Research on Test Flakiness: from Unit to System Testing

Kiet Ngo
Katalon Inc.
University of Science
Vietnam National University
Ho Chi Minh City, Vietnam
kiet.ngo@katalon.com

Vu Nguyen
Katalon Inc.
University of Science
Vietnam National University
Ho Chi Minh City, Vietnam
nvu@fit.hcmus.edu.vn

Tien Nguyen
Katalon Inc.
University of Texas at Dallas
Texas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

Test flakiness has been a common problem in the software automation testing, affecting the effectiveness and productivity of test automation. Many studies have been published to tackle test flakiness in the software research community, and existing test automation tools provide capabilities to address issues related to test flakiness. This paper describes our review of recent approaches in the research community and popular industrial tools with capabilities to tackle test flakiness. Our review indicates that while many studies focus on test flakiness in unit testing, few address the problem in the end-to-end system testing. On the contrary, industrial test automation tools tend to be more concerned with test flakiness at the system level.

CCS CONCEPTS

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

KEYWORDS

software testing, flaky test, unit testing, system testing, end-to-end testing

ACM Reference Format:
Kiet Ngo, Vu Nguyen, and Tien Nguyen. 2022. Research on Test Flakiness: from Unit to System Testing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3551349.3563242

1 INTRODUCTION

Automation testing has been a valuable tool for assisting testers in efficiently verifying software quality. Test automation saves time on repeating test cases across several software development cycles. However, due to the dynamic nature of modern software and external influences, test automation might be non-deterministic, failing to accurately reflect the quality of the product under test. This problem is referred to as test flakiness. As of now, there have been several studies regarding the overall progress of flaky test research in academics. Our paper aims to provide a brief overview of the current research progress for the flaky test problem in academics and the software industry. First, we define a non-formal concept of the flaky test problem and its impact on software engineering, along with categorizing flaky tests in the industry. Second, we discuss the current research progress regarding the tools to handle the problem of flaky tests based on several recent surveys and empirical analyses. Third, we look at how the industry is developing the current commercialized automation testing tools to handle flaky tests. Finally, we will point out future research directions in test flakiness so that researchers interested in this problem can start from there.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3563242

2 THE PROBLEM OF FLAKY TEST

Simply put, a flaky test does not produce the same results each time it is run. This means it may sometimes pass and fail at others, with no changes to the product under test or the test procedures. When we test a program, we want the test results to be consistent so that we can identify the program's flaws. If a bug in the program causes a test to fail, the test should constantly fail as long as the bug is not corrected; conversely, if the program is genuinely bug-free, there should be no reason for a test to fail at random. Test flakiness makes assessing software quality challenging for developers. We do not know whether a flaky test fails because of a defect or another reason. This reduces developers' trust in the test suite, causing them to disregard potentially hazardous bugs if they encounter a flaky test case.

Test flakiness in software engineering is regarded as detrimental to the development process. A flaky test, as previously stated, is unreliable and cannot be used to assess software quality. More dangerously, junior developers tend to disregard flaky test cases or keep repeating them until they pass, allowing potentially dangerous flaws to go unreported and polluting subsequent test suites in the development pipeline. Senior developers may try to figure out what is causing the flakiness so they can repair it. However, investigating the source of flakiness is a time-consuming operation that can squander valuable resources if the cause of flakiness turns out to be a false alarm. Overall, flaky tests have a negative impact on the software engineering cycle because they hinder development progress, conceal software flaws, and are expensive to correct in the long term. Test flakiness is a well-known issue in the software industry that even giant tech companies must deal with.

3 TYPES OF FLAKY TESTS

In this section, we classify the flaky tests into three types based on their origin of flakiness:

- **Test-based flakiness**: These are tests whose source of flakiness comes from the tests themselves or the testing framework. Possible reasons for flakiness of this type is due to faulty test scripts; incorrect random test data; unstable testing frameworks that rely on unique identifiers for the tested objects, which are randomly generated after each build; tests that require asynchronous waiting for some result from the unit under test; or order-dependent (OD) tests.
- **Environment-based flakiness**: Flaky tests of this type are influenced by external factors like network bandwidths, insufficient storage or memory, or conflicts while testing in multiple environments. This flakiness usually happens with user-interface testing or system testing, where the tester needs the whole software stack to run the tests. The more things are involved, the more potential for error to happen unexpectedly. For example, if the network suddenly goes down while testing an online program, the test will inevitably fail, but not due to program bugs. Another example is if a test only passes due to the assumption that it only works on a specific operating system, then it is executed on another, leading to failure. Software dependency conflicts after an update cycle are also a cause. Environment-based test flakiness is challenging to resolve because it involves the whole software stack, so it takes much time to determine the causes.
- **Product-based flakiness**: This type of flakiness is caused by the uncertainty of the product under test itself. It can be considered as an actual product bug that needs to be fixed. Usually, the flaky test is performed on the part of the program that uses some concurrency like multi-threading, this problem is called a race condition. Another common reason is memory leaks caused by the program under test, which slows the whole system down or even crashes it entirely before the test is completed.

4 RELATED WORK

Similar to our paper, there have been several literature reviews and survey papers about flaky tests. Parry et al.[17] conducted a thorough survey of 76 flaky test research papers to analyze the root causes of flaky tests, their costs and consequences, detection, mitigation and repair strategies that are being researched and used in the industry. Zolfaghari et al.[23] presented a comprehensive review of the recent achievements in detecting and mitigating flaky tests as well as establishing a future roadmap for flaky test research. While most research works focus on flaky tests in unit testing, Romano et al.[18] did an empirical analysis of UI-based flaky tests, which helped academic researchers better understand flaky tests when it comes to system testing. Contrary to the previous studies, our paper focuses more on the practical aspect of the current research progress in both the academic and industrial fields by reviewing the various tools and techniques researchers have developed throughout the years, along with a rundown of what some popular automation testing tools in the industry are capable of to deal with flaky tests.

5 ACADEMIC FLAKY TEST TOOLS

The field of flaky test research is still relatively new compared to other academic fields, despite the fact that test flakiness has been plaguing the software testing industry for a long time. According to Parry et al.[17], the earliest reference to the flaky testing problem in academic research is from 2009 by Lacoste et al.[12] And from then until 2018, there were only a handful of flaky test researches each year. It was only from 2019 that the flaky test problem started to gain traction among academic researchers. In recent years, this problem has become a popular research interest in both academic and industry software testing. Here we shall list some notable researches and the tools they created to try tackling the problem of flaky tests, mainly in unit testing and system testing with graphical user interface (GUI).

5.1 Unit testing

Unit testing is the lowest level of software testing and also the easiest level to perform, which is why there has been much research for tools to handle flaky tests in unit testing. When it comes to unit testing, one of the most common reasons for flaky tests is order dependency. The first academic tool to mitigate order-dependent flaky tests is VmVm by Bell et al.[2] Another framework for detecting and partially classifying order-dependent flaky tests is iDFlakies by Lam et al.[14]

More often than not, order-dependent tests rely on some global states or static fields during the testing process, leading to the state-pollution problem, one of the root causes of flaky tests. Gyori et al. tried to detect test cases causing state pollution with their tool, PolDet[6].

In some cases, the unit under test may use some non-deterministic implementation such as a HashSet, which leads to randomized test results. A tool called NonDex[5] was developed to detect and debug flaky tests that involve such randomized implementations.

Even nowadays, the most common approach for detecting flaky tests is simply to rerun the test many times to check whether or not the results are consistent. But the random nature of flaky tests means that it is difficult to know how many times the tests need to be rerun so we can be sure that their results are completely consistent. Some flaky tests only need to be rerun 2-3 times to be detected, some need tens to hundreds or even thousands of reruns to start showing inconsistent results. If the cost of running tests is large enough, rerunning may not be effective for detecting flaky tests. Bell et al. presented a tool called DeFlaker[3] for detecting flaky tests without rerunning them using differential coverage analysis. Alshammari et al. also tried to apply machine learning techniques with FlakeFlagger[1] to predict flaky tests without rerunning them.

After having created many frameworks and tools for detecting flaky tests, researchers have also focused on automatically repairing flaky tests in recent years. A framework called iFixFlakies[19] was made for automatically fixing order-dependent flaky tests. To automatically repair implementation-based flaky tests detected by NonDex, Zhang et al. developed a tool called DexFix[22]. Lam et al. also introduced the RootFinder[13] to assist in identifying root causes for repairing flaky tests.

5.2 System testing

Unlike unit testing, system testing is the topmost level in the software testing process, meaning that it is the most difficult to verify. System testing involves all components of the system under test, which leads to a higher degree of flakiness. Developers tend to limit the number of tests they perform on the system level due to the high risk of flakiness and also the costs of running tests on the whole system every time. Morán et al. introduced a technique called FlakcLoc[16] to locate the root cause of flakiness in web applications by executing them under different environmental factors. Dong et al. presented a tool called FlakeShovel[4] for detecting flaky GUI tests on Android applications using a systematic exploration of event orders. Unfortunately, research about flaky tests in system testing is still in its infancy, so there are not many tools or techniques yet for handling flaky tests at this level. Current research for flaky GUI tests is still mostly theoretical. We hope that in the near future, with flaky unit tests becoming more popular as a field, researchers would also be interested in flaky GUI tests.

6 INDUSTRY AUTOMATION TESTING TOOLS

Outside of the academic field, people in the software testing industry have also developed methods in their testing tools to assist developers with handling flaky tests. There are 3 areas where automation testing tools can help deal with flaky tests:

- Assist in identifying flaky tests;
- Assist in investigating the root causes;
- Handle when the flakiness happens during test execution.

Here are some notable commercialized automation testing tools that can mitigate flaky tests:

6.1 Katalon Studio

Katalon Studio[7], which is developed by Katalon, is one of the popular tools for an all-in-one automation testing platform, especially as an end-to-end (E2E) testing tool. It is built for continuous testing, so it has many methods to mitigate flaky tests in a continuous integration and development (CI/CD) pipeline.

First is the test case report feature[10], which generates a report each time a test suite is executed. The report given by Katalon Studio can point out which tests are flaky and how much flakiness they have. Testers can use the information from this report to investigate the root causes of flakiness. Katalon Studio also let testers choose which tests should be repeated so the flaky ones can be thoroughly vetted.

Another feature of Katalon Studio is smart wait[9], which let a test wait with multiple flexible configurations such as waiting until a UI element is present, waiting until a page has finished loading, etc. This helps avoid the asynchronous wait problem, which is a common reason for flaky tests.

Katalon Studio also generates many types of web element locators such as the relative XPath, neighbor XPath, or even locators based on the element images that can be used to determine the changing UI element during a test execution, which is a common problem leading to flakiness in system testing. The self healing feature[8] of Katalon Studio uses these locators to automatically repair the test script if it is flaky during execution.

6.2 TestProject

This is another free E2E test automation platform for web, mobile, and API testing developed by Tricentis. TestProject[21] has the capability to handle flaky tests during UI testing using artificial intelligence (AI) features like self-healing, which automatically restores an XPath locator if it is broken or missing during execution, adaptive waiting which automatically adjusts the waiting time between test steps depending on factors such as network speed, finally is the testing assistant which repairs a flaky test by finding alternative paths to reach the next test step if the current step has some error.

6.3 LambdaTest

LambdaTest[15] is a continuous quality testing cloud platform that helps developers and testers ship code faster. One of its features is the Test At Scale (TAS) which can help deal with flaky tests in the CI/CD pipeline easier. The Flaky Test Management Pipeline can find, flag, quarantine and manage flaky tests. LambdaTest also has a fully automated triaging solution to assist in determining the culprit commits that lead to failed jobs.

6.4 Kobiton

Kobiton[11] is an automation testing tool mainly for mobile apps. This tool handles the flakiness problem when running tests across multiple devices by using an AI-powered automation engine that can ensure the test execution is the same across devices and is adaptable.

6.5 Testim

Testim[20] is an AI-powered test automation platform for web applications. It has features to tackle the problem of test flakiness, such as Auto-Improving Smart Locators of web elements which is similar to the self-healing feature of TestProject, and another feature is the Root Cause Analysis that reports what makes tests fail and even analyze which causes are the most common among the test suite. These two features assist a lot in mitigating flaky tests.

7 FUTURE DIRECTIONS

After reviewing the tools available in both academics and the industry for tackling the flaky test problem, we observe that several areas are lacking research focus when it comes to this problem. In this section, we discuss several future directions for the problem domain of test flakiness that researchers can take as the domain gains more attention over the years.

• Definition, modeling and taxonomy of test flakiness.

While there are a number of studies focusing on handling flaky tests, there is a lack of a formal definition for the problem of flaky tests. One possibly important step to take in researching test flakiness should be coming up with formal modeling of flaky tests, presenting a taxonomy on flaky tests, and developing a industry standard for flakiness-free testing.

• Data sets for studying test flakiness in integration and system testing.

Most studies on test flakiness have been focused on unit testing, which has a number of publicly availble data sets. We hope that in

the near future, researchers can also start to focus on flaky tests in integration testing and system testing. A good starting point is creating a unified resource repository to enable researchers to experiment with flaky tests at the integration and system levels.

• Tools and techniques for detecting flaky tests without rerunning them.

Today, testing tools in the industry primarily detect flaky tests by rerunning them multiple times. This can be very costly in terms of time yet does not completely guarantee the flakiness of a test if the flaky reasons are related to uncontrollable environmental issues, especially for software having a sophisticated CI/CD pipeline. With the advancements of artificial intelligence, we may see more AI-based approaches to tackle the challenge of test flakiness. For example, AI-based approaches can predict whether a test is potentially flaky, thus helping developers quarantine or temporarily skip the test before wasting time rerunning it.

• Automation frameworks for detecting, classifying, analyzing, and fixing flaky tests.

One of the biggest challenges for test flakiness research is having viable methods to shorten the time for investigating the root causes of flaky tests. As a test can be flaky for many reasons, we will need a framework to reduce the efforts spent investigating and handling flakiness.

• Test flakiness in cloud-based environments.

Currently, CI/CD in cloud-based environments is a common practice in the software industry. As software continues to become more integrated with cloud environments, the need to handle the test flakiness when deploying software to these environments is also rising. This is a promising direction for researchers to take in the future.

8 CONCLUSION

Test flakiness has been a common problem in software automation testing, and research about flaky tests has been gaining more attention from researchers in recent years. However, from our review of the current literature in this domain, most studies focus on test flakiness in unit testing, while research for approaches to tackle flaky tests at the system level is still rare. On the other hand, automation testing tools in the industry mainly address test flakiness in end-to-end system testing. While the industry already has some capability to deal with flaky tests using their automation testing tools, most approaches still require much intervention from the testers to quarantine and repair flaky tests. With the growing interest in test flakiness from researchers, we hope there will be more focus on researching flaky tests at the system level.

REFERENCES

[1] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting flakiness without rerunning tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1572–1584.

[2] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering*. 550–561.

[3] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 433–444.

[4] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2020. Concurrency-related flaky test detection in android apps. *arXiv preprint arXiv:2005.10762* (2020).

[5] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 993–997.

[6] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 223–233.

[7] Katalon. 2022. Katalon Studio. Retrieved September 5, 2022 from https://katalon.com/katalon-studio/

[8] Katalon. 2022. Katalon Studio - Self Healing. Retrieved September 5, 2022 from https://docs.katalon.com/docs/katalon-studio-enterprise/test-design/web-test-design/self-healing-tests

[9] Katalon. 2022. Katalon Studio - Smart Wait. Retrieved September 5, 2022 from https://docs.katalon.com/docs/katalon-studio-enterprise/keywords/web-ui-keywords/webui-smart-wait-function

[10] Katalon. 2022. Katalon Studio - View test case report. Retrieved September 5, 2022 from https://docs.katalon.com/docs/katalon-testops/reporting/view-test-case-reports

[11] Kobiton. 2022. Kobiton. Retrieved September 5, 2022 from https://www.kobiton.com/

[12] Francis J Lacoste. 2009. Killing the gatekeeper: Introducing a continuous integration system. In *2009 agile conference*. IEEE, 387–392.

[13] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 101–111.

[14] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 312–322.

[15] LambdaTest. 2022. LambdaTest. Retrieved September 5, 2022 from https://www.lambdatest.com/

[16] Jesus Morán, Cristian Augusto, Antonia Bertolino, Claudio de la Riva, and Javier Tuya. 2019. Debugging Flaky Tests on Web Applications.. In *WEBIST*. 454–461.

[17] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–74.

[18] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An empirical analysis of UI-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1585–1597.

[19] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 545–555.

[20] Tricentis. 2022. Testim. Retrieved September 5, 2022 from https://www.testim.io/

[21] Tricentis. 2022. TestProject. Retrieved September 5, 2022 from https://testproject.io/

[22] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. 2021. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 50–61.

[23] Behrouz Zolfaghari, Reza M Parizi, Gautam Srivastava, and Yoseph Hailemariam. 2021. Root causing, detecting, and fixing flaky tests: state of the art and future roadmap. *Software: Practice and Experience* 51, 5 (2021), 851–867.