# Towards a Robust Waiting Strategy for Web GUI Testing for an Industrial Software System

Haonan Zhang
University of Waterloo, Canada
haonan.zhang@uwaterloo.ca

Lizhi Liao
Memorial University of Newfoundland, Canada
lizhi.liao@mun.ca

Zishuo Ding
The Hong Kong University of Science and Technology (Guangzhou), China
zishuoding@hkust-gz.edu.cn

Weiyi Shang
University of Waterloo, Canada
wshang@uwaterloo.ca

Nidhi Narula, Catalin Sporea, Andrei Toma, Sarah Sajedi
ERA Environmental, Canada

## ABSTRACT

Automated web GUI testing has been widely adopted since manual testing is time-consuming and tedious. Waiting strategy plays a vital role in automated web GUI testing since it significantly impacts the testing performance. Though important, little focus has been set on the waiting strategies in web GUI testing. Existing waiting strategies either wait for a predetermined time, which is not reliable in a dynamic environment, or only wait for a specific condition to be verified, which is often not robust enough to handle the complicated testing scenarios. In this work, we introduce a robust waiting strategy. Instead of waiting for a predetermined time or waiting for the availability of a particular element, our approach waits for a desired state to reach. This is achieved by capturing the Document Object Models (DOM) at the desired point, followed by an offline analysis to identify the differences between the DOMs associated with every two consecutive test actions. Such differences are used to determine the appropriate waiting time when automatically generating tests. Evaluation results with an industrial web application indicate that our approach produces more robust tests than the conventional waiting strategies used in web GUI testing. Furthermore, our generated tests are more representative of the recorded usage scenarios and are efficient with low overhead in test execution time.

## CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

## KEYWORDS

Automated web GUI testing, waiting strategy, GUI rendering, industrial experience report


*ASE '24, October 27-November 1, 2024, Sacramento, CA, USA*

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695269

**ACM Reference Format:**
Haonan Zhang, Lizhi Liao, Zishuo Ding, Weiyi Shang, and Nidhi Narula, Catalin Sporea, Andrei Toma, Sarah Sajedi. 2024. Towards a Robust Waiting Strategy for Web GUI Testing for an Industrial Software System. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695269

## 1 INTRODUCTION

Modern web applications provide rich functionalities and information to meet users' growing needs for business, daily activities, and entertainment. Graphic User Interfaces (GUIs) play a vital role in web applications as they facilitate user interaction and information access. Therefore, it is important to test the functionalities and responsiveness of GUIs to ensure a seamless user experience without software failures. A typical way to test web application GUIs is through manual verification where software testers manually interact with the application under test (AUT) by performing actions like clicking buttons, based on test specifications, to verify GUI correctness. Although manual testing accurately simulates the real user experience of interaction and ensures test actions are performed correctly, it is extremely tedious, labor-intensive, and consequently costly and error-prone [7, 17]. Our industrial collaborator also suffered the above issues of manual Web GUI testing, yet the importance of Web GUI testing makes it unavoidable in this real-life case.

To mitigate these issues, testing automation frameworks like Selenium [50] are often adopted to streamline the testing process. These frameworks support the Record-and-playback testing for Web GUI. In particular, during a recording phase, testers manually interact with the application while the framework records those interactions. Afterward, the framework can automatically play the recorded interactions, which become automated tests that are conducted on the Web GUIs of the applications. However, when employing such automation frameworks in GUI testing, testers are often required to determine appropriate waiting times between actions. The waiting time is added such that web pages can be fully loaded and interactable before executing subsequent testing actions on the web page.

Currently, the most commonly used waiting strategies in Web GUI testing [37, 43, 45, 48] are *waiting for a predetermined time* [51] and *explicit wait* [53]. *Waiting for a predetermined time* pauses the execution of the test for a certain period, allowing time for the

new page to load. *Explicit wait* operates by waiting for a specific condition of a web page element to be fulfilled, such as the visibility or clickability of an element, before proceeding with the test execution. However, both strategies have their inherent limitations, and we also encounter major challenges when attempting to integrate them into testing a large-scale enterprise web application from our industry collaborator. In particular, we find that by applying the mix of both waiting strategies, the generated tests still often fail due to flakiness. Neither *waiting for a predetermined time* nor *explicit wait* would ensure that after the wait, the web application is ready for the next action. In addition, there is a discrepancy between the generated tests and the original recording of the tests; while the existing waiting strategies cannot address such discrepancy. Finally, we find that after putting many waits into the tests, the duration of the tests often becomes very long, wasting many resources.

Despite the limitations of existing waiting strategies, there exists little work on effectively and appropriately determine the waiting time between actions in GUI automation testing. To the best of our knowledge, the work of Feng et al. [15] and its follow-up work [14] are the only studies that are attempting to address the challenges of current waiting strategies in GUI testing. However, their focus is on Android GUI testing, which differs considerably from web GUI testing [33, 48] because of the different development and execution environments. Additionally, the datasets and models used in their work are specifically tailored for Android GUI automation testing [9, 18, 31, 38, 39, 49], making their approaches inapplicable for resolving waiting issues in the context of our target industrial Web application.

Therefore, in this paper, we propose an approach to assist in conducting the Record-and-playback automated GUI testing for our industrial collaborator. Instead of waiting for a predetermined time or waiting for the availability of a particular element, our approach captures the desired state of the web application during the recording of the tests. Therefore, the tests generated by our approach wait for the appearance of the desired states to automatically perform test actions during testing. In particular, our approach captures the Document Object Models (DOM) at the desired stage of the application during recording. Then, we conduct offline analysis to compare the differences between the captured DOMs of every two consecutive test actions. Such differences are used to determine the appropriate waiting time when automatically generating tests. Our evaluation results with an industrial web application indicate that our approach produces more robust tests than the conventional waiting strategies used in web GUI testing. Furthermore, the generated tests are more representative of the recorded usage scenarios and are efficient with low overhead in test execution time.

The contributions of this paper are as follows:

- To the best of our knowledge, this is the first study that tries to address the challenges of waiting strategies in web GUI testing.
- We propose an easy-to-apply approach that only uses the differential information of the DOMs to assist in waiting for the desired states of the web application under test.
- We develop our approach into a tool, which has been adopted in testing a real-life industrial web application on a daily basis.

**Paper organization.** Section 2 presents the background information about the current practice in automation testing. Section 3 discusses the challenges encountered when applying automated GUI testing in the industry. Section 4 describes our approach to addressing the challenges. Section 5 evaluates our approach. Section 6 presents the related work to our study. Section 7 discusses the threats to the validity of our study. Finally, Section 8 concludes this paper.

## 2 BACKGROUND

In this section, we present background information about record-and-playback automated GUI testing practices and waiting strategies in automated GUI testing.

### 2.1 Record-and-playback automated GUI testing

One of the widely adopted GUI testing techniques is based on a record-and-playback loop during the testing process. Such a technique is also supported by Web GUI testing tools like Selenium IDE [22]. Normally, testers need to perform a set of actions on the web application according to their testing scenarios. During the testers' operations, these actions are recorded and stored by the recording tools. Afterward, testers can playback these recorded test actions when needed. Figure 1 gives an illustrative example of a recording-and-playback testing process for the iCloud login page[1] using Selenium IDE. As shown in the Recording Step, in this scenario, testers first need to open the login page, input their email address, and click the arrow at the end of the input area as highlighted in the red box. Afterward, the testers would need to wait for the password input field to appear, then input the password and move on to the next action.

The recording of the actions is automatically generated by Selenium. Each of the performed actions is recorded as a command that denotes what kind of action is performed, a target that denotes where the action is performed, and a value that denotes the input (if any) from the tester within the corresponding target. After the recording phase, testers can replay the recordings directly from the Selenium IDE. In the table given in the example, the recorded test actions from Selenium IDE are replayed as follows: The iCloud login page is opened using the previously recorded URL. Selenium IDE then targets the account name input field, identified by its element id "*account_name_text_field*", and enters the email address "*admin@icloud.com*". The arrow at the end of the account name input field is identified with the element id "*sign-in*" and then clicked. After several seconds the password input field should appear and Selenium IDE identifies it by its id "*password_text_field*".

In addition to playing back directly from the IDE, testers can also export the recordings to test code and playback the tests from there. As the automatically generated test code in Figure 1 shows, the generated test code adopts Selenium WebDriver APIs [54] to interact with browsers. In the given code, *driver* is an object of the browser driver that is used to relay commands to the browser. Method *findElement* is used to find the target element an action is performed on and *sendKeys* and *click* are actions to perform within the target element.

[1] https://www.icloud.com/

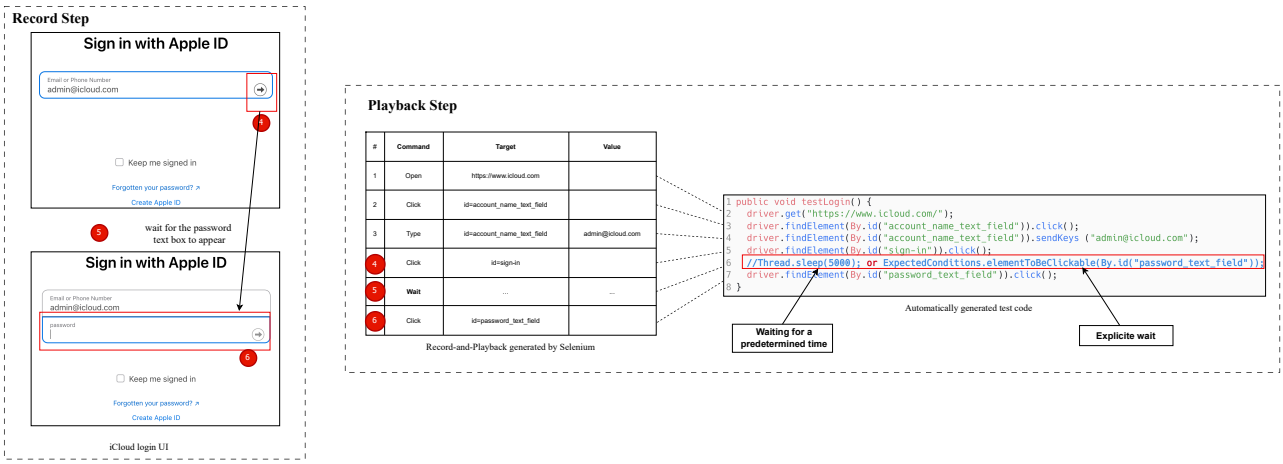



**Figure 1: Web GUI testing example for iCloud login page**

## 2.2 Waiting strategies in automated GUI testing

Although GUI testing tools like Selenium IDE can easily generate the test recordings and scripts, the generated tests can be fragile and flaky [20, 26]. One of the main reasons for the fragility and flakiness are often caused by the need for waiting time between actions [2, 15, 20, 27, 45, 48]. As shown in our example in Figure 1, the testers need to wait for the appearance of the password box in order to perform the action of inputting the password. Therefore, testers often need to manually inject some waiting commands.

There are typically two typical waiting strategies supported by these GUI testing tools. 1) *Waiting for a predetermined time.* One may simply force the test to pause for a certain period of time, (e.g., *Thread sleep*) to avoid processing to the next action too early. Intuitively, this approach would not completely address the flakiness [12, 37] of the generated tests since the responding time of GUI is often variable and unpredictable. Although one may always increase the waiting time to avoid flakiness, such an approach would drastically increase the duration of the tests [15]. 2) *Explicit wait.* One may also ask the test to wait for a certain condition, such as the availability of an element that the next action depends on. GUI test tools like Selenium provide native support on such explicit wait [20, 37, 42, 43, 45, 51, 53]. In the example shown in Figure 1, the test may wait for the password input box to be clickable before performing the action. While addressing some of the flakiness of the tests, a certain condition of an element may not be the desired condition by the testers. For example, a button being ready for clicking does not mean that it is the appropriate time to click that button. The true correct time to click the button depends on the actual testing scenario. We find that such cases often appear in testing real-world web systems and are discussed in the next section (Section 3).

# 3 CHALLENGES OF AUTOMATED GUI TESTING ON AN INDUSTRIAL SYSTEM

In this section, we discuss the challenges that are encountered when automated GUI testing is applied to an industrial software system.

## 3.1 Industrial system under study

The target industrial system of our work is a large-scale software application with a Web-based graphical user interface (GUI). We call the system *ES* in the rest of this paper. *ES* was developed and hosted by our industrial collaborator, which is a leading provider of environmental management solutions for manufacturing companies. *ES* is designed to manage various regulatory reports, such as the Toxic Release Inventory (TRI) [1] and the National Pollutant Release Inventory (NPRI) [13]. It offers reporting services on these regulations for enterprises worldwide, encompassing industries such as automotive, aerospace, oil and gas, and paints and coatings.

Since the users (customers) of *ES* are mainly non-experts in software or computer domains, the web-based GUI is extremely crucial for the adoption of the system by its customers. The web-based GUI is built using the Bootstrap 5 framework and operates on an IIS web server. The server side of *ES* is developed using the Microsoft ASP.NET framework [32]. Different from simple web applications, which primarily involve simple data create-read-update-delete (CRUD) operations, *ES* focuses on handling a large volume and diverse regulatory reports. Consequently, the customers interact with *ES* to dynamically generate and render of numerous forms and sheets through the web-based GUI. There is also a substantial amount of interaction between the client side and the server side for data retrieval and storage.

Moreover, due to the diversity of the customers, many of the forms in *ES* are dynamically generated in response to user actions and each form comprises various elements and widgets, such as drop-downs and calendars, which may have inter-dependencies, like cascading drop-downs. Therefore, the variety and the interactive nature of these forms contribute to a higher level of complexity compared to simple web applications, making it more challenging when directly applying existing automation testing strategies in the context of *ES*.

**GUI testing and waiting strategies.** Given the extensive client base and the critical nature of the service, the reliability of the web application's functionalities is a key priority for our industrial collaborator. For a long period of time, our industrial collaborator has relied on manual GUI testing of *ES*. However, due to the continuous

increase in testing scenarios, this approach has gradually become impractical. During the past two years, the first two authors have closely worked as embedded researchers with our industrial collaborator to help them automate their testing process. Inspired by the existing work in automated GUI testing [10, 14, 19, 20, 36], we have applied a Record-and-Replay testing approach to assist testers in recording their test actions once and replaying them as needed, thereby reducing manual effort. In addition, due to the fact that the test code generated by the recorder contains only the sequence of test actions, i.e., without any waiting interval between two actions, we also need to manually insert numerous waiting commands between test actions with the duration determined based on the personal experience and expertise of the system. However, throughout the collaboration process, we have also identified several major challenges when applying the existing practice in the context of our target industrial web application.

## 3.2 Encountered challenges

In this subsection, we discuss the challenges encountered when conducting record-and-playback GUI testing for the industrial system.

**Challenge 1: Substantial test failures due to flakiness.**

During GUI testing, testers often encounter the need to insert a wait between two actions. Although manually applying the mix of the two waiting strategies (cf. Section 2.2), we are still facing substantial test failures when confronted with the dynamic and complex nature of GUIs. In particular, although we injected *explicit wait* for almost every action in the tests, the complex nature of the web system makes the wait unreliable. In other words, a certain condition of a web element may not ensure the applicability of an action (cf. motivating example 1). To avoid such test failures, we had to inject further wait for a predetermined time, as long as 10 seconds to ensure the test would not crash in the middle. However, the responding time of the elements on the web pages of such a complex system is unpredictable due to factors such as internet connections, the amount of data in the database, GUI layout, and so forth. In short, we were still facing test failures that resulted from the inability to locate the element, e.g., "No Such Element Exception [11]" and had to re-run those tests.

**Challenge 2: Unrepresentativeness to actual scenarios.**

During the GUI testing of our target industrial web application, it is crucial to ensure that the testing scenarios of the system during the playback phase align with the usage scenario envisioned by testers during the test recording phase. However, existing GUI testing and waiting strategies can often lead to inconsistencies between the recorded and the playback of the tests, resulting in unrepresentativeness with actual scenarios. In fact, neither waiting for a certain period of time nor waiting for a certain state of a web element (e.g., clickable of a button) would ensure the web application under test is at the appropriate time to proceed with the next action. In particular, in many cases, waiting for a certain state of a web element (e.g., clickable of a button) often leads to performing the next action too early. We find that all too often when the time a button is ready for clicking, other important related elements are still rending (cf. motivating example 2). For example, on YouTube, the video may not be loaded while the corresponding hyperlinks to other similar videos are already clickable. If in the recording phase, the next step of the test is to view other similar videos, the test would not be representative of the recorded scenario since the video on the current page may be skipped. Such an issue may also appear when waiting for a predetermined waiting time.

**Challenge 3: Prolonged test execution.**

We also encountered the challenge of potentially prolonged test execution times when employing strategies of waiting for fixed lengths, which can adversely affect project timelines and resource allocation. In particular, in order to ensure the test cases can be successfully executed without the impact of flakiness, testers often prefer a conservative long waiting time interval between test actions. However, regardless of whether the web application under test is ready to proceed with the next action, the tests have to wait for a predetermined duration, which sometimes is over 10 seconds. Therefore, this waiting time becomes an overhead and increases the overall time required for test execution. Prolonged test execution times not only delay the feedback to development teams, impeding the project's progress but also increase the testing costs and resource consumption.

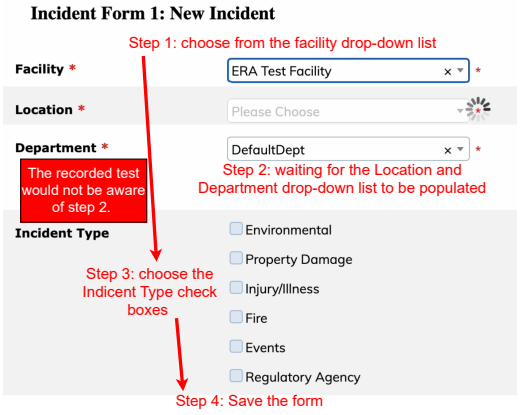



**Figure 2: Real-life motivating example #1**

## 3.3 Motivating examples

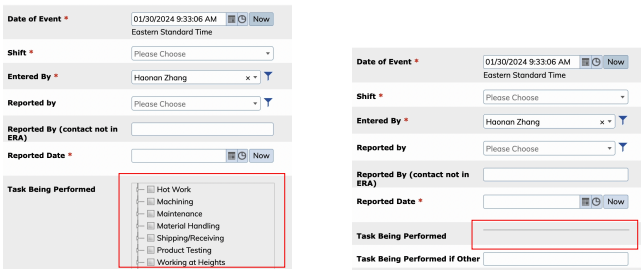
In this subsection, we present two real-life motivating examples to demonstrate the challenges that we encountered in practice.

**Motivating example 1.** Figure 2 presents a motivating example from *ES*. On this page, users of *ES* are required to interact with a sequence of drop-down lists and checkboxes before saving the entered information in a total of four steps. In particular, the user needs to 1) select from the Facility drop-down list, 2) await the dynamic population of the subsequent two drop-down lists, i.e., the Location drop-down list and the Department drop-down list. Then 3) the user needs to choose an option from the Incident Type checkboxes below. Finally, 4) the user saves the completed form.

When recording the tests, the recorded tests would only contain the three steps of actions: 1) selecting from the Facility drop-down list, 2) selecting from the Incident Type checkboxes and 3) saving the form. However, the most important waiting on the population of the Location and Department drop-down lists are missing. Moreover, since there are no direct interactions between the users and the two drop-down lists. The generated test does not even include these two elements. We tried to add *explicit wait* after the first

action. However, the *explicit wait* would wait for the availability of the Incident Type checkboxes, instead of the two drop-down lists. This is because, from the view of the test, the next action after the first step is selecting from the checkboxes. Without manually analyzing the test code to understand this issue and adding ad hoc waiting code customized for this test, we were depending on a very long waiting time to avoid the test failure. This is a real-life example that demonstrates the challenge of test failure due to flakiness (Challenge 1) presented in the previous subsection and the temporary solution of having a long waiting time also contributes to the challenge of prolonged test execution (Challenge 3).

**Motivating example 2.** Figure 3 presents a real-life example related to the discrepancy between the recorded web page and the web page during playback. As shown in Figure 3a, in the scenario under recording, testers wait for the tree of checkboxes (highlighted in the red box) to be rendered before performing the next action. However, as displayed in Figure 3b, during the playback phase, the rendering of the tree of the checkboxes is not finished when the tests start to perform the next action. In this case, the test would not fail since the tree of checkboxes is not a required field of the form. However, without having a tester to check the progress of the automated GUI test, we would not even know the rendering of this tree of check-boxes is often not even executed during tests. This playback of the testing scenario is not exactly representative of the recorded testing scenario. This is a real-life example that demonstrates the challenge that the generated tests are unrepresentative of the actual scenario (Challenge 2).




**(a) Web page during recording** **(b) Web page during playback**

**Figure 3: Real-life motivating example #2**

## 3.4 Our approach to addressing the challenges

By carefully examining the nature of the three challenges that are encountered in practice, we find that the existing automated GUI testing waiting strategies are not designed to truly represent the usage scenarios of actual users, but rather objective measurements like time, or whether an element is clickable. However, in real life, a user would not hold a timer to decide whether to perform the next action. In most cases, a real user would not use whether an element is clickable to drive the decision to go to the next page. Instead, users would see the entire web page as a whole to determine whether they have seen all the needed information to continue their actions.

Therefore, if one would generate a test that exactly replicates the usage scenario of the testers, the test would be much less flaky, since a tester would not attempt to click on something that is not there or not clickable (addressing Challenge 1). The exact replicate usage scenario would not make the test unrepresentative (addressing Challenge 2) and an actual tester's usage scenario would not have extensive waiting steps unless necessary (addressing Challenge 3). Therefore, the goal of our approach is to try to capture the actual usage scenario during the record-and-playback testing.

The state-of-the-art record-and-playback testing practices capture the usage scenario mainly focusing on the interactions between users and the applications while putting less focus on the states of the application. We argue that since the user's behavior is extensively driven by the perception of the entire states of the application under test, we should capture both the actions of users and more importantly the states of the web applications under test. To emphasize, the state information of the web application is based on what the actual users can perceive, instead of whether the page is "loaded", "stable" or "actionable".

Hence, our approach contains three steps: 1) capturing the desired states of the web applications for each action, 2) generating signatures of the states, and 3) generating tests based on the signatures. In the next section (Section 4), we present the detailed implementation of our approach that assists web GUI testing for *ES*.

# 4 IMPLEMENTATION

In this section, we present our actual implementation of a record-and-playback testing approach, that assists our GUI testing for *ES* on a daily basis. Our implementation is based on an extension to the existing Selenium IDE recorder.

## 4.1 Capturing the desired states

To wait for a desired state to be reached in a test scenario, we must first capture the desired states of each action. Selenium IDE already provides the feature that captures the interaction between users and web applications [22]. Therefore, we opt to extend the existing Selenium IDE such that during the recording phase, whenever an interaction is performed by the tester, not only the interaction is captured, we also capture the Document Object Model (DOM) of the corresponding web page and create a reference to the corresponding interaction. One may argue that DOM only represents the UI of the system while the states of the system also include other information, such as its back-end data. However, we consider the information reflected on UI is the most directly perceived by the users, especially during Web GUI testing.

When capturing the DOM of web pages, we encounter situations involving iframes within a web page. Specifically, when the corresponding element of interaction is located in the main window, we exclusively store the source code of the main window, omitting all embedded iframes. On the other hand, for interactions performed on elements nested within an iframe, only the DOM of the respective iframe is collected. This choice of implementation is based on our intuition that if the interacted element is directly embedded in the main window, the focus of the action is on the main window otherwise the users should be more concerned with the elements embedded in the iframe. This also ensures minimal overhead and precise capture of the page's current state before any transitions. It is also worth noting that for an interaction related to an iframe, the id of the iframe (if present) is also recorded to identify the iframes more precisely when playing back the test cases. This is

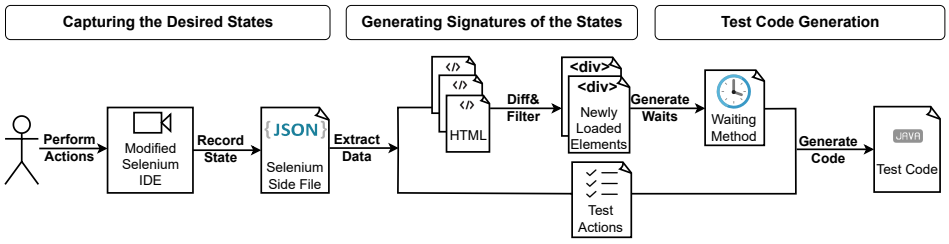



**Figure 4: Overview of our implementation**

an improvement over the default iframe identification by indexes in Selenium IDE, which is often unreliable when dealing with multiple iframes on a single web page.

## 4.2 Generating signatures of the states

Once we have gathered the desired page states, the step is to ascertain when these states are achieved during test execution. Therefore, we would need to generate a signature to represent desired states, such that during testing, the tests can match with the signature to determine whether to perform the next action.

A most naive signature would be directly using the content of the DOM, captured from the last step, to be compared during tests. However, this naive approach has two major issues. First, the direct comparison of two DOMs is costly, introducing much time overhead to the tests. If our approach costs very high runtime overhead during tests, it would have a low advantage over a simple predetermined waiting time. Second, more importantly, it lacks resilience against the trivial differences between the same web page being accessed multiple times. Much information on a web page, such as IDs of elements, is dynamically generated during runtime. Without filtering out such differences, our approach would not match to the desired state.

Instead of treating the entire page as signatures for comparison, we generate the signatures from the differences between the DOMs of the two consecutive actions, which are typically much smaller than the entire DOM. Our intuition is also based on a heuristic that if the differences between the two DOMs can all be observed, it is likely that we have reached the desired state of the web page. In particular, the differential analysis between the DOMs can be conducted offline, after the recording phase is done. With the offline analysis, our approach does not need heavy runtime comparison between signatures; while only a small set of different elements between two desired states are checked during testing. In addition, if there are no changes between the desired states of two actions, the generated tests would not even need to have wait between the two actions. Such an implementation choice would help **reduce the runtime overhead** of our approach.

The most important elements we need to wait for during testing are the added elements. In our industrial context, the content change of the text elements is also very important because the changed data in the sheet cells and other widgets are very likely subject to the actions on the database, which often take a longer time to process. Therefore, we need to apply a diff algorithm to compare two consecutive desired states on the element granularity to identify the added elements and the updated text elements during the transition from a page state to its subsequent page state. There are many algorithms for comparing HTML documents. While many studies leverage these algorithms to assess the similarity of these DOMs and identify the near-duplicated states to streamline the testing steps [56], we focus on the differences of these DOMs to guide signature generation and determine waiting conditions. Recent work [3, 34] finds that the SFTM algorithm [4] has a better performance in comparing HTML documents than others [24, 44]. Therefore we start from using the SFTM algorithm to compare our collected desired page states to identify the potential elements that need to wait for. However, we find that the default output of the SFTM can not be adopted for *ES*. This is because: 1) The SFTM algorithm does not compare the text content in the elements, which means it will take two elements similar in the criteria it cares about but different text content a match, 2) The default output contains many invisible elements that can negatively affect our approach by letting our approach wait for invisible elements to be displayed.

To adapt the SFTM output to *ES*, we first refine SFTM by further investigating the elements that directly contain text content in its output. Specifically, we compare the text content in the pairs of elements that SFTM identifies as remaining the same or updated to extract the elements that have their text content changed. To tackle the second issue, we leverage some heuristic rules to remove the invisible elements. For example, an element is invisible, if the values of the *style* attributes of the element and any of its parents contain "*display: none*" or the value of its or any of its parent's *height* or *width* attribute is *0px*. Since our target industrial web application leverages the Bootstrap 5 frameworks, we also add some framework-specific rules (e.g., *class* contains *sr-only*) to filter out the invisible elements. Similar rules to filter out the invisible elements are also used in some existing work [34]. The adapted SFTM algorithm would increase our approach's **resilience against trivial runtime differences**; while capturing all important differences between two desired states.

## 4.3 Test code generation

After identifying all the elements that we need to wait to ensure the desired state is reached, we generate our test code based on these elements and their corresponding interactions that are already captured by the Selenium IDE. To ensure the visibility of these elements by automated tests, we first need to generate a Selenium locator for each of them to locate them. We implement an approach based on JSoup [21], a Java library for parsing HTML, which generates the corresponding Selenium locators for each element. Similar to Selenium IDE, we generate the locators for an element based on

its id, attributes, and position. The details of the different kinds of locators we generate are presented in Table 1. Among these locator types, the *id* locator is typically the most reliable, whereas the *xpath position* locator is prone to change [28, 41]. However, an *id* locator may not always be available for a given element. In such cases, it is better to prioritize locators based on *link text*, *name*, and *attributes* etc. over the *xpath position* locator, due to their enhanced stability. Therefore, we store the locators in the order outlined in Table 1, selecting the first locator from the list for each element to improve the precision and consistency in our element locating process. However, for text elements, we prioritize all of the other locators before the locators that leverage the text content because the text contents in *ES* are likely to change. For example, the data in the cells of the sheets can be updated quite often. This allows our method more resistant to the dynamic changes in *ES*

**Table 1: A list of different locators used in our test generation**

| Locator name | Description |
|---|---|
| id | Locate an element by its ID |
| link text | Locate a link element by the text it displays |
| name | Locate an element by its name attribute |
| xpath link | Use xpath to find a link element |
| xpath attribute | Use xpath and its attribute to find an element |
| xpath relative id | Use xpath and its parent's or sibling's id to find an element |
| xpath image | Locate an image element with xpath and specific image attributes |
| xpath href | Loacte an anchor element with xpath and the href attribute |
| xpath inner text | Use xpath to find element with specific text content |
| xpath position | Find an element based on its position in the DOM |

At the final step, we generate test code by using FreeMarker [16], a template engine to generate text output. We generate code for actions recorded by Selenium IDE including *open*, *click*, *type*, *sendKeys*, and *selectFrame*. For each of the locators of the elements that are identified for ensuring the reach of the desired states, our waiting method uses *Selenium explicit wait* to wait for its visibility. It then ensures the clickability of the target element. In situations where no new and changed text elements are detected following an action, our method omits the creation of a waiting statement, recognizing that no changes require a pause before continuing with the subsequent action. This strategy ensures only applying waits where necessary, thereby streamlining the testing process.

# 5 EVALUATION

In this section, we introduce the experimental setup for our evaluation and discuss the evaluation results from three aspects: 1) **the robustness**, where we focus on the test failures due to flakiness (cf. Section 3-Challenge 1); 2) **the representativeness**, where we evaluate whether the playback phase aligns with the usage scenario envisioned by testers (cf. Section 3-Challenge 2) and 3) **the efficiency**, which refers to the execution times of the test cases that can be successfully executed without the impact of flakiness (cf. Section 3-Challenge 3).

## 5.1 Experimental setup

*5.1.1 Subjects.* Our evaluation is performed on a business-critical module of *ES* and this module is about creating and maintaining environment-related regulation reports. We collected 26 real-life Web GUI test cases for evaluation, which encompass a range of 11 to 51 steps each and cover almost every aspect related to creating, deleting, editing, and verifying environmental reports. Some actions in these test scenarios are performed in different *iframes*, and many of the test actions are performed on dynamically generated elements and can trigger the page rendering process and data retrieving process, making these test scenarios highly complicated. To give more details, we have made all screen recording videos of the tests and the extracted data available online[2].

*5.1.2 Baselines.* The baselines for our evaluation experiment consist of three different waiting strategies that are commonly used in current GUI testing practice. These include: running test cases using *Selenium explicit wait*, employing the *Thread sleep* strategy, and without using any waiting statements. These baselines are also used in existing work [45]. Prior study [5] points out that a typical time limit that users can tolerate for a page loading is around 3 to 5 seconds, therefore, we set the time limit for *Thread sleep* to 5 seconds. To make our evaluation more comprehensive, we also run the test cases with a doubled time limit (i.e., 10s) for *Thread sleep*.

*5.1.3 Environment.* To generate test cases from these test scenarios, we first use the Selenium IDE tailored for our industrial context to record the test actions defined in the test scenarios. Throughout this process, each action is executed on a stable page determined with human expertise. The experiment is performed in a macOS Ventura 13.1 operating system with an Apple M1 Pro CPU and 32GB RAM. The display we use is a 4K LG 32UN500-W monitor. The window resolution is set to $1920 \times 1080$ during the recording process. Google Chrome (121.0.6167.85 official build for arm64) is chosen as the browser for testing considering its popularity and widespread use in practice. After the test cases are generated, we execute them with different waiting strategies. We collect the console output and record a video of each test execution for further analysis. All test cases are executed in incognito mode with the browser cache disabled, and the browser window size is kept consistent with that used during the recording phase to ensure consistency of the test environment.

## 5.2 Evaluation of robustness

**Motivation.** In the target industrial context, test failures frequently occur due to insufficient waiting periods between actions instead of functional bugs (cf. Section 3-Challenge 1), leading to non-negligible false positives. Therefore, this research question focuses on examining the number of test failures due to flakiness when employing our waiting strategy and the baseline waiting strategies.

**Metrics collection.** We use two metrics to compare the robustness of different approaches: the number of test failures, and the steps executed when the test is terminated (no matter failed or completed). Note that all test cases can be successfully executed manually by testers, ensuring that test failures are highly unlikely to be caused by software bugs. We analyze the console output we collected to determine whether the test failed, identify the reason for the failure, and record the executed steps.

**Results and discussions. Our approach is robust against flakiness and can detect bugs that are previously unknown.** Table 2

[2]https://github.com/senseuwaterloo/ASEIndustryTrack2024

presents the detailed results regarding the performance of different approaches. Overall, our approach significantly outperforms the baselines regarding successfully replaying the test cases. The overall completion rate of the 26 test cases employing our waiting strategies is 92.31%, achieving 73.08%, 88.46%, 92.32%, and 92.32% more successful replays compared to the baselines respectively. It is worth noting that the *Sleep 5s* approach managed only a single successful playback while the *No wait* and *Selenium explicit wait* approaches have zero test successfully reproduced. In terms of executed steps, our approach outperforms all other approaches as well, having 26.85 steps executed on average, achieving an improvement range of 89.75% to 356.63% when compared with the baseline waiting strategies. The results highlight that our proposed approach exhibits better robustness compared to the baseline approaches.

**Table 2: Comparison of the overall completion rates across different approaches**

| Metrics | No Wait | Explicit Wait | Sleep 5s | Sleep 10s | Our Approach |
|---|---|---|---|---|---|
| **Completion** | 0% (0/26) | 0% (0/26) | 3.85% (1/26) | 19.23% (5/26) | 92.31% (24/26) |
| **Step** | 5.88 | 13.27 | 9.96 | 14.15% | 26.85 |

"Completion" denotes the average completion rate (*completed test cases/total test cases*), "Step" denotes the average executed test actions before tests complete or fail.

We conduct a manual analysis of the reasons behind failed test cases and find that in the baseline approaches, failures occur due to interactions with undesired pages. Specifically, some failures stem from performing subsequent actions before the current ones are completed properly, leading to unexpected follow-up actions that are different from the recorded ones, ultimately causing the test to fail (recall the industry example in Section 3.3). Additionally, failures arise when actions are interrupted by the page rendering process. A typical scenario involves the target web application displaying a loading image to signify ongoing content generation, during which the intended actions are disrupted by such visual cues. In the case of our approach, the two test failures are attributed to functional bugs in the web application that were missed by the testers. In some cases, the decision to display an alert window after an action depends on a variable whose value is subject to change through an Ajax call. There are times when, despite the page appearing stable and fully interactive, the underlying Ajax call is still in progress, leading to discrepancies in the variable's state. Such inconsistency affects whether the alert window is shown, causing discrepancies between the recording and playback phases and, consequently, test failures. We reported this issue to the developers, who identified it as a bug and have fixed it. The baseline methods would not be able to detect this bug because they failed before the buggy steps were reached. The results demonstrate that our method is more robust and capable of detecting previously unknown problems.

## 5.3 Evaluation of representativeness

**Motivation.** Prior work usually adopts *Thread sleep* and *Selenium explicit wait* waiting strategies, however, in Section 3, we find that both of them often lead to inconsistencies between the recording and the playback of the tests, resulting in unrepresentativeness of actual scenarios. Therefore, in this section, we investigate the representativeness of our proposed approach when we use it in our target industry system. Given that *Thread sleep* is a static method and often inaccurately estimates the actual loading time [15, 45], we compare our methodology with *Selenium explicit wait*—the only dynamic alternative among the baselines. To further understand whether our waiting strategy aligns with human perception, we analyze the statistical differences between the waiting time frames generated by our method and those observed during manual testing.

**Metrics collection.** We prioritize actions that require a waiting statement generated by our approach, which induces waits only in response to observable page changes. By concentrating on such actions, we effectively exclude those that either do not prompt page modifications or do not require waiting for such changes before continuation. For ease of reference, we term these as **S**ignificant-**W**ait-**R**equired **A**ctions (SWRA).

For every SWRA, we assess the page's status and interactivity at the moment of action. SWRA that are performed when pages do not reach the desired states are termed **U**nstable **A**ctions (UA). Initially, we identify the actions from the generated test code for which our method has produced waiting statements, designating them as SWRA. Subsequently, we collect all successfully executed SWRA from tests executed with *Selenium explicit wait* and our method. Upon reviewing the corresponding execution video for each test, we categorize an action as UA if it is performed before the page reaches the desired state.

To evaluate if our waiting strategy aligns with human perception, we ask an *ES* employee to manually perform the test scenarios, recording videos of each. We then extract the waiting time before each stable SWRA from these videos. To compare the distributions of waiting time from our method and the manual tests, we use the *Mann-Whitney U* test [40], which makes no assumptions about data distribution. We propose two hypotheses:

$H_0$: The distributions of the waiting time under test are the same.
$H_1$: The distributions of the waiting time under test are different.

The test is conducted at a 5% significance level. If $p$-value $\geq 0.05$, we reject $H_1$ and support $H_0$, and vice versa. Reporting only the statistical significance may lead to erroneous results as large samples can produce small $p$-value even for trivial differences [25]. Hence, we also use *Cliff's delta* [8] to quantify the magnitude of the difference between the two distributions.

**Table 3: Comparison of page stability assurance between *Selenium explicit wait* and our approach**

| Metrics | Explicit Wait | Our Approach |
|---|---|---|
| **SWRA** | 157 | 409 |
| **UA** | 29 | 8 |

**Results and discussions. Our approach is more representative of actual scenarios and has negligible statistical differences compared to manual testing.** Table 3 presents the detailed results of the page situations when tests are executed with *Selenium explicit wait* and our approach. Of the actions successfully executed when using *Selenium explicit wait*, 157 of them are SWRA, among which 29 of them are UA, making 18.47% of the Significant-Wait-Required Actions performed on unstable web pages. In terms of our approach, 409 SWRA are successfully executed and eight (1.96%) of them are performed on unstable web pages (UA). Such results indicate that compared to *Selenium explicit wait*, our method can better help

SWRA to be successfully executed and avoid UA happening. We then analyze the results of the *Mann-Whitney U* test. The *p*-value is 0.70 ($\geq$ 0.05) and the *Cliff's delta* is negligible, which indicates that there is no significant difference between our approach and manual testing regarding the waiting time and our approach aligns with user perception.

We further investigate the eight UA that happened in our approach and find that these cases are all related to the greyed-out dropdown lists that have their content already filled out. In a test scenario, we need to fill out a form and save it, and the content of the form will grey out. We then navigate to the next page and navigate back. However, sometimes after navigating back, there will be a loading sign beside the dropdown even though it is filled out and greyed out. Such a sign indicates some process (e.g., database refresh) on the server is undergoing and our method can not identify the server status therefore mistakenly takes the page as stable.

## 5.4 Evaluation of efficiency

**Motivation.** Another challenge we encounter when conducting GUI testing for the industrial system is a prolonged test execution. As the responding time of the elements on the web pages is usually unpredictable, testers often set a conservative long waiting time interval between test actions, negatively impacting testing efficiency. Therefore, in this section, we would like to explore the efficiency of our approach.

**Metrics collection.** We evaluate the efficiency from two aspects: 1) The execution time of the completed tests. We first select the completed tests and then compare the execution times between different approaches. We analyze the recorded videos to identify the period from the opening to the closing of the browser as the test execution time; 2) the overhead of certain actions. For SWRA executed via our method and not classified as UA, we further examine the video frames manually to evaluate our approach's overhead. This involves determining the timestamp of the frame at which the page becomes stable and the timestamp of the frame where the action is just executed, thereby calculating the period between these two timestamps as the overhead.

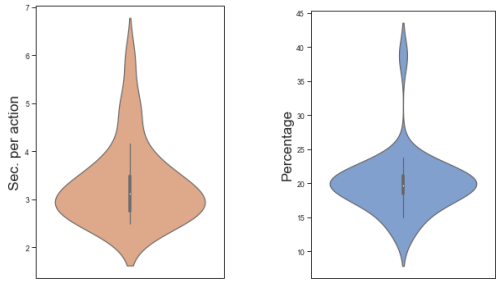
**Results and discussions. Our approach outperforms the baselines in testing efficiency and maintains a low overhead at both the action level and test case level on average.** Table 4 shows the results of the execution time of completed test cases. Compared to our approach, the *Sleep 5s* strategy requires 22.64% (29.7 seconds) more time to playback test case 317423, whereas *Sleep 10s* needs 54.96% (62.05 seconds), 70.44% (92.39 seconds), 42.12% (67.47 seconds), 56.43% (109.76 seconds), and 75.06% (131.18 seconds) additional time for test cases 317421, 317423, 317425, 317428, and 317429, respectively. This underlines the efficiency of our approach, which outpaces the *Thread sleep* strategy in time efficiency while maintaining equivalent completion rates.

For the 401 (409 - 8) SWRA (cf. Table 3) successfully executed on desired pages in our approach, we investigate the potential overhead our method may introduce on average for each action in every test case. Figure 5a demonstrates the distribution of average overhead at the action level for each test case. It is evident that the average overhead per action per test case is under 7 seconds,

**Table 4: Comparison of the execution times of the completed test cases across different approaches**

| Test | Sleep 5s | Sleep 10s | Our Approach |
|---|---|---|---|
| 317421 | N/A | 174.95 | 112.90 |
| 317423 | 160.86 | 223.55 | 131.16 |
| 317425 | N/A | 227.67 | 160.20 |
| 317428 | N/A | 304.25 | 194.49 |
| 317429 | N/A | 305.95 | 174.77 |

with the majority being less than 5 seconds. We further analyze the overhead of our approach at the test case level as outlined in Figure 5b. When examining the overhead as a percentage of the total execution time for each test case, it spans from 12.67% to 38.67%, with an average additional time of 20.13% per test case. Furthermore, the average overhead attributed to each SWRA per test case is 3.16 seconds. While the average additional time of 20.13% per test case might appear significant, we contend that the extra 3.16 seconds per related event is reasonable, considering that manual actions performed by testers on a web page also typically require several seconds.




**(a) O/StableSWRA** **(b) O/T**

**Figure 5: Analysis of the overhead of our approach at action level and test level. "StableSWRA" denotes the Significant-Wait-Required Actions executed on pages with desired states, "O" denotes the overhead time in a test, and "T" denotes the total execution time of a test in Table 2**

# 6 RELATED WORK

In this section, we discuss prior studies that are related to our work. Overall, these studies can be categorized into two categories: 1) empirical studies of web GUI testing; and 2) improving web GUI test cases.

## 6.1 Empirical studies of web GUI testing

Given the importance of web GUI testing for ensuring web application quality, significant research has been conducted to address its challenges. Luo et al. [37] identified the discrepancy between page rendering time and waiting time in test code as a major cause of flaky tests. Leotta et al. [26] found that while the Record-and-Replay approach is less costly for developing test cases, it is more expensive to maintain compared to programmable web testing. Another study by Leotta et al. [27] revealed that the main challenge with using Selenium for GUI testing is the flakiness and

fragility of tests. Presler-Marshall et al. [45] observed that different Selenium configurations impact test flakiness, particularly due to waiting strategies. Hammoudi et al. [20] discovered that insufficient waiting time is a key reason for Record-and-Replay test failures. Likewise, Nass et al. [42] identified waiting time issues as a root cause of many challenges in GUI test automation. Ricca et al. [47] highlighted brittleness as one of the three major problems hindering the automation of web GUI testing.

In almost all of the aforementioned studies, test fragility caused by improper waiting time is mentioned in their outcomes. However, little focus has been set on tackling this issue in the existing literature. In our work, we target a real-world industrial web application and propose a more robust waiting approach to improve the robustness, representativeness, and efficiency of web GUI testing.

## 6.2 Improving web GUI test cases

There are also many studies that focus on improving and fixing web GUI test cases. For example, Leotta et al. [29] proposed an approach to determine the best locator among candidates to enhance test robustness. Their later works [28, 30] introduced methods to generate more robust XPath locators by learning from fragile HTML properties. Nass et al. [41] developed a similarity-based algorithm using weighted similarity scores to locate target elements. Other studies address fixing test cases for new web application releases. Choudhary et al. [6] identified behavioral differences in test cases across successive versions and suggested fixes. Hammoudi et al. [19] proposed an incremental test repair approach, while Stocco et al. [52] used visual information for test repair. Imtiaz et al. [23] and Long et al. [36] focused on fixing test cases from Record-and-Replay automation techniques. Brisset et al. [3] used a tree-matching algorithm, Lin et al. [34] developed an iterative matching algorithm, and Qi et al. [46] leveraged semantic information from test execution to fix test cases. Xu et al. [55] explored using ChatGPT for test case repair.

Most studies focus on improving locators or fixing missing elements in test code, with few addressing waiting strategies in web GUI test cases. Olianas et al. [43] and Liu et al. [35] proposed replacing *Thread sleep* with explicit wait and deciding where to insert explicit wait, respectively. However, these methods can still lead to flaky test results (cf. Section 2.2). Unlike that, our work proposes a more robust waiting strategy by considering user actions and the state of web applications, ensuring GUI testing of real-world industrial web applications truly represents actual usage scenarios.

# 7 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our work.

**Construct validity.** The threats to our construct validity can arise from how we collect the metrics for our evaluation. Currently, there is no available way to measure exactly when the page is ready to interact in a GUI test execution, therefore we have to record a video of the test execution and manually analyze the video frame by frame to identify the timestamp of the frame where the web page just become stable and interactive. Since different people may have different standards of a stable and interactive page, our collected data may not accurately reflect other people's perceptions of the stability of the web pages. However, to the best of our knowledge, we are the first to evaluate the overhead of a waiting strategy in Web GUI testing and prior studies [14, 15] only perform evaluations related to the passing rates and execution times of the test cases. Our evaluation metrics may inspire future studies to consider this aspect for evaluation.

**External validity.** The threats to the external validity of our work can stem from our target subject. Our study is conducted on a large-scale web application that is developed with Bootstrap 5 and ASP.NET and has many years of development and maintenance history. Although our target industrial system has a certain representativeness of model web applications, some of our findings may not be directly generalizable to other web applications with different frameworks or contexts. However, our approach can be readily adapted to other industrial environments with a reasonable engineering effort. By tailoring the Selenium IDE to collect the DOM states, comparing these DOM states using a diff algorithm, and generating code using the Selenium framework and a code generation template, researchers or practitioners who are interested can easily apply our approach to their respective contexts.

**Internal validity.** The threats to internal validity can mainly come from the environment in which our experiments are performed. We have tried our best to keep our experimental settings consistent by using the same browser version, throttling the network from the browser configuration, disabling the browser cache, and using the same window size. Nevertheless, there may still exist some other factors that are beyond our control and pose an impact on our experimental results. For instance, since we do not have total control over the server side of our target industrial web application, the performance variability of the server side may result in some bias to our results.

# 8 CONCLUSION

This paper presents a robust waiting strategy in automated web GUI testing in the context of a real-world industrial web application. Instead of waiting for a predetermined time or waiting for the availability of a particular element, our approach waits for a desired state to reach. To achieve this, we first capture the Document Object Models (DOM) at the desired point and then analyze such data offline to identify the differences between the DOMs associated with every two consecutive test actions. Such differences are further used to determine the appropriate waiting time when automatically generating tests. The industrial experimental results demonstrate that with our proposed waiting strategy, we can produce more robust tests than the conventional waiting strategies used in web GUI testing. Furthermore, the generated tests are more representative of the recorded usage scenarios and are efficient with low overhead in terms of the test execution time.

# ACKNOWLEDGMENTS

We are grateful to ERA Environmental Management Solutions for providing access to the industrial system used in our study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of ERA Environmental Management Solutions and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of ERA Environmental Management Solutions' products.

# REFERENCES


[1] U.S. Environmental Protection Agency. 2024. Toxics Release Inventory (TRI) Program. Retrieved January 16, 2024 from https://www.epa.gov/toxics-release-inventory-tri-program

[2] Emil Alégroth and Robert Feldt. 2017. On the Long-Term Use of Visual Gui Testing in Industrial Practice: A Case Study. *Empirical Softw. Engg.* 22, 6 (dec 2017), 2937–2971.

[3] Sacha Brisset, Romain Rouvoy, Lionel Seinturier, and Renaud Pawlak. 2022. Erratum: Leveraging Flexible Tree Matching to repair broken locators in web automation scripts. *Inf. Softw. Technol.* 144 (2022), 106754.

[4] Sacha Brisset, Romain Rouvoy, Lionel Seinturier, and Renaud Pawlak. 2023. SFTM: Fast matching of web pages using Similarity-based Flexible Tree Matching. *Information Systems* 112 (2023), 102126.

[5] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. 2015. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 439–453.

[6] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. WATER: Web Application TEst Repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering* (Toronto, Ontario, Canada) *(ETSE '11)*. Association for Computing Machinery, New York, NY, USA, 24–29.

[7] Laurent Christophe, Reinout Stevens, Coen De Roover, and Wolfgang De Meuter. 2014. Prevalence and Maintenance of Automated Functional Tests for Web Applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 141–150.

[8] N. Cliff. 1996. *Ordinal Methods for Behavioral Data Analysis*. Erlbaum.

[9] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 845–854.

[10] Felix Dobslaw, Robert Feldt, David Michaëlsson, Patrik Haar, Francisco Gomes de Oliveira Neto, and Richard Torkar. 2019. Estimating Return on Investment for GUI Test Automation Frameworks. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 271–282.

[11] Selenium Project Documentation. 2023. No Such Element Exception. Retrieved August 21, 2023 from https://www.selenium.dev/documentation/webdriver/troubleshooting/errors/#no-such-element-exception

[12] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 830–840.

[13] Environment and Natural Resources. 2024. National Pollutant Release Inventory. Retrieved January 16, 2024 from https://www.canada.ca/en/services/environment/pollution-waste-management/national-pollutant-release-inventory.html

[14] Sidong Feng, Haochuan Lu, Ting Xiong, Yuetang Deng, and Chunyang Chen. 2023. Towards Efficient Record and Replay: A Case Study in WeChat. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1681–1692.

[15] Sidong Feng, Mulong Xie, and Chunyang Chen. 2023. Efficiency Matters: Speeding Up Automated Testing with GUI Rendering Inference. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 906–918.

[16] Apache Software Foundation. 2023. Apache FreeMarker is a template engine to generate text output based on templates and changing data. Retrieved August 21, 2023 from https://freemarker.apache.org/index.html

[17] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and evolving GUI-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*. 408–418.

[18] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications Via Model Abstraction and Refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 269–280.

[19] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: an incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 751–762.

[20] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. 2016. Why do Record/Replay Tests of Web Applications Break?. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 180–190.

[21] Jonathan Hedley. 2023. JSoup is a Java library that simplifies working with real-world HTML and XML. Retrieved August 21, 2023 from https://jsoup.org/

[22] Selenium IDE. 2023. Open source record and playback test automation for the web. Retrieved August 21, 2023 from https://www.selenium.dev/selenium-ide/

[23] Javaria Imtiaz, Muhammad Zohaib Iqbal, and Muhammad Uzair Khan. 2021. An automated model-based approach to repair test suites of evolving web applications. *J. Syst. Softw.* 171 (2021), 110841.

[24] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, Tim Roughgarden, and Scott R. Klemmer. 2011. Flexible Tree Matching. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three* (Barcelona, Catalonia, Spain) *(IJCAI'11)*. AAAI Press, 2674–2679.

[25] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software Microbenchmarking in the Cloud. How Bad is It Really? *Empirical Softw. Engg.* 24, 4 (Aug. 2019), 2469–2508.

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

[27] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. 2023. Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 339–350.

[28] Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2021. Sidereal: Statistical adaptive generation of robust locators for web testing. *Softw. Test. Verification Reliab.* 31, 3 (2021).

[29] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 1–10.

[30] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2016. Robula+: an algorithm for generating robust XPath locators for web testing. *J. Softw. Evol. Process.* 28, 3 (2016), 177–204.

[31] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-Guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 23–26.

[32] Lizhi Liao, Heng Li, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2023. Adapting Performance Analytic Techniques in a Real-World Database-Centric System: An Industrial Experience Report. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1855–1866.

[33] Jun-Wei Lin and Sam Malek. 2022. GUI Test Transfer from Web to Android. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 1–11.

[34] Yuanzhang Lin, Guoyao Wen, and Xiang Gao. 2023. Automated Fixing of Web UI Tests via Iterative Element Matching. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1188–1199.

[35] Xinyue Liu, Zihe Song, Weike Fang, Wei Yang, and Weihang Wang. 2024. WEFix: Intelligent Automatic Generation of Explicit Waits for Efficient Web End-to-End Flaky Tests. In *Proceedings of the ACM on Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 3043–3052.

[36] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: Self-Replay Enhanced Robust Record/Replay for Web Application Testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1498–1508.

[37] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653.

[38] OpenSTF Minicap. 2023. Stream real-time screen capture data out of Android devices. Retrieved August 21, 2023 from https://github.com/openstf/minicap

[39] UI/Application Exerciser Monkey. 2023. UI/Application Exerciser Monkey. Retrieved August 21, 2023 from https://developer.android.com/studio/test/other-testing-tools/monkey

[40] Nadim Nachar. 2008. The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology* 4 (03 2008).

[41] Michel Nass, Emil Alégroth, Robert Feldt, Maurizio Leotta, and Filippo Ricca. 2023. Similarity-based Web Element Localization for Robust Test Automation. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 75 (apr 2023), 30 pages.

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

[43] Dario Olianas, Maurizio Leotta, and Filippo Ricca. 2022. SleepReplacer: a novel tool-based approach for replacing thread sleeps in selenium WebDriver test code. *Softw. Qual. J.* 30, 4 (2022), 1089–1121.

[44] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.* 40, 1, Article 3 (mar 2015), 40 pages.
[45] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn Stolee. 2019. Wait, Wait. No, Tell Me. Analyzing Selenium Configuration Effects on Test Flakiness. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. 7–13.
[46] Xiaofang Qi, Xiang Qian, and Yanhui Li. 2023. Semantic Test Repair for Web Applications. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1190–1202.
[47] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. 2019. Chapter Three - Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE. *Adv. Comput.* 113 (2019), 89–133.
[48] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An Empirical Analysis of UI-Based Flaky Tests. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1585–1597.
[49] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.
[50] Selenium. 2023. Selenium. Retrieved August 21, 2023 from https://www.selenium.dev
[51] Thread Sleep. 2023. Thread Sleep in Java. Retrieved August 21, 2023 from https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#sleep-long-int-
[52] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 503–514.
[53] Explicit Waits. 2023. Selenium Explicit Waits. Retrieved August 21, 2023 from https://www.selenium.dev/documentation/webdriver/waits/#explicit-waits
[54] Selenium WebDriver. 2023. Selenium WebDriver. Retrieved August 21, 2023 from https://www.selenium.dev/documentation/webdriver/
[55] Zhuolin Xu, Yuanzhang Lin, Qiushi Li, and Shin Hwei Tan. 2023. Guiding ChatGPT to Fix Web UI Tests via Explanation-Consistency Checking. *CoRR* abs/2312.05778 (2023). arXiv:2312.05778
[56] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-Duplicate Detection in Web App Model Inference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 186–197.

Received 12 July 2024; accepted 23 August 2024