Automatic Systematic GUI Testing for Web Applications

Thiago Santos de Moura
Federal University of Campina Grande
Campina Grande, Brazil
thiago.moura@copin.ufcg.edu.br

Everton L. G. Alves
Federal University of Campina Grande
Campina Grande, Brazil
everton@computacao.ufcg.edu.br

Cláudio de Souza Baptista
Federal University of Campina Grande
Campina Grande, Brazil
baptista@computacao.ufcg.edu.br

ABSTRACT

Graphical User Interface (GUI) testing plays a fundamental role in ensuring the reliability and quality of web applications by validating functionalities and exposing faults. Automated GUI testing tools are crucial for scaling this process, particularly in identifying visible failures such as crashes, error messages, and unexpected behaviors. However, their application in real-world, large-scale systems poses persistent challenges, including: (i) the automatic and non-redundant discovery of actionable elements, (ii) reliable synchronization with dynamic content, and (iii) efficient and thorough exploration of complex GUIs. This work addresses these challenges through the design and evaluation of three novel techniques: Unique Actionable Elements Search (UAES) for accurate element detection, the Network Wait mechanism for robust synchronization, and the Iterative Deepening URL-Based Search (IDUBS) algorithm to guide scalable and effective exploration. These techniques are integrated into Cytestion, a new automated GUI testing tool for web applications. Cytestion follows a scriptless and progressive testing approach, starting from a seed test and incrementally exploring the interface by leveraging browser console errors, HTTP status codes, and GUI failure indicators as oracles. We evaluated our solution in a series of empirical studies involving four open-source and twenty industrial web applications. Results show that Cytestion significantly outperforms a state-of-the-art GUI testing tool in both fault detection effectiveness and runtime efficiency. In addition to its academic contributions, Cytestion has been adopted as part of the quality gate mechanism of an industry partner, demonstrating its practical applicability and impact.

KEYWORDS

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

1 Introduction

Web development is a fast-paced field shaped by ever-changing demands for high-quality software releases in short timeframes [5]. This context underscores the crucial need to ensure stability and reliability in web applications, where automated testing offers efficiency, cost reduction, and repeatability [3]. At the Graphical User Interface (GUI) level, testing validates functionalities and detects faults in the Application Under Test (AUT).

GUI test suites can be created manually (scripted) or automatically (scriptless) [2]. While scripted testing offers precise functionality validation, it requires intensive maintenance due to frequent GUI changes [10]. Conversely, scriptless testing automatically generates test cases by exploring the AUT, providing adaptability to GUI evolution [4]. This approach is efficient for "faults that cause visible failures", such as failing text messages presented through the GUI (e.g., errors, exceptions), browser console error messages, or unsuccessful results from requests made to the application's server (e.g., an HTTP status code of the 400 or 500 families) [12].

Despite its advantages, the use of scriptless GUI testing for systematically exploring an application can present inherent challenges:

Challenge 1: Unique Discovery of Actionable Elements. Mimicking human interpretation to detect actionable elements, such as buttons, menus, or inputs, in each GUI state is a nontrivial task [1]. A systematic exploration further complicates this process because each newly discovered GUI element must be exercised only once. This challenge involves the need to uniquely discover actionable elements in the AUT to ensure that the exploration remains finite.

Challenge 2: Synchronization. Synchronization problems are common in both scripted and scriptless testing, but they pose greater challenges in the latter because the scriptless tool cannot depend on human-inserted wait mechanisms [10]. It must automatically manage waits to ensure GUI stability before element discovery, action execution, and failure detection. Interacting too early causes flaky tests, while waiting too long risks missing failures.

Challenge 3: Efficient Systematic Exploration. A systematic exploration can generate an exponentially large number of test cases as each state leads to new actionable elements. In complex industrial web applications, this rapid growth makes the testing process lengthy, resulting in test suites that may take hours to execute. An effective integration development workflow requires optimizations that reduce the runtime [11].

Our work advances the state of the art and practice by proposing a specific approach to address each of the reported challenges (Section 2), supported by dedicated empirical studies [6, 8, 9]. In addition, we introduce a novel approach and tool for systematic GUI testing (Section 3) [7].

2 Addressing Scriptless GUI Testing Challenges

Here we discuss how we tackled each of the challenges of scriptless GUI testing.

2.1 Unique Actionable Elements Search

To address Challenge 1, the *Unique Actionable Elements Search* (UAES) approach was developed [9]. UAES automatically and uniquely identifies actionable elements such as buttons, menus, and inputs by treating the Document Object Model (DOM) of web pages as strings. It employs predefined HTML snippets and a string search algorithm to extract relevant elements. The approach then defines a unique locator (key + value) for each element, prioritizing inner text or attributes like id, name, or aria-label, to ensure non-redundancy while also providing a locator that is semantically expressive. It also manages elements across URL changes to avoid rediscovery.

We evaluated UAES in two empirical studies, comparing it against human-based markups across four open-source and twenty industrial web applications [9]. UAES demonstrated high effectiveness, achieving 94.25% accuracy in uniquely identifying actionable elements compared to manual markups. It enhanced fault detection by uncovering production code faults related to previously unmarked elements and improved GUI testing by eliminating the need for intrusive source code modifications.

2.2 The Network Wait Mechanism

Regarding Challenge 2, we introduced the *Network Wait* [6], a novel waiting mechanism that operates by observing network requests made during test execution. It ensures that a test proceeds only when all pending requests have been completed. This is achieved using a counting mechanism that tracks new requests, waits, and decrements as they are resolved. The *Network Wait* mechanism leverages Cypress[1], an automated GUI testing framework, to intercept requests and dynamically adapt to network conditions, balancing precision and efficiency during test execution. As a result, it helps prevent flaky tests and intermittent failures. A reusable implementation of this mechanism is available as an open-source Cypress dependency[2].

We evaluated the quality and impact of the *Network Wait* mechanism through empirical studies [6]. In the first study, we deliberately introduced delays into a testing environment and compared how much of a GUI test suite would break under different waiting strategies. The test suite that used *Network Wait* experienced zero breakages. In a second study, conducted with an industrial test suite, *Network Wait* reduced the breakage rate from 32% to 2%, while also uncovering new waiting issues.

2.3 Iterative Deepening URL-Based Search

To address Challenge 3, the Iterative Deepening URL-Based Search (IDUBS) algorithm was introduced [8]. IDUBS evolves the Iterative Deepening Search algorithm (IDS) by retaining minimal information from prior nodes to establish a fresh starting point in the depth search, thereby eliminating redundancies upon returning to previously visited states. It integrates each GUI state node with its associated URL. When a new URL is discovered, its node becomes a new root, allowing direct access and continued search in that branch, which is feasible in web systems. This approach optimizes test case execution, avoids redundancy of accessed GUI states, and ultimately reduces execution time.

The effectiveness of IDUBS was evaluated in two empirical studies [8]. In industrial web applications, IDUBS achieved a significant reduction in test execution time by 43.60% and decreased test case redundancy by 50%. It also detected 317 faulty states while maintaining comparable code coverage levels. In open-source applications, IDUBS reduced execution time by 39.03% and minimized test case redundancy by 36.01%.

3 Cytestion

The approaches proposed [6, 8, 9] helped us to introduce Cytestion, an automated and systematic GUI approach for systematic GUI testing, which applies a scriptless and progressive approach [7]. It systematically explores a web-based GUI and dynamically builds a test suite for detecting faults that cause visible failures. For that, it applies a practical adaptation of the IDUBS algorithm [8]. Cytestion's test generation process begins with an initial test case that accesses a given URL and waits for the state to be fully loaded using the Network Wait mechanism [6], which is strategically placed after each action to ensure element discovery and fault detection at the right moment.

For each created test case, Cytestion then executed it and verifies whether the test case passed or failed based on three sources of information: browser console failure messages, HTTP status codes of the 400 or 500 families from server requests, or default or customized failure messages displayed in the GUI. If the test case passes, Cytestion discovers new actionable elements in the current state that are not present in the already discovered set, utilizing the UAES approach [9]. This cycle repeats until no new actionable items are found and the systematic exploration is finished.

3.1 The Tool

Cytestion[3] is implemented as a Node.js open-source tool and leverages the Cypress framework for test execution. It uses our implementation dependency for the Network Wait mechanism and also applies a dependency for parallel execution[4]. The tool uses a default UAES configuration to find actionable elements such as buttons, links, menu items, and form inputs, validating their readiness before interaction. Moreover, it handles non-default tags like *div* and *span*, often used as actionable elements. Input configuration is provided via an *.env* file, and test scripts are split for parallel execution. Output includes a Cypress test script for the generated test suite, a catalog file listing all unique actionable elements, and, for detected faults, a separate Cypress test file, screenshots, and videos.

3.2 Evaluation Studies

We conducted two empirical studies to evaluate Cytestion's effectiveness in detecting faults that cause visible failures and its associated costs in terms of execution time. To guide this investigation, we defined two research questions:

RQ_1: Is Cytestion effective for detecting faults that cause visible failures?

RQ_2: How costly is it to use Cytestion?

For comparison purposes, we selected TESTAR [12], a leading scriptless GUI testing tool that adopts a monkey testing strategy.

3.3 Study with Injected Faults

This study used four different size and context open-source web applications: *petclinic*, *bistro restaurant*, *learn educational*, and *school educational*. A total of 165 faults were injected, all manually validated as causing visible failures. For each fault, we executed Cytestion and TESTAR six times with action limits ranging from 100 to 4,000 to maximize detection.

Cytestion detected 83% of all injected faults, while TESTAR's best configuration (4,000 actions) detected 67.2% of these faults. Statistical analysis indicated Cytestion significantly outperformed

[1] https://www.cypress.io/

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

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

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

TESTAR in detecting faults in the *bistro restaurant* and *school educational* apps, and performed equivalently in *petclinic* and *learn educational*, demonstrating its robustness and effectiveness in fault detection across diverse testing scenarios. We conclude, with 95% confidence, that Cytestion was effective in detecting the injected faults leading to visible failures. For RQ_2, the average execution time for Cytestion was 344.84 seconds, while TESTAR's execution time varied from a minimum of 40.53 seconds to a maximum of 1,255.82 seconds for 4,000 actions. Finally, 29 injected faults were not detected by either tool.

3.4 Study with Industrial Applications

A second empirical study involved twenty React-based industrial web applications already in production. All systems were developed by a partner software development company that had undergone testing by both their development teams and the company's QA team before release.

We ran both Cytestion and TESTAR on all twenty applications, collecting the reported faults and execution times. For TESTAR, each application was executed 15 times, with each execution consisting of a sequence of 1,000 actions, totaling 15,000 actions. Any found fault was later submitted and revised by a member of the project's team, and if confirmed as a fault, it was registered as a bug to fix.

Cytestion detected 28 real faults across twenty applications, all confirmed by developers and logged as bugs. Remarkably, 21 of these were found via failed requests and had gone unnoticed through multiple testing rounds by both development and QA teams. In comparison, TESTAR identified only two faults that were missed by Cytestion due to their reliance on random, repetitive actions. Additionally, Cytestion was more efficient, with an average runtime of 1h42min, versus TESTAR's 6h03min.

4 Cytestion on the Industry

Cytestion has delivered concrete and lasting benefits to the industry partner where it was applied. After the empirical studies, the results were presented to the QA team and project managers, who responded positively. They noted that Cytestion consistently uncovered faults often overlooked by developers and QA professionals, especially those related to edge-case interactions and subtle GUI inconsistencies affecting user experience.

Although its execution time is higher, the company found that the quality gains and risk reduction justified the cost. As a result, Cytestion was integrated into the CI/CD pipelines and made a mandatory step before new system releases.

An internal audit later showed that Cytestion identifies, on average, two previously undetected and valid faults per week, evidencing its ongoing value in ensuring system stability and strengthening the QA process.

5 Concluding Remarks

Cytestion represents a significant advancement in automated GUI testing, directly addressing important practical challenges reported by literature, such as synchronization, unique element discovery, and efficient exploration [6, 8, 9], which existing tools had not fully resolved. This dissertation holds significant relevance for the industrial context, advancing scriptless GUI testing and driving automation and reliability in web application development. Our automated approach to exploring the GUI and detecting faults leading to visible failures enables industrial projects to enhance testing processes, efficiently detect faults, and improve software reliability. The findings are also relevant to toolmakers, informing the design and enhancement of other testing tools for more robust and efficient results.

Future research will focus on integrating semantic search techniques using Large Language Models to generate specialized test cases targeting critical functionalities, thereby enhancing fault detection rates. Another priority is to better identify failures by exploring advanced filtering and validation methods, including AI-based techniques such as Convolutional Neural Networks. Additionally, we plan to expand the empirical studies to include more open-source projects, which would provide a broader perspective on Cytestion's performance and applicability across different web applications and development environments.

REFERENCES

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

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

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

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

[5] Imran Akhtar Khan and Roopa Singh. 2012. Quality Assurance And Integration Testing Aspects In Web Based Applications. *ArXiv* abs/1207.3213 (2012). doi:10.5121/ijcsea.2012.2310

[6] Thiago Santos de Moura, Everton L. G. Alves, Regina Letícia Santos Felipe, Cláudio de Souza Baptista, Ismael Raimundo da Silva Neto, and Hugo Feitosa de Figueirêdo. 2024. Addressing the Synchronization Challenge in Cypress End-to-End Tests. In *Proceedings of the XXXVIII Brazilian Symposium on Software Engineering*.

[7] Thiago Santos de Moura, Everton L. G. Alves, Hugo Feitosa de Figueirêdo, and Cláudio de Souza Baptista. 2023. Cytestion: Automated GUI Testing for Web Applications. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 388–397.

[8] Thiago Santos de Moura, Regina Letícia Santos Felipe, Everton L. G. Alves, Pedro Henrique S. C. Gregório, Cláudio de Souza Baptista, and Hugo Feitosa de Figueirêdo. 2024. Iterative Deepening URL-Based Search: Enhancing GUI Testing for Web Applications. In *Proceedings of the XXXVIII Brazilian Symposium on Software Engineering*.

[9] Thiago Santos de Moura, Francisco Igor de Lima Mendes, Everton L. G. Alves, Ismael Raimundo da Silva Neto, and Cláudio de Souza Baptista. 2024. An Automatic Approach for Uniquely Discovering Actionable Elements for Systematic GUI Testing in Web Applications. In *2024 IEEE International Conference on Software Quality, Reliability and Security*. IEEE.

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

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

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