An Empirical Study of Web Flaky Tests: Understanding and Unveiling DOM Event Interaction Challenges

Yu Pei
SnT, University of Luxembourg
Luxembourg
yu.pei@uni.lu

Jeongju Sohn*
Kyungpook National University
Korea
jeongju.sohn@knu.ac.kr

Mike Papadakis
SnT, University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

***Abstract*—Flaky tests, which exhibit non-deterministic behavior and fail without changes to the codebase, pose significant challenges to the reliability and efficiency of software testing processes. Despite extensive research on flaky tests in traditional unit and integration testing, their impact and prevalence within web user interface (UI) testing remains relatively unexplored, especially concerning Document Object Model (DOM) events. In web applications, DOM-related flakiness, resulting from unstable interactions between DOM and events, is particularly prevalent. This study conducts an empirical analysis of 123 flaky tests in 49 open-source web projects, focusing on the correlation between DOM event interactions and test flakiness. Our findings indicate that DOM events, and their associated interactions with the application, can introduce flakiness in web UI tests; these events are frequently associated with *Event-DOM* interactions (32.5%), *Event operations* (22.8%), and *Response evaluations* (16.3%). The analysis of DOM consistency and event interaction levels reveals that element-level interactions across multiple DOMs are more likely to cause flakiness than interactions confined to a single DOM or occurring at the page level. Furthermore, the primary strategies used by developers to handle these issues involve synchronizing DOM interactions (50.4%), managing conditional event completion (38.2%), and ensuring consistent DOM state transitions (11.4%). We discovered that the *Event-DOM* category has the highest fixed frequency (2.6 times), while the *DOM* category on sole takes the longest time to resolve (153.4 days). This study provides practical insights into improving web application testing practices by highlighting the importance of understanding and managing DOM event interactions.**

***Index Terms*—Flaky Tests, Web UI Tests, DOM and Event Interaction**

I. Introduction

Flaky tests pose significant challenges to software testing across various contexts [1], [2]. These tests are characterized by unpredictable behavior, leading to inconsistent results even when executed under the same conditions. For instance, tests may randomly pass or fail during repeated executions on an unchanged system. The inconsistent signals generated by flaky tests often mislead developers by introducing false test feedback and ultimately undermine developer confidence in the reliability of test results [1]–[5].

*Corresponding author

The false signal obscures the true state of the system under test: each false alert introduces additional uncertainty, making it difficult to determine whether an observed test failure is due to an actual system fault or a flaky test [6]. Consequently, developers are forced to engage in extensive debugging processes, scrutinizing both test and source code. This process is often time-consuming and labor-intensive, requiring significant effort from both developers and testers [6], [7]. As a result, this issue of flaky tests has been increasingly recognized by researchers and industry professionals, with major organizations, such as Facebook, Google, and Microsoft actively seeking solutions to address this issue [8]–[12].

Despite ongoing active investigations into the root causes and potential fixes for flaky tests, these issues are not yet comprehensively understood [13]–[16]. This problem is especially prevalent in Web User Interface (UI) testing, where the performance and reliability of flaky tests require further investigation and attention because of the complex interactions between many components under various contexts and environments [12], [17]–[20] Web UI testing focuses on testing the application under analysis by interacting and checking the application behaviour through its interface [21]. This involves assessing visual components to ensure they meet specified requirements for functionality and performance. The primary goal of UI testing is to verify that the visual elements, functionality, and usability of interfaces (e.g., Graphical User Interfaces (GUIs)) are well functioning [17], [22], [23]. Web UI tests, therefore, operate in a distinct execution environment and follow a different automation process compared to traditional unit testing.

The Document Object Model (DOM) is a crucial component of Web UI testing, which reflects the structure and content of the web page that is tested [24], [25]. As such, UI testing encompasses a wide range of DOM events, including user input handling, interactions with elements and browser APIs, and the downloading and rendering of interface resources such as images and scripts [26]. These events often occur asynchronously, leading to a non-deterministic order of user actions and other tasks. This non-deterministic behavior contributes to flakiness in web UI testing [2], despite such implications, detailed studies

979-8-3315-0814-2/25 © 2025 IEEE

ICST 2025, Naples, Italy
Technical-Research Track

on how DOM event sequences impact flaky tests, including their characteristics, causes, and repair strategies, remain notably scarce [13], [15], [16].

This study aims to address the unexplored aspects of the impact of DOM event interactions on flakiness in web UI testing by conducting an empirical analysis focused on these interactions. Specifically, the research will identify and analyze the factors contributing to test flakiness, and evaluate (i.e., summarize and identify) the main strategies developers use to manage flaky tests. Our ultimate goal is to provide practical insights that can improve the reliability of web UI testing. Particularly regarding flakiness and the impact and frequency of various DOM event interaction types, our findings could be a consideration for predicting or localizing flaky tests related to DOM issues in web tests.

In summary, the main contributions of this paper are:

- We present a comprehensive empirical analysis of DOM event interaction causes of flaky tests in web tests, identifying and exploring the specific DOM event interactions contributing to test flakiness, and providing insights into their role and impact.
- We categorized flaky tests based on their impact and interaction with DOM event interaction. The empirical evaluation indicates that interactions with flakiness are frequently associated with *Event-DOM* interactions (32.5%), *Event* operations (22.8%), and *Response* evaluations (16.3%). The analysis of DOM consistency and event interaction levels further reveals that element-level interactions across multiple DOMs are more likely to cause flakiness.
- Our examination of strategies employed by developers and built-in test frameworks to address DOM-related flakiness reveals several prevalent methods and their associated costs. The most commonly used approach involves synchronizing DOM interactions (50.4%), followed by managing conditional waits for event completion (38.2%) and ensuring consistent DOM state transitions (11.4%). *Event-DOM* interactions that events are triggered to change the DOM require the most frequent fixes (2.6 times), while *DOM-type* issues take the longest average to fix (153.4 days).
- We analyze the dataset of DOM event-related flaky tests gathered from existing studies, comprising 49 web projects and 123 test cases, evaluating the prevalence and nature of flakiness in web testing. Our investigated dataset is publicly available at *https://doi.org/10.5281/zenodo.13862302*.

II. Background

A. Web user interface(UI) testing

The web user interface is the visual aspect of a software application that dictates how users interact with the application or website and how information is presented on the screen. The user interface layer encompasses various elements such as buttons, menus, text fields, and other controls that users interact with while utilizing an application. These components are critical for facilitating user interaction and ensuring a seamless user experience [27].

Web UI tests are scripts that are designed to simulate user interactions and verify that the application behaves as expected from an end user's perspective [28]–[30]. These tests mimic various user actions, such as clicking on controls, entering text, and scrolling through pages, to ensure that the software performs its intended functions correctly. By emulating real user behavior, web UI tests help identify issues that could affect usability and functionality. However, web UI tests can be fragile, particularly when they are closely tied to the user interface. Minor changes in the UI or underlying functionality can cause these tests to fail, even if the changes do not affect the overall application performance [31], [32]. This tight coupling between the tests and UI elements requires careful management to maintain test reliability and reduce the likelihood of test failures due to insignificant modifications.

B. The DOM event interactions in flaky tests in web testing

In web application development, dependencies between Document Object Model (DOM) elements and events are critical for understanding the dynamic behavior of web pages. When a web page is loaded, the browser constructs the DOM elements for the page. The DOM links JavaScript with the web page's structure and content, allowing for a wide range of user interactions and behaviors. DOM events are actions triggered by user interactions, browser processes, or other circumstances, such as loading a page or resizing the browser window, which prompt JavaScript to respond. Common DOM events include clicking a button, hovering over an element, and scrolling. These events are essential for creating interactive and responsive web applications [33]. As shown in the Figure 1, when a user interacts with a web page, such as by clicking a specific button or typing a keyword into a search field, the corresponding event handler for the DOM element is activated. An event handler is a JavaScript function designed to execute specific code in response to an event. The browser then updates the web page accordingly, providing immediate feedback to the user. This process illustrates how the browser processes and renders user actions by executing JavaScript code. As these interactions are often asynchronous and involve a variety of elements and states, ensuring the reliability and stability of UI tests is a challenging but critical task.

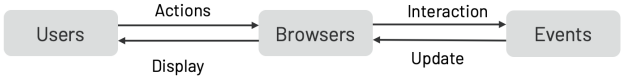

Fig. 1: Execution traces of event-driven paths in web pages

Interactions between DOM elements and events can lead to flakiness, i.e., passing and failing unexpectedly: the flakiness arises from those where DOM elements influence events and those where events influence DOM elements or the event itself contributes to flakiness. In the first scenario, some events depend on the state of specific DOM elements. If these DOM elements are not in the expected state when the event occurs, the event may fail or behave unpredictably. For example, in a form submission, unforeseen failures may occur if elements

are dynamically updated by asynchronous operations. If the event handler depends on the state of these elements, such as their values or visibility, then flaky failure may arise if the DOM has not been updated in time. On the other hand, events can affect the states of DOM elements (e.g., modifying them), often in response to user actions or other triggers. If these events are not correctly synchronized, subsequent events that depend on the updated DOM state may fail.

Previous work in this area, including static DOM event dependency analysis by Sung et al., has laid the foundation for understanding how JavaScript and DOM elements interact to create dynamic web applications [19]. However, few studies have specifically examined DOM event flakiness in web testing. This paper aims to investigate the causes and characteristics of DOM events that contribute to test flakiness in web applications.

III. Experiment Setting

A. Motivating Example

The code snippet in Figure 2 illustrates a real-world flaky test example that motivated us to explore the causes and characteristics of flakiness in web UI tests involving DOM events. Figure 2 presents an example with an explicit flaky marked commit in the *metamask-extension* project[1], one of the open-source projects we analyzed. The issue begins by clicking on the `account details` button to open the account options menu, triggering a modal dialog without waiting for it to appear fully. The flaky line (line 80) attempts to locate the QR code address element before it finishes rendering, leading to flaky failures. To resolve this issue, the fixed line (line 79, marked in green) adds a `waitFor` statement to ensure the element is fully loaded before interacting with it, preventing delayed or incomplete DOM rendering. This example illustrates how the non-determinism in DOM event interactions may cause flakiness in web UI tests.

```
75 await driver.clickElement(
76     '[data-testid="account-options-menu__account-details"]',);
77 const detailsModal = await driver.findVisibleElement('span .modal');
78
79 await driver.waitForSelector('.qr-code__address');
80 const secondAccountAddress = await driver.findElement('.qr-code__address',);
81 const secondAccountPublicAddress = await secondAccountAddress.getText();
82
83 await driver.clickElement('.account-modal__close');
84 await detailsModal.waitForElementState('hidden');
```

Fig. 2: An example of DOM event interaction flaky test case.

B. Data Collection

Previous studies have explored various flaky test datasets to understand flakiness, patching, and exploitation. Our dataset builds on two prior studies on DOM-related flaky test cases, supplemented with newly collected data. We obtained data from three main sources: (1) the UI-based flaky test dataset [2], (2) the async wait flaky test in web tests [34], and (3) relevant GitHub projects; these projects were selected based on their frequent usage and stars they received.

[1]https://github.com/oxgersa/metamask-extension/commit/a3d232ed

1) UI-based flaky test dataset: we chose the dataset because it contains a diverse set of flaky UI test samples, which allows us to easily filter out those involving DOM events interaction. Furthermore, it demonstrates the relationship between the causes of flakiness and the strategies used to address them, which aligns with our research goals.

TABLE I: Subjects. $\#_{all}$, $\#^{flaky}_{related}$ and $\#^{flaky}_{dom}$ are the number of total commits, flakiness-related commits, and flakiness-dom related commits. The three parts of our dataset, namely *D1, D2, and D3*, are from the UI-based dataset, async wait flaky dataset, and open-source projects, respectively. $\#stars$ is the number of users who have marked a repository with a star.

Project		$\#stars$	Commits $\#all$	Commits $\#^{flaky}_{related}$	Commits $\#^{flaky}_{dom}$
D1	gestalt	4.2k	5132	4	4
	angular	94.6k	30.123	93	2
	Waterfox	3.5k	908.887	760	2
	gotify/server	10.2k	595	5	2
	next.js	121k	20.833	66	1
	gatsby	55k	21.655	11	1
	influxdb	27.9k	49.241	112	1
	angular/components	24.1k	11.751	39	1
	desktop	4.4k	7804	26	1
	influxdb	27.9k	49.241	112	1
	baseweb	8.6k	3401	6	1
	plotly.js	16.6k	26.347	87	1
D2	jbrowse-components	187	9208	35	5
	elem-ing-software	1	579	5	5
	flashmap-production	0	7325	14	4
	dom-testing-extended	0	171	2	2
	preact-devtools	303	1472	12	2
	coinscan-front	6	184	4	2
	dotcom-rendering	239	29.249	24	1
	elm-select	20	584	1	1
	eui	6k	5935	18	1
	wonder-blocks	142	1942	5	1
	keptn	1.8k	8448	24	1
	shopify-theme	153	220	1	1
	azure2jira	0	20	1	1
	liquid	64	1824	2	1
	js-libp2p	2.2k	6267	2	1
	racp	6	2031	16	1
D3	wp-calypso	12.4k	67.359	74	9
	zeeve-grafana	0	42.041	65	7
	sourcegraph	9.7k	36.203	213	7
	owncloud/web	414	14.35	36	6
	aiid	156	3967	63	6
	metamask-extension	0	15.902	16	5
	evaka	32	16.574	165	5
	frontend-kaleidos	2	18.909	105	5
	devtools-frontend	451	20.526	319	4
	fictional-parakeet	1	15.677	16	3
	web-stories-wp	754	15.017	47	3
	fleet	2.2k	11.844	44	3
	wellcomecollection	37	30.126	16	2
	workbench	45	7778	57	2
	juice-shop	0	18.212	18	2
	MetaMaskTest	0	13.62	5	1
	fundamental-ngx	256	8420	12	1
	signals-frontend	8	10.384	15	1
	uwazi	210	17.965	56	1
	lisk-desktop	584	26.429	14	1
	cdap-ui	18	51.496	212	1
Total	49	-	-	3055	123

2) Async wait flaky tests in web tests: the dataset includes a reproducible collection of web projects with flaky tests related to async wait operations. The dataset is divided into time-based

and DOM-based cases, with the latter matching our focus on DOM-event flakiness.

3) Collected from GitHub: we concentrated on flaky behavior caused by DOM events issues in web applications, which could be one of the most common sources of flakiness. To collect cases of DOM event flaky cases, we retrieve commits from GitHub repositories. We began by exploring web projects that received more stars or contained a large number of commits, and whose source code was available on GitHub. Following a methodology similar to Romano et al. [2] and Pei et al. [34], we searched for relevant commits using keywords like *"e2e"*, *"flaky"*, *"flakiness"*, and further filtered with terms such as *"DOM"*, *"element"*, and *"events"* to ensure the relevance to DOM-related flakiness.

We conducted a two-round verification process to ensure the relevance of identified commits. First, we reviewed the commit messages to ensure that the issues were related to flaky tests and DOM interactions. Then, two authors reviewed the associated code changes to ensure they addressed DOM event interactions. After excluding irrelevant or redundant commits, we identified 49 web application projects with 123 commits specifically related to DOM event flakiness.

C. Categorization and Analysis

In this section, we analyzed the collected flaky tests to identify the key triggers of flaky behavior. We manually inspected relevant commits and test statements to pinpoint the code or conditions contributing to flakiness. To better understand the DOM event interactions in flaky tests, we examined all 123 commits that we collected in detail. For each commit, we analyzed the interactions leading up to (i.e., before) flakiness and those directly involved in the flakiness, along with the commit information (e.g., commit messages) and associated code changes. We then categorized these flakiness-related interactions, more specifically, the statements containing them, into six categories based on their impact on DOM modifications and event sequences to uncover trends in DOM event flakiness. These six categories are as follows:

- *ED (**E**vent-**D**OM)*: These statements involve an event directly modifying the DOM. The type is significant because it represents the connection between events and changes in the web page's structure or content. For example, in the code snippet[2], line 63 the statement `await driver.clickElement({text: "Approve", tag: "button"})`, the click event directly affects the DOM by interacting with the *Approve* button, potentially triggering DOM changes.
- *E (**E**vent)*: These statements contain only user or system events, such as clicking, triggering an event, or loading a page, without directly modifying the DOM. An example can be `cy.visit("/")`[3] in Cypress, which initiates the action of navigating to the page. While this action does not directly interact with the DOM, it triggers processes such as page loading and script execution, which can eventually result in DOM changes.
- *DE (**D**OM-**E**vent)*: Refers to statements where a DOM modification leads to the triggering of an event, like modifying an input field that triggers a change event. For instance, line 151 the statement `await page .locator(editPublicLinkRenameConfirm). click()`[4] showcases how locating an element (a DOM interaction) results in the triggering of a click event. This type may result in flakiness if the DOM is altered inconsistently and the event trigger differs due to delays in rendering.
- *D (**D**OM)*: These statements simply modify the DOM structure, for instance, adding elements, changing attributes, updating styles, or retrieving elements. In line 103 the statement[5] `const fileList = await getContentBySelector(".attach-name span:not(.attach-size)")`, a CSS selector is used to retrieve elements from the DOM, which may be further manipulated or analyzed by attached listeners or observers. If the DOM changes dynamically and inconsistently – such as elements not being fully loaded or styles not yet applied, tests may become flaky.
- *R (**R**esponse)*: These statements are used to validate or respond to the outcomes of previous DOM modifications or events typically involving assertions or verifications within a test sequence. These statements are important for ensuring that the expected changes in the DOM, triggered by events, have occurred as anticipated. The line 23 `e2e.components.LoadingIndicator. icon().should('have.length', 0);`[6] checks the DOM to ensure that a loading indicator is not present. Response statements may contribute to flakiness if they rely on DOM states that are not yet fully settled or consistent across test runs.
- *O (**O**ther)*: These statements involve other operations that are not directly related to DOM manipulation or events. These operations may contain variable declarations, function definitions, or any other relevant logic for setting up environments. For example, `var getExplorationElements = function(explorationTitle)`[7] defines a variable and declares a function rather than interacting with the DOM or events. These statements are generally less susceptible to causing flakiness.

IV. The impact of DOM-event interactions on flakiness

The primary goal of this study is to explore the causes and strategies for addressing DOM event-related flaky tests. Specifically, this research seeks to understand how different DOM event interactions contribute to flaky test occurrences,

[2]https://github.com/MetaMask/metamask-extension/commit/359f782
[3]https://github.com/influxdata/influxdb/commit/5d4248e
[4]https://github.com/owncloud/web/commit/854a0ab
[5]https://github.com/huridocs/uwazi/commit/27056d0
[6]https://github.com/Zeeve-App/zeeve-grafana/commit/86f2ed8
[7]https://github.com/BrowserWorks/Waterfox/commit/ab8e14b

one of the main factors in developing stable and reliable web test scenarios. We analyze the frequency and distribution of these DOM event interactions within our dataset, assessing the impact of DOM event interaction sequences on flakiness in web tests. Based on our observations and understanding of flaky tests, we identify four key factors that contribute to DOM-related flakiness:

- *F1 - Current-statement type*: the type of interaction that is currently investigated for it being directly involved in flaky behavior.
- *F2 - Statement-before-current type*: the type of the interaction immediately preceding the flaky behavior.
- *F3 - Same DOM or not*: Indicates whether the DOM elements involved in the flaky behavior come from the same DOM (Y) or the different DOMs (N).
- *F4 - Event interaction level*: Indicates the level of event interaction during test execution: page level (P), element level (E), or both (PE).

Specifically, both *F1* and *F2* are categorized into the six interaction types (i.e., Event (E), DOM (D), Event-DOM (ED), DOM-Event (DE), Response (R), and Other (O)) defined in Section III-C. We then further label each statement with statement detail (e.g., for `const detailsModal = await driver.findVisibleElement('span.modal');` as *find-element*), composing the final label for each statement.[8] The statement details refer to the details of the statement, usually containing specific events or DOM event pairs.

TABLE II: The DOM event flakiness categories of the statement from the motivating example. The *F1, F2, F3,* and *F4* denote the different factors, as explained above. The *flaky or not* column indicates whether the current interaction leads to flakiness.

Line *number*	F1 *Current-statement*	F2 *Statement-before-current*	F3 *Same dom*	F4 *Level*	*flaky or not*
line 77	ED (*find-element*)	ED (*click-element*)	N	E	Y
line 80	ED (*find-element*)	ED (*find-element*)	N	E	N
line 81	ED (*find-element*)	ED (*find-element*)	Y	E	N
line 83	ED (*click-element*)	ED (*find-element*)	N	E	N
line 84	ED (*wait-element*)	ED (*click-element*)	N	E	N

We categorize each statement independently within a category for each factor. Table II illustrates how the categorization can be done by categorizing statements by their interactions with the DOM and events from the motivating example in Section III-A. Here, line 77, marked as contributing to the flakiness, searches for a visible DOM element that matches the CSS selector `span.modal` and waits for it to become visible before saving it in `detailsModal`. Hence, it is categorized as ED (*find-element*) for F1 and ED (*find-element*) for F2, and the flakiness occurred at the element level and was not introduced by the same DOM element. Building on the initial categorization of interaction types, we further categorize the DOM event interactions in the statements using four distinct factors, combined with the statement details, to provide a structured framework for understanding and addressing test flakiness related to DOM event issues.

[8]The final statement label in *F1* and *F2* is composed of the statement category by six interaction types and the statement detail.

A. Results

1) ***DOM event interaction exploration*** *:* A crucial aspect of web application involves interacting with the DOM through events triggered by user actions or browser processes. The stability and reliability of web applications depend significantly on the correct functioning of these DOM and event interactions. Flaky tests in web testing, which yield inconsistent results without any code changes, often stem from the unpredictable nature of DOM event sequences. Figure 3 illustrates the interaction between the user, DOM, and the browser during event-driven web processes. These points include the handling of events and subsequent changes to the DOM, as well as the impact of DOM updates on event execution. When a user triggers an event, it initiates operations that interact with DOM elements and update the page structure. The browser then processes these updates based on its interpretation. The dashed arrows (red) in the figure indicate potential flakiness points where inconsistencies may arise if DOM changes are not fully propagated before the browser processes events.

Figure 4 exemplifies the interaction between events and the DOM across different lines of code (line 77 to line 84). Each line corresponds to an event (denoted as e_1 to e_5) that interacts with the DOM (d_1 to d_4) and is processed by the browser. The blue arrow at L80 represents the interaction that was responsible for the flakiness; in this flaky test, the interaction between DOM (d_1) and the event (e_2) was unstable (i.e., when the find event was triggered, the rendering of DOM element was not finished), causing a test to flake. The remaining statements of the test proceeded in sequence, triggering corresponding modifications in the DOM, without exhibiting any flakiness.[9]

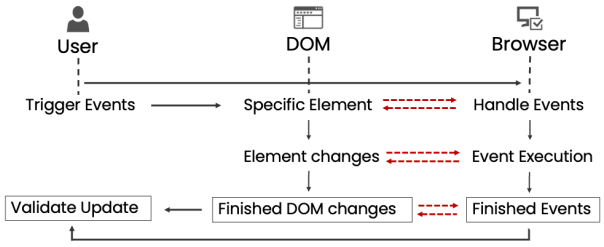

Fig. 3: The execution traces of DOM event-driven paths within a web application comprise potential flakiness points.

To enable a detailed comparison with non-flaky cases, we collected non-flaky interactions from 114 of the 123 tests in our dataset. Each case represents interactions in a flaky test that did not cause flakiness, typically the preceding or subsequent interactions of flaky-related events. The discrepancy between the number of flaky and non-flaky cases is due to the exclusion of nine tests, where only one or two DOM event statements were present, all of which contributed to flakiness. As a result,

[9]For example, when e_3 is triggered, the corresponding update to the DOM (i.e., d_2) has already been completed, and e4 then manipulates the subsequent DOM (i.e., d_3)

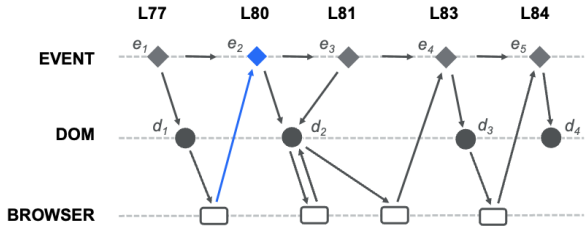

Fig. 4: Motivating example of DOM event interaction traces

once the flaky-related interactions were identified, no additional non-flaky statements remained for these tests, leaving 114 non-flaky tests. Table III compares the distribution of F1 factor values between flaky and non-flaky cases, where F1 represents interaction type involved and the details of the statements currently inspected. The goal is to identify specific interactions that are more likely to lead to flakiness than others.

From Table III, we observe distinct trends between flaky and non-flaky interactions. Overall, response-type (R) interactions—those checking or verifying the outcome of actions—are more prevalent in flaky cases (28% of the total) compared to non-flaky cases (16%); this suggests that flaky behaviors often result from preceding interactions and are then manifested during response processing. The preference for flaky behavior is also evident in the DOM type (D), where the number of flaky interactions (16) is four times higher than non-flaky interactions (4). In terms of percentage over the total (i.e., frequency), flaky cases with DOM-type interactions are more than three times as frequent (13% vs. 4%). In contrast, for the Event type (E), an opposite trend is observed, with non-flaky interactions being nearly twice as common as flaky interactions, both in absolute numbers (34 vs. 18) and frequency (30% vs. 15%), implying that event operations alone are less likely to cause flakiness.

For the Event-DOM (ED) and DOM-Event (DE) types, we observed similar numbers of flaky and non-flaky interactions. However, a closer inspection of the statement details (in the second column) reveals patterns related to the coupling between DOM elements and events. For instance, in ED-type interactions, *wait-element* is more common in non-flaky cases than in flaky ones, with the same trend for *find-element*, although the difference is smaller. In contrast, *click-element* shows similar numbers and frequencies for both flaky and non-flaky cases, while *hover-element* exhibited flaky behavior in all instances. These findings suggest that the tightness of the coupling between events and DOM elements (or their status) affects the likelihood of flaky behavior. *wait-element* is the most tightly coupled to the DOM element's status, explicitly depending on the element being in a specific state, and is associated with non-flaky behavior. *click-element* is moderately coupled, as it requires the element to be in an actionable state, with an equal number of flaky and non-flaky cases observed. In contrast, *find-element* and *hover-element* are loosely coupled with operations. *find-element* simply locates the element without requiring it to be in a particular state beyond visibility, while *hover-element* does not explicitly depend on the element's state. This relatively loose coupling of operation unexpectedly leads to flaky behavior, potentially because developers may pay less attention to these actions during testing. Thus, we posit that the degree of coupling between events and DOM elements significantly influences the probability of flaky behavior.

To further investigate what leads to flaky and non-flaky behavior, we analyzed the combinations of statement-before-current and current-statement types (i.e., F2:F1) to assess the impact of operation sequences by considering the previous statement (F2). Table IV presents the top five most frequent combinations of F1 and F2 factors, along with the top two most frequent statement detail combinations for each. Overall, we can observe the trends that we observed in Table III[10] are scrutinized in Table IV.

The first row in Table IV, *click-element:find-element*, further shows how the *find-element* operation contributes to flaky behavior. Here, we can infer that failing to wait for the element to be fully rendered or accessible after the click event leads to flakiness when attempting to access the element via the *find-element* operation. Conversely, reversing the order (i.e., *find-element:click-event* sequence in the non-flaky column) often results in stable behavior. This stability is likely because finding an element followed by clicking it involves well-defined operations, producing relatively deterministic outcomes. In this sequence, the DOm is generally in a more stable state, reducing the risk of unexpected behavior. This comparison highlights how specific sequences within *ED:ED* interactions affect the likelihood of flaky or non-flaky outcomes.

Overall, the combinations of interaction types between the previous and current statements suggest that the primary causes of flakiness may arise from inconsistent DOM modifications and poor synchronization between events and responses. The observations from the previous example in *ED:ED* - the example of *find-element:click-ele* - support this statement, showing how inconsistent DOM modifications from the previous statement affect the following event. The sequence of operations we observed in *ED:R* (e.g., *click-ele:response*) further supports this statement. The outcomes of these operations are often dependent on the state of the DOM element (e.g., an element being loaded or updated) by the previous statement, which may not always be ready at the expected time. When the execution status differs between different runs of the same test, the test's outcome varies, leading to flakiness.

Non-flaky cases often involve operations and events that are more typical (e.g., *find-element:click-ele* example), or those that naturally follow one another in a more consistent manner. For instance, the (*e-update: e-update*) reflects a relatively steady state without involving DOM elements, instead one event follows another. Similarly, (*find-element: e-wait*) involves finding an element and then explicitly waiting, which allows for sufficient time for the DOM to stabilize before the next operation. Summarized, the differences between flaky and non-flaky sequences emphasize the importance of carefully

[10]For the currently observed statement (F1), R is more frequently observed in flaky behavior, E shows the opposite, and ED is most frequent for both flaky and non-flaky behaviors but with different statement details.

managing DOM event interactions and considering the timing and state of the DOM event interaction when designing tests. Flaky tests often stem from operations dependent on unpredictable DOM or asynchronous events, while non-flaky tests involve stable and predictable sequences. Ensuring that interactions are properly arranged and synchronized with the DOM state can mitigate even-related flakiness, resulting in more reliable and consistent test outcomes.

TABLE III: Comparison of *F1* (i.e., the final statement label by the statement detail and the type of involving interactions or events) between *flaky* and *non-flaky* tests. *#number* is the number of tests belonging to each division. The value within the parentheses is the percentage of the total number of flaky tests, i.e., *frequency*. *all* is the total number of tests per type.

		F1	
		Flaky	*Non-flaky*
Type	Statement detail	#number (percent)	#number (percent)
ED (Event-DOM)	find-element	17 (0.14)	13 (0.11)
	click-element	12 (0.10)	12 (0.11)
	wait-element	5 (0.04)	13 (0.11)
	hover-element	2 (0.02)	0 (0)
all	-	36 (0.29)	38 (0.33)
E (Event)	e-click	2 (0.02)	5 (0.04)
	e-get	4 (0.03)	3 (0.03)
	e-update	8 (0.07)	13 (0.11)
	e-wait	2 (0.02)	5 (0.04)
	e-load	1 (0.01)	2 (0.02)
	e-request	1 (0.01)	6 (0.05)
all	-	18 (0.15)	34 (0.30)
DE (DOM-Event)	element-wait	2 (0.02)	0 (0)
	element-find	1 (0.01)	3 (0.03)
	element-click	15 (0.12)	17 (0.15)
all	-	18 (0.15)	20 (0.18)
D (DOM)	DOM-element	16 (0.13)	4 (0.04)
R (Response)	response-related	35 (0.28)	18 (0.16)
O (Others)	-	-	-
Total	-	123	114

*2) **The impact of DOM consistency and interaction level**:* Figure 5 illustrates the distribution of flaky tests based on two factors: F3-Same DOM or not and F4-Event interaction level types. The left chart shows that 76.4% of flaky tests occurred when elements were spread across different DOMs, while 23.6% of cases involved flakiness within a consistent DOM. The right chart indicates that 80.5% of flaky tests are related to element-level interactions, suggesting that flakiness primarily arises from issues when interacting with specific DOM elements. Moreover, 17.1% of flaky tests involve both page- and element-level interactions, while only 2.4% are associated with page-level interactions alone. These findings suggest that flaky behavior is more frequently linked to operations on DOM elements than to page-level events and may result from interactions involving multiple DOMs rather than a single DOM. Overall, the charts highlight that DOM inconsistencies and element-level interactions are major contributors to flakiness.

Summary We identified the prevalence and frequency of flaky behavior related to DOM events in web tests. The most common causes of DOM events flakiness in web UI testing are associated with element interactions, asynchronous operations, and event loads. Flaky test interactions, such as *Event-DOM*, *DOM*, and *Response*, highlight flakiness issues

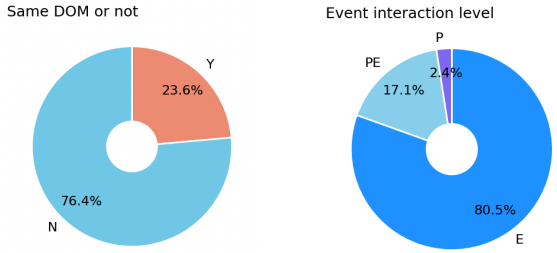

Fig. 5: The distribution of the same DOM element and event interaction levels associated with flakiness. *Y* and *N* are *yes* and *no* to whether the elements belong to the same DOM; *P*, *E*, and *PE* refer page level, element level, both.

in scenarios involving DOM element availability and event handling. The analysis of the consistency of involving DOMs and the interaction level reveals that element-level interactions across multiple DOMs are more likely to cause flakiness than the cases within a single DOM and at the page-level.

V. The Common Strategies and Time Costs to Address DOM Event Flakiness

The study in Section IV shows that we investigate various existence and frequencies of DOM event flakiness in web tests. In this section, we focus on understanding developers' strategies to fix DOM-related flaky tests and assessing the challenges of these issues. By categorizing and comparing common solution patterns for DOM event-related issues, we also examine how existing web test frameworks provide integrated solutions to address DOM event interaction flakiness.

A. Results

*1) **Fixing strategies**:* This investigation aimed to identify and categorize the common approaches developers employ to mitigate and resolve flaky tests. Based on our dataset, we identified three primary strategies and the results are as follows:

- *DOM interaction synchronization mechanism*: This strategy involves adding synchronization logic to ensure that DOM elements are fully loaded and ready for interaction before proceeding with certain actions. By employing methods like *waitFor* or *waitUntil*, developers can explicitly wait for specific DOM elements to be present, thereby reducing the likelihood of flakiness caused by elements not being available during interaction.
- *Conditional event completion waits*: This approach involves introducing specific or dynamic delays in test execution to accommodate timing issues. By using functions such as sleep, delay, or timeout, the test allows sufficient time for page loading, ongoing operations, or the completion of interactions before proceeding to the next step, thereby minimizing the chances of encountering flakiness due to timing inconsistencies.
- *Consistent DOM state transitions*: This strategy focus on ensuring that responses from asynchronous callbacks are properly received and processed before progressing to subsequent testing steps. By waiting for complete and accurate data, this approach helps prevent flakiness arising

TABLE IV: The top five flaky and non-flaky DOM event interaction combinations with detailed examples are listed below. These refer to *Statement-before-current* and *Current-statement* interactions (i.e., *F2:F1*). The top two frequent combinations for each category are provided. *#number* and *percent* indicate the test count and the test percentage for each type.

Top No.	Flaky Type		Flaky statement detail (Top-2)		Non-Flaky Type		Non-Flaky statement detail (Top-2)	
	F2:F1	*#number (percent)*	*F2: F1*	*#number (percent)*	*F2:F1*	*#number (percent)*	*F2: F1*	*#number (percent)*
1	ED:**ED**	19 (0.15)	*click-ele: find-ele* *find-ele: find-ele*	9 (0.07) 4 (0.03)	ED:**ED**	23 (0.2)	*find-ele: click-ele* *click-ele: wait-ele* *find-ele: wait-ele*	5 (0.04) 4 (0.04) 4 (0.04)
2	ED:**R**	12 (0.10)	*click-ele: response* *find-ele: response*	5 (0.04) 4 (0.03)	E:**E**	14 (0.12)	*e-update: e-update* *e-load: e-wait*	4 (0.04) 2 (0.02)
3	E:**ED**	11 (0.09)	*e-load: click-ele* *e-load: wait-ele*	6 (0.05) 6 (0.05)	ED:**E**	9 (0.07)	*find-ele: e-wait* *click-ele: e-update*	3 (0.03) 3 (0.03)
4	E:**R**	9 (0.07)	*e-update: response* *e-click: response*	6 (0.05) 2 (0.02)	DE:**DE**	8 (0.07)	*element-click: element-wait* *element-wait: element-click*	6 (0.05) 2 (0.02)
5	DE:**DE**	8 (0.07)	*element-click: element-click* *element-click: element-find*	6 (0.05) 1 (0.01)	E:**ED**	7 (0.06)	*e-load: click-ele* *e-load: wait-ele* *e-get: find-ele*	2 (0.02) 2 (0.02) 2 (0.02)

from the use of incomplete or incorrect data in rendering DOM elements.

The distribution of common approaches used to address testing concerns involving DOM events is presented in Table V. The most prevalent strategy is ensuring elements are fully loaded before interaction (50.4%), followed by introducing dynamic delays (38.2%), and ensuring stable DOM state transitions (11.4%), with corresponding occurrence rates provided for each. By understanding and using these approaches, developers can improve the reliability and robustness of web UI tests, leading to more stable and reliable web applications.

In addition to examining the strategies commonly employed by developers to address DOM-related flakiness, we further examined the methods incorporated into popular web testing frameworks to address these issues. Table VI presents the methods offered by common web testing frameworks to address DOM-related flaky tests. Each framework is described, along with methods developers can use to manage DOM-related flakiness. The commonly used test frameworks like Cypress (26.8%), Jest (26%), and Puppeteer (20.3%) offer robust synchronization mechanisms to deal with flakiness issues between DOM modifications and event handling. These frameworks provide methods such as *waitFor*, *waitForSelector*, and *waitForFunction* to ensure that the DOM is fully loaded and the events have been properly processed before proceeding with test execution. The result shows that many widely used frameworks have incorporated tools to handle DOM events interaction inconsistently, helping developers reduce the flakiness of their tests. Such features are particularly valuable in complex, dynamic web applications where the DOM structure is frequently updated and event-driven issues are common. By leveraging these tools, developers could write more stable and reliable UI tests that are less vulnerable to flakiness caused by DOM event synchronization issues.

*2) **Fixing cost**:* Table VII illustrates the relationship between various types of DOM event interaction flakiness (*ED, D, DE, E, R*) and the corresponding strategies employed to fix these issues in web testing. It provides key metrics such as the frequency of fixes, the time taken to resolve issues and the relationship with the strategies used to fix them. From the table, we see that the average fixed frequency of flaky events per type ranges from 2.1 to 2.6, with type ED (Event-DOM) having the highest average of 2.6. We speculate that, as events and DOM elements interact closely during web application execution, unstable event-to-DOM interactions could cause frequent but difficult-to-diagnose flaky behaviors; such interactions involve both event triggering and subsequent DOM modifications, further complicating root cause detection and prolonging resolution times. Flaky tests related to type D (DOM) are typically the longest to resolve, averaging 153.4 days, while DE (*DOM-Event*) interactions are resolved more quickly, averaging 58.7 days. Type DOM has a longer average days to fix because DOM manipulation alone can lead to complex states that are difficult to synchronize, particularly when dynamic content and asynchronous updates are involved. DOM state management and transitions may not be easily captured in test scripts, resulting in longer debugging and fixing times. The duration of type DE is relatively short (58.7 days), likely because DE involves more direct interactions between the DOM state and subsequent event firing, which are easier to pinpoint and fix than the asynchronous event chains found in type ED.

DOM-Sync (short for DOM interaction synchronization mechanism) is prevalent across all types, especially for ED (14 fixes) and D (13 fixes), because synchronization of DOM interactions is fundamental to resolving flaky tests. Events often interact with the DOM in a way that makes it necessary to wait for DOM updates before moving on. In ED cases, DOM-sync is necessary as both event and DOM elements highly impact each other, leading to frequent synchronization issues. Conditional event completion waits apply when tests rely on specific events to complete before moving forward. For pure DOM flakiness, the event is less relevant, hence the absence of this fix. Instead, DOM synchronization becomes the dominant solution. However, for ED and R (Event-Response), events play a major role, leading to moderate usage of event-wait strategies. DOM-Transitions (Consistent DOM state transitions) are most frequently applied to the ED and R types (15 fixes each) because both types involve interactions between events or responses and the DOM, and unstable DOM states often result from asynchronous transitions. This issue requires developers

TABLE V: Common fixed strategies for the Dom-events flaky tests

Fixed strategy category	Description	Examples	#Occurrences
DOM interaction synchronization mechanism	Ensure elements are fully loaded and visible before interaction.	`await driver.waitForSelector('.qr-code__address');`[11] wait to ensure 'qr-code__address' element available.	62 (50.4%)
Conditional event completion waits	Introduce timeout or dynamic delays in tests.	`cy.wait(500);`[12] add 500ms delay.	47 (38.2%)
Consistent DOM state transitions	Response to data procurement and requests to ensure that the test is not affected by data or network factors.	`await pWaitFor(()=> connection.streams.length === 0);`[13] add await to ensure API response before verifying DOM results.	14 (11.4%)

TABLE VI: Methods to deal with flaky tests in common web testing frameworks

Web test framework	Description	Methods to address DOM-related flaky tests	#Numbers
Chai	Chai is a BDD/TDD assertion library for the browser that can be paired with any JavaScript testing framework.	`assert.equal(); expect().tobe(); timeout();`	1 (0.8%)
BrowserTestUtils	BrowserTestUtils provide useful test utilities for working with the browser in browser mocha tests.	`TestUtils.waitForCondition(); TestUtils.waitFor.visibilityOf();`	2 (1.6%)
Cypress	Cypress is a modern JavaScript-based E2E testing framework for web apps.	`cy.wait(time); cy.waitFor('element');cy.method().should();`	33 (26.8%)
Jasmine	Jasmine is a Behavior Driven Development testing framework for JavaScript.	`waitsFor();`	1 (0.8%)
Jest	Jest is a testing framework that can adapt to any JavaScript library or framework.	`page.waitFor(); page.waitForSelector(); page.waitForTimeout();`	32 (26%)
Playwright	Playwright is a framework for web testing and automation.	`page.waitForLoadState(); page.waitForNavigation(); page.waitForSelector(); page.waitForTimeout();`	23 (18.7%)
Protractor	Protractor is an end-to-end test framework for Angular and AngularJS applications.	`browser.wait(); browser.sleep();`	5 (4.1%)
Puppeteer	Puppeteer is used for tasks like creating screenshots, crawling pages, and testing web applications.	`waitForExists(); waitForSelector(); waitForFunction(); timeout();`	25 (20.3%)
Selenium WebDriver	Selenium Webdriver are often used for testing web applications and tasks that require interaction with the browser.	`timeouts(); wait.until();`	1 (0.8%)

to ensure that the DOM reaches a stable state before subsequent actions are taken. DOM transitions are important to preventing inconsistencies and ensuring the DOM correctly reflects the state expected by the test.

TABLE VII: The correlation between DOM event interaction types and developer's fix strategies. The *Avg.* refers to the average within each type. The *Frequency* and *Days* denote the fixed frequency for each type and the number of days cost to fix. *DOM-sync, Event-waits, DOM-transitions* short for *DOM interaction synchronization mechanism*, *Conditional event completion waits*, and *Consistent DOM state transitions*.

Type	Fix Frequency (min-max) Avg.	Fix Days Avg.	Fix strategies (#number per type) DOM-sync	DOM-transitions	Event-waits
ED (Event-DOM)	(1-8) 2.6	123.2	17	2	17
E (Event)	(1-3) 2.1	117.5	9	4	5
DE (DOM-Event)	(1-8) 2.2	58.7	10	1	7
D (DOM)	(1-6) 2.0	153.4	13	0	3
R (Response)	(1-7) 2.1	71.2	13	7	15

Summary The most prevalent strategies for mitigating DOM-related flaky tests include managing DOM interaction synchronization (50.4%), followed by handling conditional waits for event completion (38.2%), and ensuring consistent DOM state transitions (11.4%). Many web testing frameworks already incorporate built-in methods to address DOM event interaction issues. Developers should actively leverage these tools to minimize flakiness and enhance the reliability of their tests. Furthermore, the *Event-DOM* type shows the highest fix frequency (2.6 times) while the DOM-type issues take the longest time to resolve (153.4 days). The findings indicate that the DOM interaction synchronization mechanism is the most commonly utilized fix across all types. Strategies such as conditional event completion waits and consistent DOM state transitions are also employed to address the DOM event interaction flakiness in certain scenarios.

VI. Discussion

The prevalence of flakiness due to DOM event interactions. Flakiness in web tests differs significantly from traditional unit tests due to the complexities of modern web applications. While unit tests focus on isolated code in controlled environments, web UI tests involve dynamic interactions between the DOM, user events, and asynchronous operations, making them more susceptible to variability. Our study highlights that flakiness frequently often arises from unstable DOM-event interactions, especially when involving multiple DOM elements or requiring cross-element synchronization. The practical impact of understanding DOM event flakiness. By prioritizing high-frequency flaky categories in web tests, such as *Event-DOM* interactions, developers can effectively address the issues with the highest impact in advance. Furthermore, managing conditional event completions and ensuring the DOM event dependencies is important to reduce the risk of test flakiness. By incorporating these insights into testing frameworks and workflows, developers could mitigate the expense of debugging and enhance the overall test stability.

VII. Threats to Validity

Construct validity threats The study is primarily focused on the flakiness commits from publicly open-source web projects on GitHub. The observations related to DOM-event pairs mainly

focus on web UI testing, which might limit their applicability to other types of testing. Nonetheless, we hold the belief that our analysis outcomes are equally applicable to scenarios involving user interaction, such as mobile testing. Furthermore, to mitigate this issue, we have selected projects that have been previously investigated in related studies to encompass a diverse range of DOM-related flakiness cases, making our data a reasonable representation of real-world scenarios.

Internal validity threats Internal validity relates to the extent to which observed relationships between variables can be attributed to the causal effects under investigation. In this study, potential threats to internal validity include subjective decisions made during data selection and analysis, which could introduce bias or errors in identifying causes of flaky tests. To ensure accuracy, we reviewed all relevant commits from prior studies on DOM flakiness. We eliminated non-relevant commits and conducted a manual examination of the remaining data to verify its relevance to flaky DOM event tests. At least two authors reviewed the dataset to reach a consensus.

External validity threats The external validity of our study's findings may be limited beyond the specific types of DOM events and web testing scenarios analyzed. Different types of DOM interactions, such as complex event sequences or interactions with third-party components, may exhibit patterns of flakiness that were not fully captured in our analysis.

VIII. Related work

The impact of flaky tests has been studied in various research domains. Research on flaky tests has gained significant attention in recent years, particularly in the context of their causes, identifying strategies, and repair strategies. Luo et al. conducted a foundational study that classified flaky tests into various types based on their causes and detection methods. They identified that asynchronous operations and timing issues were significant contributors to flakiness in UI tests [1]. Subsequent studies, such as those by Huo and Clause, highlighted concurrency issues and resource management as additional sources of flaky tests. They found that race conditions and improper handling of shared resources often lead to unpredictable test outcomes, particularly in complex UI environments where multiple threads or processes interact simultaneously [35]. Lam et al. conducted a study on the overall lifecycle of flaky tests in six large-scale Microsoft projects and found that the Async Wait issues are the most common flaky tests with the projects. Therefore, the authors confirm that the categorizations of flaky tests proposed by previous studies applied to open-source projects also apply to the proprietary Microsoft projects studied [3].

Romano et al. identified the main causes and fixed strategies for flaky tests in UI Testing projects, identifying asynchronous wait updates and event sequencing issues as prevalent causes. They also provide a foundational understanding of how DOM events contribute to test flakiness [2]. Hashemi and colleagues investigate the prevalence and causes of flaky tests within JavaScript projects and identify common sources of flakiness, such as asynchronous operations and timing issues [36]. Pei et al. further emphasized the importance of synchronization techniques and event-handling strategies in minimizing test flakiness in web projects, which aligned closely with our study's objectives [34]. Dong et al. present a method for identifying flaky tests by systematically varying event order during test execution, revealing inconsistencies that indicate non-deterministic behavior in Android applications [37]. Wang et al. introduce a strategy to identify and mitigate flaky tests by focusing on tests that produce different outcomes when executed multiple times under the same conditions [38]. Addressing flaky tests involves both preventive and fixed strategies. There have also been several studies that have been targeted to detect certain types of flaky tests. Bell et al. introduced a test repair tool that fixes flaky tests to improve their reliability. The tool employs dynamic analysis to detect flaky behavior and applies corrective actions, such as adjusting wait times or isolating external dependencies, to mitigate flakiness [39]. Lam et al. developed a preliminary tool, RootFinder, which analyzes the logs of passing and failing executions of the same test to suggest method calls that could be responsible for the flakiness [40]. Later, Shi et al. proposed iFixFlakies, an automated framework of fixing order-dependent tests. The findings possess the potential to significantly improve the quality and stability of web testing within open-source projects. This study builds on these foundations by specifically targeting DOM event interaction flakiness in web UI testing, an underexplored area in the existing literature.

IX. Conclusion

This study provides an in-depth empirical analysis of test flakiness arising from DOM event interactions in web UI testing. Our findings reveal that flaky tests frequently arise from the unpredictable interactions between DOM and events. The findings demonstrate that unforeseen DOM interactions, asynchronous operations, and event execution handling processes are the primary contributors to flakiness. By examining various interaction types, including current and preceding interactions, the analysis reveals a strong correlation between these factors and the occurrence of flaky tests. Moreover, major strategies for mitigating DOM-related flakiness involve synchronizing DOM interactions, implementing conditional event completion waits, and ensuring consistent DOM state transitions. These findings emphasize the complexity of managing flakiness in web tests and the necessity for robust testing practices, especially given the dynamic nature of DOM interactions. Our study offers practical insights to improve web application testing reliability and encourages future research into advanced methods for detecting and localizing this flakiness to further enhance the testing stability and dependability.

X. Acknowledgments

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C20/IS/14761415/TestFlakes and partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A5A1021944).

References

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014, nov 2014, pp. 643–653.

[2] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of ui-based flaky tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1585–1597.

[3] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020.

[4] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–74, 2021.

[5] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 244–255.

[6] G. Haben, S. Habchi, J. Micco, M. Harman, M. Papadakis, M. Cordy, and Y. L. Traon, "The importance of accounting for execution failures when predicting test flakiness," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, Industry Showcase, ASE 2024, Sacramento, California, United States*. IEEE / ACM, 2024.

[7] M. Eck, M. Castelluccio, F. Palomba, and A. Bacchelli, "Understanding Flaky Tests: The Developer's Perspective," *arXiv*, pp. 830–840, 2019.

[8] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 91–100.

[9] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1074–1083.

[10] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 233–242.

[11] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," *Proceedings - International Conference on Software Engineering*, pp. 1471–1482, 2020.

[12] J. Lampel, S. Just, S. Apel, and A. Zeller, "When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1381–1392.

[13] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "IDFlakies: A framework for detecting and partially classifying flaky tests," *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pp. 312–322, 2019.

[14] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. Le Traon, "What made this test flake? pinpointing classes responsible for test flakiness," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 352–363.

[15] F. P. V. Pontillo and F. Ferrucci, "Static test flakiness prediction: How far can we go?" in *Empirical Software Engineering*, 2022.

[16] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models," *Empirical Software Engineering*, vol. 28, no. 3, p. 72, 2023.

[17] A. Marchetto, F. Ricca, and P. Tonella, "An empirical validation of a web fault taxonomy and its usage for web testing," *Journal of Web Engineering*, pp. 316–345, 2009.

[18] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, "Assessing transition-based test selection algorithms at google," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 101–110. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00019

[19] C. Sung, M. Kusano, N. Sinha, and C. Wang, "Static dom event dependency analysis for testing web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 447–459.

[20] D. Olianas, M. Leotta, F. Ricca, and L. Villa, "Reducing flakiness in end-to-end test suites: An experience report," in *International Conference on the Quality of Information and Communications Technology*. Springer, 2021, pp. 3–17.

[21] A. M. Memon, *A comprehensive framework for testing graphical user interfaces*. University of Pittsburgh, 2001.

[22] J. Strecker and A. M. Memon, "Testing graphical user interfaces," in *Encyclopedia of Information Science and Technology, Second Edition*. IGI Global, 2009, pp. 3739–3744.

[23] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian, "Terminator: Better automated ui test case prioritization," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 883–894.

[24] A. Mesbah, A. Van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 35–53, 2011.

[25] K. Barbosa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test flakiness across programming languages," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2039–2052, 2022.

[26] A. Heiskanen, "Robotic process automation in automated gui testing of web applications," 2021.

[27] H. M. El-Bakry, A. M. Riad, M. Abu-Elsoud, S. Mohamed, A. E. Hassan, M. S. Kandel, and N. Mastorakis, "Adaptive user interface for web applications," in *Recent Advances in Business Administration: Proceedings of the 4th WSEAS International Conference on Business Administration (ICBA'10)*, 2010, pp. 20–22.

[28] A. Mesbah and A. Van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 210–220.

[29] V. Garousi, A. Mesbah, A. Betin-Can, and S. Mirshokraie, "A systematic mapping study of web application testing," *Information and Software Technology*, vol. 55, no. 8, pp. 1374–1396, 2013.

[30] S. Doğan, A. Betin-Can, and V. Garousi, "Web application testing: A systematic literature review," *Journal of Systems and Software*, vol. 91, pp. 174–201, 2014.

[31] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, "Leveraging existing tests in automated test generation for web applications," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 67–78.

[32] P. Aho and T. Vos, "Challenges in automated testing through graphical user interface," in *2018 ieee international conference on software testing, verification and validation workshops (icstw)*. IEEE, 2018, pp. 118–121.

[33] B. A. Myers and M. B. Rosson, "Survey on user interface programming," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1992, pp. 195–202.

[34] Y. Pei, J. Sohn, S. Habchi, and M. Papadakis, "Non-flaky and nearly-optimal time-based treatment of asynchronous wait web tests," *ACM Transactions on Software Engineering and Methodology*, 2024.

[35] C. Huo and J. Clause, "Improving oracle quality by detecting brittle assertions and unused inputs in tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 621–631.

[36] N. Hashemi, A. Tahir, and S. Rasheed, "An empirical study of flaky tests in javascript," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 24–34.

[37] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in android via event order exploration," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 367–378.

[38] A. Wei, P. Yi, Z. Li, T. Xie, D. Marinov, and W. Lam, "Preempting flaky tests via non-idempotent-outcome tests," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1730–1742.

[39] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 433–444.

[40] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root Causing Flaky Tests in a Large-Scale Industrial Setting," in *Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. Beijing, China: ACM Press, 2019, pp. 101–111.