2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-7281-9630-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/SANER50967.2021.00081

Research Progress of Flaky Tests

Wei Zheng†, Guoliang Liu†, Manqing Zhang†, Xiang Chen‡, Wenqiao Zhao†
†*Software Institute*, *Northwestern Polytechnical University*, China
‡*School of Information Science and Technology*, *Nantong University*, China
wzheng@nwpu.edu.cn, lzh1505413361@gmail.com, zmqgeek@gmail.com, xchencs@ntu.edu.cn, 892096557@qq.com

***Abstract*—A flaky test is a test that both passes and fails periodically without any code changes, and its uncontrolled uncertainty will destroy the value of the test suites and even cause developers to distrust the test results. Recently, researches in the flaky test have received broad attention in the software test community to reduce the manual maintenance cost of flaky tests by developers. In this survey, we conducted comprehensive research progress on the flaky test. We identified 31 relevant studies and summarized the following aspects of the flaky test: root causes and factors, analyzing the impact, detecting and classifying techniques, and fixing approaches. This survey also identifies open research challenges to be further explored in future work.**

***Index Terms*—automated testing, flaky test, root cause, research progress**

I. Introduction

Regression testing is a crucial activity of software development performed by developers. It checks whether new software changes have broken existing functionality. A crucial assumption to make regression testing effective is that test outcomes are deterministic. Unfortunately, this is not always the case due to some tests often called flaky tests, i.e., tests that exhibit non-deterministic outcomes with the same code.

A flaky test is a test that both passes and fails periodically with the same configuration (such as the same test environment, product, and code). For example, flaky tests may fail or pass over different system time zone with the same code [1]. Flaky tests make the test results uncertain, resulting in the inability to determine whether the software has real bugs. It will significantly reduce the trust of software developers and project managers in automated testing. In addition, developers need to spend a lot of time to confirm flaky tests. The flakiness of a test will reduce developers' confidence in testers, which greatly damages software project development.

In recent years, flaky tests have obtained considerable research progress. The first extensive and explicit empirical analysis of flaky tests in the literature can be traced back to a conference paper by Luo et al. in 2014 [2]. Some studies present overview of the flaky tests, but are either preliminary or do not make flaky tests the main focus [3]–[13]. For instance, to draw attention to the fundamental role of diversity in test case selection strategies, Alessio et al. [14] presented PARDET to automatically and practically detect manifest dependencies that might lead to flaky failures. Similarly, Nie et al. [11] presented an orchestrated experience report related to testing isolation, which could eliminate test-order dependencies. Consequently, there is currently no up-to-date, exhaustive research progress analyzing both the state-of-the-art and new potential research directions of flaky tests. Therefore this survey can fill this gap in the previous studies.

We followed the systematic literature review guidelines provided by Kitchenham and Charters [15] and presented comprehensive research progress on the flaky tests covering primary studies published between 2014 and 2020.

The rest of this survey is organized as follows. Section II presents our literature review methodology. Section III gives the root causes of flaky tests in existing studies. Section IV discusses the impacts of flaky tests in current studies. Section V analyzes the-state-of-the-art of flaky tests detection techniques. Section VI gives a detailed analysis of the various fix strategies and techniques of flaky test. Section VII provides some potential challenges to be addressed in future work. Finally, Section VIII concludes the survey.

II. Methodology

Guided by Kitchenham and Charters [15], we followed a structured and systematic method to perform the survey on the flaky test studies. The detailed methodology is described in the rest of this section.

A. Research Questions

Our goal was to gather and categorize the available literature on the flaky test. To achieve this goal, we designed the following research questions (RQs):

RQ1: What are the root causes of the flaky test?

RQ2: What are the impacts of the flaky test?

RQ3: What different strategies and approaches in detecting the flakiness?

RQ4: What are the previous studies in fixing the flaky test?

The answer to RQ1 will summarize the root causes of the flaky tests. RQ2 will identify the field that the flaky test that has been impacted. RQ3 will identify the state-of-the-art in flaky test detecting strategies and techniques, including a description, summary, and classification. RQ4 will summarize how the various flaky test studies involving fix strategies and fix technologies.

B. Literature Search and Selection

Following previous research progress studies on other software engineering topics [16], [17], we chose the following five literature repositories: (1) ACM Digital Library, (2) IEEE Xplore Digital Library, (3) Elsevier Science Direct, (4) Springer Online Library, (5) Scopus.

978-1-7281-9630-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SANER50967.2021.00081

After determining the five literature repositories, each repository was searched by using the exact phrase "flaky test" and titles or keywords containing "flaky test". Fig. 1 shows a summary of the search and selection process. We defined the selection criteria below.

- The publications are written in English.
- Studies explicitly mention they are targeting flaky tests.
- Studies contain empirical results (e.g., case study, experiments and surveys).

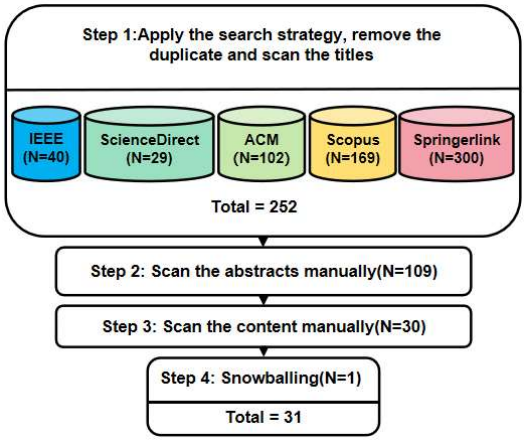

Fig. 1. Search and selection process

- Step 1: We applied our search strings in five literature repositories. Next, we discarded the duplicate papers. Then, we scanned the title and excluded the irrelevant papers (252 findings).
- Step 2: To extract the results, the first author manually scanned the abstracts based on selection criteria. Then, the second and third authors checked a sample of papers randomly. The differences were resolved in the discussion among all the authors. We included 109 studies in this step.
- Step 3: At the beginning, the first author scanned the content of the papers based on the selection criteria and marked the records as relevant/irrelevant studies. The second and third authors reviewed the findings and selected the relevant studies separately. We found 30 studies in this step.
- Step 4: Finally, the second author performed a forward snowballing on our included studies that have high citations. Other authors reviewed the new findings. We found one additional papers in this step. In total, we identified 31 relevant studies.

We acknowledge the apparent infeasibility of finding all flaky test papers through our search. However, we are confident that we have included the majority of relevant published papers and that our survey provides the overall trends and the-state-of-the-art studies of flaky tests.

III. Result Analysis for RQ1

In this section, according to the different factors leading to flaky tests, we summarized the root causes of flaky tests into three categories: root causes related to software under test, root causes related to test cases, and root causes related to environment.

A. Root Causes Related to Code Under Test

Async Wait. This kind of flaky tests occur when the test execution makes an asynchronous call and does not properly wait for the result of the call to be available before using it [18]–[20].

Concurrency. This kind of flaky tests is due to different threads interaction in an undesired way (but not due to asynchronous calls from the Async Wait category). For example, tests may pass or fail due to data races, atomicity violations, or deadlocks [18]–[20].

UI. When developers carelessly design the widget layouts on UI or misunderstand the underlying UI rendering process, some widgets were not rendered correctly, or a blank screen showed up now and then [21].

IO. Not correctly handling input/output channels can also result in flaky tests [18].

Program logic. Developers make incorrect assumptions about the behavior of the application and fail to implement functions correctly to handle corner cases [21]. Fig. 2 illustrates a program logic scenarios.

```
...
} catch (IOException e) {
  Util.closeQuietly(MockSpdyPeer.this);
-  throw new RuntimeException(e);
+  e.printStackTrace();
}...
```

Fig. 2. A commit to fix program logic error

In the commit, the tested class MockSpdyPeer may throw an IOException due to various reasons unrelated to the actual test. In the buggy version, the program caught the exception and rethrew another exception RuntimeException, which could abruptly terminate the whole program. In the fixed version, the program was altered to log the exception without throwing an exception. In this way, the code could ward off the intermittent crash due to any RuntimeException and reduce the overall flakiness problem.

Floating Point Operations. Floating point operations could cause flaky test failures if potential precision overflows and underflows are not properly considered(e.g., calculating the sum of an array). [18].

Randomness. Flaky tests can be caused by using a random number generator without considering all the possible values that could be generated [18], [19], [22].

Resource leak. When an application fails to properly manage (get or release) one or more of its resources (such as memory allocations), a resource leak occurs, leading to flaky tests [18].

B. Root Causes Related to Test Cases

Test Order Dependency. Test results depend on the order in which tests were run, and different test cases order will lead to uncertain results [18].

Too Restrictive Range. In this category, some effective output value is not in the test design consideration of assertion, so the test fails when they appear. [22].

Test Suite Timeout. The test suites will grow over time, and the maximum run-time value will not always be adjusted accordingly, which leads to the passing or failing of the test suite depending on different random variables. [22].

Test Case Timeout. This kind of flaky tests is associated with a single test that experiences non-deterministic timeouts [22].

C. Root Causes Related to Environment

Network. Flaky tests frequently appear since the network is a difficult resource to control [18].

Time. System time dependence caused the flaky outcome [18].

Platform Dependency. Test failures occur only on specific platforms [22].

Summary for RQ1: The root causes of flaky tests mainly have the following three dimensions: 1) root causes related to code under test, 2) root causes related to the test cases, 3) root causes related to the environment.

IV. Result Analysis for RQ2

Flaky tests directly impact software quality assurance because they hinder development progress and hide real bugs [23]. This section summarizes two primary impacts (i.e., impact on projects and test techniques).

A. Impact on Projects

User Crash Reports: Firefox. Rahman et al. [24] conducted a study related to the link between builds with flaky tests and the number of user crash reports on the Firefox web browser. They found that ignoring flaky test failures will increase the number of browser crashes.

B. Impact on Test Techniques

Test case Prioritization. Test case prioritization (TCP) refers to prioritizing test cases in the test suite on the basis of different factors [25]. Peng et al. [26] conducted a study on enhancing TCP. They found that flaky test failures can substantially affect the results from evaluating and comparing different TCP techniques. In particular, for change-aware TCP techniques, their study demonstrated that all change-aware TCP techniques perform better on non-flaky tests than on flaky tests.

Automated Repair. Matias et al. [13] conducted an experiment on automatically fixing real bugs in Java based on the Defects4J database [27]. They uncovered that flaky tests have a direct impact on automatic repair. If the failing test case is flaky, the fixing system might identify a revised patch while it is actually right. If one of the passing test cases is flaky, the fixed system might identify that a patch has introduced a regression while it is not. More specifically, Cordy et al. [28] proposed a test flakiness assessment and experimentation platform named Flakime, which supports the seeding of a (controllable) degree of flakiness into the behavior of a given test suite. Their experiment found fault localization (FL) step of automated repair techniques is the most sensitive to test flakiness.

Summary for RQ2: The current studies indicate that the impacts of flaky tests mainly include: 1) user crash reports of Firefox, 2) test-case prioritization, 3) automated repair especially in fault localization.

V. Result Analysis for RQ3

In this section, we summarized and analyzed the existing detection methods, among which detect flaky tests can be divided into two types: Rerun and non-Rerun. We further divided the Rerun techniques into detecting causes of flaky tests and detecting flaky tests.

A. Rerun

Rerun is a common way to combat flaky tests: if some reruns pass and some not, the test is definitely flaky, but if all reruns still fail, the status is unknown.

1) Detect Causes of Flaky Tests:

state pollution, data dependencies, and manifest dependencies. State pollution might result in data dependencies, which might cause manifest dependencies and flaky test failures. Alex et al. [7] proposed a technique, called PolDet, focused on practical detection of state polluting tests. In contrast, Bell et al. [6] proposed a tool named ElectricTest, which focused on practical detection of data dependencies between tests. It is worth noting that the presence of state pollution does not imply that there will be data dependencies between tests, and the presence of data dependencies does not imply that tests will fail if the tests are re-ordered. Simply detecting tests that could cause data dependencies or tests that have data dependencies between each other and could manifest as flaky test failures is insufficient to provide clear, actionable guidance to developers. Hence, Alessio et al. [14] presented a novel technique, called PRADET, for automatically and practically detecting these manifest dependencies, reporting these results to developers, and allowing them to be aware of and respect that dependency.

Time, Randomness, Async Wait, and Concurrency. Lam et al. [19] put forward an end-to-end framework to help identify flaky tests and understand the root causes. Their work first uses the CI pipeline inside Microsoft, a CloudBuild system, to identify flaky tests by rerunning failing tests and storage the Torch log information during the identification. Then they develop a tool called RootFinder to parse Torch logs of passing and failing executions to identify potential root causes of certain types of flaky tests, including time, randomness, async wait, and concurrency. For Algorithmic

randomness, Dutta et al. [29] proposed FLASH, a technique for detecting flaky tests due to the non-deterministic behavior of algorithms. At a high-level, FLASH runs tests multiple times but with different random number seeds, which leads to different sequences of random numbers between executions. FLASH then checks for differences in actual values in test assertions and even different test outcomes.

Code level. Testing at the test level can only determine whether a test is a flaky test. Celal et al. [30] proposed a method to locate the root causes of flaky tests on the code level to help developers debug and fix them. This technology compares the first divergence point in the code control flow by discovering failures and passing the test run trajectory to use this novel divergence algorithm to identify the code's earliest flaky places.

Flaky tests with fuzzy root causes. Different from previous studies, Terragni et al. [31] put forward a method to explain the execution results instead of focusing on specific root causes. Their core idea is to execute a flaky test under different execution clusters repetitively. Each cluster explores a certain non-deterministic dimension (e.g., concurrency, I/O, and networking) with dedicated software containers and fuzzy-driven resource load generators.

2) Detect Flaky Tests:

Flaky tests due to order dependency. Developers who run regression tests are usually concerned with whether a single test passes or fails and whether the entire test suite has failed any tests. Lam et al. [32] put forward an end-to-end framework iDFlakies. The core of their framework is their tool that can automatically detect flaky tests and classify it into OD (Order Dependency) test and NOD (Non-Order Dependency) test, and supports the use of this tool in different projects.

Flaky tests due to ADINS Code. First and foremost, the term ADINS code refers to code that Assumes a Deterministic Implementation of a method with a Non-deterministic Specification. August et al. [33] proposed a tool, called NONDEX, to detect flaky tests due to ADINS code. They identify 31 methods with non-deterministic specifications, provided models for these methods to create various non-deterministic choices, and use an execution environment that can explore various combinations of these non-deterministic choices.

Flaky tests of time-constrained. Because of the high cost and uncertainty of Rerun, Silva et al. [34] proposed a fantastic technique, called SHAKER, to improve the ability of Rerun to detect time-constrained flaky tests. SHAKER is a lightweight approach to detect flaky tests due to time constraints. SHAKER's key insights assume that: (1) such tests can be detected by adding noise in the environment where test cases will be executed, and (2) compared with rerunning tests without noise, rerunning tests in a noisy environment will more instantly expose flaky tests.

Others. Some other studies are not mainly aimed at the flaky tests but can also be used to detect flaky tests with specific root causes in a certain circumstance. Andre et al. [9] proposed an approach NODERACER, which aims to detect event races in Node.js applications by selectively postponing events, guided by happens-before relations. Castro et al. [35] provided a tool ASTRO, which can visualize log information from multiple perspectives. This tool can compare the tests between different time points and detect flaky tests caused by time.

B. Non-Rerun

Rerun is a common way to deal with flaky tests, but the rerunning cost is high, which will slow down the development process. Hence many studies began to explore the field of non-rerun, namely, to detect flaky tests without rerunning them.

Bell et al. [36] proposed a new efficient technology Deflaker to detect flaky tests by tracking the changed code coverage, which they call differential coverage. It is a supplement to Rerun. For tests without code changes, Deflaker will mark the failed test as a flaky test immediately without re-running. More important, for tests with changed code, the crucial insight in Deflker is that one needs not to collect the coverage of the *entire code base*. Instead, one only needs the differential coverage from each program source file. Finally, when each test run, it monitors change execution, generating an individual change-coverage report. Once the test is completed and executed, Deflaker will analyze the coverage information report. If the test fails and it does not cover any recent code changes during regression testing, the test is marked as flaky. DeFlaker works only on failing tests.

As a supplement of Deflaker, Dong et al. [37] proposed a method FlakyShovel to detect flaky tests in the pass tests by systematically exploring the event orders. Based on the premise that a flaky test can be exposed by exploring event execution order space that may result from different execution environments. Instead of running tests in all possible environment-related the space of event execution orders, they validate a test by exploring the space of feasible event execution orders.

Furthermore, Tariq et al. [38] proposed a supervised ML method to classify and predict flaky tests through the machine learning method. They treat the test flakiness problem as a disease by specifying its symptoms and possible causes. Pinto et al. [39] applied a variety of machine learning algorithms (including Random Forest and Support Vector Machines, etc.) based on the vocabulary of tests to automatically classifying them as flaky or not. They observed that job, action, and services are generally correlated with flaky tests.

Summary for RQ3: Comparison and summary between different detection technologies can be found in Table I.

VI. Result Analysis for RQ4

In this section, we divided studies related to fixing flaky tests to fix strategies and fix technologies. Fix strategies refer to how to fix the common flakiness in the development process, and we conducted the details in Table III. Fix technologies is to assist programmers in fixing flaky tests and improving efficiency and quality, mainly introduced in three aspects: refactoring, strategically re-running, and test suite patches.

TABLE I
COMPARISON OF DIFFERENT DETECTION METHODS

Study	Technique Type	Technique	Detect Type	Concrete Detection
Bell et al. [40]	Non-Rerun	DeFlaker	Flaky Tests	Failed Flaky Tests
Dong et al. [41]	Non-Rerun	FlakyShovel	Flaky Tests	Passed Flaky Tests
Tariq et al. [42]	Non-Rerun	Bayesian	Flaky Tests	Flaky Tests Based on Source Code
Lam et al. [32]	Rerun	iDFlakies	Flaky Tests	Flaky Tests Due to Order Dependency
August et al. [33]	Rerun	NONDEX	Flaky Tests	Flaky Tests Due to ADINS Code
Silva et al. [34]	Rerun	SHAKER	Flaky Tests	Flaky Tests of Time-constrained
Andre et al. [9]	Rerun	NODERACER	Flaky Tests	Flaky Tests Due to Event Races
Castro et al. [35]	Rerun	ASTRO	Flaky Tests	Flaky Tests Due to Time
Alex et al. [7]	Rerun	PolDet	Root causes	State Pollution
Bell et al. [6]	Rerun	ElectricTest	Root causes	Data Dependencies
Alessio et al. [14]	Rerun	PRADET	Root causes	Manifest Dependencies.
Dutta et al. [43]	Rerun	FLASH	Root causes	Algorithmic Randomness
Lam et al. [1]	Rerun	RootFinder	Root causes	Time, Randomness, Async Wait, Concurrency
Celal et al. [30]	Rerun	Flakiness Debugger	Root causes	Code Level Factors
Terragni et al. [31]	Rerun	Container-Based Infrastructure	Root causes	Fuzzy root causes of flaky tests

* Technique Type means the technique type we have classified.
** The detection type refers to the detection of flaky tests or root causes, and Concrete Detection means the explicit type of detection Type.

A. Fix Strategies

In exploring the root causes of flaky tests, developers have some common fix strategies for some specific causes. For instance, the key to solve flaky tests caused by Async Wait is to solve the order conflict between different threads or processes. Developers fix it by using waitFor and calling sleep(). We summarize and detail fix strategies in Table III.

B. Fix Technologies

1) Strategically Re-running: **Mutation test flakiness.** Mutation testing implicitly assumes that tests exhibit deterministic behavior in terms of their coverage and the outcome of a test (not) killing a certain mutant. Such an assumption does not hold in the presence of flaky tests. August et al. [44] evaluated the impact of flaky tests on mutation testing through empirical research and proposed a technique to manage flakiness during the entire mutation testing process. The technology is mainly based on strategic re-running tests and achieved by modifying the open-source mutation testing tool PIT (JAVA mutation testing tool) to improve mutation testing reliability.

Asynchronous calls flakiness. Lam et al. [45] studied the lifecycle of flaky tests on six large-scale proprietary projects at Microsoft. They found that the universal type of flaky tests is the Async Wait. To mitigate these tests, they proposed an automated solution FaTB (Flakiness and Time Balancer). FaTB will try various time-values and outputs the minimum time values that developers should use based on their tolerance for test-flakiness failures.

2) Test Suite Patches: Test order dependency is a common root cause of flaky tests. However, it is troublesome to analyze the order dependency among tests and fix it.

August et al. [46] proposed a framework iFixFlakies, which includes core components of Minimizer and Patcher, for automatically repairing order-dependent flaky tests. They divided the order-dependent flaky tests into two types: (1) victim: pass when running independently, but fail when running with some other tests; (2) brittle: fail when running alone, but pass when run with other tests together. And other tests related to order-dependent tests are divided into three different roles: polluter, cleaner, and state-setter. iFixFlakies accepts an order-dependent test, a passed test order, and a failed test order as the input. Call Minimizer to obtain the type of order-dependent test, the minimized polluter/state-setter, and the minimized cleaner. According to the type of order-dependent test, iFixFlakies then calls Patcher to create a patch corresponding to each helper(tests of test suites) for the order-dependent test.

Lam et al. [47] conducted a further study, and they proposed a universal technique for enhancing regression testing algorithms to make them dependent-test-aware. They applied the technique to 12 algorithms. The outcomes manifest that compared with the orders produced by unenhanced algorithms, the orders produced by enhanced algorithms (1) have overall 80% fewer flaky-test failures because of Order-Dependent tests, and (2) could increase extra tests but run only 1% slower on average.

3) Refactoring: Palomba et al. [48], [49] conducted experiments to refactor test smell according to the guidelines provided by Van Deursen et al. [50] and evaluate the ability to remove flakiness. They evaluate the effect of refactoring, showing that it can remove design flaws and fix all 75% flaky tests causally co-occurring with test smells.

Summary for RQ4: The current fix technologies and fix strategies are summarized in Table II and Table III.

VII. CHALLENGES AND OPPORTUNITIES

The current studies on flaky tests are still not enough to aroused widespread concern in the academic community. There are still many challenges to solve the problem of flaky tests in software testing. This section discussed future research directions, challenges, and possible solutions to flaky tests from four aspects: benchmarks, reproduce, detection, and fix.

TABLE II
COMPARISON OF FIX TECHNOLOGIES

Study	Technique Type	Technique	Repair Flakiness Type
Shi et al. [51]	Strategically Re-running	Modified PIT	Mutation Test Flakiness
Lam et al. [45]	Strategically Re-running	FaTB	Asynchronous Calls Flakiness
August et al. [52]	Test Suite Patches	iFixFlakies	Order-Dependent Flakiness
Lam et al. [47]	Test Suite Patches	Dependent-Test-Aware	Order-Dependent Flakiness
Palomba et al. [48], [53]	Refactoring	Refactoring of Test Smells	Test Sells Flakiness

TABLE III
COMPARISON OF FIX STRATEGIES

Study	Root Causes	Fix Strategies
Luo et al. [18],	Async wait	Call of waitfor(), sleep(), etc
	Concurrency	Add lock, modify concurrency condition, assertion, etc
	Test Order dependency	Set/clean state, removing dependencies or merging testing
	Network	Use simulation for testing
	Float point operation	Make test assertions as independent of floating point results as possible
	Unordered collection	Try to avoid some any specific ordering on collections
	Resource leak	Managing related resources through resource pools
	Time	Avoid using platform-dependent values
	Randomness	Control the seeds of the random generator
Moritz et al. [22]	Too Restrictive Range	Fix assertion, adjust fuzzing, disable test, missing code added
	Test Case Timeout	Increase timeout, skip non-Initialized Part, split test, disable test
	Platform Dependency	Run additional tests, correct directories
	Test Suite Timeout	Skip non-initialized part, split test Suite
Thorve et al. [21]	Dependency	Replace implementation
	Program logic, UI	Improve implementation, modify code or assertion

Benchmarks. Currently, the unified benchmark has not been established to verify the effectiveness of the flaky test's detection and fix. Since flaky tests are a common problem in software projects and it is difficult for academia to obtain it through small-scale experiments. However, there are very few open-source flaky tests data in industrial software, so it is challenging to verify the effectiveness of detection and repair technologies in different languages and actual projects. In order to promote the research progress in the field of flaky tests, it is very necessary to construct a unified large-scale data set in the future. At the same time, it is hoped that the industry and academia will work together to promote the development of the field.

Reproduce flaky tests. A common challenge for developers is to identify and replay flaky tests [22]. Due to the occasional nature of flaky test failures, the test is difficult to reproduce. Therefore, it is necessary to explore the root causes of flaky tests and obtain the accurate test case execution path by recording the coverage and detailed information. In addition, Exploring how to streamline the test reproduction process and retain key test path information will help reproduce flaky tests efficiently and accurately.

Detect flaky tests. Previous works [9], [32], [34], [35], [39]–[42] has made a lot of efforts to detect flaky tests. However, there are many root cause for flaky tests [1], [6], [7], [14], [29]–[31], [43], these detection techniques are only for certain specific flaky tests. Some root causes of flaky tests are difficult to detect, and developers often choose to ignore them, but this can introduce serious vulnerabilities. Therefore, it is necessary to construct an intelligent detection technology to synthesize various types of flaky test reasons, and to model flaky tests uniformly. In the future, detection tools for flaky tests should support multiple types of projects and most kind of flakiness. Using machine learning to detect unstable tests is a research direction that can be explored in the future.

Fix flaky tests. In order to repair the unstable test, it is necessary to locate the root causes of flaky tests first. However, flaky tests may be caused by multiple dependencies. Therefore, it is necessary to explore and construct the patch set of flaky tests, which can automatically match and repair flaky tests based on the patch set. Moreover, the previous research aimed at the specific reasons of flaky tests and fix technologies. In the future, a new method is needed to cover different levels of flaky tests.

VIII. CONCLUSION

The existence of flaky tests destroys the reliability of regression testing. It is essential to solving the problems caused by a flaky test for software testing and software quality assurance. This survey reviews the previous studies on the flaky test and discusses the root cause, fix method, and detection method about the flaky tests. The main challenges and corresponding research directions are also pointed out to help the researchers conduct related studies in the future.

ACKNOWLEDGEMENT

This research was partially supported by the Key Laboratory of Advanced Perception and Intelligent Control of High-end Equipment, Ministry of Education (GDSC202006) and the Seed Foundation of Innovation and Creation for Graduate Students in Northwestern Polytechnical University (CX2020246).

REFERENCES

[1] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 101–111.

[2] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[3] A. M. Memon and M. B. Cohen, "Automated testing of gui applications: models, tools, and controlling flakiness," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1479–1480.

[4] O. R. Valdes, "Finding the shortest path to reproduce a failure found by testar," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1223–1225.

[5] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 79–90.

[6] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 770–781.

[7] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: detecting state-polluting tests to prevent test dependency," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 223–233.

[8] J. Bell and G. Kaiser, "Unit test virtualization with vmvm," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 550–561.

[9] A. T. Endo and A. Møller, "Noderacer: Event race detection for node. js applications," 2020.

[10] A. Shi, P. Zhao, and D. Marinov, "Understanding and improving regression test selection in continuous integration," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 228–238.

[11] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, "Debugging the performance of maven's test isolation: Experience report," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 249–259.

[12] L. Ma, C. Zhang, B. Yu, and H. Sato, "An empirical study on the effects of code visibility on program testability," *Software Quality Journal*, vol. 25, no. 3, pp. 951–978, 2017.

[13] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[14] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 1–11.

[15] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[16] T. W. W. Aung, H. Huo, and Y. Sui, "A literature review of automatic traceability links recovery for software change impact analysis," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 14–24.

[17] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, "A survey on adaptive random testing," *IEEE Transactions on Software Engineering*, 2019.

[18] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[19] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 101–111.

[20] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840.

[21] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 534–538.

[22] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840.

[23] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. Di Penta, "Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 327–337.

[24] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 857–862.

[25] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999, pp. 179–188.

[26] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 324–336.

[27] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[28] M. Cordy, R. Rwemalika, M. Papadakis, and M. Harman, "Flakime: Laboratory-controlled test flakiness impact assessment. a case study on mutation testing and program repair," *arXiv preprint arXiv:1912.03197*, 2019.

[29] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 211–224.

[30] C. Ziftci and D. Cavalcanti, "De-flake your tests: Automatically locating root causes of flaky tests in code at google," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 736–745.

[31] V. Terragni, P. Salza, and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 69–72.

[32] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 2019, pp. 312–322.

[33] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 80–90.

[34] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! detecting flaky tests caused by concurrency with shaker," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 301–311.

[35] D. Castro and M. Schots, "Analysis of test log information through interactive visualizations," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 156–166.

[36] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[37] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Concurrency-related flaky test detection in android apps," *arXiv preprint arXiv:2005.10762*, 2020.

[38] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a bayesian network model for predicting flaky automated tests," in *2018*

IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2018, pp. 100–107.

[39] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 492–502.

[40] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[41] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Concurrency-related flaky test detection in android apps," *arXiv preprint arXiv:2005.10762*, 2020.

[42] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a bayesian network model for predicting flaky automated tests," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018, pp. 100–107.

[43] D. Knott, *Hands-on mobile app testing: a guide for mobile testers and anyone involved in the mobile app business*. Addison-Wesley Professional, 2015.

[44] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 112–122.

[45] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1471–1482.

[46] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "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*, 2019, pp. 545–555.

[47] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 298–311.

[48] F. Palomba and A. Zaidman, "The smell of fear: On the relation between test smells and flaky tests," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2907–2946, 2019.

[49] ——, "Does refactoring of test smells induce fixing flaky tests?" in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 1–12.

[50] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*, 2001, pp. 92–95.

[51] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 112–122.

[52] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "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*, 2019, pp. 545–555.

[53] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 1–12.