International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 4, Issue 3, 172-174, 2023
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V4I3P120

Original Article

Reducing Selenium Test Flakiness in Edge Environments with Element Synchronization Strategies in Intellij Ide

Udayan Verma
Denver, USA.

Abstract *- Selenium is a leading tool for automated UI testing, but test flakiness, intermittent pass/fail outcomes without code changes, remains a critical challenge, especially in Microsoft Edge environments. Flakiness often arises from timing mismatches, dynamic DOM updates, and asynchronous UI rendering. At Charter Communications, these issues affected enterprise-scale automation, reducing CI/CD reliability. This report investigates advanced element synchronisation strategies implemented within IntelliJ IDE, including explicit waits, conditional locators, and custom polling algorithms. Leveraging IntelliJ's plugin and automation APIs, the framework stabilises element interactions, automates build and assembly workflows, and significantly reduces intermittent failures, improving regression reliability and cross-browser automation consistency.*

Keywords *- Selenium, Test Flakiness, Element Synchronisation, Intellij Ide.*

1. Introduction

Selenium WebDriver has become the industry standard for UI automation due to its flexibility, extensive language bindings, and support for multiple browsers. However, Selenium-based tests often exhibit flakiness, defined as tests producing inconsistent results across runs without code changes. Flaky tests are particularly problematic in enterprise-scale projects where applications are large, complex, and subject to frequent updates. Inconsistent test results lead to wasted debugging effort, unreliable defect reporting, and delays in continuous integration and deployment (CI/CD) processes. Most browsers usually have no problems with W3C WebDriver standards, but that is not the case with Microsoft Edge. Although it is Chromium-based, Edge has different timing, rendering, and asynchronous characteristics of behaviour. Edge environment: Flaky tests are common due to differences between the dynamic behaviour of the DOM and the execution of a script of automation. As an example, slow page rendering, postponed messaging of JavaScript and dynamically loaded content can result in automation scripts initiating interactions before the target elements are ready. Such are exacerbated in applications where the client-side is richly interactive, e.g. a single-page application (SPA) or an Angular frontend. Automated regression testing is significant at Charter Communications, and it is used to maintain quality in several web applications. Such applications are constantly changing, and new features are regularly added together with changes in the UI.

2. Problem Statement

The overall problem discussed in this report is Selenium test flakiness in Microsoft Edge environments, whereby a test occasionally fails with no modifications to the application code. Flakiness is often timing-based and occurs when the tests are trying to interact with elements before they are loaded or in transient DOM changes, so they occasionally fail. Element relocation, mutation, or temporary disappearance of elements are the results of dynamic DOM shifts in the present web applications. Client-side frameworks can further desynchronize execution of tests because of asynchronous UI updates [1]. Also, traditional Selenium waits, e.g. implicit waits, can be inadequate when dealing with complicated behaviours of the UI, or browser idiosyncrasies. These make the execution of the tests unstable, debugging inefficient, engineering effort wasted, regression reporting unreliable, and confidence in automated testing on an enterprise scale across multiple browsers diminished.

Fig 1: Element Synchronisation

3. Methodology

A methodology of complex synchronisation strategies was applied to solve Selenium test flakiness issues in the Microsoft Edge environment. The first element, project-specific expected UI conditions, entailed the definition of conditions that were specific to each project so that only interaction between elements was made when an application was in a stable state. Other conditions like element visibility, clickability and attribute validation were added, and they are usually checked using external monitoring utilities to maintain accuracy in dynamic conditions. The second strategy was encapsulation of UI reactivity and concentrated on mutable or postponed elements [2]. Synchronisation logic had been divorced through specialised wrappers, and test scripts were able to communicate such that repetitive waits were not required, enhancing maintainability and reducing delays in test scripts.

The third one involved decoupled monitor classes, which established settling of the UI elements where polling, event discovery, and conditional checks were made instead of waiting based on time, so that synchronisation attracts greater predictability. The fourth one was a multi-team synchronisation utility on shared IntelliJ IDE tools, which provides support for cross-team consistency, which consists of reusable resources, debugging and test execution. Finally, IDE integration passed IntelliJ IntelliJ IntelliJ specialises in the integration of the three systems of IntelliJ, namely build, assembly and deployment, into synchronisation tools, inserting syncing logic into test procedures, neglecting manual work, consistency and optimisation of synchronisation plans, accelerating automated regression testing of Edge setups.

4. Implementation

Focus was made on the implementation stage to make sure that better coordination strategies were integrated in the Selenium framework of tests with emphasis on breadth, sustainability, and robustness. The former one was unconditional locators given as explicit waits. It additionally utilised the WebDriverWait Selenium offers, as well as application-specific elements, such as visibility of elements, possibility to click on the elements, and availability of qualities [3]. This allowed dynamically changing tests based on tests of varying load without arbitrary delays, thereby reducing timing flakiness. That was succeeded by the implementation of personal polling algorithms on those elements that were updated regularly. These were lightweight mechanisms that kept on checking element availability at the target time repeatedly until the conditions were met or timeouts.

Here also, resiliency on the transient nature of the changes in the UI was made by polling on the UI, typically on Microsoft Edge, where asynchronous rendering was prone to introducing some failures occasionally. The third step referred to IntelliJ integration. Synchronisation tools came in the form of a plug that automated the application construction and assembly, applied the qualities of code, and enabled testers to treat the utilities within the IDE itself. This was easier to develop, ensured consistency in carrying out the pattern of synchronisation and enhanced the quality of the code. Better monitoring instruments were then applied at the team level. Such classes always guaranteed the resting of the elements and offered diagnostic impressions in the cases of failures, reduced unnecessary work, and maximised the efficiency of triage [4]. Finally, validation and tuning were performed by looking at the test executions to detect flakiness patterns. Parameters in synchronisation that include wait times, polling periods, conditioning, among others, were narrowed down via an iterative process and resulted in maximisation of stability without necessarily extending test time in a bid to obtain consistent and repeatable automation.

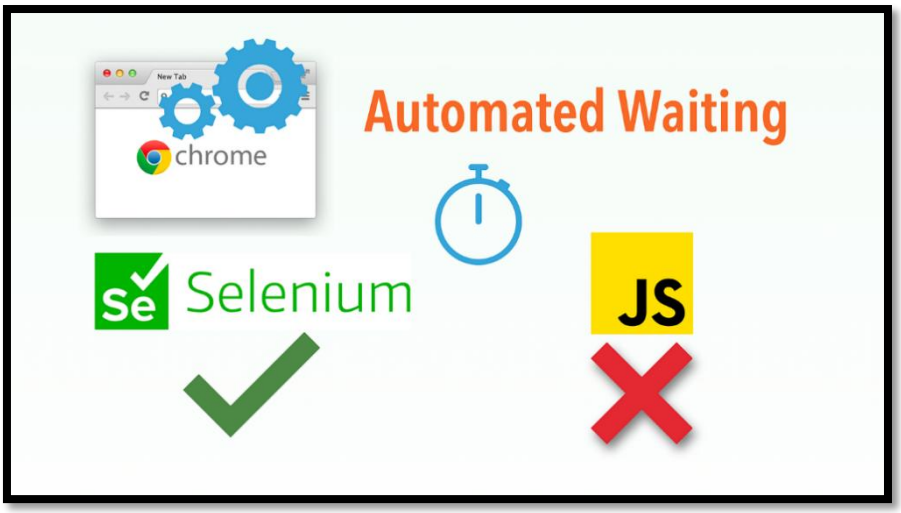

Fig 2: Selenium Flakiness

5. Results and Impact

In case complex elements synchronisation arrangements are confessed to the IntelliJ IDE, the extent of influence on the Selenium test Ethical platforms is apparent. Flaky failures were minimised, intermittent failures were minimised, and nightly regression runs were also more reliable. The result of this was simplified CI/CD productivity as it minimised delays as well as enhanced deployment predictability. Deterministic synchronisation was also useful in enhancing debugging and triage, which gave engineers time to address actual defects, rather than false positives due to errors in timing, or an out-of-flight user interface [5]. There were also common IDE tools and monitoring systems helping to create uniformity across teams, standardise the habits of synchronisation across different projects and teams, eliminate unnecessary work, and enhance coordination. Although originally developed to work with Edge, the modular and flexible design of the framework can also work with other browsers, such as Chrome and Firefox, adding to the overall automation strength, scalability, and enterprise levels of reliability with testing in a wide variety of settings.

6. Discussion

Although the solutions to flakiness of Selenium tests in Microsoft Edge are being implemented to significantly mitigate the issue, there are still several limitations. Although such applications may be monolinguistic, even in the context of the highly asynchronous forms, they never fail to require manual adjustments to ensure robust element interaction. Application of the IntelliJ IDE style could present a constraint since it would be hard to work on them on other IDEs or established tooling organisations. There are equally contrasting edge rendering problems which are not in sync with the Model but are cross-browser without further customisation [6]. Despite these weaknesses, the technique would support the effectiveness of the utilisation of automation and combinations of the IDE tools to solve the problems of testing in real life.

7. Conclusion

Microsoft Edge flaky tests also present a major problem to enterprise UI automation because they result in a complete waste of resources, lack of confidence and delayed veteran release. Teams can stabilise test execution and reduce flakiness by using improved element synchronisation techniques consisting of explicit waits, conditional locators, polling algorithm, monitor classes, and common utility. IntelliJ IDE's plugin architecture provides an effective platform for automating the integration, build, and assembly of these synchronisation strategies, enabling consistent, reusable, and maintainable practices. The framework implemented at Charter Communications demonstrates substantial reductions in intermittent failures, improved regression run reliability, and enhanced debugging efficiency.

References

[1] Alshammari, A., Morris, C., Hilton, M. and Bell, J., 2021, May. Flakeflagger: Predicting flakiness without rerunning tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (pp. 1572-1584). IEEE.

[2] 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," *arXiv preprint* arXiv:2112.04919, 2021.

[3] J. Micco, "Flaky tests at Google and how we mitigate them," *Google Testing Blog*, May 27, 2016. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[4] Sawant, K., Tiwari, R., Vyas, S., Sharma, P., Anand, A. and Soni, S., 2021, February. Implementation of selenium automation & report generation using selenium web driver & ATF. In *2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)* (pp. 1-6). IEEE.

[5] Varghese, N. and Sinha, R., 2020, October. Can commercial testing automation tools work for iot? A case study of selenium and node-red. In *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society* (pp. 4519-4524). IEEE.

[6] A. Tahir, S. Rasheed, J. Dietrich, and H. Rasool, "Test flakiness' causes, detection, impact and responses: A multivocal review," *Journal of Systems and Software*, vol. 206, p. 111837, Feb. 2023.