences with 50 steps each, the number of actions per sequence was

Table 4.3: Number of generated test steps before the model became non-deterministic using a single widget attribute for state abstraction.

Attribute	Mean	Total
WidgetTitle	95.5	14,74,74,79,79,92,92,93,108,136,145,160
WidgetBoundary	90.6	5,61,64,77,79,83,84,103,105,115,155,156
WidgetHasKeyboardFocus	82.1	38,55,67,68,70,72,78,87,100,105,118,128
WidgetIsKeyboardFocusable	21.6	9,12,14,16,17,17,19,20,26,28,28,53
WidgetSetPosition	20.4	10,11,12,12,13,16,18,21,21,22,26,63
WidgetIsContentElement	20.3	7,9,14,15,17,17,18,21,24,27,35,39
WidgetIsOffscreen	19.7	6,9,11,14,14,15,15,18,19,27,29,59
WidgetGroupLevel	19.1	7,10,11,11,12,13,15,18,22,29,33,48
WidgetClassName	19.0	11,11,15,16,17,19,19,19,21,22,26,32
WidgetIsControlElement	16.9	8,11,11,12,13,13,16,16,20,24,28,31
WidgetIsEnabled	16.8	7,8,14,15,15,16,16,19,19,20,25,28
WidgetControlType	16.3	8,13,13,13,13,13,17,17,18,19,25,27
WidgetOrientationId	16.2	8,9,12,13,16,16,17,19,19,21,21,23
WidgetIsPassword	15.8	6,10,10,12,14,14,17,17,19,20,22,28
WidgetZIndex	15.6	9,11,12,12,13,14,15,16,16,19,22,28
WidgetIsPeripheral	15.4	7,8,9,13,14,14,16,19,20,21,22,22
WidgetSetSize	15.0	7,9,10,12,14,15,16,17,17,18,21,24
WidgetFrameworkId	15.0	8,11,12,13,13,14,15,16,17,18,21,22
WidgetRotation	14.7	8,9,12,12,13,14,16,16,16,20,20,20

upgraded to 100. This upgrade was based on the hypothesis that these combinations should last longer before the state model inference module encounters non-determinism. Each combination is tested 16 times, making for a total of 2736 test runs. In summary, none of the 171 combinations was able to produce a deterministic model. Moreover, after the 48 best-performing combinations, the average number of steps executed per test run declines quickly. Within these 48 combinations, three attributes occur 17 times, whereas the next best attribute occurs only 3 times. Again, the three best-performing attributes are WidgetTitle, WidgetBoundary, and WidgetHasKeyboardFocus.

A combination of these three attributes was employed and executed 16 times for the next experiment. Table 4.4 shows the results. Unfortunately, none of the test runs reached the complete 400 possible steps. Moreover, the average

and median are lower than the highest results from the 2-attribute combinations. However, the highest number of steps reached during a test run was 293, where this was 229 for the two-attribute combinations.

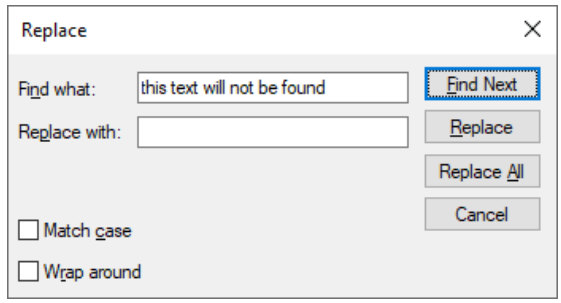
Table 4.4: Selection of attributes for state abstraction

Attributes	Total	Mean	Median
WidgetTitle, WidgetBoundary, WidgetHasKeyBoardFocus	1781	111.3	92.5

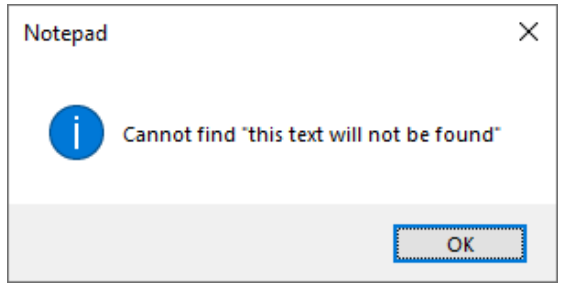
In the next experiment, combinations of five attributes were evaluated by selecting the three highest-scoring attributes from the two-attribute experiment and combining them with all the possible pairs of two additional widget attributes from the remaining 16 attributes. This resulted in 120 combinations, each executed eight times. Once again, no combination resulted in a deterministic model. Surprisingly, the more concrete abstraction using five attributes resulted in non-determinism faster than the three attributes in the previous experiment. This is probably due to the dynamic nature of some of the attributes.

As using five attributes for abstraction also resulted in non-determinism, the model was made even more concrete by incorporating all 32 control pattern properties into the tests. To make some headway, the three high-scoring general properties (WidgetTitle, WidgetHasKeyBoardFocus, and WidgetBoundary) were once again added and combined with all the combinations of two control patterns. This results in 492 possible combinations, and running each one eight times makes a total of 3936 test runs.

Several widget combinations reached the limit of 400 sequence actions without encountering non-determinism in the model, and all of these combinations included the 'Value' control pattern. Even though some combinations made it to 400 sequence steps 3 or 4 times out of the 8 test runs, they also encountered certain actions that led to non-determinism in the model. 'Value' pattern is a very 'concrete' attribute: 1) Using the 'Value' pattern can lead to models of infinite size, in the case that the application accepts text input that is not bounded. Hence, ideally, this attribute would be excluded from the abstraction mechanism. 2) While using this very concrete widget attribute, plenty of non-determinism in the state models was still encountered.



(a) Notepad 'Replace' dialogue



(b) Notepad 'Cannot find' popup

Figure 4.9: Notepad examples of non-determinism

Analysing the reasons for non-determinism, it was observed that certain actions lead to different states depending on the history of actions and states traversed before. For example, if the 'Replace' option in the 'Edit' menu is clicked in Notepad, the 'Replace' dialogue is opened (see Figure 4.9a). If the text written in the 'Find what' field is not found in the Notepad document, clicking the 'Find Next' or 'Replace' buttons will result in the same popup dialogue (see Figure 4.9b), having only an 'OK' button. Clicking that button will lead back to the 'Replace' dialogue. Still, the focus remains on the button that was pressed before, and if WidgetHasKeyBoardFocus was used in the state abstraction, clicking the 'OK' button leads to two different states based on the action that was taken in the previous state. In this case, altering the abstraction level by adding more widget attributes would not remove the non-determinism because the concrete states for the two visitations of the popup screen are also the same.

4.4.2.3 Including the predecessor state

Another solution is incorporating the state's incoming action into the state identifier [31]. In some situations, the state could depend on the previous state, requiring taking the previous state into account in the state identification algorithm. Consequently, the predecessor state and the incoming action were included in the state abstraction.

The first experiments run with all the combinations of widget attributes used in the experiments from Section 4.4.2 including the previous state identifier. The results showed that including the previous state in the abstraction resulted in a lower best performance and higher worst performance compared to using the same attribute combinations without the previous state. The average performance over all the combinations and test runs is comparable. The top-performing widget attributes are almost the same in both cases. Non-determinism related to viewing the status bar was still happening.

Subsequently, the incoming action was included in addition to the previous state in the abstraction identifier. Using the same attributes as the experiments in Section 4.4.2 allowed for comparison of the results. The average number of steps executed before encountering non-determinism significantly increased when using 1 or 2 widget attributes and including the incoming action. However, the results seemed to get worse with more widget attributes, probably because, in those experiments, the widget attributes were selected for their good performance without incoming action. With incoming action, the best-performing widget attributes were different. When executing the experiments including pattern attributes with incoming action, the ValuePattern and the 'incoming action' combination seems very successful. The following three combinations did not encounter any non-determinism during their 8 test runs of 400 actions:

1. Boundary, HasKeyboardFocus, LegacyIAccessiblePattern, Title, ValuePattern.
2. Boundary, DropTargetPattern, HasKeyboardFocus, Title, ValuePattern.
3. Boundary, ExpandCollapsePattern, HasKeyboardFocus, Title, ValuePattern.

As some of the detected cases of non-determinism were due to various lengths of text inputs, an additional experiment was executed, disabling input actions and

only allowing left-click actions. Results showed an increased average number of executed steps before encountering non-determinism. However, the model may be partial, and some functionality of the SUT may be excluded from the model. After conducting additional experiments to produce a deterministic model, it was concluded that this is not a trivial task.

Another aspect that might be important when using the inferred models, but has not been monitored so far in these experiments, is the size of the inferred abstract state model. For using the model programmatically, the size of the model probably affects the computation, but if the model is analysed by a human, the size restrictions have to be more strict. To have a first estimation of the model size, a selected set of different abstraction configurations was explored, measuring the number of abstract states in the model after a long execution of 50 000 actions.

The quest for inferring a deterministic model by making the abstraction more concrete resulted in a huge increase in the number of abstract states, as depicted in Figure 4.10. The implications of possible future research directions are discussed in Section 4.5. The results demonstrate the inherent tension between model determinism and usability. More concrete models achieve better determinism but face state explosion, while more abstract models remain manageable but encounter non-determinism more frequently.

RQ2 answer: *While no single or simple combination of widget attributes consistently produces deterministic models, including action history and carefully selected attribute combinations can significantly reduce non-deterministic behaviour. However, achieving determinism requires increasingly complex state representations, leading to potential state explosion and reduced model usability.*

4.5 Discussion

This section presents findings highlighting key challenges encountered during the state model inference process and the application of inferred models in testing.

Ab1: Current+previous+incoming action; all; Boundary, IsControlElement, IsKeyboardFocusable
Ab2: Current; all; Title
Ab3: Current; all ; Boundary, HasKeyboardFocus, Title
Ab4: Current+previous; all; Boundary, HasKeyboardFocus, Title
Ab5: Current+previous+incoming action; all actions; Boundary, HasKeyboardFocus, Title
Ab6: Current; left click only; Boundary, HasKeyboardFocus, ScrollItemPattern, Title, ValuePattern
Ab7: Current; all; Boundary, HasKeyboardFocus, ScrollItemPattern, Title, ValuePattern
Ab8: Curr+prev+incom; all; Boundary, HasKeyboardFocus, ScrollItemPattern, Title, ValuePattern

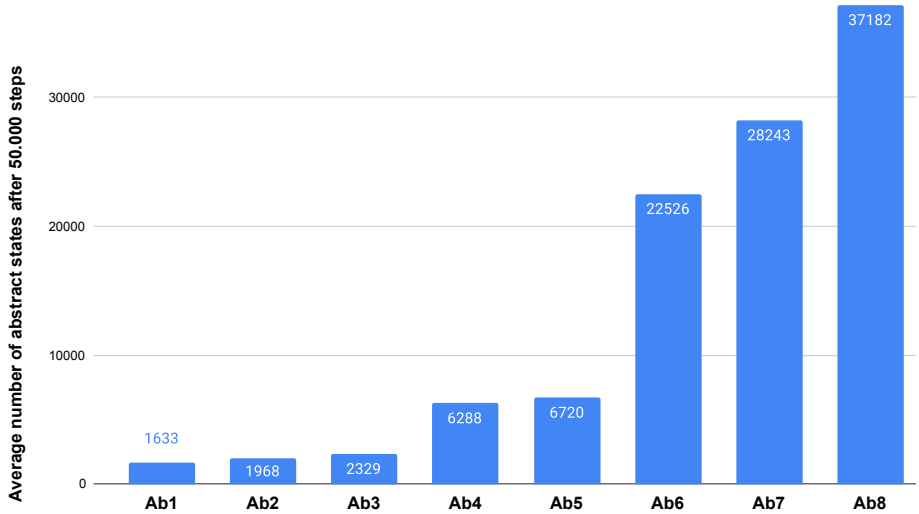


Figure 4.10: Average number of abstract states after 50000 actions for abstractions Ab_i: state; actions; attributes for $i \in 1, \ldots, 8$

These challenges and lessons learned offer insights into the complexities of creating effective and usable state models for automated GUI testing. Additionally, potential directions for addressing these challenges and improving the utility of inferred models in testing are discussed.

4.5.1 State abstraction

The first finding is that tuning the abstraction level for model inference seems highly dependent on the specific SUT. While tuning the abstraction level, the

following SUT-specific characteristics should be taken into account:

- **Dynamic increment of widgets**: Some applications, for example, Rachota, contain dynamic lists of elements where new items can be continuously added. This constantly creates new widgets and states in the model, causing a state and action space explosion.
- **High number of combinatorial elements**: Some applications, such as Notepad, include multiple scroll lists with a large number of different elements, and from a functional point of view, it is not important to cover all of these options (e.g., Notepad font selection).
- **Slide actions**: In some applications where scrolling actions are required, the exact scrolling coordinates from start to end can cause a change in the number of widgets visible in the state. Depending on the state abstraction and how the widget tree is obtained, this can create new states and cause a high number of combinatorial possibilities.
- **Popup information**: In some applications, like Rachota, a descriptive popup message may appear for a few seconds in the GUI when the mouse hovers over some of the widgets. This could result in a new state for the model, and it might cause non-determinism if hovering over a widget was not an intentional action that was executed on purpose.

A second finding is that trying to produce a deterministic state model is far from a trivial effort. There are various options to address the challenges of non-determinism in the inferred models.

1. Let the models have non-determinism and deal with it when using them. Action selection requires detecting when the modelled behaviour deviates from the observed behaviour and temporarily adjusting the action selection to avoid getting stuck during GUI exploration. Another solution, and an interesting future research topic, could be using the concrete state model to navigate through states that have non-determinism in the abstract state model.

2. Try to infer a deterministic model. This would require more SUT-specific ways to dynamically adjust the abstraction, for example, based on the widget type or even a specific widget in a particular state. TESTAR currently supports triggered behaviour that overrides normal action selection. Similarly, a mechanism is planned to trigger changes in the calculation of state identifiers, such as ignoring a specific widget during state abstraction. An example of a widget that can be ignored from the state model is a dynamically changing advertisement on a website.

3. Correct non-deterministic models after runtime. However, this technique has not yet been observed in existing model-based testing tools.

4.5.2 Applying the inferred models in testing

One of the core objectives for this chapter was to use the inferred models for a new action selection mechanism (ASM) for TESTAR. The new ASM was presented in Algorithm 1, and initial experiments show that it is better than random. Although this is a good result, it is also a step towards implementing more advanced ASMs. For example, ASMs-based artificial intelligence (AI) needs some model for learning, and the inferred model can serve that purpose.

Another advantage of the inferred state models is that human testers can use them during testing. For instance, it is interesting to have an overall view of an application's execution flow: to see the details about a certain state or executed action, to identify the path to a state where an application failed, and to obtain various metrics about the state model. Although some of this information can be obtained by querying the OrientDB database and outputting it as textual data, e.g., in tabular format, it is argued that the data would be best presented through visualisation, making it more intuitively understandable for humans.

Abstract state models can also allow performing conformance testing, which determines how a system under test conforms to meet the individual requirements of a particular standard. Before using inferred abstract models, the domain experts must validate them to use the automatically generated test cases for conformance testing. This also requires suitable visualisation.

4.6 Conclusions

This chapter describes the state model inference extension for TESTAR and reports experiments on the impact of various state abstraction mechanisms for the purpose of producing a deterministic model and on the evaluation of the performance of an action selection algorithm using the inferred models.

The experiments on using various state abstraction mechanisms show that inferring a deterministic abstract state model is difficult, especially when trying to prevent the state space explosion. Based on these experiences and the fact that, in the literature, many approaches using inferred models for GUI exploration or testing do not explain the details about state abstraction, more research and new, more flexible abstraction mechanisms are needed. Also, dealing with non-determinism in the inferred models is an important direction for future research.

Based on the experiments on the impact of various levels of state abstraction on the performance of an ASM using the inferred models, it can be concluded that an appropriate level of abstraction enhances the performance of GUI exploration, as measured by code coverage. Having a too-abstract or too-concrete model has a negative impact on performance. However, in the experiments, the ASM_*statemodel* performed better than the ASM_*random* with all tested abstractions.

5

Adding intelligence

"You will have to lose hundreds of games before becoming a good player."

José Raúl Capablanca, A Primer of Chess

This chapter examines the use of Reinforcement Learning (RL) [147] to implement more sophisticated ways to select actions. RL is a branch of machine learning that is directly applicable to scriptless testing. It is trained using rewards after every interaction with an environment. As observed in Chapter 2, a growth of papers applying Artificial Intelligence techniques has been observed in GUI testing during the last decade, mainly for mobile Android applications. RL uses *reward functions* to guide the selection of actions and explore the search space of the system being tested. These rewards are usually based on the difference between consecutive states, with high rewards given to actions that lead to very different states.

Using rewards based on the difference between consecutive states can lead to unwanted behaviour such as *Jumping Between States* (JBS) instead of exploring other areas. Pan et al. [148] describe the problem: if a large reward is obtained as the result of moving between two very different states, the RL agent will

frequently jump between these states instead of exploring other areas of the SUT. A solution to the JBS problem was proposed based on neural networks to extract main features from the states to detect if two states are different and save them as a vector in a memory buffer to avoid repeating states. However, neural networks require much data and time to work well. Additionally, the inability to debug and explain the reasoning and evolution of a neural network over time is a disadvantage of this technique. Instead, the approach proposed in this study uses state abstraction to compare states and includes memory, based on the frequency of executed actions, to the reward itself.

Testing Web applications has more diverse and complex states than mobile or desktop applications due to their frequent dynamic content updates and elaborate workflows. More effective exploration is required to test Web applications, considering their huge search space and interactive nature. Therefore, this chapter focuses on testing Web applications in addition to proposing an alternative solution to the JBS problem. The contributions of this chapter are:

- an implementation of an RL framework that can be used to compare different rewards used during scriptless testing of Web applications;
- an empirical evaluation showing the reduction of the JBS problem for rewards containing memory based on the frequency of executed actions
- a comparison of the exploration effectiveness of the different rewards looking at URL coverage and state exploration.

The rest of the chapter is organised as follows: Section 5.1 presents background about *Q*-Learning, and Section 5.2 offers relevant related work. Section 5.3 presents the proposed approach for smart exploration, Section 5.4 presents the empirical evaluation, and Section 5.5 presents the results. Section 5.8 concludes the chapter.

5.1 *Q*-Learning

Reinforcement Learning (*RL*) [147,149] is a machine learning technique that consists of an agent that learns to behave in an interactive environment. The agent

works as an independent entity that executes actions within the environment and obtains information through trial-and-error interactions. RL algorithms contain four basic elements:

- The **state** describes the situation of the environment at every step.
- An **action** is a possible move from one specific state to another.
- The **policy** defines the strategy used to select an action in a given state and the learning approach of the algorithm.
- The **reward** defines the goal to achieve.

The environment can be formalised with a Markov Decision Process (MDP), defined as a 4-tuple $M = \langle S, A, T, R \rangle$, where S is the set of states, A is the set of actions. At each time step t, the agent executes an action $a_t \in A$ and will receive a reward $r_t \in R$. The transition probability function T describes the probability $P(s_{t+1}|s_t, a_t)$ of transitioning into state s_{t+1} from state s_t after executing action a_t. The goal of RL is to maximise the rewards obtained over time, formalised as *expected return* and defined as:

$$E\left(\sum_{t=0}^{\infty} r_t \mid s_0 = s\right)$$

This value represents the expected sum of rewards starting from an initial state $s_0 = s$. This formulation assumes that all future rewards contribute equally to the return. However, in many real-world scenarios, sooner rewards are often more valuable than those received later. To incorporate this preference, the concept of a discounted return is introduced.

The discounted return modifies the expected return by multiplying future rewards by a discount factor $\gamma \in [0, 1]$, used to define the balance between long-term and immediate rewards. The discounted return is defined as:

$$R_t = \sum_{k=0}^{\infty} R_{t+k+1} \gamma^k$$

When γ is close to 1, the agent values long-term rewards almost as much as immediate rewards. When γ is close to 0, the agent focuses primarily on immediate rewards.

To make this practical, the expected return is associated with an action-value function $Q(s, a)$, which estimates the expected return after taking an action a in a state s and following a specific policy π. Formally:

$$Q^{\pi}(s, a) = E\left(R_t \mid s_t = s, a_t = a\right)$$

The *policy* π defines the agent's action-selection strategy. It can be deterministic, where $\pi : S \rightarrow A$, mapping each state s to a specific action a, or probabilistic, where $\pi : S \times A \rightarrow [0, 1]$, assigning a probability distribution over actions for each state.

Q-learning [150] is a model-free RL approach to learn the value of an action in a particular state and, hence, find an optimal action-selection policy π. An action-value function estimates the expected return $Q^{\pi}(s_t, a_t)$ after executing an action a_t in state s_t following policy π. The action-value function can be defined in terms of its successor state-action pair as:

$$Q(s, a) = Q(s, a) + \alpha * [reward + \gamma * \max_{a' \in A} Q(s', a) - Q(s', a')] \tag{5.1}$$

The Q-function refers to the maximum Q expected for a given (state, action) pair over all possible policies. Furthermore, Q can also be interpreted as the optimal strategy at each step, maximising the sum of the immediate reward r of the current step and the Q-value $Q(s', a')$ of the next step. The parameter α (step size) defines the learning rate of the algorithm. A higher learning rate (closer to 1) makes the algorithm adapt more quickly to recent experiences but may lead to instability, as it can overwrite previous information too aggressively. Conversely, a lower learning rate (closer to 0) makes the algorithm update more conservatively, allowing it to retain more of its past knowledge but potentially slowing down learning.

The idea behind using Q-learning for GUI testing is to reward each selection of possible actions over the SUT (see Algorithm 2). Choosing an action (line 5)

and executing it (line 6) moves the agent from the current state s to a new state s' (line 7). The agent is rewarded with a *reward* upon executing the action a (line 8). This reward is calculated by the reward function R. The main objective of Q-learning is to *learn* how to act in an optimal way that maximises the cumulative reward (line 9). An approximation for the optimal Q-function that simplifies the problem and enables early convergence is shown in algorithm 5.1, where *learn* is defined as the action-value in equation 5.1.

Algorithm 2 Q-Learning

Input: γ, α ▷ *discount factor, learning rate*

```
1:  initialiseQValues()
2:  s ← getStartingState()
3:  repeat
4:      availableActions ← deriveActions(s)
5:      a ← selectAction(availableActions)          ▷ Select an action
6:      executeAction(a)
7:      s' ← getReachedState()
8:      reward ← R(s, a, s')                        ▷ Reward the action
9:      learn(s, s', a, reward, γ, α)               ▷ Learn from the experience
10:     s ← s'
11: until s' is the last state of the sequence
```

In this context, online Q-learning is employed, where the Q-values are updated incrementally after each action and reward observed during interaction with the environment. Due to its dynamic and exploratory nature, online Q-learning is particularly well-suited for scriptless GUI testing. Unlike offline Q-learning, which relies on pre-collected datasets, online Q-learning allows the testing tool to continuously adapt its knowledge of the SUT's state space as new states and transitions are encountered.

A critical aspect of online Q-learning is balancing exploitation (choosing the action with the highest known Q-value to maximise immediate rewards) and exploration (selecting less-frequented actions to discover potentially better rewards). This trade-off is usually managed using an ϵ-greedy strategy, where the agent selects the best-known action with probability $1-\epsilon$ and explores a random action with probability ϵ. This approach ensures the agent avoids getting stuck

in suboptimal strategies while still improving its policy.

The following section reviews the related work to contextualise this method within the field of scriptless GUI testing. The related work focuses on existing reinforcement learning techniques applied to software testing and scriptless methods.

5.2 Related Work

RL has been used for GUI testing in different ways. Offline *Q*-learning, uses only previously collected offline data in [151, 152]. These works learn from existing SUTs about actions that are useful to reach a particular objective and then apply it to new SUTs. Online *Q*-learning is applied independently for every SUT with state-based rewards and the approaches are summarized in Table 5.1. Since this chapter focuses on online learning, each of these aspects is discussed next.

TESTAR, as described in Chapter 3, has implemented *Q*-learning with a frequency-based reward function for Java desktop applications [121]. This reward aims to encourage the selection of the least executed actions, thus guiding exploration towards unvisited areas of the application. Also applying *Q*-Learning, Adamo et al. [153] use the same frequency-based reward but in the scope of Android applications.

AutoBlackTest [154] also uses *Q*-Learning to test Java desktop applications. Their reward function favours actions that increase the difference between two consecutive states, measured by comparing their respective widget trees. Similarly, a combination of frequency and widget tree rewards can be found in [155].

AimDroid [156] uses RL to guide the exploration of Android applications. A positive reward is obtained if a new activity or crash is observed. More recently, ARES [157] uses a similar reward to AimDroid (larger values) but with Deep Neural Network as a technique to learn the best exploration strategy. In the scope of Deep Reinforcement Learning, Collins et al. [158] use a reward based on the code coverage obtained during the execution.

Table 5.1: Related work with state-based rewards

Publication	RL Algorithm	Reward	Policy	SUT types
Vos [10]	Q-Learning $\alpha = 1$ multiple γ	Frequency-based	Greedy	Windows desktop, Web
Adamo [153]	Q-Learning $\alpha = 1$ dynamic γ	Frequency-based	Greedy	Android
Mariani [154]	Q-Learning	Widget Tree difference between consecutive states	ϵ-Greedy, $\epsilon = 0.8$	Windows (Java)
Vuong [155]	Q-Learning $\alpha = 1$ $\gamma = 0.9$	Combination of event distance between consecutive states and frequency-based rewards	ϵ-Greedy ($\epsilon_{initial} = 1$) Gradually decreased up to 0.5	Android
Cao [156]	SARSA	Small reward (1) if crash or new activity is found. Low reward (-1) if activity is out of SUT. Very low reward (-10) otherwise.	ϵ-Greedy	Android
Romdhana [157]	Q-Learning and Deep RL algorithms	Very large reward (1000) if crash or new activity is found. Large reward (100) if activity is out of SUT. Small reward (1) otherwise.	ϵ-Greedy $\epsilon = 0.8$ or $\epsilon = 0.5$	Android
Collins [158]	Deep RL algorithm	Large reward if code coverage increases or new activity is found. Small reward if state does not change.	-	Android
Pan [148]	Q-Learning	Similarity between consecutive states (using Neural Network)	ϵ-Greedy, $\epsilon = 0.2$	Android
Degott [159]	multi-armed bandit [160]	1 if interface changes between consecutive states. 0 otherwise.	ϵ-Greedy Thompson Sampling	Android

Similar to AutoBlackTest, Pan et al. [148] give a large reward when a very different state is reached. They described the JBS problem that arises with this reward: if two states are too different, the agent will continuously receive high rewards, and consequently, the testing tool might frequently jump between them. As mentioned in the introduction, to address this problem, Pan et al. used a neural network to compare the reached state with the previously visited states.

Degott et al. [159] build a general model with the goal of sharing models between different apps. Their reward is either 1 or 0, according to visual changes.

A different and more straightforward solution to the JBS problem is proposed in this study by incorporating memory, based on the frequency of executed actions, into the reward. To achieve this, the visited states and executed actions are tracked. An additional novelty of this proposal is using RL to explore web-based applications. Little work has been done on RL-based web exploratory testing without prior knowledge, with most studies focusing on Android or Windows applications.

5.3 Smart Scriptless Testing

This section outlines the method for integrating Q-Learning into TESTAR. The SUT is the environment, and states are determined by TESTAR along with all available actions that can be executed. Initially, TESTAR has no knowledge of the SUT, but as the tool learns to select the most optimal action at each step, it updates its knowledge to find the best policy. These actions generate test sequences. For the 4-tuple MDP $M = \langle S, A, T, R \rangle$:

- a state $s \in S$ is represented as the set $(w_1, \ldots, w_n)$ of widgets that together constitute the widget tree.
- an action $a \in A$ is represented as a 2-dimensional: (*action type*, *widget*).
- the execution of the SUT causes the transition T: TESTAR executes an action and observes the new state.

- each time TESTAR executes an action a in state s that results in state s', a reward $R(s, a, s')$ is calculated.

5.3.1 Rewarding test behaviours

To apply RL to automated GUI testing using TESTAR and define smart exploration strategies, rewards must be tailored towards improved testing. The following defines four types of state-based rewards: frequency, state-change, state, and combined.

Frequency-based Rewards: This reward is based on the previous work described by Vos et al. [10], where actions with low execution count are favoured (see 5.2). The reward function is inversely proportional to the number of times $ec(a, s)$ the action a has been executed in state s. The R_{max} parameter determines the initial reward assigned to unexplored actions. High values of R_{max} might bias the search towards executing new actions.

$$R_{frequency}(s, a, s') = \left\{ \begin{array}{ll} R_{max}, & \texttt{if } ec(a, s) = 0 \\ \frac{1}{ec(a,s)} & \texttt{otherwise} \end{array} \right\} \tag{5.2}$$

Rewarding State Changes: By simply observing the interface of the state, it is possible to observe the changes, similar to how the user will experience the exploration of the SUT. Two screenshots corresponding to the previous and the current state are obtained and scanned pixel by pixel to compare their RGB value. The reward consists of calculating the ratio between the total number of different pixels $dp(s, s')$ and the total of pixels tp. If all pixels are equal, no observable state change is detected; hence, the reward is 0. Otherwise, the maximum reward value of 1 is returned if all pixels differ.

$$R_{state-change}(s, a, s') = \frac{dp(s, s')}{tp}, \tag{5.3}$$

Rewarding Reached State: While tuning the RL parameters, a problem described by Pan et al. was encountered [148]. An agent is rewarded with large values after finding very different states, which might result in constantly jumping between them. The problem persists even when *State Changes* rewards are combined

with frequency-based rewards, as the initial situation reemerges once most states and actions have been visited or executed multiple times. To solve this problem, in [148], a memory buffer was proposed, where the reached state is compared with a set of previously visited states instead of with only the immediately previous state. This research uses TESTAR's state model to keep a memory of which actions have been executed on every abstract state.

Initially, a single reward is calculated according to the level of exploration of the reached state, as shown in 5.4. The value is 1 (maximum possible) if none of the available actions has been executed, i.e. the state has not been explored yet. On the other hand, as the actions are executed, the reward decreases until it reaches the minimum value of 0. The main advantage of this reward is that it is independent of the previously visited states, acting as a pure measure of how *useful* the current state is. This research hypothesises that this reward will exhibit a lower incidence of the JBS problem.

$$R_{state}(s, a, s') = \frac{\sum_{a' \in A(s')}[ec(a') = 0]}{||A(s')||} \tag{5.4}$$

Combining Rewards: A final reward is proposed, combining all previous rewards to address the JBS problem and provide the agent with enhanced information for effectively exploring the state space of the SUT. However, the weights of this reward may need to be adjusted for each specific SUT. For the sake of simplicity, the reward function weights were assigned equal values.

$$R_{combined}(s, a, s') = w_1 * R_{frequency} + w_2 * R_{state-change} + w_3 * R_{state} \tag{5.5}$$

5.3.2 RL Framework

The interface-based architecture (from Section 3.6) results in a modular and maintainable framework since the RL functionality is independent of the rest of the tool. The framework consists of an implementation of the ActionSelector interface called QLearningActionSelector as depicted in Figure 5.1. This class implements `selectAction` and is responsible for selecting the following action using *Q*-

Learning. The QLearningActionSelector class uses three main interfaces:

- RewardFunction is an abstraction for reward functions. In Figure 5.1, the four implementations of the reward functions from Section 5.3.1 are shown.
- Policy is an abstraction for possible policies. For this study, ϵ-Greedy was implemented: a strategy that defines a probability ϵ for exploration.
- QFunction represents the action-value function, implemented by QLearningFunction to calculate Formula 5.1.

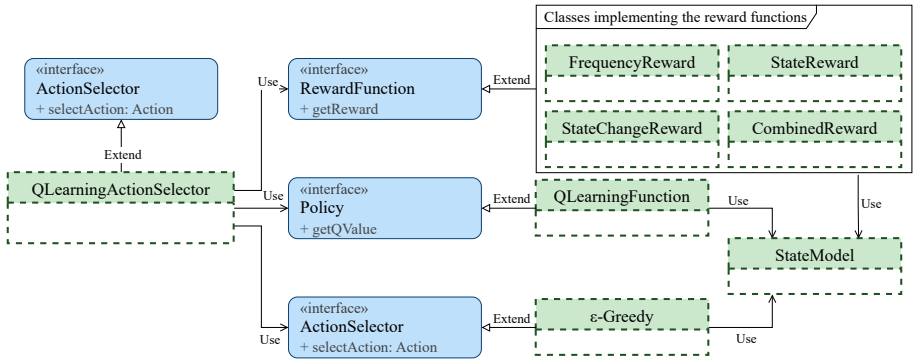


Figure 5.1: `QLearningActionSelector` implements `selectAction` with Q-Learning.

Algorithm 3 implements `selectAction` using the interfaces described earlier. Lines 1 and 2 in Algorithm 3 correspond to lines 8 and 9 in Algorithm 5.1, respectively. Once the Q value is computed, it is stored in the TESTAR State Model (see Section 4.2) as a property of the action, since every action is represented as an outgoing transition of a state. Therefore, the Q-values are associated with both the abstract representation of the state and the action. Finally, the QLearningActionSelector uses the policy to select the next action.

Configuration of a specific Q-Learning set-up is done through TESTAR's configuration file (`test.settings`). This is used to configure the reward function, policy and Q-function that the ASM will use and to define the parameters such as ϵ for the policy and δ and α for the Q-learning algorithm (see Algorithm 2).

Algorithm 3 `selectAction` in class QLearningLActionSelector

Input: $RewardFunction$

Input: $QFunction$

Input: $Policy$

Input: s, a, s' ▷ *Previous State, Executed Action, Current State*

1: $reward \leftarrow RewardFunction.getReward(s, a, s')$
2: $q \leftarrow QFunction.getQValue(s, a, reward)$
3: $updateQValue(s, a, q)$
4: $a \leftarrow Policy.applyPolicy(s')$
5: return a

Factory method patterns[1] are used to select and initialise the applicable implementation class. During the initialisation of the applicable classes, the configured parameters are set. It is possible to generalise this implementation, as it is independent of the internal implementation of TESTAR and could be applied to any other tool capable of abstracting states from a SUT. However, an adapter class was implemented to integrate it with the TESTAR implementation. This adapter class could be easily modified or replaced to integrate the implementation with a different tool.

5.4 Experiment Design

This study investigates the effectiveness of the different rewards and their influence on the JBS problem. To achieve this, the following research questions were formulated:

RQ1 *Which reward-based ASM most effectively explores the SUTs?*

RQ2 *Does $R_{state-change}$ result in a higher occurrence of the JBS problem compared to $R_{frequency}$, R_{state} or $R_{combined}$?*

[1] A design pattern that handles object creation by delegating the instantiation decision to specialised classes.

Our experiment, based on guidelines presented in [114,144], aims to answer the research questions by analysing the exploration effectiveness of four different rewards compared to random action selection. Additionally, the effectiveness of frequency-based and state-based rewards as memory-based solutions for the JBS problem is evaluated. Null hypotheses are formulated to facilitate statistical analysis of the experiment.

$H0^1$: The exploration performance of the ASMs, as measured by state, action, and URL coverage, is statistically equivalent across all evaluated ASMs.

$H0^2$: The occurrence of states associated with the JBS problem is statistically equivalent for $R_{state-change}$, R_{state}, $R_{combined}$ and $R_{frequency}$.

5.4.1 Objects: Selection of SUTs

The SUTs selected for the experiment should comply with the following: 1) The SUTs have a GUI; 2) TESTAR can detect the widgets on the GUI of the SUTs; 3) The SUT contains a high difference between consecutive states to increase the probability of encounter loops. Since the focus was on web exploration, the following three SUTs were selected.

Shopizer is an e-commerce sales management software that allows the creation of online stores, marketplaces or product listings. The home page comprises a search form, an item menu and a banner. Shopizer is an open-source website containing 126 Java packages for a total of 410 Java classes and 23330 lines of code. Shopizer was selected as a demo application for the initial experiments of this work. To increase the search space, 10000 fake products were added to 6 categories and 60 subcategories. Complex actions such as pagination and searching are required to access the products.

Craigslist is a classified advertisement website with more than 80 million new classifieds each month. Similar to Shopizer, Craigslist divides the products into multiple categories and subcategories. However, the product listing view is more complex: each category has specific search options for refining the displayed product list. During the execution of the experiments, a total of 105 search options were observed among all the categories.

Bol.com is a large webshop with many different products. It was selected due to its similarity with both Shopizer and Craigslist. Nevertheless, Bol.com provides a more complex interface: extensive sequences of complex actions are required to unblock certain areas of the application. Moreover, small images of the products are always displayed, making the comparison based on screenshots between states more sensitive. Furthermore, the home page consists of the more recent products visited by the user, adding extra dynamism to the website. Finally, as with Craigslist, specific search options are provided for every category.

5.4.2 Independent and Dependent Variables

The ASMs based on the different rewards from Section 5.3.1 are compared with the random ASM. This means the factors are the rewards, and all other independent variables are kept constant.

Independent Variables: The independent variables in this study are the parameters manipulated to observe their impact on exploration effectiveness. These variables include:

- **State abstraction:** The abstract state representation affects how TESTAR detects widgets. In this work $ABS_{PROP} = \{WidgetID, WidgetTextContext\}$. This abstraction was selected after several trials.
- **Action derivation:** The widget associated with the action is always part of the action's abstract representation. To differentiate actions originating from the same widget, the role of the action (e.g., click or type) is added to its representation.
- **Filters:** Different parts of the web applications were filtered out for every SUT, such as payment checkouts or registration, because they require specific sensitive information to work correctly.

Other independent variables for the experiment are constant values:

- **Execution time:** Each action has an execution time of 1 second

- **Time between actions:** a minimum waiting time of 1 second between executed actions.
- **Length and number of test sequences:** A test run consists of 300 sequences of 100 actions each.
- **Exploration policy:** An ϵ-Greedy policy was selected as the policy probability of exploration, with $\alpha = 1$, $\gamma = 0.7$ as the Q-learning parameters (based on the related work).

Dependent Variables: The dependent variables represent the outcomes measured to assess exploration effectiveness and the presence of the JBS problem.

To answer RQ1, it is necessary to measure the exploration performance of each ASM. While the SUT is explored, new states and actions will be discovered and/or visited by the RL agent. To measure the exploration performance of each RL algorithm in terms of state and action space size, the available information is extracted from TESTAR's state model. Although code coverage is a good indicator of the exploration effectiveness of a testing tool, in the case of real web applications, this is not always available. Alternatively, the number of unique URLs visited on each website was counted.

The JBS problem occurs when two states are significantly different. This problem arises because the agent receives high rewards for transitioning between such states. As a result, the agent may frequently "jump" between these states, limiting the exploration of other areas. States that frequently appear in short sequences are often very different from their consecutive reachable states. It is possible to obtain a path $S_1, S_2, ..., S_n$ from every test sequence, where S_i is the abstract state visited after executing action a_{i-1}. A loop in a path means that certain abstract state was revisited. A *jumping state* can be defined as a state that appears excessively in multiple loops, which intuitively indicates the presence of the JBS problem. When a test execution is finished, every loop in every path is extracted. If a state s is the initial and final state of the loop, the length l_i of the loop is associated with that state. Thus, a tuple (s, l_i) is obtained for every loop in the test sequence.

The Dependent Variables measured to answer the RQs are:

- Number of different abstract states visited
- Number of different abstract actions executed
- Number of different abstract actions discovered
- Average number of distinct URLs visited
- Number of state-loop pairs (S_i, l_i) identified per test sequence.

The number of (S_i, l_i) pairs represents the loops detected during a test sequence. Here, S_i is the state that starts and ends the loop, while l_i is the loop's length. By analysing these pairs, "jumping states" (i.e. states that frequently appear in loops) can be identified, serving as a key indicator of the JBS problem. For instance, if a specific state S_i is involved in multiple loops with short lengths, it suggests that the agent often returns to this state without exploring new areas, highlighting the presence of the JBS problem.

The goal for identifying such states is to detect outliers, i.e., states exhibiting extreme looping patterns, such as high loop frequency and short loop length. By analysing the proportion of outliers across different rewards and SUTs, the influence of each reward on mitigating (or failing to mitigate) the JBS problem can be evaluated.

5.4.3 Experimental Process

In this experiment, the general design principle of blocking was used. This means fault-detection mechanisms are disabled to ensure that errors and exceptions do not interrupt the test runs, as the primary focus is exploration rather than fault detection. As a result, oracles (that define the errors and exceptions that TESTAR will check) will not be used during this experiment because finding an error or exception will interrupt the test sequence, and the goal of this experiment is exploration.

To address the randomness of TESTAR ASMs, all test runs of the experiment will be repeated 20 times using concurrent Virtual Machines (VMs) to execute

the tests. Each virtual Windows machine is configured with a 4.5 GHz CPU and 16 GB RAM.

To analyse the data and evaluate the hypotheses, the Mann-Whitney-U test was used for pairwise comparisons between ASMs due to the non-normal distribution of data. The effect size of significant differences was calculated using Cliff's Delta, with thresholds for large, medium, and small effects.

States exhibiting unusual loop behaviour are identified using DBSCAN, a density-based clustering algorithm. DBSCAN explicitly labels points that do not belong to any cluster as noise and does not require predefining the number of clusters, making it ideal for unknown state-loop distributions [161].

DBSCAN requires three parameters: the neighbourhood radius (ϵ-DBSCAN), the minimum number of points (*minPoints*) required to form a dense region and the features used for clustering. To determine the optimal radius, the k-Nearest Neighbors (kNN) distance plot is often used [162]. In statistics, the kNN distance plot is a reliable method for determining density thresholds, as it objectively identifies the transition between high-density clusters and sparse regions.

To configure kNN to predict the optimal neighbourhood radius, the parameters k (for KNN) and *minPoints* (for DBSCAN) are both set to the same value: $\sqrt{n}$, where n represents the total number of loop states. This approach aligns with standard practices in kNN classification, where k is often chosen as the square root of the total number of data points [163]. Table 5.2 shows the parameter configuration per SUT. Moreover, loop frequency and average loop length per loop state were used as the features for clustering with DBSCAN.

SUT	k	ϵ-DBSCAN
Craigslist	33	0.75
Bol.com	23	0.45
Shopizer	23	0.75

Table 5.2: DBSCAN parameters configuration for each SUT.

5.5 Results

This section presents the findings of the experiments, addressing the formulated research questions through statistical analyses. The exploration effectiveness of the evaluated ASMs is compared, and the incidence of the JBS problem across the different rewards is analysed.

5.5.1 RQ1: Exploration Effectiveness

The ability of the different ASMs to explore a web application was tested by monitoring the URLs visited during execution. Additionally, TESTAR's state model provides information about states and actions that have been discovered or visited.

The number of distinct abstract states visited, different actions executed, and unvisited actions were counted for each application. An action is considered unvisited when derived by TESTAR but has yet to be visited. Table 5.3 shows the average values per SUT and ASM, while Figures 5.2, 5.3 and 5.4 show the results for URL performance and space-related variables.

Table 5.3: Average values of dependent variables for every SUT

ASM	$R_{combined}$	Random	$R_{frequency}$	R_{state}	$R_{state-change}$
			Shopizer		
Abstract States (mean)	291.50	296.00	312.50	321.95	290.33
Abstract Actions (mean)	1097.85	1109.20	1146.17	1177.65	1065.67
Unvisited Actions (mean)	5135.65	4684.85	5551.56	5687.40	5180.39
URL (mean)	115.80	115.05	122.67	125.40	114.56
			Craigslist		
Abstract States (mean)	1010.30	1022.05	991.14	959.20	1003.26
Abstract Actions (mean)	2234.65	2464.25	2226.00	2229.70	2223.11
Unvisited Actions (mean)	35625.15	25781.40	37752.00	38824.10	36861.42
URL (mean)	1028.55	1031.70	963.10	946.50	1049.63
			Bol.com		
Abstract States (mean)	813.10	701.60	803.63	733.70	722.40
Abstract Actions (mean)	1544.47	1365.80	1546.42	1443.80	1422.70
Unvisited Actions (mean)	17445.26	14402.20	17455.95	15686.30	14748.50
URL (mean)	217.53	222.10	227.42	224.85	199.75

Pairwise comparisons between each ASM were conducted following established guidelines [144], and the effect size was measured in each case. Table 5.4 summarises the findings after applying the Mann-Whitney-U test to compare the exploration of abstract states. The statistical results shown in Table 5.4 confirm that R_{state} obtained the best performance. Also $R_{frequency}$ outperforms $R_{combined}$.

Table 5.4: The p-values calculated pairwise for Abstract States. When $p < 0.05$, the effect size is calculated with Cliff's delta.

ASM	$R_{combined}$	Random	$R_{frequency}$	R_{state}	$R_{state-change}$
			Shopizer		
Random	0.61	–	–	–	–
$R_{frequency}$	<0.01 (l)	0.07	–	–	–
R_{state}	<0.01 (l)	0.01 (l)	0.43	–	–
$R_{state-change}$	0.60	0.94	<0.01 (l)	<0.01 (l)	–
			Craigslist		
Random	0.72	–	–	–	–
$R_{frequency}$	0.33	0.02 (m)	–	–	–
R_{state}	0.18	0.03 (l)	0.39	–	–
$R_{state-change}$	0.55	0.53	0.35	0.22	–
			Bol.com		
Random	0.01 (l)	–	–	–	–
$R_{frequency}$	0.20	0.18	–	–	–
R_{state}	0.03 (m)	0.21	0.49	–	–
$R_{state-change}$	0.01 (l)	0.52	0.36	0.54	–

Figure 5.2 shows high variability for Random across all dependent variables in Shopizer, with some executions achieving excellent results while others exhibit the worst performance. Shopizer always lists the same products in the same order, making it challenging to browse different items. The pagination is based on a "load more button". Consequently, new actions only appear if that button is clicked. On the contrary, RL ASMs had less variability and obtained better results. Especially, R_{state} visited a larger amount of new abstract actions while also discovering more unvisited actions than any other ASM.

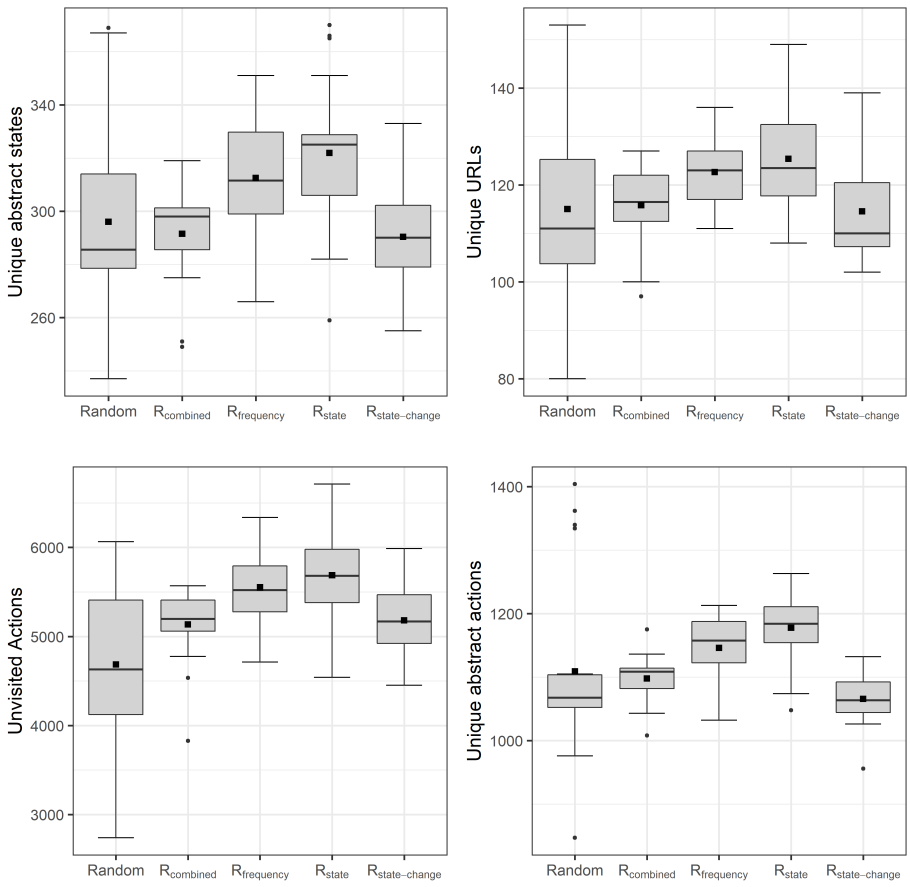


Figure 5.2: Exploration performance of Shopizer

Figure 5.3 shows that Random ASM visits more unique abstract actions and executes more unique actions for Craigslist. Table 5.4 indicates that there is a significant statistical difference between Random and R_{state} or $R_{frequency}$ with regards to Abstract State exploration. This may be due to Craigslist's multiple search options available in most states, resulting in a vast set of possible actions to execute. Conversely, RL ASMs repeat many actions already executed to learn

from the experience. Further research is needed to improve abstraction to handle multiple search options or increase the testing time to train the RL agent.

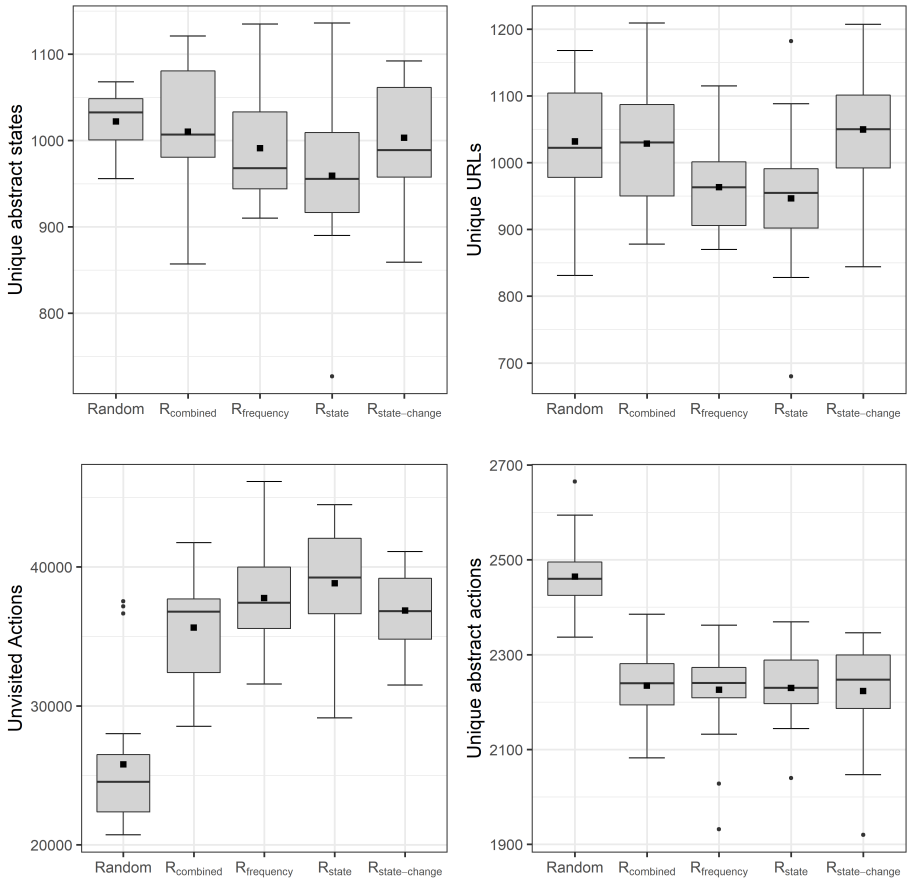


Figure 5.3: Exploration performance of Craigslist

To verify if this is also the case for Craigslist, a single run of 100 test sequences, each consisting of 100 actions, was executed using Random and $R_{combined}$. Table 5.5 shows the results for state exploration. Both ASMs reached a similar number of abstract states, while $R_{combined}$ visited considerably fewer concrete states than

Random as a sign of better exploration. The model generated can be used in future test sessions or in different versions of the same SUT. Theoretically, the RL agents will need less execution time to reach the same exploration level.

Table 5.5: State exploration after 10000 actions

ASM	Abstract States	Concrete States	URL coverage
Random	3274	6479	1136
$R_{combined}$	3166	4576	1301

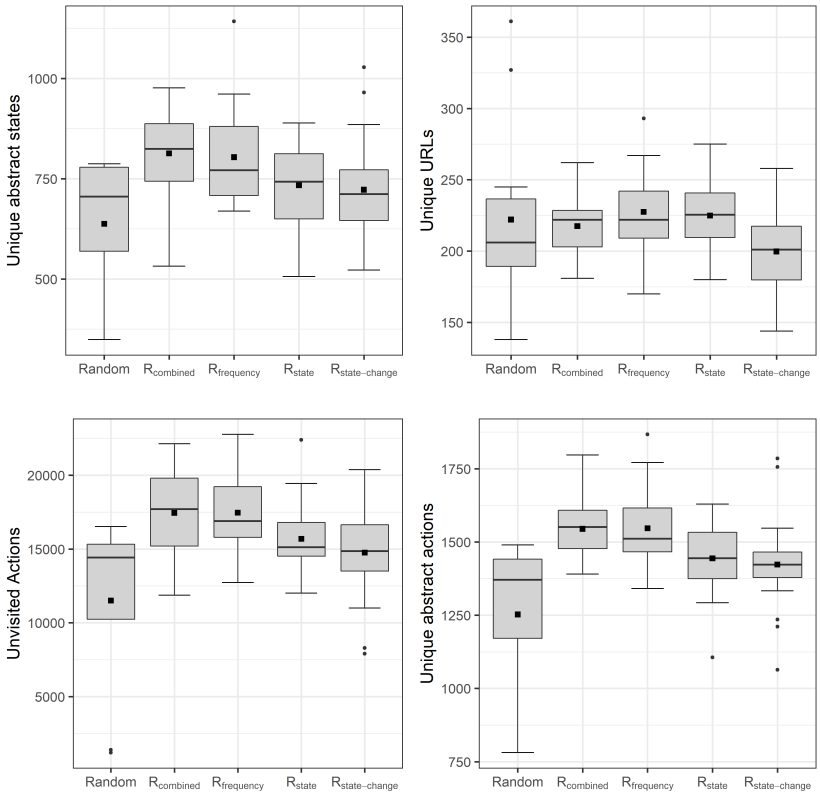


Figure 5.4: Exploration performance of Bol.com

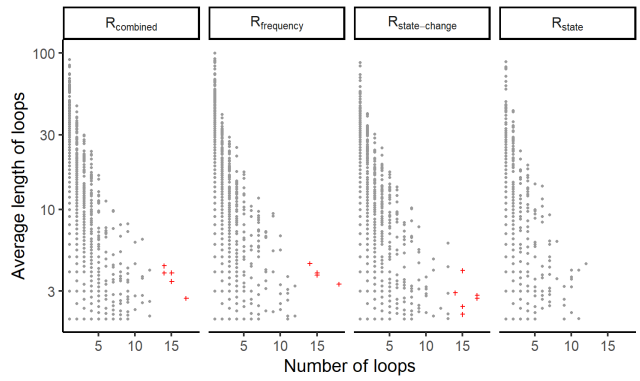
For Bol.com, Figure 5.4 indicates that the RL approaches generally outperformed Random. In particular, $R_{combined}$ outperformed R_{state} and $R_{state-change}$ in terms of search space exploration. Table 5.4 shows that there is a statistical difference at a significant level for the exploration of new abstract states.

However, there are no significant differences in terms of URL coverage. An analysis of the extracted URLs during the executions revealed that most URLs include multiple search parameters. Since this is an e-commerce site, new URLs are obtained after accessing a product view or refining the product search on the listing views. This highlights the importance of domain-specific preprocessing, such as URL normalisation, to distinguish meaningful navigational actions from parameter variations.

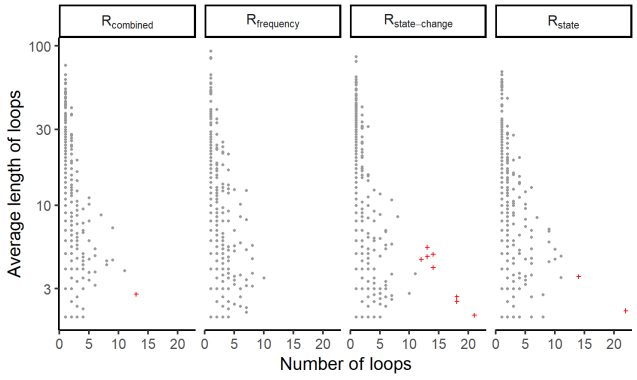
RQ1 answer: *State-based reward (R_{state}) generally exhibited superior performance in exploring diverse areas of the SUTs, as indicated by metrics like abstract state discovery and action coverage. However, the frequency-based reward ($R_{frequency}$) proved to be effective in promoting balanced exploration across more complex and interconnected state spaces.*

5.5.2 RQ2: JBS Problem

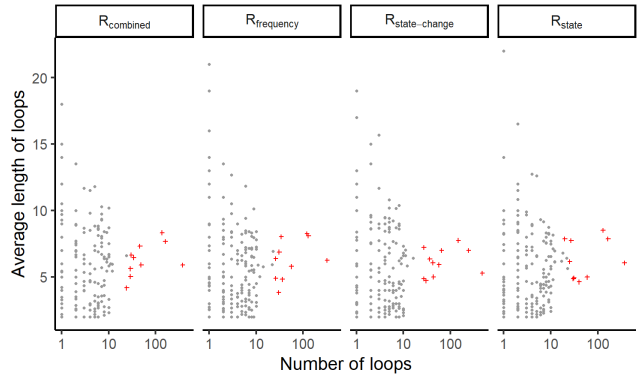
Intuitively, jumping between states happens when states appear multiple times with short sequences of actions between each occurrence. However, the length and frequency required to classify a state as a jumping state are hard to predict for every SUTs. Loops were extracted to analyse the distribution of jumping states, with a focus on the initial state and loop length. Figure 5.5 shows the relation between the number of times a state starts a loop and the average size of those loops. The pattern is consistent across ASMs. The interesting sections rely on the bottom right of every chart: states (represented as red dots) with many loops of small size. DBSCAN's outlier detection was applied for all rewards across the three SUTs. Each point in the data represents a state-loop pair, characterised by the average loop length and the number of loops.



(a) Craigslist



(b) Bol.com



(c) Shopizer

Figure 5.5: Distribution of states starting loops and their mean loop length

Figure 5.5a presents the clustering for Craigslist. $R_{state-change}$ and $R_{combined}$ seem to have a higher accumulation of points in this area, while R_{state} contains no problematic states.

In Figure 5.5b, Bol.com's clustering results reveal fewer outliers overall despite its complex navigation requirements. Similar to Craigslist, $R_{frequency}$ tends to have fewer loops per test sequence in Bol.com, with the states being visited fewer times. $R_{state-change}$ again shows a higher number of potential jumping states, while R_{state} and $R_{combined}$ appear to balance exploration and revisitation, leading to a moderate number of outliers. Additionally, the graphs show fewer loops for Bol.com. This could be an important factor in explaining the better performance of the rewards in this web application.

Figure 5.5c shows the clustering results for Shopizer, which contains a more significant presence of jumping states, and in general, all the loops are of small size. There is no noticeable difference between the rewards. This can be explained by the nature of Shopizer, which has smaller state space and simple navigability to access the main parts of the SUT. This result is expected: R_{state} is a reward based solely on the reached state, not depending on any characteristic of the consecutive states.

To further evaluate the incidence of the JBS problem in each ASM, the number of jumping states was statistically analysed using the Mann-Whitney U test. The goal is to measure if there is any statistical difference between the reward $R_{state-change}$ and the other three rewards, respectively. Table 5.6 summarises the p-values calculated for this comparison for Craigslist and Bol.com. $R_{state-change}$ is significantly different from $Rstate$ and $R_{frequency}$ ($p - value < 0.05$). $R_{state-change}$ tends to visit the same states more frequently with fewer actions in between. The hypothesis for $Rcombined$ in Craigslist could not be rejected. The weight parameter could give too much influence to the state-change reward within the combined reward.

Table 5.6: p-values calculated between every reward and $R_{state-change}$

	$R_{combined}$	$R_{frequency}$	R_{state}
Craigslist	0.48	$p < 0.001$	$p < 0.001$
Bol.com	0.03	$p < 0.001$	0.03

RQ2 answer: *The frequency-based reward ($R_{frequency}$) was most effective in mitigating the JBS problem, reducing the prevalence of states involved in short, repetitive loops. The reward based on state-change ($R_{state-change}$) exacerbated the issue by rewarding transitions between a limited set of states. In contrast, the state-based reward (R_{state}) showed resilience against this problem in most cases.*

5.6 Discussion

The analysis demonstrated that the state-based reward tends to excel in environments where exploration breadth is critical, as it prioritises the discovery of new states. Conversely, the frequency-based reward promotes a more uniform exploration. In contexts where navigation complexity is high, such as e-commerce platforms, a frequency-based reward provides a balance between revisiting key states and discovering new regions.

The JBS problem arises when rewards incentivise behaviour that limits the exploration potential, leading to frequent transitions between a small subset of states. Frequency-based strategies discourage such behaviour by penalising revisitation patterns, enabling more balanced exploration. In contrast, rewards based on state differences, if not calibrated, risk reinforcing the problem, even in large state spaces.

Several factors might affect the incidence of the JBS problem. Web applications with a high probability of accessing previously visited states can present this problem regardless of the selected reward. This can be observed, for example, when the state space is small, when there is a bottleneck to access other unvisited parts of the SUT or when there is high interconnectivity between the main states. This is the case with Shopizer, where new pages are only accessible through a button that is not always visible, and the state space is small.

5.7 Threats to Validity

In this study, several threats to validity were identified and addressed to ensure the reliability and applicability of the results. These threats are categorised into four main types, following the guidelines proposed by [114].

5.7.1 Internal Validity

Although multiple runs (20 repetitions) were conducted to mitigate randomness, the selection of specific web applications might inherently favour certain exploration strategies over others. Three web applications with varying complexities and navigation structures were chosen to reduce this threat.

Furthermore, the choice of RL parameters can significantly influence the agent's performance. While default values were selected based on related work, different configurations might yield different results. Future studies could explore the sensitivity of the outcomes to these parameters.

Regarding TESTAR's configuration, the state and action abstraction method plays a crucial role in how states (and actions) are represented and differentiated. An inappropriate abstraction could lead to misleading state representation, as observed in Chapter 4, affecting the learning process. Although this abstraction was selected after several trials during the Implementation Phase (see Figure 3.10), alternative abstraction methods might produce different exploration behaviours.

5.7.2 External Validity

Only three web applications were evaluated, each with its unique characteristics. While they represent a range of complexity, the results may not generalise to all types of web applications, especially those with highly dynamic content. Future research could include a broader spectrum of applications to enhance generalizability.

Moreover, the integration of Q-Learning with TESTAR is tailored to its architecture and state modelling capabilities. Ensuring modularity in the implementation allows for easier adaptation and testing across different tools.

5.7.3 Construct Validity

The study used metrics such as the number of abstract states visited, actions executed, and URL coverage as proxies for exploration effectiveness. While these metrics provide valuable insights, they might not fully capture the qualitative aspects of exploration, such as the relevance of the visited states.

The methodology for detecting the JBS problem relied on loop detection and clustering using DBSCAN. The choice of parameters and features might influence the identification of outliers. Alternative clustering techniques or parameter settings could yield different interpretations of what constitutes a jumping state. To mitigate this threat, parameter tuning was performed on the kNN distance plot to objectively identify optimal clustering thresholds.

5.7.4 Conclusion Validity

Each ASM was executed 20 times to account for randomness. While this number provides a reasonable balance between computational feasibility and statistical power, larger sample sizes might offer more robust estimates and detect smaller effect sizes. Future experiments could increase the number of repetitions to enhance statistical reliability.

The implementation details of the Q-Learning algorithm, including the calculation of the rewards and the application of the policy, could introduce inconsistencies. Ensuring that the algorithm is correctly implemented and that all ASMs are evaluated under identical conditions is crucial for drawing valid conclusions. Rigorous testing and validation of the implementation were performed to minimise this threat.

The rewards were tailored based on existing state models. The ASMs may have performed well on the selected SUTs but might not generalise to unseen applications or different testing scenarios. The study acknowledges this and recommends validating the approach across additional and more diverse SUTs in future research.

5.8 Conclusions

A *Q*-learning approach for automated GUI testing was presented, utilising rewards based on the state model of web applications. This is an understudied yet valuable area for understanding an agent's behaviour when executing actions to attain rewards. This approach offers a unique advantage in this domain.

An experiment was conducted on three complex web applications to evaluate the performance of four rewards, using Random as a baseline for comparison. The results showed that frequency, state, and combined rewards had the best performance for state exploration. The state-change reward, however, was outperformed by or did not improve upon the results of the other rewards. Additionally, a solution to the Jumping Between States (JBS) problem was proposed by incorporating memory information into the reward. The results demonstrated that rewards based solely on the reached state effectively circumvent the JBS problem. However, in domains with high connectivity between states, where most pages are accessible within a few clicks, the JBS problem is more prevalent, regardless of the reward. Nevertheless, state or frequency rewards performed better in web applications with complex navigability.

6 Applying it at a company: Marviq

"Software never was perfect and won't get perfect. (...) The missing ingredient is our reluctance to quantify quality."

Boris Beizer, Software testing techniques

The increasing reliance on complex web applications demands robust software testing practices to prevent bugs that could cause user dissatisfaction, data breaches, and reputational harm [63]. For companies, testing at the GUI level is essential, as it provides insights into the customer experience. However, manually executing GUI tests is resource-intensive and error-prone, particularly in regression testing, prompting a shift toward automation.

The industry case-based studies discussed in Chapter 3 have demonstrated that scriptless GUI testing complements traditional scripted testing techniques. Despite these advantages, industrial adoption remains limited.

In collaboration with the private company Marviq [1] and under the European IVVES (Industrial-grade Verification and Validation of Evolving Systems) project [2], several critical needs were identified influencing the adoption of scriptless testing

[1] Official website: https://marviq.com/
[2] Official website: https://www.ivves.eu

in industrial settings:

1. the optimisation of test session length to balance coverage and time efficiency.
2. the evaluation of random GUI testing as a complement to existing manual testing processes.
3. the introduction of code smell coverage to address maintainability and technical debt.
4. the assessment of correlations between code smell coverage and traditional coverage metrics to identify testing gaps.

To address these needs, it is essential to consider the breadth of code exercised during testing and the underlying quality of the code being tested. Traditional metrics, such as line, branch, and complexity coverage, have been widely used to measure how much of the code is covered during testing. While these metrics offer valuable insights, they have long been debated for their limitations in fully capturing the quality of testing [164,165]. High code coverage can be deceptive, as it does not guarantee thorough testing or the detection of more subtle defects, leaving critical areas of software quality unaddressed.

This chapter will explore the use of code smells as a metric for evaluating traditional coverage metrics within the context of an industrial web application. Code smells indicate potential issues that may lead to maintainability problems and hidden bugs [166]. Covering these smells during testing can reflect the ability of the testing tool to detect deeper quality issues. While static analysis identifies potential smells, dynamic testing ensures these smells are encountered during real application use, increasing confidence in addressing areas that may contribute to defects and boosting confidence in overall software quality.

This study uses SonarQube [167] for code smell detection to assess the impact of test sequence length on coverage and investigates correlations between code smells and traditional coverage metrics. Additionally, this work compares the complementarity of scriptless testing with Marviq's existing manual testing process. This collaboration brings significant relevance to this research, as it adds

a real-world and practical component and another industrial validation of scriptless GUI testing needed for case study generalisation through the architectural analogy presented in Chapter 3. The contribution of this chapter is threefold:

1. An empirical study to analyse the influence of test sequence length on traditional coverage metrics.

2. The use of known code smells in an industrial application for evaluating the effectiveness of traditional coverage metrics in exploring a system.

3. A comparison of random scriptless testing with Marviq's manual testing process to demonstrate their complementarity and the potential of scriptless testing.

This study offers insights for software testing professionals and researchers interested in expanding traditional coverage metrics to include aspects of code maintainability. By integrating code smell detection, this work contributes to the ongoing development of testing techniques that address both functionality and software quality. Moreover, new research directions could emerge in software testing, code quality, and the relationship between GUI testing and code smells.

The chapter is structured as follows. Section 6.1 presents the state of the art in random scriptless GUI testing, adequacy metrics and code smell analysis. Section 6.2 describes the industrial context of this study. Section 6.3 describes the experiments with random testing. Section 6.4 shows the results and answers to the research questions. Section 6.5 discusses Marviq's perspective of the findings, while Section 6.6 presents the validity threats and mitigation actions. Finally, Section 6.7 concludes the work.

6.1 Related Work

This section presents relevant studies that provide insights into random scriptless GUI testing, test adequacy metrics, and the relationship between testing and code smells.

6.1.1 Random Scriptless GUI testing

Although random scriptless GUI testing has been shown to effectively identify a range of faults, its success largely depends on how the randomisation parameters are configured. Recent studies demonstrate that test outcomes can be significantly influenced by settings such as the length of the test sequences, the state abstraction (as observed in Chapter 4), and the stopping criterion [168,169].

However, improving the effectiveness of random GUI testing requires careful adaptation of the random strategy to overcome specific challenges presented by GUI components, such as blocking GUI (i.e., GUIs that need specific user interactions to be unlocked). To address this, recent work has explored the use of novel techniques to enhance the ability of random agents to navigate and test sophisticated interfaces, improving overall testing adequacy [170].

Furthermore, a body of research compares random testing with manual testing approaches. These studies highlight that random and manual techniques are complementary: while manual testing is capable of covering parts of the code that random testing might miss, random approaches can explore unexpected interactions and pathways that manual testers may overlook. This complementarity suggests that a hybrid testing strategy, leveraging both methods, can offer improved coverage and fault detection capabilities [64,171].

6.1.2 Test adequacy metrics

Code coverage has been widely used in the literature to evaluate the quality of testing by relating coverage to the test effectiveness [172], [173], [174], [175].

The most commonly used coverage criteria for GUI testing include:

- *Line of Executable Code Coverage (LC):* percentage of executable lines of code covered by an execution of random scriptless GUI testing session. An executable line of code is considered covered when it is executed by the random scriptless GUI testing session.

- *Statement Coverage (SC):* percentage of executable statements that have been executed, focusing on individual operations regardless of how they are

arranged on lines of code. Multiple statements can be on a single line, or one statement can span multiple lines.

- *Instruction Coverage (IC):* percentage of bytecode instructions that have been executed during testing.
- *Branch Coverage (BC):* percentage of branches in the code that were executed at least once during testing. A branch is a decision point in the code where the program can take different paths based on a condition.
- *Complexity Coverage (CoC):* percentage of the code's cyclomatic complexity that has been tested. Cyclomatic complexity measures the complexity of a program's control flow by counting the number of independent paths through the code.
- *Method Coverage (MC):* percentage of methods (or functions) in the codebase that have been executed at least once during testing.
- *Class Coverage (ClC):* percentage of classes in the codebase that have been instantiated or had their static members accessed during testing.
- *State Space Exploration (SSE):* number of distinct states explored during testing.
- *Action Space Exploration (ASE):* number of distinct actions executed during testing.

Choudhary et al. [176] conducted a comprehensive comparison of Android GUI testing tools, including Monkey, using line of executable code coverage. Similarly, Wang et al. [177] compared various automated Android testing tools, focusing on method coverage, activity coverage and fault detection. Branch coverage was used as a primary metric when comparing random testing with a search-based test data generation study for web applications in [178]. Van den Brugge et al. [179] assessed effectiveness using instruction, branch, and accumulated instruction and branch coverage in Java applications. Likewise, in [180], line and statement coverage were used to propose a framework to evaluate the effectiveness of different

Android GUI testing tools. Recently, Collins et al. [181] examined the effectiveness of a reinforcement learning testing approach for Android using instruction, branch, line and method coverage. Additionally, in [182], an image-based GUI testing approach was empirically evaluated in Android and Web applications using line coverage and branch coverage.

Thus, coverage metrics act as surrogate measures of the thoroughness of testing efforts. Nevertheless, there is also evidence in the literature arguing that coverage criteria alone are not a sufficient indicator of test quality [183], and that new solutions should be explored to improve test quality by looking beyond just code coverage [184].

Memon in [185] recognises that code coverage metrics do not address problematic interactions between the GUI user events and the application and argues that coverage criteria for GUI testing require new perspectives. Subsequently, in [186], an empirical evaluation to analyse the impact of the test suite size on the capability to detect faults is presented, showing that larger test suites identify more faults previously seeded in toy projects. This conclusion is straightforward since high coverage means that the system has been explored more deeply, and then the quality of the test process should improve to detect faults. However, this conclusion raises the question of which of the traditional coverage metrics is more likely to be a good indicator of test quality in real projects.

6.1.3 Code Smells

Code smells [70] refer to indicators of design flaws or issues in source code that can lead to future problems. Consequently, finding these smells during testing can reflect the ability of the testing tool to cover code with deeper quality issues, such as code smells. Most of the existing research on code smells focuses on prioritisation [187–189], filtration [190], and the code smells-faults correlation [191–193].

Spadini et al. [194] examined how the presence of code smells affects the coverage of test suites, revealing that classes with code smells tend to have lower test coverage. Bavota et al. [195] studied the relations between quality metrics, or the presence of code smells, and refactoring activities performed by

developers. Results highlighted that only 7% of the refactorings performed on classes affected by smells actually removed those smells. These findings indicate that developers focus on mitigating the problem without necessarily removing completely the code smell.

While these studies have made important steps in linking code smells to test quality, a gap remains in assessing how the different traditional coverage metrics relate to code smells.

Based on existing research, this study is the first to assess the power of traditional coverage metrics to predict testing quality by using known code smells in an industrial application. This approach offers a new perspective on GUI testing effectiveness, bridging the gap between testing and code quality assessment.

By applying TESTAR, this work builds on scriptless testing theory using the architectural analogy and strategies from [196]. This work provides a deeper explanation of the testing effectiveness of the industrial application of scriptless testing presented in [197], contributing this way to the generalisation of the findings.

6.2 Industrial case

Marviq is a software development company offering Team as a Service, Software Development as a Service and IoT development, with a focus on integrating skilled professionals into client teams and managing entire development projects. Marviq is a small company with 35 professionals who work with agile practices on typically eight concurrent development projects while serving 25 clients.

As these projects are tailored to the client's needs, Marviq applies a tailor-made Quality Assurance (QA) process consisting of the following steps: (1) conduct a workshop with the client for business alignment and scoping; (2) develop the MVP (Minimum Viable Product) as a prototype of the project; (3) develop the product and environment following agile practices based on SCRUM [46]; (4) and provide support channels to the client when the project is released. It is important to mention that QA for small companies like Marviq faces several challenges [47], [48], such as unclear requirements, the illusion that the prototype is

the final product, mapping existing software to new business process mismatch or running out of time for testing.

The selected SUT is Yoho [3], a digital solution developed by Marviq to enhance operations and communications within industrial environments. Yoho offers functionalities such as alert and notification management, task handling, work instructions and enhanced communication tools (see Figure 6.1). Yoho is a software as a service (SaaS) platform with the typical web application functionality.

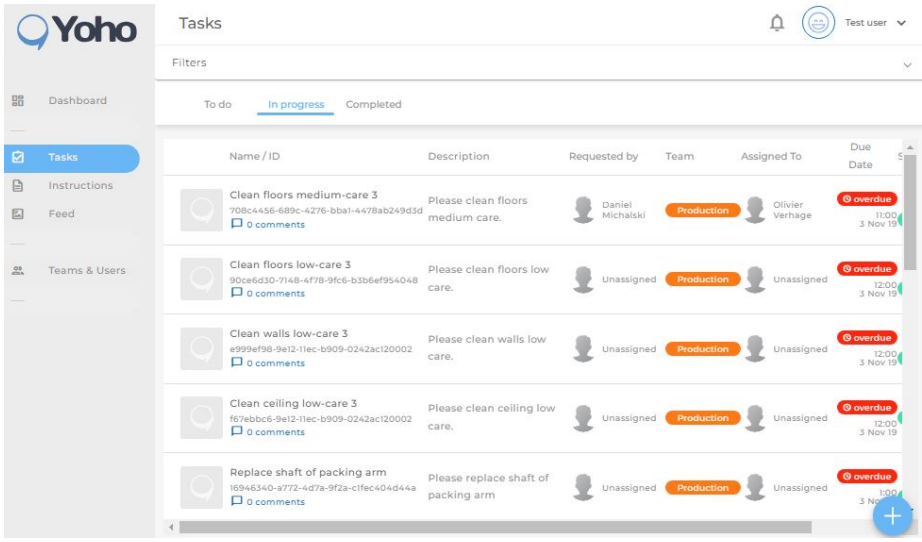


Figure 6.1: Excerpt of Yoho SUT.

The Yoho Software as a Service (SaaS) platform initially began as a well-defined minimum viable product. However, market circumstances caused the product scope to rapidly change with each potential customer, resulting in a system more akin to a prototype than a functional product. At this stage, Marviq was brought in to adopt the project and transform it into a market-ready product. With the first customers in sight, the product focus became clear, and development resumed a straightforward path towards a production-ready state, resulting in the product as it stands today.

[3] Yoho showcase: https://marviq.com/our-showcases/yoho-factory-management-platform/

At its core, Yoho has been designed with highly configurable options and a role-based access mechanism to support future requirements and customer-specific demands. The design includes interaction units tailored by roles, which provokes that while executing tests a specific role and customer would result in a relatively low percentage of code coverage as not all functionality would be revealed for this user.

Table 6.1 presents an overview of the size of Yoho. As can be observed, the metrics presented are representative of a real-world application. Additionally, this SUT exposes relevant challenges, such as the dynamism of modern web applications (i.e., dynamic identifiers for the GUI widgets).

Metrics	Yoho
Java Classes	569 (709 incl. interfaces)
Methods	3033
SLOC	25099
LLOC	9059
Branches	1622
Instructions	37180
Cyclomatic Complexity	3856

Table 6.1: Overview of the size of Yoho

Marviq uses SonarQube to identify code smells in the Yoho application. This information provides a rich basis for evaluating the performance of different coverage metrics to test the parts of the code that may be causing problems. Nevertheless, Marviq faces several challenges in ensuring effective software testing while managing limited resources. To address these challenges, the company identified the need to explore random testing, manual testing, and new coverage metrics to enhance testing efficiency and code quality. Below are the specific needs that motivated the study.

(a) *The Need to Conduct Random Testing with Different Session Lengths:* The company needs to optimise testing resources to ensure the process is effective and efficient. Since testing resources are limited, finding the optimal session length that balances **coverage** and **time** is important. Testing sessions that are

too short may miss critical issues, while longer sessions may be inefficient. By experimenting with different session lengths, the company aims to **identify the best trade-off** between test coverage and resource expenditure. This is especially important in environments with **rapid development cycles** like Agile, where the ability to quickly adapt testing to available time windows is critical.

(b) *The Need for a New Coverage Metric: Code Smell Coverage:* The company aims to ensure not only functional correctness but also long-term **high code quality** and maintainability. Traditional metrics like code or method coverage focus on functionality but do not fully capture the maintainability and readability aspects of the codebase. Introducing **code smell coverage** addresses the need to track potential **technical debt** that could accumulate unnoticed. This metric helps the company ensure that, while functional coverage may be high, the code remains maintainable and scalable, reducing the risk of future issues as the software evolves.

(c) *The Need to Assess Correlations between Code Smell Coverage and Traditional Metrics:* The company seeks to understand whether traditional metrics like code and method coverage correlate with **overall code quality**, as high coverage does not guarantee well-structured or maintainable code. By exploring the relationship between code smell coverage and traditional metrics, the company aims to **identify gaps** in its testing process. Low correlation would suggest traditional testing may not fully address long-term maintainability concerns. This insight can help the company **develop a more holistic testing approach**, including both functional correctness and code quality to ensure robust, maintainable software.

(d) *The Need to Compare Random Testing with Manual Testing:* Given the limited resources for manual testing, there is a need to evaluate whether **random testing can complement** the existing manual testing processes. Manual testing is labour-intensive and expensive, so the company seeks a solution that can **reduce the time and cost** associated with it. By comparing the two approaches, the company aims to determine if random testing can identify different types of issues that manual testers may miss. The ultimate goal is to **enhance test coverage** while reducing the burden on manual testers, allowing them to focus on more critical scenarios.

6.3 Experiment Design

An experiment to explore the application of random testing on an industrial web application was conducted, addressing the industrial needs discussed in Section 6.2. Specifically, the study focuses on optimising testing resources, assessing random testing's complementarity to manual testing, and introducing innovative metrics like "code smell coverage" to monitor code quality and maintainability. The following three research questions were formulated along with their rationales to achieve this goal.

RQ1: How do the number and length of random scriptless GUI testing sequences impact the coverage of testing adequacy metrics?

This question investigates ***Need 1***. The company aims to determine how the number and length of random scriptless GUI testing sequences influence testing adequacy metrics. Understanding this relationship will help optimise the testing process by balancing thoroughness with resource limitations in Agile environments with tight testing cycles.

RQ2: How do traditional coverage metrics (e.g., code and method coverage) relate to code smell coverage?

This question addresses ***Need 2 and Need 3***. The company seeks to examine the relationship between traditional coverage metrics and new metrics, like code smell coverage, to determine how well they align in assessing overall code quality.

RQ3: How can random testing complement or reduce the reliance on manual approaches?

This question tackles ***Need 4***. The company aims to assess whether random testing can supplement or reduce the need for manual testing.

The experiment was designed following the guidelines proposed by Wohlin [114]. Moreover, this study follows a methodological framework [198] specifically designed to evaluate testing tools in order to encourage future secondary studies. The experiment is described in the following subsections.

6.3.1 Independent and Dependent Variables

The independent and dependent variables were defined as follows to address the research questions.

Independent Variables: The independent variables refer to the parameters used to configure the random scriptless GUI testing tool. These variables include:

- *Number of random testing sequences:* the total number of random test sequences executed.
- *Number of GUI actions per sequence:* the number of actions executed within each test sequence.
- *Time delay between actions:* the time interval (in seconds) between two consecutive random actions.
- *Action duration:* the time (in seconds) taken for each GUI action to complete.
- *State abstraction:* defined by the properties of the widgets used to represent the state of the system under test (SUT).
- *Initial sequence needed:* for example, to pass a login screen.
- *Form filling enabled:* to fill detected forms with meaningful data.

Additionally, the parameters used for detecting code smells are treated as independent variables.

Dependent Variables: To answer the research questions, several traditional coverage metrics were measured, including *Line Coverage* (LC), *Instruction Coverage* (IC), *Branch Coverage* (BC), *Complexity Coverage* (CoC), *Method Coverage* (MC), and *Class Coverage* (ClC). In addition, the following variables were defined to analyse coverage within the state models and to quantify code smells:

- *Abstract State Coverage (AbSC):* The number of abstract states covered in the state model during testing.
- *Abstract Transition Coverage (AbTC):* The number of transitions covered in the abstract state model.

- *Concrete State Coverage (CoSC):* The number of concrete states covered in the concrete state model.

- *Concrete Transition Coverage (CoTC):* The number of transitions covered in the concrete state model.

- *Code Smell Coverage (CSC):* The number of unique code smells encountered. A code smell is considered "covered" when the Java method containing it is executed at least once during testing.

- *Code Smell Occurrences (CSO):* The total number of code smell instances triggered during testing, including multiple occurrences of the same code smell.

6.3.2 Experimental Setting

To carry out the experiment, both the TESTAR testing tool and the SonarQube static analysis platform were configured to suit the needs of this study, ensuring an effective evaluation of test coverage and code quality metrics.

TESTAR Configuration: For testing the Yoho application using TESTAR, several key configurations were implemented to optimise the testing process. First, the SUT was specified by defining Yoho's URL and establishing the necessary login procedures. This ensured that TESTAR could consistently access and interact with the application. The blocking principle [114] was applied to focus on exploring the SUT, turning off TESTAR's oracles to prevent test interruptions.

For widget identification, TESTAR was configured to consider attributes such as name, ID, control type, and text content. In cases where clickable elements were defined by CSS classes instead of standard attributes, clickability was manually configured to ensure accurate testing. State abstraction (*SA*) consisted of the WebWidgetId, WebWidgetName, WebWidgetTextContent, and WidgetControlType properties. The action abstraction strategy followed TESTAR's default configuration. Additionally, certain actions, such as logging out or file uploads, were excluded to avoid interactions outside the test scope.

A mandatory login sequence was defined to execute at the start of each test run, using consistent credentials across all tests to ensure a uniform starting point. Based on preliminary trials, time parameters were optimised, with the action duration set to 0.5 seconds and the delay between actions to 0.8 seconds.

Additionally, a BTrace[4] server was integrated alongside TESTAR. BTrace allows real-time instrumentation of Java methods without modifying the source code or interrupting the normal execution of the application. BTrace intercepted and logged method calls triggered by GUI actions while operating in a separate environment. Each traced method (along with its class name, timestamp, and relevant parameters) was stored in a database. This information was cross-referenced with known code smells, enabling us to track and measure variables such as CSC and CSO, as described in Section 6.3.1.

SonarQube Configuration: SonarQube [69] was used to perform static analysis of the Yoho codebase, identifying code smells and other violations. SonarQube classifies violations by severity: Blocker, Critical, Major, Minor, or Info. In Yoho's analysis, SonarQube detected a total of 173 code smell instances, categorised as shown in Table 6.2 using both Fowler's [70] original classification of code smells and a more recent classification system [199].

Most detected code smells were categorised as Object-Orientation Abusers. Conditional Complexity was the most frequent, suggesting a need for better adherence to object-oriented design patterns in the Yoho codebase. Although only one security-related issue was found, it was classified as Critical. This analysis provided valuable insights, allowing us to assess both the prevalence and severity of code smells in relation to the executed test sequences. Furthermore, code smells in comments and dead code were excluded from the study, as they are not executable, to ensure accurate coverage analysis and responses to the research questions.

[4]Source code available at: https://github.com/btraceio/btrace

Table 6.2: Code Smell Classification and Severity

Code Smell	Type	Critical	Major	Minor
Bloaters (26)	Data Clumps	1	0	0
	Long Parameter List	0	11	0
	Primitive Obsession	1	11	2
Couplers (11)	Indecent Exposure	0	11	0
Dispensables (29)	Comments	12	3	0
	Dead Code	0	8	0
	Lazy Class	0	0	1
	Speculative Generality	0	4	1
Lexical Abusers (3)	Inconsistent Naming	0	0	3
Obfuscators (8)	Clever Code	0	1	3
	Inconsistent Style	0	0	4
Abusers (95)	Conditional Complexity	0	70	0
	Refused Bequest	0	3	20
	Switch Statements	0	0	1
	Temporary Field	0	1	0
Security (1)	Vulnerability	1	0	0
Total (173)		15	123	35
Total excluding comments and dead code (150)		3	112	35

6.3.3 Experimental Procedure

The experiment was designed with three different configurations of test processes consisting of 10,000 actions, i.e. TP100, TP500 and TP1000. Table 6.3 shows the details of these configurations. Moreover, the best configuration was selected (that turned out to be TP500 for the answer to RQ1 in Section 6.4.1) to run it with the advanced *form-filling* feature of TESTAR, to conduct the comparison with manual testing for RQ3. Table 6.3 also shows the details for this enhanced configuration (TP500Forms).

TESTAR's *form-filling* feature automatically populates forms with data. As the scriptless tool randomly navigates through the different states of the SUT, it automatically detects forms. Once a form is identified, TESTAR generates an XML file with each key representing an editable widget within the form, and the corresponding value is an automatically generated input.

The generated input data type depends on the widget type (e.g., random text

Table 6.3: Test Process Configurations

Variable	TP100	TP500	TP1000	TP500Forms
Test sequences	100	20	10	20
Actions per sequence	100	500	1000	500
Time delay (s)	0.8	0.8	0.8	0.8
Action duration (s)	0.5	0.5	0.5	0.5
State abstraction	*SA*	*SA*	*SA*	*SA*
Login sequence	yes	yes	yes	yes
Form-filling	no	no	no	yes

for text fields or a valid email address for email fields). The XML file can contain multiple input sets for the same form, each with an associated weight indicating the likelihood of selection. These XML files can be manually edited to replace or add specific values and weights, enabling the form to be tested with varied data combinations based on assigned probabilities.

Example 6.1 illustrates an automatically generated XML, representing a simple form containing two fields: a description and an email. The XML file contains two input sets with different values for the form fields, and equal weight values.

Example 6.1: Example of XML form

```
<form>
  <data>
    <description>RandomText1</description>
    <email>first.email@example.com</email>
    <weight>50</weight>
  </data>
  <data>
    <description>RandomText2</description>
    <email>second.email@example.com</email>
    <weight>50</weight>
  </data>
</form>
```

During the configuration of TESTAR, 23 forms were automatically identified

within the SUT, ranging from one to six fields in complexity. Two distinct input profiles were created for each form to thoroughly test these forms: one with baseline values across fields and another with varied, alternative values to cover broader data cases. The edition and small-scale testing (during the Implementation Phase, as described in Chapter 3) of the 23 respective XML files took one working day (8 hours). With this functionality added to TP500Forms, if TESTAR detects a form during testing, a *form-filling* action will be added to the list of available actions in this state. If selected randomly, one of the input profiles defined in the XML file will be chosen based on its weight.

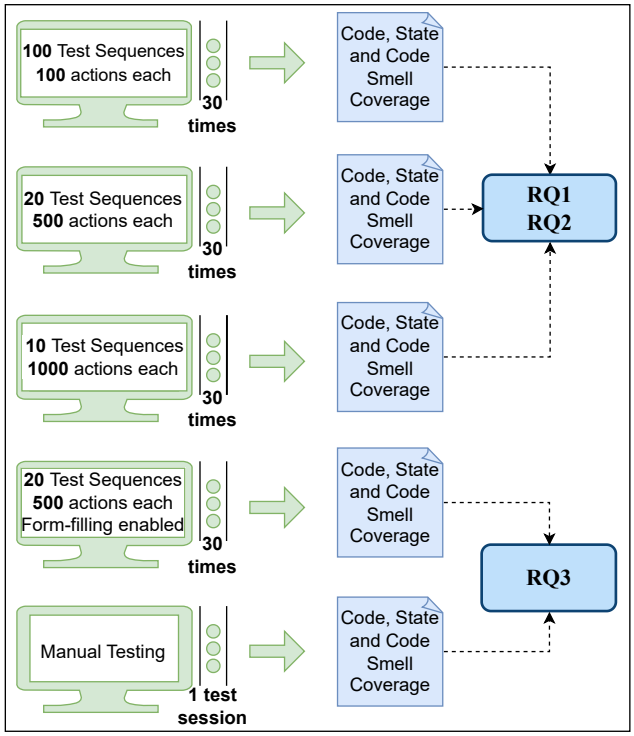


Figure 6.2: Overall experimental design architecture.

Figure 6.2 presents an overview of the experimental design. Each configura-

tion was repeated 30 times to deal with randomness. TESTAR was configured to restore the initial state of the SUT after each sequence. This setup was used to evaluate the influence of sequence length on coverage metrics and the relationship of code smells to traditional coverage metrics. Manual testing was performed by an experienced tester with prior knowledge of the Yoho application. The tester explored the application thoroughly during a manual test session that lasted one working day.

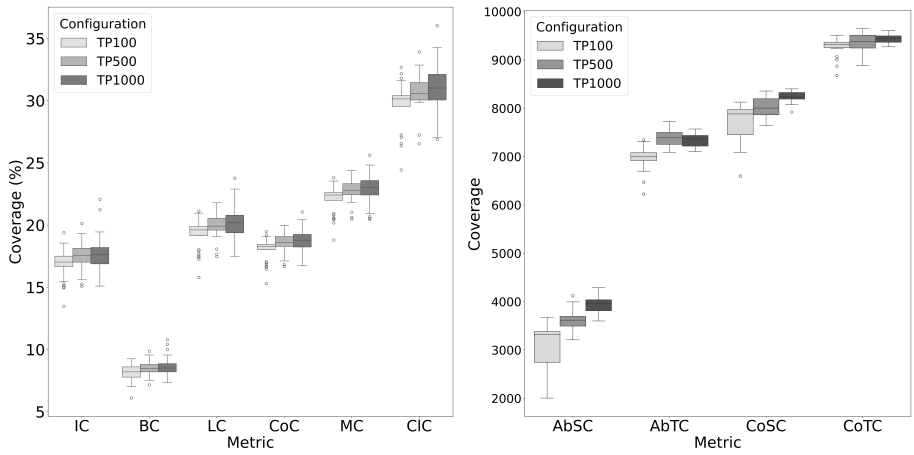
6.4 Results

This section presents the results obtained to understand the influence of sequence length on traditional test adequacy metrics, the relationship between code smells and traditional coverage metrics, and the comparison of random with manual testing.

6.4.1 RQ1: Number and length of test sequences

Figure 6.3a presents the box plot graphs comparing the different traditional coverage metrics across the three test runs. The box plots reveal a consistent trend across all metrics, with coverage generally increasing from TP100 to TP1000, though the magnitude of the increase varies by metric. Instruction Coverage (IC) and Branch Coverage (BC) show relatively lower percentages with minimal variation across test processes. Line Coverage (LC) and Complexity Coverage (CoC) show moderate coverage with slightly more variability. In contrast, Method Coverage (MC) and Class Coverage (ClC) show the highest coverage levels and the most noticeable differences across configurations. Notably, TP1000 consistently achieves higher median coverage and often larger variability, particularly for MC and ClC. Several metrics, especially ClC, show outliers, indicating instances of exceptionally high or low coverage in some test runs.

For the state coverage metrics, Figure 6.3b illustrates how test runs with longer sequences lead to significantly better coverage of both abstract and concrete states and transitions.



(a) Code Coverage **(b)** State Model Coverage

Figure 6.3: Distribution of coverage metrics.

Figure 6.4 shows the distribution of unique code smells covered by each configuration. Similarly, the data suggest a trend towards higher code smell coverage with test processes featuring longer sequences.

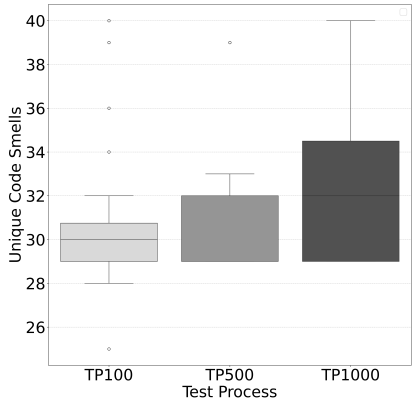


Figure 6.4: Distribution of code smell coverage

A detailed analysis revealed that 40 code smells were covered by at least one run in each test process. Although TP1000 covered more code smells on average, three smells were never covered by this test process. TP100 and TP500 uniquely covered a smell related to the *Delete Post* functionality, while TP500 uniquely covered two smells associated with *Delete User*. The three aforementioned code smells are classified as Major severity and fall under the Conditionals Complexity subcategory.

Figure 6.5 shows the distribution and density of code smell occurrences across test processes. Occurrences refer to the total number of times code with existing code smells is executed during testing. TP100 shows the lowest total of occurrences, with a relatively narrow distribution centred around 7500 occurrences per run. In contrast, TP500 and TP1000 configurations present broader distributions with longer upper tails, suggesting these configurations occasionally produce runs with a higher number of interactions with smelly code.

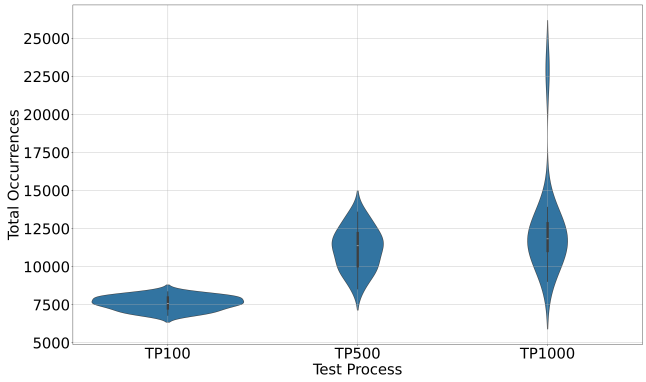


Figure 6.5: Distribution of code smells occurrences.

Statistical analysis was done to test whether the observed differences in metrics across the test configurations (TP100, TP500, and TP1000) are meaningful or likely due to random variation, as shown in Table 6.4. Kruskal-Wallis test was used to determine whether there was at least one significant difference among

Table 6.4: Statistical Analysis of Code Coverage Metrics

Metric	KW[a]	Mann-Whitney U (Effect Size)			Significant Pairs
	p-value	TP1000 vs TP500	TP1000 vs TP100	TP500 vs TP100	(M-W U / Dunn's test)
CSC	0.28	0.71 (0.05)	0.13 (0.22)	0.22 (0.18)	–
CSO	**0.001**	0.08 (0.27)	**0.001 (1)**	**0.001 (1)**	TP1000 vs TP100[b,c], TP500 vs TP100[b,c]
LC	**0.049**	0.77 (0.04)	**0.04 (0.31)**	**0.03 (0.33)**	TP1000 vs TP100[b], TP500 vs TP100[b]
IC	0.13	0.89 (0.02)	0.09 (0.26)	0.08 (0.27)	–
BC	0.09	0.71 (0.06)	0.051 (0.29)	0.07 (0.27)	–
CoC	**0.02**	0.65 (0.07)	**0.02 (0.36)**	**0.01 (0.38)**	TP1000 vs TP100[b,c], TP500 vs TP100[b]
MC	**0.01**	0.70 (0.06)	**0.01 (0.37)**	**0.01 (0.38)**	TP1000 vs TP100[b,c], TP500 vs TP100[b,c]
ClC	**0.03**	0.65 (0.07)	**0.02 (0.35)**	**0.03 (0.34)**	TP1000 vs TP100[b,c], TP500 vs TP100[b]
AbSC	**0.001**	**0.001 (0.74)**	**0.001 (0.99)**	**0.001 (0.78)**	all pairs[b,c]
CoSC	**0.001**	0.28 (0.16)	**0.001 (0.88)**	**0.001 (0.89)**	TP1000 vs TP100[b,c], TP500 vs TP100[b,c]
AbTC	**0.001**	**0.001 (0.64)**	**0.001 (0.94)**	**0.001 (0.45)**	all pairs[b,c]
CoTC	**0.001**	0.25 (0.18)	**0.001 (0.68)**	**0.04 (0.31)**	TP1000 vs TP100[b,c], TP500 vs TP100[b,c]

[a] KW: Kruskal-Wallis test
[b] Significant according to Mann-Whitney U test ($p < 0.05$)
[c] Significant according to Dunn's test with Bonferroni correction ($p < 0.05$)
Note: Bold values indicate statistical significance ($p < 0.05$). Effect sizes (Cliff's delta) are shown in parentheses.

the test configurations for each metric without assuming normal distributions. Mann-Whitney U tests followed up for pairwise comparisons, while Dunn's test further confirmed significance across multiple comparisons.

Code coverage metrics across the three configurations revealed significant differences in several metrics. TP100 showed significantly lower coverage across all metrics (except IC and BC) when compared to both TP1000 and TP500, with moderate effect sizes (0.31 to 0.38), indicating practically meaningful differences. No significant differences were found between TP1000 and TP500 for traditional metrics.

Regarding state metrics, Kruskal-Wallis tests indicated significant differences across all metrics ($p < 0.001$). Post-hoc analysis revealed that TP100 resulted in

significantly lower coverage than TP500 and TP1000 for both AbSC and AbTC, with large effect sizes highlighting the substantial impacts of shorter sequences on coverage levels.

Kruskal-Wallis and Dunn's tests confirm significant differences in code smell occurrences among the configurations, with large effect sizes in comparisons between TP100 and the other test processes.

In summary, these results show that longer test sequences lead to significantly higher values in traditional coverage metrics and an increase in code smell occurrences. Additionally, the distribution pattern suggests that longer sequences lead to better code smell coverage.

RQ1 answer: *longer random test sequences improve traditional coverage metrics and code smell coverage metrics.*

6.4.2 RQ2: Relationship between code coverage metrics

To investigate the relationship between different coverage measures, Spearman's rank correlation coefficients were calculated between code smell coverage and each traditional adequacy metric for each configuration. Spearman's correlation was selected because the data is not normally distributed. The results are presented in Table 6.5.

Table 6.5: Spearman's Correlation: Code Smell Coverage vs Traditional Metrics

	Code Coverage Metrics						State Model Coverage			
Test Process	IC	BC	LC	CoC	MC	ClC	AbSC	AbTC	CoSC	CoTC
TP100	.459*	.393*	.508**	.611***	.664***	.578***	-.049	.043	-.050	-.048
TP1000	.597***	.627***	.597***	.586***	.585***	.590***	.078	-.097	-.256	-.224
TP500	.401*	.536**	.433*	.451*	.435*	.455**	.150	.212	.302	.243

* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

All correlation coefficients between code smell coverage and traditional code metrics are statistically significant. Results mostly show a moderate correlation between code coverage metrics and code smell coverage. The correlation with

state metrics is generally weak and not statistically significant.

This finding suggests that relying solely on traditional coverage metrics might not fully represent a test suite's effectiveness at uncovering deeper issues like code smells. This highlights the need for complementary metrics or deeper analysis beyond basic coverage percentages.

Testers should consider these correlations when designing test suites and possibly combine traditional metrics with newer, more code-quality-focused metrics. Among traditional metrics, Method and Complexity Coverage show the highest correlations with Code Smell Coverage across all test processes, suggesting these metrics are more reliable indicators for exposing quality issues like code smells. However, widely used metrics like Instruction and Branch Coverage appear less reliable as standalone indicators of test quality.

RQ2 answer: *traditional metrics are useful, but not sufficient alone at reflecting the ability of the test suite to detect deeper quality issues, such as code smells. Code Smell Coverage can be a valuable metric to be considered along with the traditional coverage metrics to obtain a more holistic view of software quality and test effectiveness*

6.4.3 RQ3: Comparison of random with manual testing

Following the analysis of RQ1 (see Section 6.4.1), TP500 covered all code smells reached by TP1000, and additional ones, while achieving similar levels of code smell coverage in most test runs and using fewer resources than TP1000. For that reason, TP500 was enhanced with the form-filling feature to conduct the comparison with manual testing.

Figure 6.6 presents a comparison between the scriptless testing processes, including the enhanced test process (TP500Forms), and the manual testing results, in terms of Code Smell Coverage. TP500Forms significantly outperformed the original TP500 test process, closing the gap with the 88 code smells detected by manual testing. Some runs of TP500Forms even detected up to 99 unique code smells, surpassing the manual testing results.

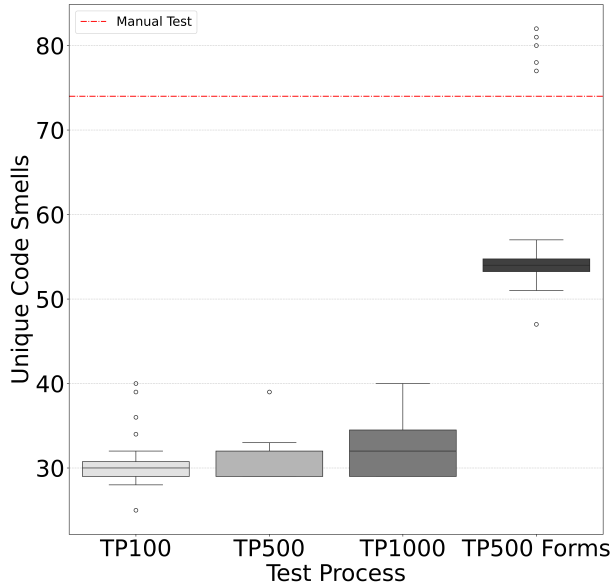


Figure 6.6: Distribution of code smell coverage

The coverage metrics in Table 6.6 provide further insight. While manual testing achieved slightly higher or similar code coverage, TP500Forms exhibited broader class exploration. Furthermore, TP500Forms discovered more unique code smells (101) than manual testing (88), suggesting that the enhanced approach can match the thoroughness of manual testing in terms of traditional adequacy metrics and surpass it for code smell coverage.

Figure 6.7 depicts a detailed analysis of the type of code smells covered (or not) by random or manual testing. This analysis revealed that all scriptless approaches (TP100, TP500, TP1000, and TP500Forms) covered 12 code smells that were not reached during manual testing. Two code smells were of Minor severity, categorised as Clever Code and Inconsistent Style, while the remaining ten were classified as Conditional Complexity with Major severity. These code smells

Table 6.6: Manual Testing Coverage Results

Metric	Manual Testing	TP500Forms	
		Mean	Max
IC	43.03%	43.21%	54.50%
BC	20.53%	17.42%	21.95%
LC	49.48%	47.76%	60.01%
CC	42.09%	39.20%	49.22%
MC	51.47%	48.00%	60.27%
ClC	57.47%	72.28%	82.95%
CSC	88	70.5	99

were associated with two specific functionalities of the application (deleting feed and task commenting), whose corresponding user stories were not covered by the manual testing process.

Despite not consistently outperforming manual testing in individual runs, the TP500Forms configuration, when considered in aggregate across all runs, covered three additional code smells that were not reached during manual testing or by the other random test processes. Two of these code smells were classified as Major. Furthermore, these code smells were triggered in multiple runs.

One of the new covered smells was reached as a consequence of a random input combination of a filtering functionality within the SUT. This was the only newly reached code smell that did not result from the predefined form field values. Notably, every code smell triggered by manual testing was also triggered by at least one test run of TP500Forms. Notably, every code smell covered by manual testing was also covered by at least one test run of TP500Forms.

In summary, random testing identified code smells missed by manual testing, demonstrating (again [10, 64]) its potential as a complementary approach. However, random testing struggled with forms requiring specific inputs, which manual testing handled better. The enhanced form-based approach demonstrated comparable and even surpassed manual testing by covering all manually reached smells and additional ones.

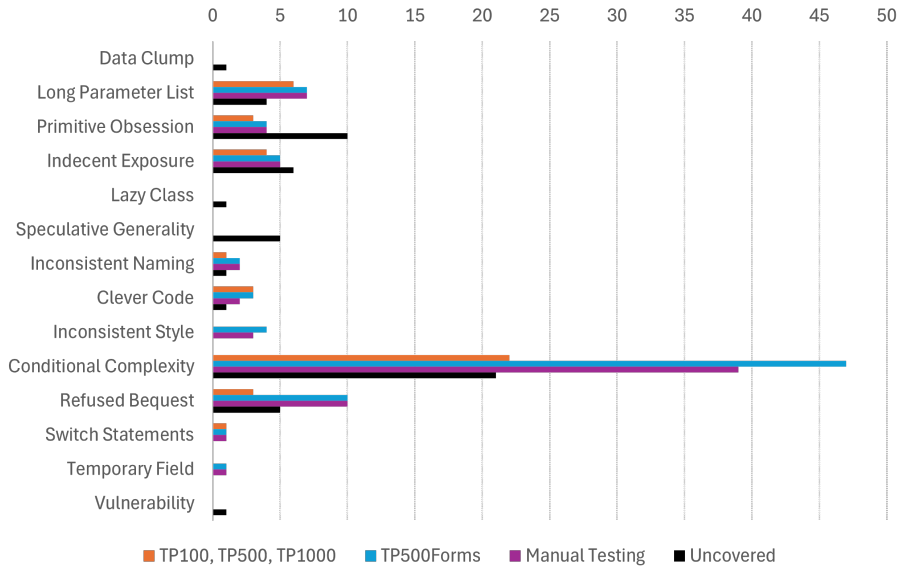


Figure 6.7: Coverage of Code Smell types

RQ3 answer: *findings suggest that random testing offers promising complementary effectiveness in test coverage and identifies unique smells that manual testing might overlook.*

6.5 Discussion

To understand the impact of adopting random testing and introducing the new code smell coverage metric on Marviq's QA process, two one-hour focus groups were conducted with three of their test engineers. Marviq shared that these additions significantly enhanced their workflow. While manual testing leverages testers' domain expertise, random testing complements it by uncovering unexpected navigation paths, providing a balanced approach that strengthens Marviq's quality control.

The team further emphasised the value of scriptless testing as a complementary tool within their established QA practices. Running these scriptless tests overnight and integrating them into the CI/CD pipeline enables a continuous and efficient testing cycle. This enhances software robustness and supports the move toward continuous delivery, reducing the need for separate acceptance testing phases and optimising both time and effort per release.

Marviq also observed that, once configured for a specific project as demonstrated with the Yoho project (Section 6.3.2), the testing setup can be easily adapted for other projects using similar technologies, making it a scalable and reusable solution.

Finally, Marviq noted that monitoring coverage metrics closely linked to code smells acts as an effective early warning system. This proactive insight into potential code issues enables the team to address quality concerns early in the development lifecycle, ultimately supporting the delivery of more robust software.

6.6 Threats to Validity

Potential threats to the validity of the study and the mitigation actions taken to address them within the available means are discussed. They are classified into four categories following [114]: internal, external, construct, and conclusion validity.

6.6.1 Internal Validity

Internal validity refers to factors that may introduce bias in this experiment. One possible threat is the random nature of the scriptless GUI testing process. Since this study relies on a random testing algorithm, the specific sequence of actions generated during the testing process may influence the coverage of code smells. Different executions could lead to different levels of coverage, which may not fully represent the testing tool's effectiveness. To mitigate this threat, multiple testing sessions with varied configurations were executed to observe trends and reduce the impact of randomness.

6.6.2 External Validity

External validity concerns the generalizability of the findings to other contexts. This study focused on a specific industrial web application, selected for its role as a core system for the company with functionalities commonly used in web applications. While we advocate that the selected SUT is representative of other industrial web applications, the results might not be directly applicable to different types of applications, such as mobile or desktop software. Additionally, the reliance on the SonarQube tool for detecting code smells and the TESTAR tool for GUI testing could limit the generalisation of the findings. Other tools may produce different results regarding code smell detection and coverage. Future research should replicate this approach with various applications, frameworks, and testing tools to validate the general applicability of the findings.

6.6.3 Construct Validity

Construct validity refers to how well this experimental setup measures its intended outcomes. This study used code smells as a proxy for software quality and testing effectiveness. While code smells are widely recognised as indicators of potential maintainability and quality issues, they may not always directly correlate with defects in the system. Additionally, code smell detection from SonarQube was relied upon, which may not capture all relevant issues. To address this, efforts were made to ensure that detected smells were representative of common issues, but the limitations of the tools should still be acknowledged.

6.6.4 Conclusion Validity

Conclusion validity relates to the reliability of the relationship between the treatments and the observed outcomes. One potential threat is the sample size of the testing actions and configurations. Although 10,000 actions were executed, this may still be insufficient to generalise the findings across all possible scenarios in the application. Moreover, the impact of different configuration settings on code smell coverage should be interpreted carefully, as certain configurations

may favour specific types of code smells over others. This risk was minimised by conducting experiments with varied configurations; however, future work should explore a broader range of parameters to draw more robust conclusions.

6.7 Conclusions

This study explored the potential of random scriptless GUI testing as a complementary approach to traditional testing in an industrial setting, focusing on Marviq's Yoho web application. The results indicate that increasing the length of random test sequences significantly enhances both traditional coverage metrics and code smell coverage, suggesting that longer test sequences can lead to more thorough and effective testing even within resource constraints.

The findings further suggest that while traditional coverage metrics offer valuable insights into testing adequacy, they are insufficient to capture the full scope of quality issues, particularly regarding code maintainability. Integrating code smell detection with traditional coverage metrics provides a more comprehensive perspective on software quality, addressing areas of technical debt and maintainability that may be overlooked with conventional coverage alone.

Moreover, random GUI testing also demonstrated a unique strength in identifying code smells missed by manual testing, including some critical ones. While manual testing benefits from the tester's domain knowledge, random testing offers the potential of unexpected navigation paths. Therefore, the study highlights the complementary role of random testing alongside manual testing, as random testing effectively identifies unique code smells that manual efforts might miss. This synergy between testing methods enhances overall test coverage, potentially reducing reliance on manual testing and enabling a more resource-efficient approach to quality assurance in software development.

In conclusion, combining coverage metrics with maintainability-focused analyses, such as code smell detection, provides a robust and efficient testing framework that better aligns with industrial needs. This integrated approach offers a deeper and more accurate assessment of software quality, covering aspects of both functionality and maintainability.

7

Going mobile: the Android plugin

"There are two ways to write error-free programs; only the third one works"

Alan J. Perlis, Epigrams on Programming

Existing scriptless tools for Android lack efficient prioritisation and advanced oracles to detect problems in the SUT. They also require significant manual input for specific SUT information. This chapter presents MINTestar: an implementation of the TESTAR scriptless testing approach [197] for Android that solves some of these drawbacks. The implementation encompasses: (1) customisable rules, (2) probabilistic exploration to improve coverage, (3) composable oracles, (4) effortless integration, and (5) improved reporting. Furthermore, we conducted a preliminary empirical comparison with two relevant scriptless testing tools applied to one mobile application. The selected testing tools are TESTAR [197] and DroidBot [200].

The contribution of this chapter is:

- a novel scriptless testing tool for Android that uses a probabilistic model and composable oracles for the selection of actions;
- a preliminary comparison with TESTAR and DroidBot regarding effectiveness

and efficiency.

The rest of the chapter is structured as follows. Section 7.1 presents a brief discussion of tools for Android testing. Section 7.3 introduces MINTestar along with its key features and contributions. Section 7.4 presents preliminary results from the initial comparative experiment. Section 7.5 concludes the chapter.

7.1 Scriptless Android GUI testing

This section presents a detailed overview of the existing scriptless testing tools for Android. The literature review phase used Google Scholar as the primary digital platform. The search process leveraged keyword-based queries focusing on: *android testing*, *GUI testing*, and *scriptless testing*.

After a review of paper titles and abstracts, 12 tools in the field of scriptless Android GUI testing were identified. Each paper was evaluated based on criteria including novelty, presence in previous experiments, and alignment with current study metrics. Table 7.1 presents key information about each tool, analysing fundamental aspects, such as testing techniques, oracles, state representation, available actions, and whether or not the tool is publicly available and actively maintained.

The review of scriptless testing tools for Android apps showed diverse approaches. Tools like Dynodroid [201] and DroidBot [200] use **random action selection**, with DroidBot adding a **model-based approach** for state recognition. Similarly, Stoat [202] and APE [203] employ a model-based strategy to guide the exploration towards areas of the application that have to be explored yet.

Table 7.1: Summary of scriptless Android GUI testing tools

Tool	Publicly available	Actively maintained	Testing technique	Concept of state	Fault detection
Dynodroid	Yes	No	Random (with three different event selection strategies).	Yes (widget tree)	Implicit
Sapienz	No	No	Systematic, Search (multi-objective search-based testing with genetic algorithms.)	Yes (widget set)	Implicit
DroidBot	Yes	Yes	Systematic, Random (model-based exploration testing)	Yes (Widget tree)	–
Humanoid	Yes	No	Unsupervised deep learning (with supervised learning using human interactions)	No	–
Stoat	Yes	No	Stochastic model based	Yes (based on static analysis)	Implicit
DroidMate2	Yes	No	Random, model-based strategies	Yes (widget tree)	–
PUMA	Yes	No	Programmable exploration testing with customisable action selection	Yes	Programmable oracles
RegDroid	Yes	Yes	Differential (comparing behaviours between two versions)	Yes (widget tree)	Implicit
QTesting	Yes	No	Reinforcement Learning with curiosity-driven exploration	Yes (widget tree)	Implicit
AimDroid	Yes	No	Reinforcement Learning (SARSA algorithm)	Yes (widget tree)	Implicit
ARES	Yes	No	Deep Reinforcement Learning	Yes (widget tree)	Implicit
ComboDroid	Yes	No	Combinatorial	Yes (widget tree)	Implicit

Some existing tools use more advanced techniques to improve their ASM. Sapienz [204] uses fuzzing and search-based methods with evolutionary algorithms. Humanoid [205] applies deep-learning to mimic human interactions, while ARES [206] uses deep-learning for better exploration strategies. QTesting [207] and AimDroid [208] use reinforcement learning, the former for prioritising unfamiliar states and the latter for predicting events likely to trigger new activities or crashes. ComboDroid [208] uses combinatorial exploration for identifying unvisited states, and RegDroid [209] focuses on finding functional bugs via differential regression testing.

Although the landscape of scriptless GUI testing tools for Android is diverse, these tools often have limitations. Random testing tools lack efficient prioritisation, while tools with more advanced ASM such as RL require more extensive training periods. Generally, these tools rely on implicit oracles, detecting application crashes or exceptions during the execution of tests. Additionally, most of these testing tools are not actively maintained, which may result in compatibility issues with the evolving Android ecosystem. Furthermore, outdated tools will likely lack crucial updates addressing security vulnerabilities and adapting to new testing requirements.

7.2 Extending TESTAR to support mobile testing

To support mobile application testing, TESTAR has been extended to handle both Android and iOS platforms. This section describes how the mobile testing capabilities were integrated into TESTAR's architecture.

TESTAR's modular architecture 3.6 allows the integration of new platforms while maintaining a unified testing approach. To minimise maintenance costs, Appium was chosen to be integrated into TESTAR as the bridge between TESTAR's core architecture and both mobile platforms. This choice was driven by Appium's implementation of the WebDriver API for mobile apps, which aligns well with TESTAR's existing web testing capabilities. Additionally, Appium provides a unified interface for both Android and iOS, abstracting many platform-specific details.

Figure 7.1 depicts the adapted TESTAR loop for mobile testing, extending the

generic scriptless testing loop discussed in Section 4.2. TESTAR uses the Appium automation driver to initialise the SUT and to capture the current application state as an XML document containing the complete widget tree with its attributes. After some ASM selects the next action, TESTAR instructs the Appium driver to execute it on the SUT.

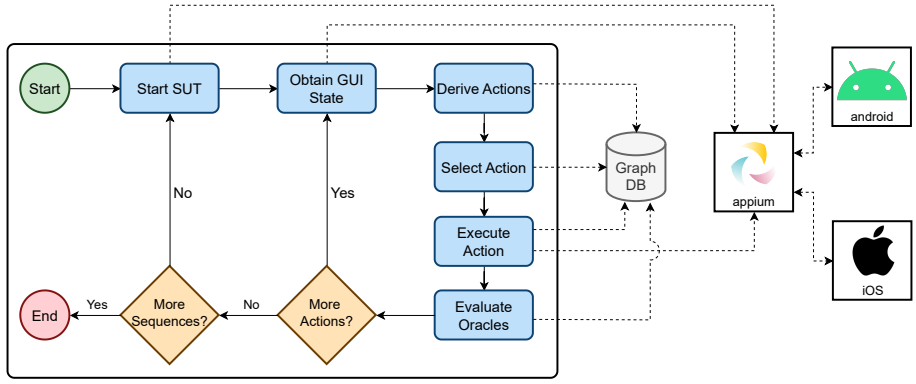


Figure 7.1: TESTAR testing cycle with mobile capabilities.

While Appium claims to abstract away Operating System differences, separate handlers were necessary in practice due to fundamental differences in how each platform exposes UI information. For example, the attributes available for certain widgets were different across platforms. As depicted in Figure 7.2, these differences result in the creation of two specific protocols for Android and iOS: **AndroidProtocol** and **IOSProtocol**, respectively, which complement the already existing **DesktopProtocol** and **WebProtocol** (as explained in Section 3.6).

Platform-specific protocols parse and normalise the data obtained from the **Appium**-specific drivers for Android and iOS before integrating it into TESTAR's state model. This normalisation process ensures that despite the underlying platform differences, TESTAR's core testing algorithms can work with a consistent state representation.

Consequently, the action derivation was also implemented separately for both Android and iOS, as it depends on the widgets and their attributes. For instance, Android relies on explicit properties such as a *clickable* boolean, while iOS might

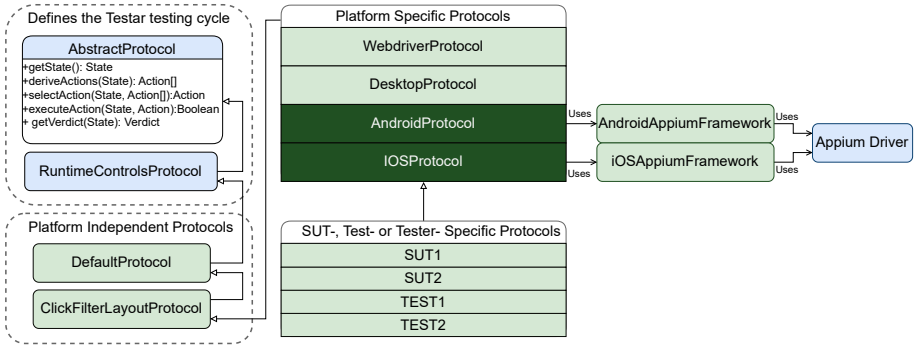


Figure 7.2: Layers of the different TESTAR protocols

determine action availability through widget class types like *XCUIElementType-Button*. For Android, support was implemented for *clicking*, *long clicking*, *scrolling*, *typing*, *back navigation*, and system actions. For iOS, support focuses on *clicking*, *scrolling*, and *typing*, as iOS does not support certain Android-specific actions like long-clicks or back navigation.

Finally, TESTAR's ASM and oracles work for both Android and iOS, as they are platform-independent. Once an action is selected, the automation driver configured for Appium will connect and interact with the mobile environment. This cross-platform approach through Appium provides significant advantages for organisations testing applications across multiple platforms. However, the additional abstraction layer can introduce performance overhead, which is particularly noticeable in scenarios requiring rapid state inspection or action execution. As an alternative, complementary approaches were explored to leverage platform-specific capabilities while the core principles of scriptless testing were maintained.

In the following section, MINTestar is introduced, through which it is demonstrated how TESTAR's scriptless testing philosophy can be adapted for specialised Android testing needs.

7.3 MINTestar: scriptless and seamless

While TESTAR provides a comprehensive mobile testing solution through Appium integration, the mobile testing ecosystem offers opportunities for more specialised approaches. MINTestar[1] was developed as a dedicated Android testing tool, through which TESTAR's core principles of scriptless testing are preserved. The same fundamental testing loop is followed, yet rather than Appium being utilised as an intermediary layer, direct interfaces with Android's native testing frameworks are established. Through this direct integration, the testing process is streamlined and natural alignment with existing Android development workflows is achieved.

MINTestar's ASM is built upon a probabilistic exploration approach augmented by customisable rules to identify and (de) prioritise GUI interactions during action selection. Once configured, MINTestar autonomously explores native Android applications, obviating the need for manual script development and maintenance. Unlike scripts, which require detailed programming to define every test case, MINTestar's rules allow testers to specify testing criteria and priorities more abstractly and intuitively, facilitating rapid adaptation to application changes.

MINTestar relies on the Espresso API to locate and interact with UI elements. Espresso is an Android testing framework specifically designed for writing UI tests to automate the testing through the GUI of Android apps. Espresso uses matches to detect View elements, representing any visible or interactive element that users can see and interact with on the GUI.

The SUT's intended behaviour is verified through oracles, as is done with TESTAR, producing verdicts that record specific aspects of the SUT, which facilitates a comprehensive evaluation of the SUT. The oracles are composable, providing a continuous and multidimensional assessment of the SUT. Additionally, an interactive reporting tool accompanies MINTestar, providing detailed analysis of system interactions and insights into the SUT's behaviour, crucial for enhancing the quality and Customer Experience (CX).

[1] Available at: https://github.com/ing-bank/mint

MINTestar starts by accessing the top-level **View** container, encompassing all GUI elements, and recursively traverses the GUI hierarchy, aggregating View elements to form a concrete state representation saved as an XML document. Figure 7.1 illustrates the XML state representation with one container **layout** and three actionable **widgets**.

Example 7.1: XML State representation

```
<View class="Layout" ...>

    <View class="Checkbox" id="like" .../>

    <View class="TextField" id="comment" .../>

    <View class="Button" id="done" .../>

</View>
```

MINTestar is conceived to seamlessly integrate as a plugin into a testing pipeline. Therefore, MINTestar tests can be executed as a task similar to unit or integration testing tasks. Consequently, Android app developers can effortlessly and expeditiously incorporate an exploratory GUI testing solution into their testing processes. MINTestar introduces three key features: customisable rules, probabilistic exploration, and composable oracles.

7.3.1 Core Architecture

Figure 7.3 depicts the high-level architecture of MINTestar, divided into two main parts: the MINTestar Core and the Android Environment.

The **Core** of MINTestar is built from several core components, each with a specific testing function: the **Test Engine** manages test execution, the **Rules Engine** controls test behaviour, the **Oracle Manager** monitors application state, the **Interaction Engine** handles GUI actions, the **State Collector** captures the SUT's state, and the **Report Generator** creates test results.

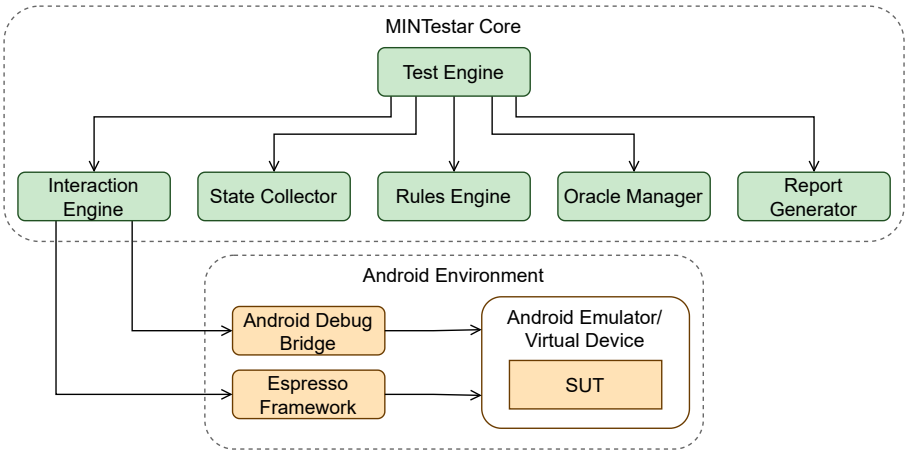


Figure 7.3: MINTestar Architecture Overview

The **Android Environment** involves the components that allow MINTestar to interact with the Android testing infrastructure through two main paths:

- Android Debug Bridge (ADB), which communicates with the Android Emulator/Virtual Device where the SUT runs.
- Espresso Framework, which provides direct UI testing capabilities for interacting with the SUT.

This dual-path approach allows MINTestar to both control the Android environment through ADB and perform precise UI interactions through Espresso. The SUT runs within an Android Virtual Device, which can be either an emulator or a physical device.

7.3.2 Test Engine

The Test Engine acts as MINTestar's central orchestrator of the entire testing process. This component coordinates the test execution flow, ensuring that each test sequence is properly initialised, executed, and completed. It maintains state

information throughout the testing process, enabling the framework to make informed decisions about test progression and to adapt its behaviour based on the application's current state.

The sequence diagram in Figure 7.4 illustrates MINTestar's testing process, summarising how a tester can utilise this system and understand the sequence of operations that occur during testing.

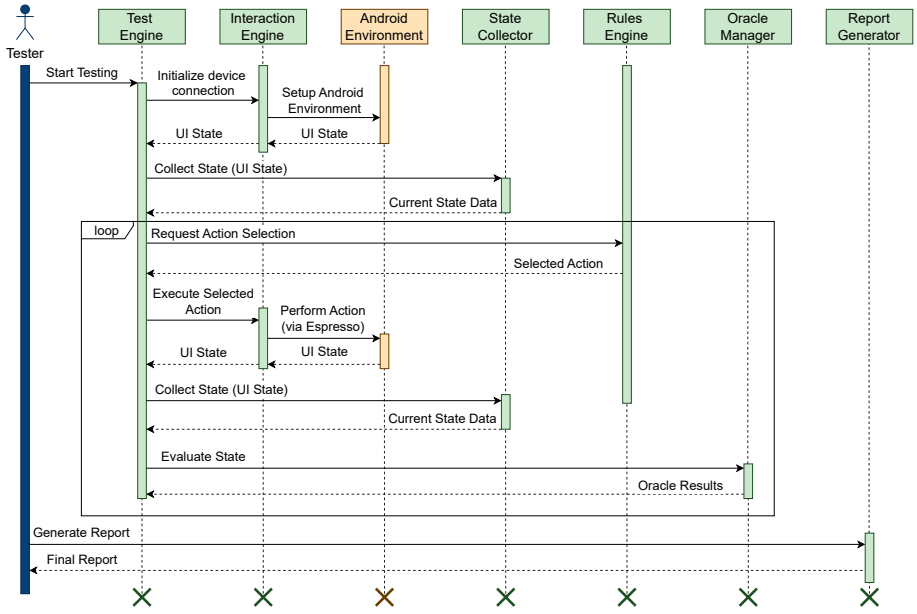


Figure 7.4: MINTestar Testing Process

The process begins with the initial setup of a MINTestar test run. The Test Engine begins its initialisation sequence. The diagram shows that the Test Engine first establishes the device connection and sets up the Android Environment. This initialisation phase is crucial as it prepares the testing environment and establishes proper communication channels between MINTestar and the SUT.

Following initialisation, the sequence moves into state management. The Interaction Engine obtains the application's initial GUI state, which is then passed to the State Collector for analysis. This state collection process provides the

foundation for understanding the application's current condition and determining possible actions.

The main testing loop (inspired by the scriptless generic testing cycle from Figure 3.1), which forms the core of MINTestar's testing process, begins its execution. The Test Engine requests action selection from the Rules Engine. Upon receiving the selected action, the Test Engine coordinates with the Interaction Engine to execute it through Espresso. After each action, the Oracle Manager collects and evaluates a new GUI state, ensuring the application behaves as expected.

This testing loop continues until predetermined conditions, such as time limits, are met. Upon completing the testing loop, the Report Generator creates a comprehensive final report.

7.3.3 Customizable Rules

MINTestar's functionality revolves around the adherence to predefined rules. MINTestar's Rule Engine governs how the framework interacts with SUTs through a system of hierarchical rules. These rules define the interactions with the SUT, such as clicks or text inputs. Every rule is assigned a relative importance, collectively forming a model that intelligently guides MINTestar's exploration through various SUT states without prior knowledge.

A rule is a tuple denoted by $R = (P, A, \pi)$. The predicate function $P : S \rightarrow \{0, 1\}$ serves the fundamental role of mapping the state $s \in S$ to a binary set $\{0, 1\}$, such as $P(s) = 1$ if and only if the rule is applicable under the conditions represented by state s. The finite set A represents all possible actions that can be executed. Lastly, priority $\pi \in \mathbb{R}$ quantifies the importance and precedence of the rule relative to other rules. This numerical priority plays an important role in decision-making processes, influencing the order in which rules are considered and executed.

A rule also encompasses attributes such as *name*, *description*, and *modifier*. The *modifier* characterises the adjustment applied to a rule's priority, offering fine-tuning for its significance in decision-making. For example, if a rule with an original priority $\pi = 0.5$ has a *multiplicative* modifier with a factor 2, the modified

priority becomes $\pi = 1.0$, increasing its importance by the multiplicative value.

Rules can be categorised into three types:

- Generic rules: applicable to all SUT (e.g., interacting with clickable elements).
- Specific rules: tailored for particular application types, addressing unique scenarios such as inputting email addresses in relevant email input fields.
- Domain-Specific rules: designed for internal use with specific account numbers or identifiers.

Tables 7.3 and 7.2 show all existing rules provided by MINTestar. These include navigation rules such as scrolling to and clicking any yet hidden widget, or de-prioritising previously executed actions. Every de-prioritising rule contains a *multiplicative* modifier to reduce the priority of the rules. MINTestar also features input rules, facilitating the generation of various input types (e.g., emails, names, dates, postal codes) as needed for different testing scenarios. For instance, the generic rule *simpleClickableRule*, defined as:

Example 7.2: Simple Click defined as a rule

```
GenericRule(action = Action.CLICK,
            pred = xpred(".[@isClickable = 'true'
                   and @isDisplayed = 'true']"),
            prio = 0.5)
```

uses Espresso to evaluate whether an element is **clickable** and **visible** and applies the action of **clicking** with a given **priority**. The rules can be defined using regular expressions, such as emails or numbers, to generate specific input types. The library JavaFaker was used to generate fake, realistic, and non-sensitive data, like phone numbers or postal codes.

Once all rules have been assigned individual priorities, the Rule Engine proceeds to generate a *model* containing the final set of available actions and their

Table 7.2: Generic rules provided by MINTestar

Rule Name	Description
simpleClickableRule	Click on any widget that has 'isClickable' as true and is displayed.
scrollingClickableRule	Scroll to and click any widget that is clickable, not yet displayed, and can be scrolled to.
clickableRuleForItemWithTag	Click on any displayed, clickable widget that has a specific tag.
clickableRuleBasedOnPosition-InViewHierarchy	Click on any displayed, clickable widget.
clickableRuleBasedOnPosition-InViewHierarchyForPopupItem	Click on any displayed, clickable widget in a pop-up window.
deprioritizeClickingOnPopupItem-OnCurrentRoot	Deprioritize clicking on widgets in pop-up windows.
deviceRotationRule	Change the device's rotation to check the responsiveness of the UI.
deviceThemeRule	Change the device's theme to check the responsiveness of the UI.
clickableRuleForAdapterViewItems	Click on an item in a list backed by an adapter.
clickableRuleForSpinnerItems	Click on an item in a spinner list.
spinnerSimpleClickDeprioritizRule	Deprioritize clicking of a Spinner.
adapterViewClickDeprioritizeRule	Deprioritize clicking of an AdapterView.
scrollingPagerRightRule	Select scrolling to the right within pagers for horizontal navigation.
scrollingPagerLeftRule	Select scrolling to the left within pagers for horizontal navigation.
timePickerInputRule	Generate time input for TimePickers.
datePickerInputRule	Generate date input for DatePickers.
clickableRuleBasedOnPosition-InViewHierarchyForBottomSheet	Click on any displayed, clickable widget that has a bottom sheet as an ancestor.
defaultPreviousActionDeprioritize-Rule	De-prioritized actions that were already taken historically.

Table 7.3: Specific rules provided by MINTestar

Rule Name	Description
defaultUTF8InputRule	Generate UTF8 text streams for anything accepting text.
defaultTextInputRule	Generate generic text for anything accepting text.
defaultMultilineTextInputRule	Generate generic text for anything accepting text.
defaultEmailAddressInputRule	Generate text in email address format for widgets accepting email addresses.
defaultNumberInputRule	Generate number input for widgets accepting input of numbers type.
defaultDecimalNumberInputRule	Generate decimal number input for widgets accepting input of decimal number type.
defaultSignedNumberInputRule	Generate signed number input for widgets accepting input of signed number type.
defaultPersonNameInputRule	Generate text for widgets accepting person name input.
defaultUriRule	Generate text for widgets accepting input of URI type.
defaultPhoneNumberInputRule	Generate phone number for widgets accepting text in phone format.
defaultPostalAddressInputRule	Generate postal address input for widgets accepting a postal address as input.
defaultDateInputRule	Generate date input for widgets accepting date as input.
defaultTimeInputRule	Generate time input for widgets accepting time as input.
defaultGenericTextInputRule	Generate generic text for anything accepting any input.
defaultUneditableTextClick-DeprioritizeRule	De-prioritize the input of text in uneditable text fields.
defaultTextClickDeprioritizeRule	De-prioritize the clicking of text elements.
defaultTextClickAtPosition-DeprioritizeRule	De-prioritize the clicking of text elements based on position.

relative importance with the goal of selecting the next action. The ASM explained in Algorithm 4 works in two phases: **priority computation** and **probabilistic selection**.

Algorithm 4 ASM: Select an Action

```
Input: actionRules                                          ▷ Set of available action rules
 1: pairs ← []                                              ▷ List of (action, priority) pairs
 2: prioritySum ← 0                                         ▷ Total sum of priorities
 3: for all rule in actionRules do
 4:     if rule has modifier attribute then
 5:         modPriority ← ModifyPriority(rule)              ▷ Adjust priority if modified
 6:     else
 7:         modPriority ← rule.Priority                     ▷ Use default priority
 8:     pairs ← pairs + (rule.action, modPriority)
 9:     prioritySum ← prioritySum + modPriority
10: accumPriority ← 0
11: dice ← RandomFloat() * prioritySum                      ▷ Random threshold for selection
12: for all (action, modPriority) in pairs do
13:     accumPriority ← accumPriority + modPriority
14:     if accumPriority ≥ dice then                        ▷ Select the action when threshold is met
15:         return action
```

Initially (lines 1-9), the algorithm iterates through all applicable action rules to compute their effective priorities (*modPriority*). Each rule initially has an associated priority, which may be *modified* based on certain conditions (lines 4-7). If a rule has a modifier attribute, its priority is adjusted; otherwise, the default priority is used. As each rule is processed, a pair consisting of the action and its resulting modified (or not) priority is stored, and the total sum of all priorities (*prioritySum*) is updated (lines 8 and 9). This results in a set of (*action*, *modified priority*) pairs, where actions with higher priority will have a greater influence in the selection process.

In the second phase (lines 10-15), the probabilistic action selection takes place. A random value (*dice*) is generated within the range $[0, prioritySum]$ (line 11). The next action to be executed is selected (lines 12-15) by choosing the first action that surpasses this random value. Actions with higher priorities are more likely to be selected while still allowing some degree of randomness in the action selection mechanism.

7.3.4 State Collector

MINTestar's state management system forms a critical component of its testing intelligence. The system creates detailed representations of each unique state encountered during testing, capturing not only the visible UI elements but also their properties, relationships, and interactive capabilities.

MINTestar builds a comprehensive model of the application's behaviour and structure. This model enables the framework to make informed decisions about test progression, identifying unexplored states and potential paths through the application. The state management system also detects state changes resulting from user interactions, system events, or background processes, ensuring thorough testing coverage.

The representation of the state is extended by MINTestar through incorporating actions to its nodes. As shown in Algorithm 5, the process starts by iterating over all available action rules (*actionRules*) (line 1). Each rule contains a predicate function, which determines whether the rule is applicable to a specific state node. For each action rule, a nested loop is used to recursively traverse all widget nodes within the state structure (line 2). At each node, if the rule's predicate is satisfied (equivalent to $P(s) = 1$), the rule's action is associated with the node (line 4). This step ensures that each widget is correctly associated with applicable actions.

Algorithm 5 Annotate State With Rules

Input: *state* ▷ *The current state representation*

Input: *actionRules* ▷ *A set of rules to be applied*

```
1: for all rule in actionRules do                    ▷ Iterate through all action rules
2:     for all node recursively in state do          ▷ Traverse all nodes in the state
3:         if rule.predicate(node) then              ▷ Check if rule applies to the node
4:             node.append(rule)                     ▷ Associate rule's action with node
```

Following the Example 7.1, an XML hierarchy with annotated actions and their corresponding priorities is shown in Example 7.3. Note that the Button has an associated multiplicative action, which will modify its final priority.

Example 7.3: Annotated state representation

```
<View class="Layout" ...>

    <View class="Checkbox" id="like" ...>
        <action type="click" prio="1.0" .../>
    </View>

    <View class="TextField" id="comment" ...>
        <action type="text" prio="2.0" value="text" .../>
        <action type="text" prio="1.0" value="1282" .../>
    </View>

    <View class="Button" id="done" ...>
        <action type="click" prio="2.0" .../>
        <action type="multiplicative" prio="3.0" .../>
    </View>

</View>
```

7.3.5 Composable oracles

MINTestar implements a module for oracles designed to evaluate different aspects of the SUT. This module classifies oracles into distinct categories, each addressing specific testing objectives. These categories include Accessibility, Internationalisation, Performance, Stability, Aesthetics and Miscellaneous. The structure of an oracle is defined by an interface that incorporates key information such as the category, probe, and evaluation function. Probes serve as a data source for oracles, enabling them to form judgments about the SUT. For instance, oracle *AndroidLogOracle* creates a probe with the information extracted from the system logs to assess the presence of faults.

A significant subset of implemented oracles is dedicated to accessibility checks. These oracles scrutinise the SUT for adherence to standard accessibility guidelines [210]. They assess factors like text readability or image contrast to ensure

a positive CX. Currently available oracles (Table 7.4) also include *CrashOracle*, which detects a crash of the SUT, and *AndroidDeviceOracle*, to monitor the CPU and memory usage. Oracles to check internationalisation and stability are yet to be added.

Table 7.4: Implemented Oracles Provided by MINT

Category	Oracle Name	Description
Stability, Performance	AndroidDeviceOracle	Monitors the Android device for system metrics.
	AndroidLogOracle	Checks the Android system log for relevant events.
	CrashOracle	Detects application crashes during execution.
Accessibility	ClassNameCheckOracle	Verifies if the class name is appropriate for accessibility.
	ClickableSpanCheckOracle	Ensures that ClickableSpan is not misused within a TextView.
	DuplicateClickableBoundsCheckOracle	Detects cases where a clickable container overlaps entirely with a child view, leading to unexpected interactions.
	DuplicateSpeakableTextCheckOracle	Checks if two views in the hierarchy have the same speakable text.
	EditableContentDescCheckOracle	Ensures that an editable TextView is not labelled with a content description.
	ImageContrastCheckOracle	Validates that images have sufficient foreground-background contrast for visibility.
	LinkPurposeUnclearCheckOracle	Warns about links whose purpose is unclear to assistive technologies.
	RedundantDescriptionCheckOracle	Identifies cases where speakable text may contain redundant or irrelevant information.
	SpeakableTextPresentCheckOracle	Ensures that elements requiring speakable text have appropriate descriptions.
	TextContrastCheckOracle	Ensures text has sufficient contrast against its background for readability.
	TextSizeCheckOracle	Detects text scaling issues that may affect visibility.
	TouchTargetSizeCheckOracle	Ensures that touch targets meet the minimum recommended size (e.g., 48x48dp).
	TraversalOrderCheckOracle	Identifies problems in the accessibility traversal order defined by developers.
	UnexposedTextCheckOracle	Detects texts that might be blocked from OCR (Optical Character Recognition) and unreadable by accessibility services.

MINTestar's oracle framework emphasises extensibility, allowing developers to compose sets of oracles or add custom oracles. Testers can create compre-

hensive test suites by combining oracles from different categories, providing a versatile and adaptable testing environment. For instance, the following code shows the definition of a rule, accompanied by the oracle step of checking errors in the system logs without monitoring the memory usage. MINTestar also offers the inclusion (*withAllOracles*) or exclusion (*withoutAllOracles*) of all oracles, in combination with specific ones.

Example 7.4: Rule definition, with explicit oracle inclusion/exclusion

```
Mint.Rule(DefaultBuilder.withOracle(AndroidLogOracle)
                        .withoutOracle(AndroidDeviceOracle)
                        .build())
```

7.3.6 Interaction Engine

MINTestar's interactions with the GUI involve two main components: Android Debug Bridge (ADB) and Espresso. By integrating with the ADB, MINTestar establishes reliable communication with the test device or emulator, ensuring consistent and accurate testing. This communication layer manages all aspects of device interaction, from application installation and launch to command execution and state monitoring. The system implements robust error handling and recovery mechanisms, ensuring stable testing sessions even in the presence of device-level issues or communication interruptions.

The device communication system supports both physical Android devices and emulators, automatically adapting its behaviour to accommodate the specific characteristics of each target environment. It manages device-specific features and limitations, ensuring consistent test execution across different device types and Android versions.

Additionally, MINTestar leverages Espresso's precise view matching and interaction capabilities to execute actions on the GUI. The engine translates MINTestar's high-level testing directives into specific UI interactions, automatically handling complexities such as view synchronisation and wait conditions.

The framework allows testers to define specific sequences of actions using Espresso's familiar syntax. These predefined steps serve multiple purposes, from setting up specific application states to validating critical user flows. Testers can create precise sequences of actions when needed, such as logging into an application or navigating to a specific screen, before allowing MINTestar's automated exploration to take over.

Test creation in MINTestar follows a straightforward pattern, allowing testers to define test scenarios through a combination of automated exploration and, when needed, specific scripted steps. Basic exploration can be started with:

Example 7.5: Basic test instruction

```
Mint.explore()
```

For more controlled testing, it is possible to combine scripted steps with scriptless exploration:

Example 7.6: Exploration with predefined steps

```
Mint.step {
    onView(withId(R.id.button)).perform(click())
    onView(withId(R.id.input)).check(matches(isDisplayed()))
}.explore()
```

7.3.7 Reporting the results

MINTestar saves the testing process information in XML format. The previously described plugin provides a reporting task that parses the XML data and generates an overview HTML page with all the oracle outputs, individual pages for each test sequence, and screenshots associated with the test sequences. Each report (see Figure 7.5) contains a chronological record of actions taken, states encountered, and any issues detected by the various oracles.

The report also allows the search of elements of the application through their XPath and highlights their location within the screenshots, making it easier for testers and developers to understand and reproduce any identified issues. This reporting structure enhances the comprehensibility of test results by organising them into structured HTML pages, significantly improving the clarity and analysis of test results.

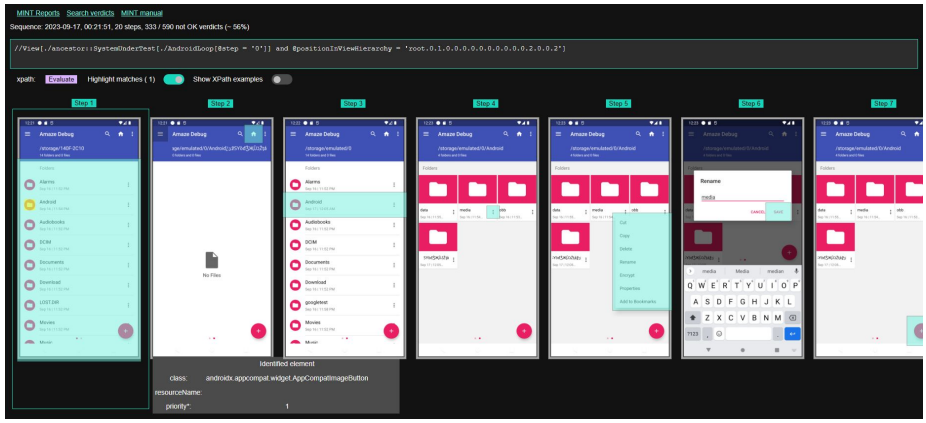


Figure 7.5: Excerpt of a MINTestar report

7.3.8 Seamless integration

MINTestar provides a plugin that serves as a gateway for the integration of MINTestar-based tests into the Android testing process. This integration aims to align with the execution patterns of classical integration and unit tests in Android, following established testing practices [211], promoting a consistent and familiar testing experience for developers and testers without requiring substantial changes to existing testing practices.

The plugin enables the incorporation of specific Gradle tasks, such as data collection and report generation, into the standard Android testing framework. Gradle is the default automation tool used by Android Studio, the official IDE for Android development. Hence, developers can effortlessly incorporate MINTes-

tar into their existing Gradle-based projects by simply adding the plugin to the Gradle configuration file:

Example 7.7: Integration with Gradle

```
apply plugin: 'mint-tooling'
```

Thus, any MINTestar test run can be defined and executed similarly to existing unit and integration tests, as demonstrated in the simplified example code below.

Example 7.8: Integration with Android testing

```
@org.junit.Test
fun MintExploratoryTestRun() {
    Mint.explore()
}
```

The practical implementation of MINTestar in a testing environment involves several carefully structured steps, each designed to ensure proper setup and effective test execution. The implementation process begins with the integration of MINTestar into the project's build system, typically through Gradle configuration.

The framework's API provides easy-to-use methods for defining test behaviour, setting test conditions, and specifying validation criteria. MINTestar provides flexible configuration options through its builder pattern:

Example 7.9: Rule configuration

```
var mint = MintRule(
    Mint.DefaultBuilder
        .withRule(customRule)        // Add custom rules
        .withSequences(5)            // Set sequence count
        .withStepsPerSequence(30)    // Set steps per sequence
        .build { e -> fail(e) }      // Configure failure
 handling
)
```

7.4 Preliminary evaluation

A preliminary evaluation was done to compare MINTestar with actively maintained random testing tools reported in Section 7.1. This evaluation aims to assess the effectiveness of MINTestar compared to existing random testing tools. Specifically, the following research questions are addressed:

RQ1: How does MINTestar compare to existing scriptless Android testing tools in terms of fault detection capabilities?

RQ2: What is the code coverage effectiveness of MINTestar compared to existing tools?

To answer these research questions, a comparative study was conducted. First, variables were defined to ensure a controlled comparison, followed by a detailed description of the tools and applications used in the evaluation.

7.4.1 Independent and Dependent Variables

The Independent Variables refer to the configuration of the test processes and the selection of the testing tools:

- **Testing tools used:** DroidBot, TESTAR, MINTestar. Each tool was configured with its default settings.

- **Subject applications:** Two open-source applications were selected based on their different functionalities and active maintenance status.
- **Test sequence length:** Fixed 300 actions per run to ensure comparable execution time across tools.
- **Number of test runs:** 10 runs per tool-application combination to account for the random nature of testing.

To answer these research questions, DroidBot and TESTAR (with Random as the ASM for baseline) were selected as the testing tools. For their selection, tools not actively maintained or not publicly available were excluded.

The SUTs were selected randomly from F-Droid, a platform for distributing free and open-source Android apps. The first SUT, *Amaze File Manager*, is an advanced file explorer that allows different operations over the Android file system. Next, Arity is a scientific calculator with function graphing. Both applications were instrumented with Jacoco for code coverage measurement.

To ensure consistency and reliability, a standardised testing protocol was established. The evaluation protocol consisted of executing 300 test actions per run, with 10 test runs conducted for each tool.

To evaluate fault detection capabilities (RQ1) and code coverage effectiveness (RQ2) of the tools, the following Dependent Variables were defined:

- **Fault Detection:** Recorded through Android Log Oracle for runtime exceptions and derived oracles included by default (or not) by each tool.
- **Code Coverage:** Measured using Jacoco, capturing instruction code coverage (ICC).

After each test run, the final code coverage was measured, and all issues detected were recorded for subsequent analysis.

7.4.2 Results

The experimental results reveal distinct differences between MINTestar and existing tools for both bug detection and coverage metrics. The findings are presented

according to the research question.

The comparative analysis (see Table 7.5) of the testing tools showed a nuanced distinction in their bug detection capabilities. MINTestar uniquely excelled in identifying specific types of accessibility issues.

Table 7.5: Comparison of Testing Tools on Various APKs

AUT	LOC	Metric	Droidbot	TESTAR (Random)	MINTestar
Amaze File Manager	84247	ICC	15.6%	**23.7%**	22.5
		Faults	0	0	46
Arity	5197	ICC	26.0%	**66.6%**	40.7%
		Faults	0	0	5

MINTestar found two different types of accessibility problems on multiple widgets: not speakable text and touch target size not large enough, detected by oracles *SpeakableTextPresentCheckOracle* and *TouchTargetSizeCheckOracle* respectively. Figure 7.6 depicts a sample of states where such accessibility issues were found. A total of 13 different widgets did not have a speakable text. Additionally, 10 widgets did not meet the minimum size suggested by the accessibility guidelines. For both problems, widgets with the same functionality were counted as one, such as *Three dots menu items* that indicate "advanced options".

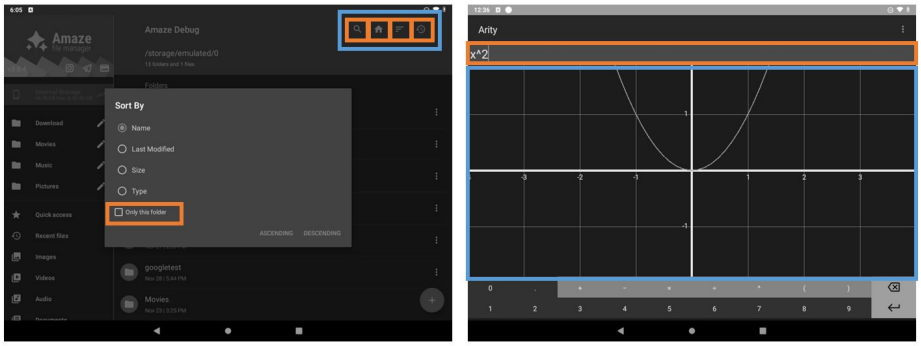


Figure 7.6: Sample of accessibility issues detected by MINTestar

Orange: The touch widget size is not large enough. Blue: Widget without speakable text.

Moreover, multiple exceptions obtained through the Android Log Oracle were

discarded as false positive bugs, except for one identified in Arity. Neither DroidBot nor TESTAR detected any faults during their test runs.

RQ1 answer: *MINTestar shows superior fault detection capabilities specifically for accessibility issues. However, all tools performed similarly for runtime exception detection.*

Moreover, analysis of code coverage metrics (see Table 7.5) shows that MINTestar's effectiveness presents mixed results when compared to existing tools. For the Amaze File Manager application, MINTestar achieved higher instruction coverage than Droidbot but slightly lower than TESTAR. However, when testing the Arity application, MINTestar notably outperformed Droidbot's coverage, though still falling short of TESTAR's results.

MINTestar's lower coverage can be explained by its requirement for a manual fine-tuning of the rules, as it is designed for use by testers of the SUT itself, leveraging their knowledge of its specific nature. This contrasts with the plug-and-play approach of the existing tools, which require minimal setup.

RQ2 answer: *code coverage effectiveness exhibits mixed results, suggesting that MINTestar's specialised ASM can achieve competitive but not superior coverage compared to random exploration approaches.*

7.4.3 Discussion

The experimental evaluation reveals important insights about the trade-offs between code coverage and specialised fault detection. While MINTestar showed lower coverage, it excelled in identifying specific types of accessibility issues that went undetected by other tools. This reflects the understanding in software testing that code coverage is a helpful but not definitive indicator of test quality [212].

Furthermore, MINTestar's specialised focus on certain failure categories, like accessibility, adds a valuable dimension to the testing landscape that the existing tools have not addressed (see Section 7.1).

TESTAR's ASM could be augmented with a configurable rule system similar to MINTestar's, through which more sophisticated testing strategies could be implemented while maintaining TESTAR's platform independence. Such rules could be defined at an abstract level that would work across platforms. Furthermore, platform-specific oracles could be implemented within TESTAR's architecture, allowing specialised checks when testing mobile applications while maintaining compatibility with existing cross-platform oracles.

7.5 Conclusions

This chapter presents MINTestar, a scriptless Android application testing tool that uses probabilistic rule-based exploration. Preliminary results show that MINTestar fills a critical gap in detecting faults like accessibility issues, which are increasingly important in creating inclusive and user-friendly applications. Therefore, rather than viewing MINTestar's performance in isolation, it should be considered part of a diverse toolkit, hence its seamless design for easy integration with standard Android testing frameworks.

MINTestar's approach to Android application testing offers several significant advantages over traditional testing methodologies. The framework's ability to combine automated exploration with scripted testing provides unprecedented flexibility in test creation and execution. This hybrid approach allows teams to leverage the benefits of both testing styles - the thoroughness of automated exploration and the precision of scripted tests.

The framework's state management and oracle systems provide comprehensive coverage and validation capabilities, helping teams identify issues that conventional testing approaches might miss. The integration with Espresso enhances these capabilities further, providing reliable interaction with GUI elements while maintaining the benefits of scriptless testing.

8 Conclusions and future work

"I may not have gone where I intended to go, but I think I have ended up where I needed to be."

Douglas Adams, The Long Dark Tea-Time of the Soul

This chapter synthesises the findings of this thesis by addressing the research questions posed in the Introduction Chapter and providing a perspective on the contributions. The chapter is organised into two main sections. The first section presents a detailed discussion of the research questions and their corresponding answers, while the second section proposes future research directions that build upon the work presented herein.

8.1 Answers to the Research Questions

This section thoroughly examines the six primary research questions, each contributing to a comprehensive understanding and practical application of scriptless GUI testing across diverse software environments.

8.1.1 Evolution of Automated GUI Testing

> RQ1: How has automated GUI testing evolved over time regarding size, research trends, collaboration, authors and publication patterns?

The bibliometric analysis conducted by this thesis (see Chapter 2: Thirthy years of automated GUI testing) has provided a broad historical overview of automated GUI testing, identifying key trends, research patterns, and shifts in testing methodologies. Over the years, GUI testing has transitioned from early manual scripting to scriptless methodologies enhanced by artificial intelligence. This evolution reflects the increasing complexity of software applications and the need for more efficient testing solutions. The field has also shown significant growth, with 41.4% of all papers being published in the last five years.

Figure 8.1 shows how GUI testing techniques have evolved. The timeline marks important milestones, like the shift from manual testing to script-based and model-based approaches and eventually to scriptless and AI-driven methods. Although each approach offers advantages, recent trends favour exploration-based and AI-driven techniques. Additionally, there has been a marked increase in research targeting mobile-based SUTs, demonstrating the growing demand for mobile testing strategies.

Rapid advancements in hardware and software platforms, coupled with the rising complexity of systems, have prompted researchers and practitioners to seek more adaptive, efficient, and intelligent testing solutions. This trajectory underscores the importance of specialised methods for state abstraction and action selection, which this thesis investigates in detail.

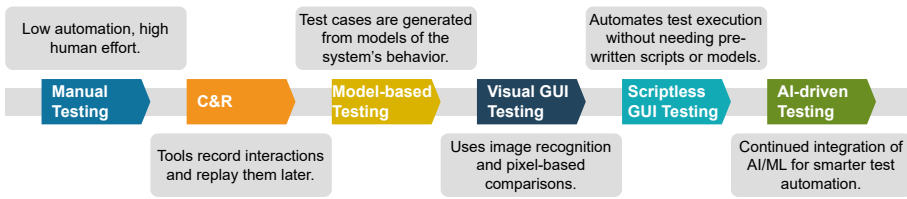


Figure 8.1: The Evolution of GUI Testing Techniques: A timeline depicting the transition from manual to AI-driven GUI testing.

8.1.2 Industrial Insights on Using TESTAR for GUI Testing

RQ2: What general insights do industrial case studies provide about using TESTAR for GUI testing in industry?

The industrial studies (as observed in Chapter 3: TESTAR demonstrate that TESTAR, as a vehicle to explore scriptless GUI testing, is a valuable addition to the testing process, particularly for identifying faults in less probable sequences of actions. Its exploratory nature complements traditionally used testing approaches, such as Capture and Replay, enhancing overall testing coverage and effectiveness. While its efficiency improves over time, as setups are refined and reused, the initial learning curve and configuration effort can be challenging. Subjective satisfaction among testers was generally positive, but improvements in usability and reporting could further enhance its adoption and impact.

A significant takeaway from the case studies is that the setup process is crucial: the case studies underscored the importance of a structured and systematic process for setting up TESTAR. The generic process developed from these industrial studies (see Figure 3.10) provides a structured framework for deploying TESTAR with minimal overhead. This iterative approach ensures that the test environment evolves with each run, improving its coverage and fault detection capabilities over time.

The architectural analogy (see the simplified version in Figure 8.2) for TESTAR's integration, derived from the industrial case studies, illustrates how the tool fits into diverse testing environments. This architectural analogy emphasises the separation of concerns between different actors (i.e., clients, developers, and testers) and illustrates the tool's complementary role in enhancing testing effectiveness.

The findings from those case studies collectively support the conclusion that TESTAR is an effective and resource-efficient exploratory testing tool in industrial contexts. While initial setup and usability challenges persist, its iterative refinement process and ability to reduce manual effort make it a compelling addition to GUI testing strategies. This architectural analogy synthesises the findings from various case studies and provides a scalable blueprint for companies looking to

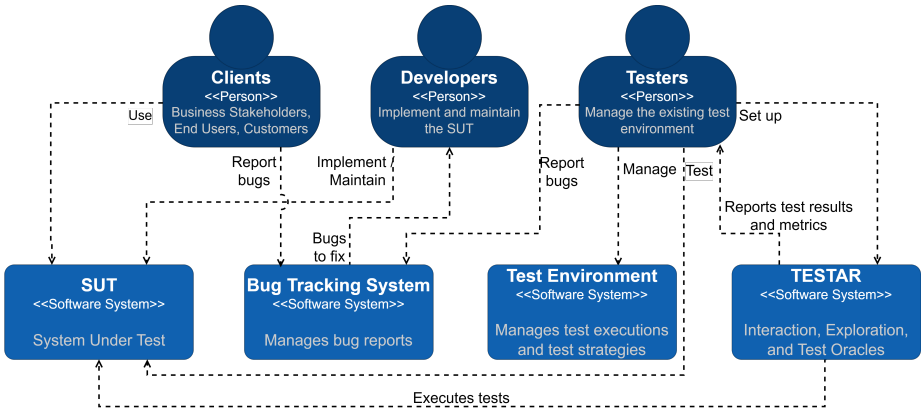


Figure 8.2: Simplified architectural analogy showing the most important components

adopt scriptless testing solutions. It emphasises the collaborative and iterative nature of the testing process, where the complementarity between automated scriptless testing and traditional scripted approaches can lead to more comprehensive software validation with reduced manual effort over time.

8.1.3 Impact of State Abstraction on State Model Inference

RQ3: How does state abstraction in TESTAR influence the inference of state models during on-the-fly exploration with scriptless testing?

The choice of state abstraction in scriptless GUI testing plays a crucial role in the effectiveness of inferred state models, as observed in Chapter 4: *Inferring state models with TESTAR*. Figure 8.1 illustrates the impact of state abstraction. An overly fine-grained abstraction (i.e., using many dynamic attributes), where every minor GUI change results in a distinct state, generates a state explosion, making the inferred model very large and expensive to maintain and interpret. Conversely, if the abstraction is too coarse (e.g., ignoring important widget properties), the model becomes overly abstract and can introduce non-deterministic transitions, reducing its ability to guide test exploration effectively.

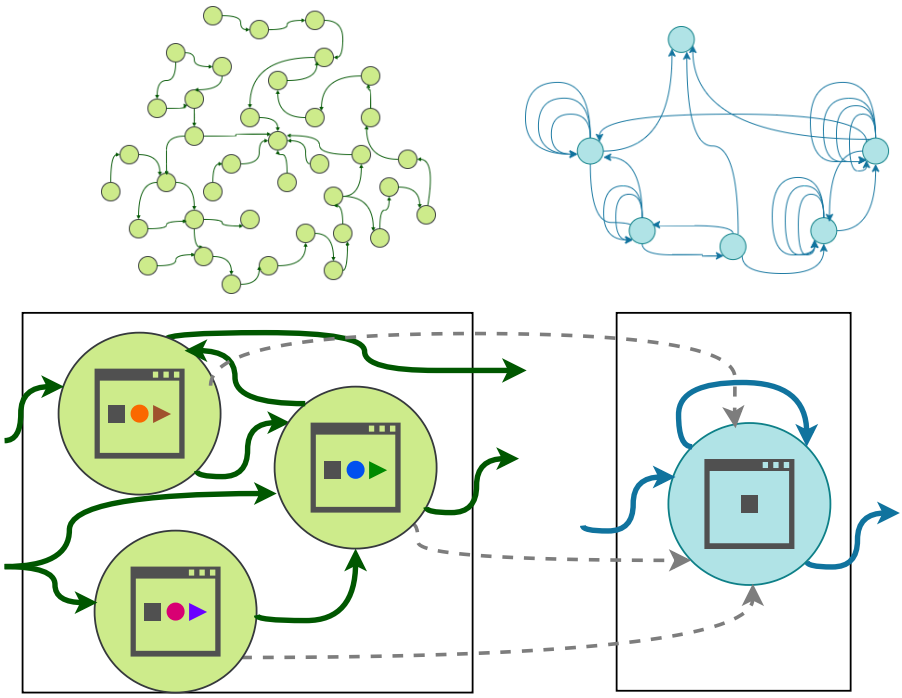


Figure 8.3: Effect of state abstraction in TESTAR.

Striking the right balance, where each state is meaningful but not excessively detailed, improves test coverage (e.g., code coverage, state coverage) and reduces model complexity. The experiments show that an appropriately chosen set of stable widget attributes (sometimes augmented by action-history information) can substantially improve coverage and detect more states than overly abstract or concrete configurations. Different state abstraction mechanisms can either empower or hinder scriptless GUI testing. A thoughtful, SUT-specific abstraction (one that filters out noise yet preserves functionality-relevant properties) yields the most compelling exploration and best coverage results in automated GUI testing.

Additionally, using a model-based Action Selection Mechanism (ASM) that

prioritises unvisited actions/states (rather than purely random action selection) further boosts effectiveness, regardless of the specific abstraction level.

8.1.4 Reward Mechanisms for Exploratory Testing with Reinforcement Learning

RQ4: Which reward mechanism is most effective for exploratory testing with reinforcement learning in TESTAR?

Random exploration can become inefficient in scriptless testing tools by repeatedly visiting states or actions that do not lead to improved coverage. Intelligent or probabilistic approaches address this limitation through adaptive policies that focus on uncovered or critical areas of the GUI. Reinforcement Learning (RL) is one such approach: it rewards transitions leading to previously unexplored states, gradually refining its exploration strategy to balance exploration (of new states) and exploitation (of known high-impact actions).

This thesis investigated several reward mechanisms to enhance exploratory testing using reinforcement learning (see Chapter 5: *Adding intelligence*). Each approach directs the testing process by assigning higher values to certain actions or transitions, encouraging the test generator to select those actions more frequently. Four different rewards were evaluated: State Difference Reward, Action Frequency Reward, State Reward, and a Combined Reward combining these strategies. Table 8.1 presents a summary of these rewards = mechanisms.

The State Difference Reward is designed to encourage exploration by assigning higher rewards to actions that lead to significantly different states. While this promotes novelty, it introduces the Jumping Between States (JBS) problem, where the agent oscillates between highly different states without exploring new states. To address this, rewards based on action frequency memory were introduced.

The Action Frequency Reward tracks action repetition and discourages the agent from over-relying on a particular action, ensuring that different interactions are explored over time. While this approach helps prevent excessive reliance on specific actions, it can also discourage the agent from repeating actions that might

Table 8.1: Comparison of Reward Mechanisms

Reward Mechanism	Description	Key Advantage	Key Limitation
State Difference Reward	Rewards actions that lead to significantly different states	Encourages exploration of new states	Causes JBS (agent jumps between distinct states)
State Reward	Rewards states with many unexplored actions	Best for discovering new interactions	May not ensure broad coverage in complex SUTs
Action Frequency Reward	Discourages the repetition of frequently used actions	Promotes balanced exploration across interconnected states	Can discourage repeating actions necessary for reaching unexplored areas
Combined Reward	Combines the reward strategies	Achieves a trade-off between exploration and JBS mitigation	More complex to implement and tune

lead to states with many unexplored interactions. This limitation can result in suboptimal exploration when certain actions must be executed multiple times to reach deeper or more complex areas of the application.

The State Reward, in contrast, evaluates the level of exploration of the reached states by the number of available unexplored actions. This encourages deeper exploration of individual states while still allowing necessary repetitions when beneficial.

Empirical results indicate that the **State Reward was the best at maximizing exploration** regarding abstract state discovery and action coverage, as it enabled the RL agent to systematically discover unexplored actions. The **Action Frequency Reward was more effective in mitigating repetitive loops** between previously explored paths, and achieved better exploration across interconnected state spaces. The Combined Reward, which combines aspects of both, achieved a well-balanced trade-off, optimizing exploration while mitigating JBS, but it may be more complex to implement and tune.

In general, compared to purely random methods, reward-based exploration guides scriptless testing toward a more thorough exploration of the state space by adapting over time. As some areas of the GUI are covered, the approach naturally shifts attention to the less-explored regions, maximising the overall testing efficiency and effectiveness.

8.1.5 Scriptless GUI Testing and Code Smell Coverage

RQ5: To what extent can scriptless GUI testing with TESTAR provide meaningful coverage of code smells, and how does this relate to traditional test adequacy metrics?

Traditional coverage criteria (e.g., line, branch and state coverage) have long been used to measure the thoroughness of testing activities. While these metrics provide valuable insights into how extensively a software application is exercised during testing, they may fail to capture deeper structural or maintainability problems. Quality-oriented metrics, by contrast, directly measure the extent to which tests exercise areas of the software known to have maintainability issues.

The collaboration with Marviq (see Chapter 6: *Applying it at a company: Marviq*) investigates the extent to which scriptless GUI testing with TESTAR provides meaningful coverage of code smells and its relation to traditional test adequacy metrics.

TESTAR effectively exercises code containing detected code smells, but its ability to cover unique smells varies with testing configurations. Longer test sequences result in more executed smelly code, but the number of unique code smells covered plateaus. The introduction of meaningful form inputs leads to improved exploration, increasing the likelihood of exposing code smells that depend on realistic user input. However, on average, TESTAR covered fewer unique code smells per test sequence when compared to manual testing. Despite this, across all test sequences, TESTAR covered code smells that were not encountered during manual testing.

To understand whether traditional test adequacy metrics can serve as indicators for code smell coverage, Spearman correlation analysis was conducted as show in Figure 8.4. The study revealed that high coverage alone does not necessarily translate into coverage of subtler quality issues. Correlations between code smell coverage and traditional metrics were **moderate** to **weak**, suggesting that higher code coverage does not automatically translate into covering code with deeper structural or maintainability issues. The findings suggest that incor-

porating quality-oriented metrics like code smell coverage can serve as a helpful indicator of test effectiveness.

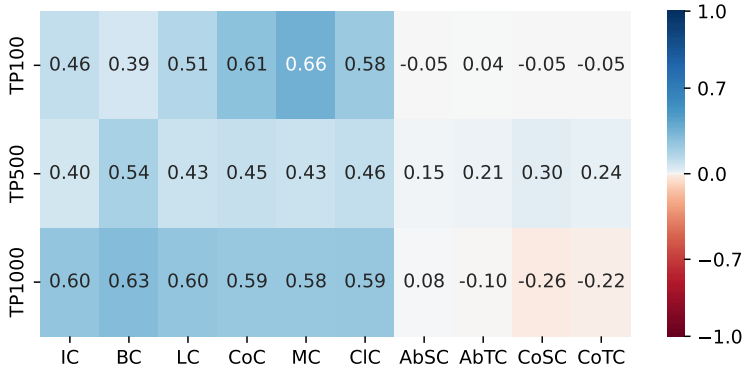


Figure 8.4: Spearman's Correlation: Code Smell Coverage vs Traditional Metrics[1] for different test lengths[2]

While traditional coverage metrics remain valuable for measuring how extensively the code is exercised, code smell coverage highlights parts of the code prone to design flaws. Consequently, using code smell detection alongside traditional coverage metrics results in a more holistic view of test quality.

8.1.6 Adapting Scriptless GUI Testing for Mobile Applications

> RQ6: How can scriptless GUI testing be adapted for mobile applications by improving exploration strategies and integrating mobile-specific testing oracles?

Mobile application testing introduces unique challenges due to platform-specific behaviours, diverse hardware configurations, and touch-based interac-

[1]Metrics: **IC** = Instruction Coverage, **BC** = Branch Coverage, **LC** = Line Coverage, **CoC** = Complexity Coverage, **MC** = Method Coverage, **ClC** = Class Coverage, **AbSC** = Abstract State Coverage, **AbTC** = Abstract Transition Coverage, **CoSC** = Concrete State Coverage, **CoTC** = Concrete Transition Coverage,

[2]Test process: **TP100** = 100 actions, **TP500** = 500 actions, **TP1000** = 1000 actions.

tions. This research investigated improvements to TESTAR's exploration strategies for mobile SUTs (see Chapter 7: *Going mobile: the Android plugin*).

Two complementary solutions were developed to adapt scriptless GUI testing for mobile applications. Table 8.2 summarises the differences between these two approaches. The first involved extending TESTAR to provide cross-platform mobile testing, ensuring a seamless integration with the existing TESTAR framework. This approach allows TESTAR to test Android and iOS applications while maintaining platform independence. This is particularly beneficial for organisations that develop applications for multiple platforms, as it reduces maintenance effort and ensures consistency in testing results. However, this approach introduces some performance overhead due to the additional abstraction layer that Appium provides.

The second solution, MINTestar, was explicitly designed as a lightweight Android-focused tool fully integrated into the Android testing ecosystem. MINTestar enables seamless execution within Android environments, offering direct interactions with native frameworks. This allows for faster execution, deeper integration into development pipelines, and a more seamless experience for testers and developers working within Android environments.

Table 8.2: Comparison of TESTAR (with Appium) and MINTestar for Mobile Testing

Feature	TESTAR + Appium	MINTestar
Platform Support	Android and iOS	Android
Integration	WebDriver-based (Appium)	Direct Android APIs (Espresso, ADB)
Exploration Strategy	All ASMs available with TESTAR	Probabilistic rule-based exploration
Oracles	Generic cross-platform oracles	Mobile-specific oracles (e.g., accessibility)
Ease of Use	Generic TESTAR setup process	Seamless integration into Android workflows

This extension to mobile has introduced adaptations that enhance scriptless testing capabilities for Android applications. These enhancements include mobile-specific oracles, gesture-based interactions, and rule-based probabilistic exploration strategies to improve test execution in mobile environments.

The mobile-specific oracles aim to detect UI inconsistencies and accessibility issues. Domain specificity becomes critical in contexts such as mobile platforms,

where the literature review indicated limited exploration of accessibility-related problems. Incorporating oracles tailored to mobile accessibility, for instance, has the potential to identify subtle or device-specific issues that remain invisible to general-purpose oracles. Addressing such gaps in oracle design can encourage the adoption of scriptless testing tools in industrial environments specialised in mobile applications. Preliminary experiments suggest that these adaptations improve coverage and enhance the detection of subtle, mobile-specific defects, thereby making TESTAR a versatile tool for desktop, web and mobile environments.

8.2 Future Research Directions

Building upon the findings from this research, some areas for future exploration emerge, each offering potential advancements in scriptless GUI testing methodologies. The proposed future directions provide a clear roadmap for ongoing research, enabling both academic and industrial communities to further enhance and adopt scriptless GUI testing methodologies. Figure 8.5 presents an overview of the key research directions categorised into four main areas: **Interaction**, **Exploration**, **Oracles**, and **Test Results and Evaluation**.

- **Expanding to Emerging Interfaces:** Emerging technologies such as **augmented reality** (AR), **virtual reality** (VR), and **voice-controlled applications** present new challenges for GUI testing. Traditional scriptless testing methodologies may not directly translate to these interfaces, necessitating novel research directions to adapt and expand automated testing capabilities.

- **Adaptive State Abstraction Techniques**: State Abstraction is a crucial factor in determining the effectiveness of GUI testing. A promising future research direction includes developing **dynamic state abstraction** methods that adjust based on real-time feedback during testing execution, potentially incorporating **computer vision** for analysing the screenshots in addition to the attributes of the widget tree, and/or visualising the results of

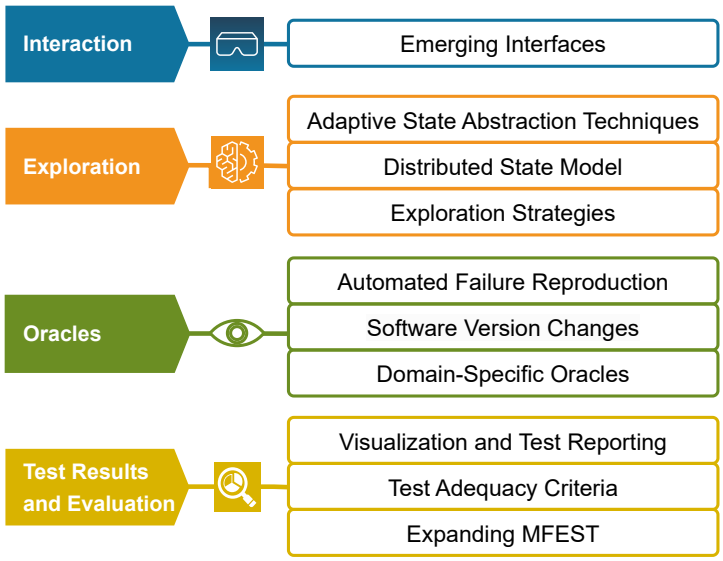


Figure 8.5: Future Research Directions in Scriptless GUI Testing.

state abstraction for the user and learning from the user input to find a suitable level of abstraction.

A novel approach could involve an AI-based system that **continuously refines abstraction levels** during test execution, adjusting dynamically based on the complexity of the GUI. Researchers can improve test efficiency while maintaining **model interpretability** by developing an adaptive mechanism that iteratively refines the abstraction strategy. Such an approach could benefit large-scale or continuously evolving software environments.

- **Distributed State Model:** Beyond abstraction, improving the scalability of GUI testing remains a challenge. Even when the execution is automated, GUI testing is significantly slower than lower-level tests, such as unit testing, due to the need for GUI updates after each action. Future research should explore **parallelising scriptless GUI testing** to speed up execution. For instance, running multiple TESTAR instances in parallel could improve

efficiency, but this requires a robust method for synchronising state models across distributed test executions. Recent research [213] has demonstrated the feasibility of inferring a model with a distributed approach, thus reducing the time required to infer a similar-size state model.

- **Exploration Strategies:** Reinforcement learning has already demonstrated its potential in guiding GUI testing, but there is room for improvement. One primary challenge this research identified is the trade-off between exhaustive exploration and efficient test execution. Future work could investigate the integration of advanced **AI-driven adaptive exploration strategies**, where reinforcement learning (RL) and deep learning models dynamically adjust exploration policies based on real-time feedback.

  Future research should also explore **adaptive reward mechanisms** that evolve dynamically based on real-time testing goals, such as deeper GUI coverage or exploring areas with known technical debt. Code smells could be exploited as part of these reward mechanisms to guide the exploration process toward more critical software weaknesses.

  Recent research [214] has studied the potential of evolutionary-based approaches in test case generation, allowing structured and adaptable exploration strategies. Future work should investigate how **evolutionary algorithms** can be integrated into GUI testing to dynamically refine exploration policies.

  The explosion of Large Language Models (LLMs) offers a new paradigm for intelligent action selection in scriptless testing. Future work could explore how LLMs can assist in dynamically generating GUI interactions by understanding application context and user behaviour patterns. Research should assess the feasibility of **fine-tuning LLMs** to predict meaningful and diverse GUI actions that maximize coverage while reducing redundant interactions. Furthermore, LLMs could be leveraged for adaptive exploration, where they guide test execution toward complex or undertested GUI components based on real-time feedback.

  An additional research direction involves integrating LLMs with reinforce-

ment learning (RL) to balance between random exploration and **goal-directed testing**. LLMs could provide semantic reasoning capabilities, ensuring that generated actions align with realistic user behaviours while reinforcement learning fine-tunes the testing policy over time.

Implementing a **Domain-Specific Language** (DSL) can enhance the usability of scriptless GUI testing by providing a human-readable mechanism for defining test strategies. A DSL could allow users to specify action derivation rules, state abstraction configurations, and oracle definitions in an intuitive and structured format. Future work should focus on designing a flexible and extensible DSL that enables testers to fine-tune test behaviour without requiring deep programming expertise. Incorporating AI-driven abstraction refinements within the DSL could enable self-adaptive scriptless testing, where the exploration model evolves based on application changes and user feedback.

Another area for improvement is **human-in-the-loop** techniques, which explore reinforcement learning strategies where testers or users can guide the training process. This would allow a balance between full automation and human expertise. By leveraging imitation learning, these agents could learn from seasoned testers and gradually develop sophisticated testing strategies. Explainable AI techniques could ensure these agents' decisions remain transparent and interpretable.

Beyond action selection, further research is needed to explore methods for generating more effective actions. For instance, improving **input generation** is a crucial area for future research, aiming to produce meaningful values for text fields automatically.

- **Automated Failure Reproduction:** Debugging remains one of the most time-consuming aspects of software development. Future research should push the boundaries of automated fault reproduction by leveraging inferred state models to trace the shortest paths to faults. This requires recognising whether a failure is unique or duplicates a previously encountered issue. Advanced anomaly detection and clustering techniques could be used to

automate failure classification and reproduction, reducing debugging effort significantly.

A key challenge in failure reproduction is determining whether an observed failure is truly reproducible or influenced by non-determinism. Since GUI testing often encounters dynamic behaviours, failures may not always occur in the same sequence of actions. Future work could explore probabilistic models that assess failure recurrence likelihood, helping testers **prioritise** debugging efforts based on statistical confidence in the reproducibility of an issue.

- **Software version changes:** The inferred models can aid regression testing by automating change detection between consequent versions of the same SUT. Similar approaches have been explored in the Murphy tools [31], and future research could further refine these methodologies by integrating automated GUI comparison techniques. This would allow GUI testing frameworks to efficiently identify regressions and determine the potential impact of software updates. This aligns with ongoing research [215] on automatic inference and recognition of GUI changes between versions for delta testing.

- **Domain-specific Oracles:** The *test oracle problem* remains one of the biggest challenges in automated GUI testing. While existing oracles primarily focus on detecting crashes or functional misbehaviour, future research should explore domain-specific oracles tailored for different testing goals, such as accessibility, usability and security.

  Accessibility and **usability** remain underexplored in automated GUI testing. Future work could investigate new accessibility oracles, heuristic evaluations, and user behaviour simulations. By simulating diverse interaction patterns (e.g., using screen readers or navigating with a keyboard), scriptless testing tools could proactively detect usability issues.

  Security testing is another underexplored area in scriptless GUI testing. Future research should investigate security-specific oracles capable of detecting authentication weaknesses, insecure data handling, and potential

injection vulnerabilities. Automating security validation through exploratory testing could significantly improve software robustness.

Another relevant research direction is improving **internationalisation** testing through automated layout validation. Many localisation issues arise when translated text does not fit within designated UI elements, causing layout distortions. Future work could explore image recognition and text extraction techniques to compare rendered UI elements against expected translations.

- **Test Adequacy Criteria:** Traditional coverage metrics, such as code coverage and state coverage, provide valuable insights into test thoroughness but fail to capture broader aspects of software quality, such as maintainability, performance, and usability. Future research should explore expanding test effectiveness metrics to include additional **quality indicators**, enabling a more comprehensive evaluation of GUI testing.

  By integrating dynamic performance monitoring into GUI testing frameworks, testers could identify laggy UI interactions and inefficient rendering processes, ensuring a smooth user experience. Additionally, quality-driven metrics could help identify parts of the application that require further testing attention.

  Another crucial research area is defining effective **stopping criteria** for GUI testing. One potential criterion is the saturation effect [216], where test execution is halted once no new states, transitions, or faults are discovered over a certain number of iterations. Research could explore the automated detection of test saturation by analysing coverage trends and fault detection rates over time.

  Future work should explore alternative stopping heuristics, such as mutation-based coverage, state-model-based saturation, or diversity-based exploration metrics. These approaches could help define when enough testing has been performed, particularly in complex and dynamically changing GUIs. One possible direction for future research is switching to a different action selection algorithm after reaching saturation. This could involve

transitioning from GUI exploration to combinatorial testing, aiming to discover new state transitions by varying the order of actions within a specific state or the sequence of state transitions.

- **Visualization and Test Reporting:** While TESTAR has proven effective in exploratory testing, its adoption in industrial settings could be further improved by enhancing its reporting and result interpretation mechanisms. One promising direction is the development of interactive dashboards that automatically analyse, categorise, and visualise test results. Such dashboards could integrate:
  - NLP-based log analysis to summarise execution traces and highlight potential issues.
  - Visual heatmaps of explored states to show areas of high interaction density.
  - Automated clustering of failure cases to detect patterns and minimise duplicate issue reporting.

  Future research can enhance test reporting, visualisation, and failure analysis mechanisms to ensure that scriptless GUI testing frameworks provide clear, interpretable, and actionable insights, making them more practical for industrial adoption.

- **Expanding MFEST with Architectural Components:** The architectural analogy from industrial studies highlights key actors and systems in the test automation process. These components interact structurally, influencing the effectiveness, efficiency, and usability of test automation strategies. Future research should extend MFEST (explained in Chapter 3) by explicitly integrating these components into the evaluation methodology.

  While subjective satisfaction is already assessed, further evaluation could focus on how well the testing tool integrates into the existing testing pipeline. The Test Environment, comprising Test Strategy and Test Execution, defines how automated testing tools operate. Future work should

assess whether the tool aligns with predefined test strategies and integrates effectively into execution pipelines. The Bug Tracking System is another crucial element, serving as the primary feedback loop for detected faults. Research should determine how seamlessly the tool communicates findings to the bug tracking system and whether reported faults facilitate debugging for developers.

Fault Detection Rate (FDR) analysis could be enhanced by breaking it down across different layers of the SUT. Research should explore how many faults are detected in GUI components versus business logic layers to identify strengths and weaknesses depending on the software architecture. Evaluating FDR across architectural levels will provide a deeper understanding of coverage and refine testing approaches to better target critical fault-prone areas.

Beyond raw FDR, analyzing the **nature and impact of detected faults** is essential. Future research should examine a new scenario for fault classification and prioritisation, comparing scriptless testing results with the existing manual or scripted approaches. Measuring how often detected faults are marked as critical in the Bug Tracking Systems and how frequently they lead to actual fixes can provide insight into their relevance. Additionally, studies should explore whether detected faults contribute to long-term software improvements, in order to understand the broader impact of scriptless testing beyond detection rates.

The future directions outlined above highlight key opportunities to enhance the capabilities and impact of scriptless GUI testing. By integrating AI-driven adaptive strategies, expanding testing to new domains such as accessibility or security, and improving result interpretation mechanisms, the research community can further advance automated testing methodologies. These directions will play a crucial role in ensuring that scriptless testing remains a viable, scalable, and industry-adopted solution, addressing the challenges posed by emerging technologies, evolving software architectures, and increasing demands for automation.

This thesis has embraced the philosophy of exploring beyond the happy path,

advocating for scriptless approaches that push boundaries. While this work represents a step forward, the journey does not end here. Future research can further refine these techniques, integrating more intelligent decision-making and domain-specific knowledge. As software systems evolve, so too must the approaches used to test them, ensuring that even the less-travelled paths are not overlooked. As Frodo and Sam demonstrated, sometimes the unexpected route is the one that changes everything.

Bibliography

[1] J. R. R. Tolkien. *The Two Towers*. George Allen & Unwin, London, 1954. Part of *The Lord of the Rings*. (Cited on page 1)

[2] What went wrong with hawaii's false emergency alert. Cable News Network (CNN), 2018. [Online; accessed: December 22, 2024]. (Cited on page 3)

[3] Tsb chief paul pester steps down after it meltdown. The Guardian, 2018. [Online; accessed: December 22, 2024]. (Cited on page 3)

[4] Southwest's $140m penalty 'should put all airlines on notice' after travel debacle. Politico, 2023. [Accessed: December 22, 2024]. (Cited on page 3)

[5] LeadDev Staff. Crowdstrike disaster: A lesson about testing. LeadDev, 2024. [Online; accessed: 2025-01-10]. (Cited on page 4)

[6] Ing customer suddenly had access to a complete stranger's account. Nederlandse Omroep Stichting (NOS), 2024. [Online; accessed: 2025-01-10]. (Cited on page 4)

[7] MA Johnson. Automated testing of user interfaces. In *Pacific North West Software Quality conference*, pages 285–293, 1987. (Cited on page 5)

[8] Emil Alégroth and Robert Feldt. *Industrial Application of Visual GUI Testing: Lessons Learned*, pages 127–140. Springer International Publishing, Cham, 2014. (Cited on page 5)

[9] Pekka Aho, Emil Alégroth, Rafael A. P Oliveira, and Tanja E. J. Vos. Evolution of automated regression testing of software systems through the graphical user interface. In *The First International Conference on Advances in Computation, Communications and Services (ACCSE 2016)*, pages 16–21, May 2016. (Cited on page 5)

[10] Tanja E. J. Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, and Ad Mulders. TESTAR – scriptless testing through graphical user interface. *STVR*, 31(3), 2021. (Cited on pages 5, 11, 13, 89, 121, 123, and 169)

[11] Atif M Memon and Mary Lou Soffa. Regression testing of guis. *ACM SIGSOFT software engineering notes*, 28(5):118–127, 2003. (Cited on page 6)

[12] Emil Alégroth, Robert Feldt, and Pirjo Kolström. Maintenance of automated test suites in industry: An empirical study on visual gui testing. *Information and Software Technology*, 73:66–80, 2016. (Cited on page 6)

[13] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 571–579, 2005. (Cited on page 6)

[14] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Robula+: an algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016. (Cited on page 6)

[15] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 503–514, New York, NY, USA, 2018. ACM. (Cited on page 6)

[16] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, page 24–29, New York, NY, USA, 2011. Association for Computing Machinery. (Cited on page 6)

[17] Z. Gao, Z. Chen, Y. Zou, and Atif M. Memon. Sitar: Gui test script repair. *IEEE Transactions on Software Engineering*, 42(2):170–186, Feb 2016. (Cited on page 6)

[18] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. Gui-guided repair of mobile test scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 326–327. IEEE, 2019. (Cited on page 6)

[19] E. Alegroth, M. Nass, and H.H. Olsson. Jautomate: A tool for system- and acceptance-test automation. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 439–446, March 2013. (Cited on page 7)

[20] Sikulix. `http://sikulix.com/`. [Online; accessed 20-12-2019]. (Cited on page 7)

[21] Eyeautomate. `https://eyeautomate.com/`. [Online; accessed 20-12-2019]. (Cited on page 7)

[22] José L. Silva, José Campos, and Ana Paiva. Model-based user interface testing with spec explorer and concurtasktrees. *Electronic Notes in Theoretical Computer Science*, 208:77 – 93, 2008. Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems (FMIS 2007). (Cited on page 7)

[23] Vivien Chinnapongse, Insup Lee, Oleg Sokolsky, Shaohui Wang, and Paul Jones. Model-based testing of gui-driven applications. In Sunggu Lee and

Priya Narasimhan, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 203–214, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (Cited on page 7)

[24] Rodrigo M. L. M. Moreira, Ana Paiva, Miguel Nabuco, and Atif M. Memon. Pattern-based gui testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*, 27(3):e1629, 2017. e1629 stvr.1629. (Cited on page 7)

[25] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4:326–345, 2005. (Cited on page 7)

[26] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based gui testing of an android application. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST, pages 377–86. IEEE Computer Society, 2011. (Cited on page 7)

[27] João Carlos Silva, Carlos Silva, Rui D. Gonçalo, João Saraiva, and José Creissac Campos. The GUISurfer tool: Towards a language independent approach to reverse engineering GUI code. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 181–186. ACM, 2010. (Cited on pages 7 and 91)

[28] Rui Couto, António Nestor Ribeiro, and José Creissac Campos. A patterns based reverse engineering approach for java source code. In *35th IEEE Software Engineering Workshop*, pages 140–147, 2012. (Cited on pages 7 and 91)

[29] Bao N Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated software engineering*, 21(1):65–105, 2014. (Cited on pages 7, 75, and 92)

[30] Pekka Aho, Tomi Räty, and Nadja Menz. Dynamic reverse engineering of GUI models for testing. In *2013 International Conference on Control, Decision and*

Information Technologies (CoDIT), pages 441–447, 2013. (Cited on pages 7 and 92)

[31] Pekka Aho, M. Suarez, T. Kanstren, and Atif M. Memon. Murphy tools: Utilizing extracted gui models for industrial software testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 343–348, March 2014. (Cited on pages 7, 8, 91, 92, 108, and 217)

[32] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1), March 2012. (Cited on pages 7 and 92)

[33] Andres Kull. Automatic GUI model generation: State of the art. In *2012 IEEE 23rd ISSRE Workshops*, pages 207–212. IEEE, 2012. (Cited on pages 7 and 92)

[34] Pekka Aho, Teemu Kanstrén, Tomi Räty, and Juha Röning. Automated extraction of GUI models for testing. volume 95 of *Advances in Computers*, pages 49–112. Elsevier, 2014. (Cited on pages 7, 8, and 92)

[35] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 121–130, 2008. (Cited on pages 7 and 92)

[36] Antonia Bertolino, Andrea Polini, Paola Inverardi, and Henry Muccini. Towards anti-model-based testing. In *In Proc. DSN 2004 (Ext. abstract*, pages 124–125, 2004. (Cited on page 7)

[37] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 260–269, Nov 2003. (Cited on pages 7 and 91)

[38] André MP Grilo, Ana CR Paiva, and João Pascoal Faria. Reverse engineering of GUI models for testing. In *5th ICIST*, pages 1–6. IEEE, 2010. (Cited on pages 8 and 91)

[39] Tanja E. J. Vos and Pekka Aho. Searching for the best test*. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, pages 3–4, May 2017. (Cited on page 8)

[40] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. (Cited on pages 8 and 27)

[41] Gunel Jahangirova. Oracle problem in software testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 444–447, New York, NY, USA, 2017. ACM. (Cited on page 8)

[42] Rafael A.P. Oliveira, Upulee Kanewala, and Paulo A. Nardi. Automated test oracles: State of the art, taxonomies, and trends. *Advances in Computers*, 95:113 – 199, 2014. (Cited on page 8)

[43] Marcel Böhme and Soumya Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, 42(4):345–360, 2015. (Cited on page 9)

[44] Pekka Aho and Tanja E. J. Vos. Challenges in automated testing through graphical user interface. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 118–121, Los Alamitos, CA, USA, Apr 2018. IEEE Computer Society. (Cited on page 9)

[45] Pekka Aho, Tanja E. J. Vos, Otto Sybrandi, Sorin Patrasoiu, Joona Oikarinen, Olivia Rodriguez Valdes, and Lianne V. Hufkens. IVVES (industrial-grade verification and validation of evolving systems). In João Araújo, Jose Luis de la Vara, Isabel Sofia Brito, Nelly Condori-Fernández, Leticia Duboc, Giovanni Giachetti, Beatriz Marín, Estefanía Serral, Alessandra Bagnato, and Lidia López, editors, *Joint Proceedings of RCIS 2022 Workshops and Research Projects Track co-located with the 16th International Conference on Research Challenges in Information Science (RCIS 2022), Barcelona, Spain, May 17-20, 2022*, volume 3144 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022. (Cited on page 10)

[46] Manuela Andreea Petrescu and Simona Motogna. A perspective from large vs small companies adoption of agile methodologies. In *ENASE*, pages 265–272, 2023. (Cited on pages 11 and 151)

[47] M Hossain. Challenges of software quality assurance and testing. *International Journal of Software Engineering and Computer Systems*, 4(1):133–144, 2018. (Cited on pages 11 and 151)

[48] Nelson Vargas, Beatriz Marín, and Giovanni Giachetti. A list of risks and mitigation strategies in agile projects. In *2021 40th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–8. IEEE, 2021. (Cited on pages 11 and 151)

[49] Ing erkent technische storing na problemen met overboekingen. Tweakers, 2023. [Online; accessed 30-January-2025]. (Cited on page 13)

[50] Vijftien uur niet online bankieren bij ing: 'we zijn het niet meer gewend'. Nederlandse Omroep Stichting (NOS), 2023. [Online; accessed 30-January-2025]. (Cited on page 13)

[51] Ing-klant had plots toegang tot de rekening van een wildvreemde. Nederlandse Omroep Stichting (NOS), 2023. [Online; accessed 30-January-2025]. (Cited on page 13)

[52] Uia windows. `https://docs.microsoft.com/en-us/windows/win32/winauto/uiauto-entry-propids`. [Online; accessed 25-12-2019]. (Cited on pages 13 and 57)

[53] Selenium. `https://selenium.dev/`. [Online; accessed 20-12-2019]. (Cited on pages 13, 34, and 56)

[54] Java Access Bridge. `https://docs.oracle.com/javase/8/docs/technotes/guides/access/`, 2025. [Online; accessed 30-January-2025]. (Cited on page 13)

[55] Sebastian Bauersfeld, Tanja E. J. Vos, Nelly Condori-Fernández, Alessandra Bagnato, and Etienne Brosse. Evaluating the TESTAR tool in an industrial

case study. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 4, 2014. (Cited on pages 14, 78, and 79)

[56] Fernando Pastor Ricós, Pekka Aho, Tanja Vos, Ismael Torres Boigues, Ernesto Calás Blasco, and Héctor Martínez Martínez. Deploying testar to enable remote testing in an industrial ci pipeline: A case-based evaluation. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, pages 543–557, Cham, 2020. Springer International Publishing. (Cited on pages 14 and 79)

[57] Sebastian Bauersfeld, A de Rojas, and Tanja E. J. Vos. Evaluating rogue user testing in industry: An experience report. In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–10, May 2014. (Cited on pages 14, 78, 79, and 81)

[58] Hatim Chahim, Mehmet Duran, Tanja E. J. Vos, Pekka Aho, and Nelly Condori Fernandez. Scriptless testing at the gui level in an industrial setting. In Fabiano Dalpiaz, Jelena Zdravkovic, and Pericles Loucopoulos, editors, *Research Challenges in Information Science*, pages 267–284, Cham, 2020. Springer International Publishing. (Cited on pages 14 and 79)

[59] P. Aho, G. Buijs, A. Akin, S. Senturk, F. Pastor Ricos, S. de Gouw, and T. Vos. Applying Scriptless Test Automation on Web Applications from the Financial Sector. In S. Abrahão, editor, *Actas de las XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021)*. Sistedes, 2021. (Cited on pages 14 and 79)

[60] Pekka Aho, Tanja E. J. Vos, Sami Ahonen, Tomi Piirainen, Perttu Moilanen, and Fernando Pastor Ricos. Continuous piloting of an open source test automation tool in an industrial environment. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 1–4. Sistedes, 2019. (Cited on pages 14 and 80)

[61] Mireilla Martinez, Anna I. Esparcia, Urko Rueda, Tanja E. J. Vos, and Carlos Ortega. Automated localisation testing in industry with testar. In Franz Wotawa, Mihai Nica, and Natalia Kushik, editors, *Testing Software and Systems*, pages 241–248, Cham, 2016. Springer International Publishing. (Cited on pages 14 and 80)

[62] Tanja E. J. Vos, Peter M. Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. TESTAR: Tool support for test automation at the user interface level. *Int. J. Inf. Syst. Model. Des.*, 6(3):46–83, July 2015. (Cited on pages 14 and 80)

[63] Axel Bons, Beatriz Marín, Pekka Aho, and Tanja EJ Vos. Scripted and scriptless gui testing for web applications: An industrial case. *Information and Software Technology*, 158:107172, 2023. (Cited on pages 14, 80, 81, and 145)

[64] Thorn Jansen, Fernando Pastor Ricós, Yaping Luo, Kevin Van Der Vlist, Robbert Van Dalen, Pekka Aho, and Tanja EJ Vos. Scriptless gui testing on mobile applications. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 1103–1112. IEEE, 2022. (Cited on pages 14, 80, 148, and 169)

[65] Rafael Ball. *An introduction to bibliometrics : new developments and trends*. Chandos Information Professional Series. Chandos Publishing, Cambridge, Massachusetts, 2018 - 2018. (Cited on page 14)

[66] Roel Wieringa and Maya Daneva. Six strategies for generalizing software engineering theories. *Science of Computer Programming*, 101:136 – 152, 2015. Towards general theories of software engineering. (Cited on pages 14, 15, and 82)

[67] Richard S Sutton. Reinforcement learning: An introduction. *A Bradford Book*, 2018. (Cited on page 17)

[68] Claes Wohlin and Per Runeson. Guiding the selection of research methodology in industry–academia collaboration in software engineering. *Information and software technology*, 140:106678, 2021. (Cited on page 17)

[69] G Ann Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013. (Cited on pages 18 and 158)

[70] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018. (Cited on pages 18, 150, and 158)

[71] Olivia Rodriguez Valdes. Finding the shortest path to reproduce a failure found by testar. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1223–1225, 2019. (Cited on page 21)

[72] Olivia Rodriguez-Valdes. Towards a testing tool that learns to test. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 278–280. IEEE, 2021. (Cited on page 22)

[73] Pekka Aho, Tanja EJ Vos, Otto Sybrandi, Sorin Patrasoiu, Joona Oikarinen, Olivia Rodriguez Valdes, and Lianne V Hufkens. Ivves (industrial-grade verification and validation of evolving systems). In *RCIS Workshops*, 2022. (Cited on page 22)

[74] Selmin Nurcan, Andreas L Opdahl, Haralambos Mouratidis, and Aggeliki Tsohou. Research challenges in information science: Information science and the connected world: 17th international conference, rcis 2023, corfu, greece, may 23–26, 2023, proceedings. 2023. (Cited on page 22)

[75] Henry Small. Visualizing science by citation mapping. *Journal of the American Society for Information Science*, 50(9):799–813, 1999. (Cited on page 25)

[76] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. Graphical user interface testing: Systematic mapping and repository. *IST*, 55(10):1679–1694, October 2013. (Cited on page 25)

[77] Brian C Vickery. Bradford's law of scattering. *Journal of documentation*, 4(3):198–203, 1948. (Cited on pages 26 and 35)

[78] Alfred J Lotka. The frequency distribution of scientific productivity. *Journal of the Washington academy of sciences*, 16(12):317–323, 1926. (Cited on page 26)

[79] M.J. Cobo, A.G. López-Herrera, E. Herrera-Viedma, and F. Herrera. Science mapping software tools: Review, analysis, and cooperative study among tools. *Journal of the American Society for Information Science and Technology*, 62(7):1382–1402, 2011. (Cited on page 28)

[80] Elizabeth S. Vieira and José A. N. F. Gomes. A comparison of Scopus and Web of Science for a typical university. *Scientometrics*, 81(2):587–600, 2009. (Cited on page 28)

[81] Claudio Bustos, Maria Malverde, Pedro L., and Alejandro Díaz-Mujica. Buhos: A web-based systematic literature review management software. 7, 11 2018. (Cited on page 31)

[82] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014. (Cited on page 31)

[83] Andreas Thor, Werner Marx, Loet Leydesdorff, and Lutz Bornmann. Introducing citedreferencesexplorer (crexplorer): A program for reference publication year spectroscopy with cited references standardization. *Journal of Informetrics*, 10(2):503–515, 2016. (Cited on page 31)

[84] Massimo Aria and Corrado Cuccurullo. bibliometrix: An r-tool for comprehensive science mapping analysis. *Journal of informetrics*, 11(4):959–975, 2017. (Cited on page 31)

[85] José A Moral-Muñoz, Enrique Herrera-Viedma, Antonio Santisteban-Espejo, Manuel J Cobo, et al. Software tools for conducting bibliometric analysis in science: An up-to-date review. 2020. (Cited on page 31)

[86] Eric Paulos. The rise of the expert amateur: Diy culture and citizen science. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 181–182, 2009. (Cited on page 34)

[87] Shir Aviv-Reuven and Ariel Rosenfeld. Publication patterns' changes due to the covid-19 pandemic: a longitudinal and short-term scientometric analysis. *Scientometrics*, 126(8):6761–6784, 2021. (Cited on page 34)

[88] Massimo Franceschet. The role of conference publications in cs. *Communications of the ACM*, 53(12):129–132, 2010. (Cited on page 34)

[89] Ferdinand Leimkuhler. An exact formulation of bradford's law. *J. of Documentation*, 1980. (Cited on page 36)

[90] Alberto Martín-Martín, Enrique Orduna-Malea, Mike Thelwall, and Emilio Delgado López-Cózar. Google scholar, web of science, and scopus: A systematic comparison of citations in 252 subject categories. *Journal of Informetrics*, 12(4):1160–1177, 2018. (Cited on page 39)

[91] Aravind MacHiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. pages 224–234, Saint Petersburg, 2013. cited By 397; Conference of 2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 ; Conference Date: 18 August 2013 Through 26 August 2013; Conference Code:99148. (Cited on pages 39 and 40)

[92] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore de Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. pages 258–261, Essen, 2012. cited By 343; Conference of 2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 ; Conference Date: 3 September 2012 Through 7 September 2012; Conference Code:92925. (Cited on page 39)

[93] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? pages 429–440.

Institute of Electrical and Electronics Engineers Inc., 2016. cited By 245; Conference of 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015 ; Conference Date: 9 November 2015 Through 13 November 2015; Conference Code:118982. (Cited on page 39)

[94] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. Cary, NC, 2012. cited By 231; Conference of 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2012 ; Conference Date: 11 November 2012 Through 16 November 2012; Conference Code:94505. (Cited on page 39)

[95] Anneliese Amschler Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4(3):326–345, 2005. cited By 227. (Cited on page 39)

[96] Tom Yeh, Tsunghsiang Chang, and Robert C. Miller. Sikuli: Using gui screenshots for search and automation. pages 183–192, Victoria, BC, 2009. cited By 217; Conference of 22nd Annual ACM Symposium on User Interface Software and Technology, UIST 2009 ; Conference Date: 4 October 2009 Through 7 October 2009; Conference Code:78541. (Cited on page 39)

[97] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. pages 94–105. Association for Computing Machinery, Inc, 2016. cited By 208; Conference of 25th International Symposium on Software Testing and Analysis, ISSTA 2016 ; Conference Date: 18 July 2016 Through 20 July 2016; Conference Code:122744. (Cited on page 39)

[98] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd D. Millstein. Reran: Timing- and touch-sensitive record and replay for android. pages 72–81, San Francisco, CA, 2013. cited By 202; Conference of 2013 35th International Conference on Software Engineering, ICSE 2013 ; Conference Date: 18 May 2013 Through 26 May 2013; Conference Code:100317. (Cited on page 39)

[99] Atif M. Memon. An event-flow model of gui-based applications for testing. *Software Testing Verification and Reliability*, 17(3):137–157, 2007. cited By 193. (Cited on page 39)

[100] Shuai Hao, Bin Liu, Suman Kumar Nath, William G.J. Halfond, and Ramesh Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. pages 204–217, Bretton Woods, NH, 2014. Association for Computing Machinery. cited By 192; Conference of 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2014 ; Conference Date: 16 June 2014 Through 19 June 2014; Conference Code:105809. (Cited on page 39)

[101] Werner Marx, Lutz Bornmann, Andreas Barth, and Loet Leydesdorff. Detecting the historical roots of research fields by reference publication year spectroscopy (rpys). *Journal of the Association for Information Science and Technology*, 65(4):751–764, 2014. (Cited on page 40)

[102] Atif M. Memon. *A comprehensive framework for testing graphical user interfaces*. 2001. Advisors: Mary Lou Soffa and Martha Pollack; Committee members: Prof. Rajiv Gupta (University of Arizona), Prof. Adele E. Howe (Colorado State University), Prof. Lori Pollock (University of Delaware). (Cited on page 40)

[103] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical gui test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, 2001. cited By 187. (Cited on page 40)

[104] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for gui testing. pages 256–267, Vienna, 2001. Association for Computing Machinery (ACM). cited By 166; Conference of 8th Eiropean Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9) ; Conference Date: 10 September 2001 Through 14 September 2001; Conference Code:60512. (Cited on page 40)

[105] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore de Carmine, and Gennaro Imparato. A toolset for gui testing of android ap-

plications. pages 650–653, Riva del Garda,Trento, 2012. cited By 38; Conference of 28th International Conference on Software Maintenance, ICSM 2012 ; Conference Date: 23 September 2012 Through 28 September 2012; Conference Code:95267. (Cited on page 40)

[106] Wontae Choi, George C. Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. pages 623–639, Indianapolis, IN, 2013. cited By 129; Conference of 2013 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2013 ; Conference Date: 29 October 2013 Through 31 October 2013; Conference Code:100913. (Cited on page 40)

[107] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. *International Conference on Fundamental Approaches to Software Engineering, FASE*, 7793 LNCS:250–265, 2013. cited By 185; Conference of 16th International Conference on Fundamental Approaches to Software Engineering, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013 ; Conference Date: 16 March 2013 Through 24 March 2013; Conference Code:95779. (Cited on page 40)

[108] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. pages 641–660, Indianapolis, IN, 2013. cited By 158; Conference of 2013 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2013 ; Conference Date: 29 October 2013 Through 31 October 2013; Conference Code:100913. (Cited on page 40)

[109] Brendan Rousseau and Ronald Rousseau. Lotka: A program to fit a power law distribution to observed frequency data. *Cybermetrics: International Journal of Scientometrics, Informetrics and Bibliometrics*, (4):4, 2000. (Cited on page 41)

[110] Guo Chen and Lu Xiao. Selecting publication keywords for domain analysis in bibliometrics: A comparison of three methods. *J. of Informetrics*, 10:212–223, 02 2016. (Cited on page 44)

[111] Hsin-Ning Su and Pei-Chun Lee. Mapping knowledge structure by keyword co-occurrence: a first look at journal papers in Technology Foresight. *Scientometrics*, 85(1):65–79, October 2010. (Cited on page 44)

[112] Edward Loper and Steven Bird. Nltk: The natural language toolkit. *arXiv 0205028*, 2002. (Cited on page 44)

[113] Olivia Rodríguez, Tanja EJ Vos, Pekka Aho, and Beatriz Marín. 30 years of automated gui testing: a bibliometric analysis. In *QUATIC*, pages 473–488. Springer, 2021. (Cited on page 44)

[114] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. (Cited on pages 50, 127, 141, 155, 157, and 171)

[115] Appium. `http://appium.io/`. [Online; accessed 25-12-2019]. (Cited on page 56)

[116] Mirella Martínez, Anna I. Esparcia-Alcázar, Tanja E. J. Vos, Pekka Aho, and Joan Fons i Cors. Towards automated testing of the internet of things: Results obtained with the testar tool. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems*, pages 375–385, Cham, 2018. Springer International Publishing. (Cited on page 56)

[117] Pekka Aho, M. Suarez, T. Kanstrén, and Atif M. Memon. Murphy tools: Utilizing extracted gui models for industrial software testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 343–348, March 2014. (Cited on page 57)

[118] Noel Nyman. Using monkey test tools - how to find bugs cost-effectively through random testing. *Software Testing & Quality Engineering*, Jan/Feb:18–21, 2000. (Cited on page 69)

[119] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. On the effectiveness of random testing for android. In *A-TEST*, 2018. (Cited on page 69)

[120] Nataniel Borges, Jenny Hotzkow, and Andreas Zeller. Droidmate-2: a platform for android test generation. In *33rd ACM/IEEE ASE*, pages 916–919, 2018. (Cited on page 69)

[121] Anna I Esparcia-Alcázar, Francisco Almenar, Mirella Martínez, Urko Rueda, and Tanja EJ Vos. Q-learning strategies for action selection in the testar automated testing tool. *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)*, pages 130–137, 2016. (Cited on pages 69 and 120)

[122] Pekka Aho, Matias Suarez, Teemu Kanstrén, and Atif M. Memon. Industrial adoption of automatically extracted gui models for testing. volume 1078, pages 49–54. CEUR-WS, 2013. cited By 6; Conference of 3rd International Workshop on Experiences and Empirical Studies in Software Modeling, EESSMod 2013 - Co-located with 16th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2013 ; Conference Date: 1 October 2013; Conference Code:111113. (Cited on page 75)

[123] Pekka Aho, Nadja Menz, Tomi D. Räty, and Ina Schieferdecker. Automated java gui modeling for model-based testing purposes. pages 268–273. IEEE Computer Society, 2011. cited By 21. (Cited on page 75)

[124] Ali Mesbah, Engin Bozdag, and Arie Van Van Deursen. Crawling ajax by inferring user interface state changes. pages 122–134, Yorktown Heights, NY, 2008. cited By 133; Conference of 8th International Conference on Web Engineering, ICWE 2008 ; Conference Date: 14 July 2008 Through 18 July 2008; Conference Code:73518. (Cited on page 75)

[125] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. Webmate: a tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, pages 11–15, 2012. (Cited on page 75)

[126] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. volume 2018-January, pages 280–290. IEEE Computer Society, 2018. cited By 14; Conference of 40th International Conference on Software Engineering, ICSE 2018 ; Conference Date: 27 May 2018 Through 3 June 2018; Conference Code:137142. (Cited on page 75)

[127] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE fifth international conference on software testing, verification and validation*, pages 81–90. IEEE, 2012. (Cited on page 75)

[128] Tanja E. J. Vos, Beatriz Marín, María José Escalona, and Alessandro Marchetto. A methodological framework for evaluating software testing techniques and tools. In *12th International Conference on Quality Software, Xi'an, China, August 27-29*, pages 230–239, 2012. (Cited on pages 77 and 78)

[129] Tanja E. J. Vos. Evolutionary testing for complex systems. *ERCIM News*, 2009(78), 2009. (Cited on page 77)

[130] Tanja E. J. Vos. Continuous evolutionary automated testing for the future internet. *ERCIM News*, 2010(82):50–51, 2010. (Cited on page 77)

[131] B. Kitchenham, L. M. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *Software, IEEE*, 12(4):52 –62, July 1995. (Cited on page 77)

[132] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software eng. *Empirical Softw. Engg.*, 14(2):131–164, 2009. (Cited on page 77)

[133] Martin Host and Per Runeson. Checklists for software engineering case study research. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 479–481, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 77)

[134] B. Kitchenham, S. Linkman, and D. Law. Desmet: a methodology for evaluating software engineering methods and tools. *Computing Control Engineering Journal*, 8(3):120 –126, June 1997. (Cited on page 77)

[135] Floren de Gier, Davy Kager, Stijn de Gouw, and E.J. Tanja Vos. Offline oracles for accessibility evaluation with the TESTAR tool. In *13th RCIS*, pages 1–12, 2019. (Cited on pages 89 and 99)

[136] Pekka Aho, Emil Alégroth, Rafael AP Oliveira, and Tanja EJ Vos. Evolution of automated regression testing of software systems through the graphical user interface. In *1st Int. Conf. on Advances in Computation, Communications and Services*, pages 16–21, 2016. (Cited on pages 90 and 91)

[137] Pekka Aho, Matias Suarez, Atif Memon, and Teemu Kanstrén. Making GUI testing practical: Bridging the gaps. In *2015 12th International Conference on Information Technology-New Generations*, pages 439–444. IEEE, 2015. (Cited on page 90)

[138] Karl Meinke and Neil Walkinshaw. Model-based testing and model inference. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 440–443, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (Cited on page 90)

[139] Alexandre Canny, Philippe Palanque, and David Navarre. Model-based testing of gui applications featuring dynamic instanciation of widgets. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 95–104, 2020. (Cited on page 91)

[140] Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, and Mika Katara. Model-based testing through a gui. In Wolfgang Grieskamp and Carsten

Weise, editors, *Formal Approaches to Software Testing*, pages 16–31, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited on page 91)

[141] Domenico Amalfitano, Anna R. Fasolino, Porfirio Tramontana, and Nicola Amatucci. Considering context events in event-based testing of mobile applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 126–133, March 2013. (Cited on page 92)

[142] Y. Miao and X. Yang. An fsm based gui test automation model. In *2010 11th International Conference on Control Automation Robotics Vision*, pages 120–126, Dec 2010. (Cited on page 92)

[143] Aaron van der Brugge, Fernando Pastor Ricos, Pekka Aho, Beatriz Marín, and Tanja E.J. Vos. Evaluating TESTAR's effectiveness through code coverage. In S. Abrahão Gonzales, editor, *XXV JISBD*. SISTEDES, 2021. (Cited on page 99)

[144] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *33rd ICSE*, page 1–10. ACM, 2011. (Cited on pages 99, 127, and 133)

[145] Jacoco coverage tool. `https://www.jacoco.org/jacoco/`. last accessed 17 Jan 2022. (Cited on page 99)

[146] Rachota timetracker. `http://rachota.sourceforge.net`. Last accessed: 17 Jan 2022. (Cited on page 100)

[147] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. (Cited on pages 115 and 116)

[148] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In *29th SIGSOFT*. ASM, 2020. (Cited on pages 115, 121, 122, 123, and 124)

[149] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012. (Cited on page 116)

[150] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992. (Cited on page 118)

[151] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez. Qbe: Qlearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 105–115, Los Alamitos, CA, USA, apr 2018. IEEE Computer Society. (Cited on page 120)

[152] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. Search-based energy testing of android. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. (Cited on page 120)

[153] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, pages 2–8, New York, NY, USA, 2018. ACM. (Cited on pages 120 and 121)

[154] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 81–90, 2012. (Cited on pages 120 and 121)

[155] Thi Vuong and Shingo Takada. A reinforcement learning based approach to automated testing of android applications. In *9th ACM A-TEST Workshop*, 2018. (Cited on pages 120 and 121)

[156] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *ICSME*. (Cited on pages 120 and 121)

[157] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. Deep reinforcement learning for black-box testing of android apps. *arXiv preprint arXiv:2101.02636*, 2021. (Cited on pages 120 and 121)

[158] Eliane Collins, Arilo Neto, Auri Vincenzi, and José Maldonado. Deep reinforcement learning based android application gui testing. In *SBES*. ACM, 2021. (Cited on pages 120 and 121)

[159] Christian Degott, Borges Jr., Nataniel P., and Andreas Zeller. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 296–306, New York, NY, USA, 2019. ACM. (Cited on pages 121 and 122)

[160] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952. (Cited on page 121)

[161] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996. (Cited on page 131)

[162] Youguang Chen, William Ruys, and George Biros. Knn-dbscan: a dbscan in high dimensions. *ACM Transactions on Parallel Computing*, 2020. (Cited on page 131)

[163] Marcel Jirina. Using singularity exponent in distance based classifier. In *2010 10th international conference on intelligent systems design and applications*, pages 220–224. IEEE, 2010. (Cited on page 131)

[164] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014. (Cited on page 146)

[165] Lech Madeyski. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology*, 52(2):169–184, 2010. (Cited on page 146)

[166] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering*, 29(1):47–94, 2022. (Cited on page 146)

[167] Sonarcube. `https://www.sonarsource.com/products/sonarqube/`. (Cited on page 146)

[168] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, Atif Memon, and Anna Rita Fasolino. Developing and evaluating objective termination criteria for random testing. *ACM Trans. Softw. Eng. Methodol.*, 28(3), July 2019. (Cited on page 148)

[169] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana, and Anna Rita Fasolino. A general framework for comparing automatic testing techniques of android mobile apps. *Journal of Systems and Software*, 125:322–343, 2017. (Cited on page 148)

[170] Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino. Combining automated gui exploration of android apps with capture and replay through machine learning. *Information and Software Technology*, 105:95–116, 2019. (Cited on page 148)

[171] Sergio Di Martino, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. GUI testing of android applications: Investigating the impact of the number of testers on different exploratory testing strategies. *J. Softw. Evol. Process.*, 36(7), 2024. (Cited on page 148)

[172] Hadi Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE, 2015. (Cited on page 148)

[173] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 560–564. IEEE, 2015. (Cited on page 148)

[174] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313, 2013. (Cited on page 148)

[175] Sonali Pradhan, Mitrabinda Ray, and Srikanta Patnaik. Coverage criteria for state-based testing: A systematic review. *International Journal of Information Technology Project Management (IJITPM)*, 10(1):1–20, 2019. (Cited on page 148)

[176] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? pages 429–440, 2015. (Cited on page 149)

[177] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. An empirical study of android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 738–748, 2018. (Cited on page 149)

[178] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3–12. IEEE, 2011. (Cited on page 149)

[179] Aaron van der Brugge, Fernando Pastor-Ricós, Pekka Aho, Beatriz Marín, and Tanja Ernestina Vos. Evaluating testar's effectiveness through code coverage. *Actas de las XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021)*, pages 1–14, 2021. (Cited on page 149)

[180] Domenico Amalfitano, Nicola Amatucci, Atif M Memon, Porfirio Tramontana, and Anna Rita Fasolino. A general framework for comparing automatic testing techniques of android mobile apps. *Journal of Systems and Software*, 125:322–343, 2017. (Cited on page 149)

[181] Eliane Collins, Arilo Neto, Auri Vincenzi, and José Maldonado. Deep reinforcement learning based android application gui testing. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, pages 186–194, 2021. (Cited on page 150)

[182] Shengcheng Yu, Chunrong Fang, Xin Li, Yuchen Ling, Zhenyu Chen, and Zhendong Su. Effective, platform-independent gui testing via image embedding and reinforcement learning. *ACM Transactions on Software Engineering and Methodology*, 33(7), 2024. (Cited on page 150)

[183] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. On the danger of coverage directed test case generation. In *Fundamental Approaches to Software Engineering: 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24-April 1, 2012. Proceedings 15*, pages 409–424. Springer, 2012. (Cited on page 150)

[184] Dávid Tengeri, Árpád Beszédes, Tamás Gergely, László Vidács, Dávid Havas, and Tibor Gyimóthy. Beyond code coverage—an approach for test suite assessment and improvement. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–7. IEEE, 2015. (Cited on page 150)

[185] Atif M Memon. Gui testing: Pitfalls and process. *Computer*, 35(08):87–88, 2002. (Cited on page 150)

[186] Atif M Memon and Qing Xie. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE transactions on software engineering*, 31(10):884–896, 2005. (Cited on page 150)

[187] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. Context-based approach to prioritize code smells for prefactoring. *Journal of Software: Evolution and Process*, 30(6):e1886, 2018. (Cited on page 150)

[188] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Riccardo Roveda. Towards a prioritization of code debt: A code smell intensity in-

dex. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 16–24. IEEE, 2015. (Cited on page 150)

[189] Zadia Codabux and Byron J Williams. Technical debt prioritization using predictive analytics. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 704–706, 2016. (Cited on page 150)

[190] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection. In *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, pages 44–53. IEEE, 2015. (Cited on page 150)

[191] Md Masudur Rahman, Toukir Ahammed, Md Mahbubul Alam Joarder, and Kazi Sakib. Does code smell frequency have a relationship with fault-proneness? In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 261–262, 2023. (Cited on page 150)

[192] Steffen M Olbrich, Daniela S Cruzes, and Dag IK Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *2010 IEEE international conference on software maintenance*, pages 1–10. IEEE, 2010. (Cited on page 150)

[193] Iker Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195, 2008. (Cited on page 150)

[194] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE international conference on software maintenance and evolution (ICSME)*, pages 1–12. IEEE, 2018. (Cited on page 150)

[195] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015. (Cited on page 150)

[196] Roel Wieringa and Maya Daneva. Six strategies for generalizing software engineering theories. *Science of computer programming*, 101:136–152, 2015. (Cited on page 151)

[197] Tanja EJ Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, and Ad Mulders. TESTAR–scriptless testing through graphical user interface. *Software Testing, Verification and Reliability*, 31(3):e1771, 2021. (Cited on pages 151 and 175)

[198] Tanja EJ Vos, Beatriz Marín, Maria Jose Escalona, and Alessandro Marchetto. A methodological framework for evaluating software testing techniques and tools. In *2012 12th international conference on quality software*, pages 230–239. IEEE, 2012. (Cited on page 155)

[199] Marcel Jerzyk and Lech Madeyski. Code smells: A comprehensive online catalog and taxonomy. In *Developments in Information and Knowledge Management Systems for Business Applications: Volume 7*, pages 543–576. Springer, 2023. (Cited on page 158)

[200] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *IEEE/ACM ICSE-C*. IEEE, 2017. (Cited on pages 175 and 176)

[201] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. *Proc. de ESEC/FSE '13*, 2013. (Cited on page 176)

[202] Ting Su. Fsmdroid: Guided gui testing of android apps. In *IEEE/ACM ICSE-C*, 2016. (Cited on page 176)

[203] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In *IEEE/ACM 41st ICSE*, pages 269–280, 2019. (Cited on page 176)

[204] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. *ISSTA'16*, 2016. (Cited on page 178)

[205] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019. (Cited on page 178)

[206] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. Deep reinforcement learning for black-box testing of android apps. *ACM TOSEM*, 2022. (Cited on page 178)

[207] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020. (Cited on page 178)

[208] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lu. Aimdroid: Activity-insulated multi-level automated testing for android applications. *Int. Conf. on Software Maintenance and Evolution*, 2017. (Cited on page 178)

[209] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. An empirical study of functional bugs in android apps. In *ACM SIGSOFT*, 2023. (Cited on page 178)

[210] Android. Android accessibility overview. (accessed: 26.11.2023). `https://developer.android.com/guide/topics/ui/accessibility`. (Cited on page 191)

[211] Android Developers. *Fundamentals of Testing Android Apps*. Google, 2024. Accessed: 2024-11-25. (Cited on page 195)

[212] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004. (Cited on page 200)

[213] Fernando Pastor Ricós, Arend Slomp, Beatriz Marín, Pekka Aho, and Tanja EJ Vos. Distributed state model inference for scriptless gui testing. *Journal of Systems and Software*, 200:111645, 2023. (Cited on page 215)

[214] Lianne V Hufkens, Tanja EJ Vos, and Beatriz Marín. Novelty-driven evolutionary scriptless testing. In *International Conference on Research Challenges in Information Science*, pages 100–108. Springer, 2024. (Cited on page 215)

[215] Fernando Pastor Ricós, Beatriz Marín, Tanja EJ Vos, Rick Neeft, and Pekka Aho. Delta gui change detection using inferred models. *Computer Standards & Interfaces*, page 103925, 2024. (Cited on page 217)

[216] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, Atif Memon, and Anna Rita Fasolino. Developing and evaluating objective termination criteria for random testing. *ACM Trans. Softw. Eng. Methodol.*, 28(3), July 2019. (Cited on page 218)