Exploring Tools for Flaky Test Detection, Correction, and Mitigation: A Systematic Mapping Study

Pedro Anderson Costa Martins
Federal University of Ceará (UFC)
Quixadá, Ceará, Brazil
pedroanderson10@gmail.com

Victor Anthony Alves
Federal University of Ceará (UFC)
Quixadá, Ceará, Brazil
victorpa@alu.ufc.br

Iraneide Lima
Federal University of Ceará (UFC)
Quixadá, Ceará, Brazil
IraneideLima

Carla Bezerra
Federal University of Ceará (UFC)
Quixadá, Ceará, Brazil
carlailane@ufc.br

Ivan Machado
Federal University of Bahia (UFBA)
Salvador, Bahia, Brazil
ivan.machado@ufba.br

ABSTRACT

Flaky tests, characterized by their non-deterministic behavior, present significant challenges in software testing. These tests exhibit uncertain results, even when executed on unchanged code. In the context of industrial projects that widely adopt continuous integration, the impact of flaky tests becomes critical. With thousands of tests, a single flaky test can disrupt the entire build and release process, leading to delays in software deliveries. In our study, we conducted a systematic mapping to investigate tools related to flaky tests. From a pool of 37 research papers, we identified 30 tools specifically designed for detecting, mitigating, and repairing flakiness in automated tests. Our analysis provides an overview of these tools, highlighting their objectives, techniques, and approaches. Additionally, we delve into the highest-level characteristics of these tools, including the causes they address. Notably, approximately 46% of the tools focus on tackling test order dependency issues, while a substantial majority (70%) of the tools are analyzed in the context of the Java programming language. These findings serve as valuable insights for two key groups of stakeholders: (Software Testing Community:) Researchers and practitioners can leverage this knowledge to enhance their understanding of flaky tests and explore effective mitigation strategies; (Tool Developers:) The compilation of available tools offers a centralized resource for selecting appropriate solutions based on specific needs. By addressing flakiness, we aim to improve the reliability of automated testing, streamline development processes, and foster confidence in software quality.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **General and reference → Surveys and overviews**.

KEYWORDS

Flaky tests, tools, systematic mapping.

1 INTRODUCTION

The software testing process constitutes an important stage in software development. With shorter cycles and faster and more continuous deliveries in the agile development process, implementing automated tests has become essential for providing quicker feedback [33]. However, the reliability of this feedback may come into question due to the history of testing with ambiguous and uncertain results [34]. Tests with inconsistent results, commonly referred to as flaky tests, are tests that can pass or fail when run on the same version of the software, even without changes to the code [34, 38].

Several studies offer insights into the significant problems caused by flaky tests [11]. Failures in test suites due to flaky tests can occur frequently in software development, leading to issues such as delayed product delivery or decreased reliability of automated tests [5]. For instance, Luo et al. [31] conducted a study covering 51 open-source projects and analyzing 201 commits. They found that flaky tests accounted for 73 thousand out of 1.6 million (4.6%) instances of test failure in the Google TAP system. Studying the causes and detection strategies for flaky tests presents a significant challenge due to their non-deterministic nature, and it has become an increasingly active area of research [27]. According to Luo et al. [31], the primary causes of flaky tests include asynchronous waiting, concurrency, and order-dependent tests [25]. Other causes identified over time by [11], such as test case timeout and platform dependence, have been recognized. Flaky tests resulting from order dependence are often distinguished from those caused by other factors, leading to two main categories: order-dependent and non-order-dependent flaky tests [17].

Among the detection strategies, re-execution of tests is the most commonly used. However, this technique can be time-consuming, particularly with many tests. Alternative approaches have been proposed, including identifying flaky tests based on differences in coverage between consecutive software versions and monitoring [5, 34]. By understanding the primary causes and detection strategies for flaky tests, efforts can be directed towards identifying and selecting tools that can efficiently and automatically detect these issues [17, 34]. Several tools, as highlighted by Parry et al. [34] and Gruber et al. [17], are available for the test re-execution process or for identifying flaky tests, particularly in order-dependent scenarios.

This work aims to identify and analyze a selection of tools designed for detecting, mitigating, and correcting flaky tests, providing a comprehensive overview of their main characteristics. To achieve this objective, we conducted a systematic mapping study to understand better flaky test detection, mitigation, and repair tools. Through this study, we compare these tools and delve into the techniques and approaches they employ. By synthesizing the findings of our study, we aim to offer professionals a framework of

flaky testing tools, facilitating the selection of the most suitable tool for their specific needs and circumstances. This framework will enable professionals to make informed decisions when addressing flaky tests within their software development and testing processes.

2 BACKGROUND AND RELATED WORK

2.1 Flaky tests

Flaky Tests behave non-deterministically, meaning they can pass or fail when executed repeatedly on the same code under test [15]. One of the pioneering empirical studies conducted on flaky tests by [31] examined 201 commits in open-source projects to address flaky tests. This investigation identified and classified several general causes of flaky tests. `Asynchronous Wait`, `Concurrency`, and `Test Order Dependency` were identified in 125 out of the 201 inspected commits, accounting for approximately 62% of the commits. Consequently, these factors are considered the primary causes of flaky tests.

Flaky tests, like bugs, are recognized as issues that cannot be entirely eliminated from software. Consequently, numerous studies have introduced and assessed various mitigation strategies and techniques to manage this problem [34].

Gruber and Fraser [15] highlighted several approaches proposed to mitigate flaky tests, including: **re-executing tests:** Rerunning tests that previously exhibited flakiness issues; **Disabling Tests:** Temporarily disabling tests that consistently demonstrate flakiness; **Automated Test Re-execution:** Automatically rerunning tests using annotations or scripts; and **Utilizing Automated Detection Tools:** Employing tools specifically designed for the automatic detection of flaky tests.

While completely eliminating flaky tests may not be feasible, developers employ various techniques to address their causes in different scenarios and specific situations. Table 1 presents some of these corrective techniques, as indicated by the study conducted by Luo et al. [31]. These are just a few examples of the methods employed to address flaky tests effectively. Each approach has its advantages and limitations, and the choice of mitigation strategy may vary depending on the specific context and requirements of the software project.

Table 1: Techniques for correcting flaky tests across main categories [31]

Categories	Type of Correction
Async wait	Add/modify waitFor
	Add/modify sleep
	Reorder execution
Concurrency	Block atomic operation
	Make it deterministic
	Change condition
	Change assertion
Test Order Dependency	Configure/change status
	Remove dependency
	Merge tests

2.2 Related work

In their study, Lam et al. [25] investigated the complete lifecycle of flaky tests across six extensive Microsoft projects, shedding light on these tests' detrimental effects on such projects. Through an analysis of the implemented solutions, the authors identified the most prevalent category of flaky tests in the examined projects as the `Async Wait` category, which encompasses tests making asynchronous calls without waiting for the call's return. Consequently, the authors confirmed that categorizations of flaky tests proposed in prior studies, as applied to open-source projects, are also applicable to the proprietary Microsoft projects under study.

Parry et al. [34] examined a wide range of studies focusing on the origins, consequences, detection, mitigation, and repair of flaky tests. Analyzing a total of 76 studies, the authors identified 18 flaky test tools as a partial outcome of their investigation. However, it's worth noting that these tools were not thoroughly analyzed according to their specific characteristics.

Gruber and Fraser [16] engaged 335 software developers and testers across various domains to gain insights into how developers perceive flaky tests, their prevalence, their impact on daily routines, and expectations from academia regarding this issue. The survey revealed that developers consider flaky tests to be a widespread and significant problem. Their primary concern is losing confidence in test results rather than the computational costs associated with rerunning unstable tests. Additionally, developers wanted plugins to identify problematic code sections directly within their Integrated Development Environments (IDEs). They also emphasized the need for more training and information regarding flaky tests.

Parry et al. [36] explored developers' experiences with flaky tests through a multi-source approach, incorporating numerical and thematic analyses of surveys with developers and an examination of threads from StackOverflow. The findings offer valuable insights into various aspects of flaky tests, including their definitions, significant impacts, common causes, and the actions taken by developers to mitigate them. Furthermore, the study provides actionable recommendations for developers and researchers to enhance understanding and address the issue of flaky tests within the software industry.

Tahir et al. [44] undertook a multivocal review of the literature on flaky tests, combining insights from academic and gray literature to offer a comprehensive understanding of the state of practice in this domain. The research summarizes existing work, detection methods, strategies for preventing and eliminating flaky tests, and their impact. Furthermore, the authors identify prevailing challenges and propose future research directions.

In summary, while several studies have explored various aspects of flaky tests, including their definitions, impacts, and developers' perceptions, none of the reviewed papers provided an in-depth analysis of flaky test detection tools. Despite addressing important facets of the issue, such as prevention, impacts, and developers' experiences, none of the studies dedicated a specific section to thoroughly examining the available tools for detecting flaky tests.

3 SYSTEMATIC MAPPING

This study aims to explore flaky test detection, mitigation, and correction tools available in both the market and literature through systematic mapping. The systematic mapping approach adopted in this work draws inspiration from the processes outlined by Petersen et al. [37] and Keele et al. [23]. Additionally, we employed the backward snowballing technique as outlined by Wohlin [47]. To achieve our objective, we have formulated the following research questions (RQs):

RQ$_1$: *What are the objectives of flaky test tools, and what techniques do they employ?* This question seeks to clarify the specific functions of each collected tool, including the flaky testing contexts in which they operate and the techniques they utilize. Furthermore, it aims to compare tools with similar functionalities.

RQ$_2$: *What are the main characteristics of flaky test tools, and which detection causes do they address?* This RQ aims to compile a comprehensive list of all the tools and their key characteristics. Additionally, it aims to identify the causes of flaky tests targeted by each tool, providing researchers and professionals in the field with an overview of current tools to facilitate their selection process.

3.1 Search strategy

For conducting the systematic mapping, we utilized the Parsif.al tool.[1] Initially, we conducted a pilot search in the IEEE and ACM digital libraries. During this manual process, publications that contained the terms *flaky test* or *flaky tests* in their title or abstract were examined to identify relevant words and terms.

Following the pilot search, we identified the following keywords: *Flaky Test*, *Flaky Tests*, *Flakiness*, *Test Tool*, and *Test Code*. These keywords were used to construct a search string for the primary search, which was applied exclusively to the title and abstract of publications to minimize the risk of false positive results.

> *Title:("flaky test" OR "flaky tests" OR "flakiness") AND Abstract:("flaky test" OR "flaky tests" OR "flakiness" OR "test tool" OR "test code")*

In addition, to IEEE and ACM, we selected other digital libraries with publications in Software Engineering and Computer Science for the primary research. The final step in this stage involved using the defined search string to gather the publications used for mapping. Afterward, the string was applied to five libraries: ACM[2], SpringerLink[3], IEEE Xplore[4], ScienceDirect[5], and Scopus[6].

3.2 Selection

Following the results obtained in the selection process, we applied four additional filtering steps during the mapping execution:

- **Step 1 - Removal of Duplicate Publications**: Initially, we removed all duplicate publications from the list;
- **Step 2 - Application of Inclusion and Exclusion Criteria**: Next, we filtered further based on defined inclusion and exclusion criteria, aiming to exclude studies irrelevant to addressing the previously defined research questions;
- **Step 3 - Reading of Filtered Articles**: Afterward, each publication underwent a comprehensive assessment to ensure its content provided relevant information for the study;
- **Step 4 - Application of the Snowballing Technique**: Finally, we implemented the Backward Snowballing technique. This involved reviewing the reference lists of collected publications, selecting papers meeting the basic inclusion and exclusion criteria, and identifying new publications that may not have been initially found [22]. By the end of this stage, we expected to reduce the number of publications, containing only those studies contributing to the subject of the work.

Library paper selection took place from February 11 to March 26, 2023. Initially, 341 papers were identified across the five libraries. During **Step 1**, 95 duplicate papers were detected and removed. After removing duplicates, 246 papers remained for analysis in the subsequent steps. In **Step 2**, we conducted a manual filtering process to identify publications that best aligned with the objectives of the proposed work. Inclusion and exclusion criteria were established for this purpose, which can be viewed in detail in Table 2.

Table 2: Criteria for inclusion and exclusion

Inclusion	The paper proposes, mentions or uses a tool for detecting flaky tests
	The abstract cites detection, mitigation or correction of the flaky test
	The abstract cites some tool for detecting flaky tests
Exclusion	Paper outside the scope of flaky test detection
	Paper of less than two pages
	Paper not in English
	Websites, books, pamphlets and gray literature
	Full text not available online
	Paper that do not cite flaky test detection tools

During this activity, the criteria were applied to the abstract and body of publications. As a result, 173 papers were removed, leaving us with 73 papers for use in the next filtering process. During **Step 3**, a more detailed reading process was conducted on the remaining 73 papers to ascertain whether they presented relevant information. Subsequently, 44 publications were removed, leaving 29 papers for further analysis. In **Step 4**, the Backward Snowballing technique was applied to the remaining 29 papers to uncover additional references not identified during the previous stages. As a result of this step, eight new papers were selected, bringing the total number of valid papers for data extraction during the systematic mapping analysis stage to 37. The execution stage started with 341 papers and concluded with a pool of 37 primary studies, comprising 29 obtained from the automated search, and another eight from the snowballing process (see Figure 1).

In Table 3, it is possible to observe the number of publications filtered by the digital library at each step of the selection process. Following the final filtering of studies, all 37 publications were reviewed to gather information about flaky test detection tools. Subsequently, data were extracted to address the research questions defined earlier. The selected papers were mapped in Table 4.

Table 3: Filtering by digital library

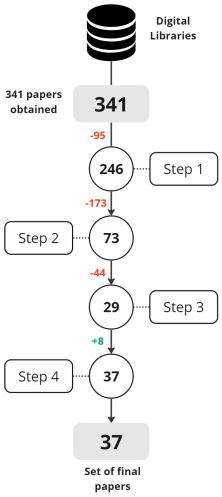
Library	Papers	Step 1	Step 2	Step 3	Step 4
ACM Library	49	10	0	0	7
SpringerLink	151	139	1	0	0
IEEE Xplore	45	5	2	1	2
ScienceDirect	3	3	1	0	0
Scopus	93	89	69	28	28
Total	**341**	**246**	**73**	**29**	**37**

[1]https://parsif.al/
[2]https://dl.acm.org/
[3]https://link.springer.com/
[4]https://ieeexplore.ieee.org/
[5]https://www.sciencedirect.com/
[6]https://www.scopus.com/

Table 4: Final set of papers

ID	Title	Reference
S1	DeFlaker: Automatically Detecting Flaky Tests	Bell et al. [5]
S2	FlakeFlagger: Predicting Flakiness Without Rerunning Tests	Alshammari et al. [2]
S3	Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker	Silva et al. [43]
S4	Detecting Flaky Tests in Probabilistic and Machine Learning Applications	Dutta et al. [10]
S5	Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications	Shi et al. [41]
S6	Know You Neighbor: Fast Static Prediction of Test Flakiness	Verdecchia et al. [45]
S7	Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests	Fatima et al. [12]
S8	Peeler: Learning to Effectively Predict Flakiness without Running Tests	Qin et al. [40]
S9	A Multi-factor Approach for Flaky Test Detection and Automated Root Cause Analysis	Ahmad et al. [1]
S10	iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests	Lam et al. [26]
S11	Practical Test Dependency Detection	Gambi et al. [14]
S12	Efficient dependency detection for safe Java test acceleration	Bell et al. [4]
S13	Empirically revisiting the test independence assumption	Zhang et al. [49]
S14	Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency	Gyori et al. [19]
S15	Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests	Parry et al. [35]
S16	Flaky Test Detection in Android via Event Order Exploration	Dong et al. [9]
S17	Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests	Huo and Clause [21]
S18	iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests	Wang et al. [46]
S19	Web Test Dependency Detection	Biagiola et al. [6]
S20	Evolution-Aware Detection of Order-Dependent Flaky Tests	Li and Shi [29]
S21	FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment	Cordy et al. [8]
S22	Unit Test Virtualization with VMVM	Bell and Kaiser [3]
S23	FlakyLoc: Flakiness Localization for Reliable Test Suites in Web Applications	Morán Barbón et al. [32]
S24	A Framework for Automated Test Mocking of Mobile Apps	Fazzini et al. [13]
S25	Root causing flaky tests in a large-scale industrial setting	Lam et al. [24]
S26	A Study on the Lifecycle of Flaky Tests	Lam et al. [25]
S27	IFixFlakies: A framework for automatically fixing order-dependent flaky tests	Shi et al. [42]
S28	Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications	Zhang et al. [48]
S29	FlakeRepro: Automated and Efficient Reproduction of Concurrency-Related Flaky Tests	Leesatapornwongsa et al. [28]
S30	Repairing order-dependent flaky tests via test generation	Li et al. [30]
S31	Shaker: A Tool for Detecting More Flaky Tests Faster	Cordeiro et al. [7]
S32	NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specifications	Gyori et al. [18]
S33	A Large-Scale Longitudinal Study of Flaky Tests	Lam et al. [27]
S34	A Survey of Flaky Tests	Parry et al. [34]
S35	Root causing, detecting, and fixing flaky tests: state of the art and future roadmap	Zolfaghari et al. [50]
S36	Static test flakiness prediction	Pontillo [39]
S37	A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests	Habchi et al. [20]

Figure 1: Filtering of papers during the execution stage



4 RESULTS AND DISCUSSIONS

4.1 *RQ$_1$: Objectives and techniques of flaky testing tools*

The first **RQ** aims to provide an overview of the tools identified in the 37 selected primary studies. This overview covers the objectives, techniques, approaches, and the study from which each tool was collected. We found 30 tools with their respective articles published between 2014 and 2022 (Table 10).

The test re-execution technique involves running the test suite one or more times to determine whether the tests pass or fail and to pinpoint any inconsistencies in the results. This approach represents a traditional method for identifying faulty tests. This study found that 25 out of 30 tools employ this technique.

The next subsections will provide an overview of various tools categorized based on their functions concerning flaky tests: detection, mitigation, and repair. Certain studies highlight tools that focus specifically on detecting flaky tests in order-dependent scenarios, a significant cause of test flakiness. Consequently, these tools will receive dedicated explanations within their own subsection.

Table 5: Flaky tests detection tools

Tool	Techniques / Approach	Study	ID
DeFlaker	Differential coverage Re-execution	Bell et al. [5]	S1
FlakeFlagger	Machine learning in detection	Alshammari et al. [2]	S2
Shaker	Stressful Execution Environment Re-execution	Silva et al. [43] Cordeiro et al. [7]	S3 S31
FLASH	Machine learning in detection Re-execution	Dutta et al. [10]	S4
NonDex	Nondeterministic specifications Re-execution	Shi et al. [41] Gyori et al. [18]	S5 S32
FLAST	Machine Learning in detection	Verdecchia et al. [45]	S6
Flakify	Applying prediction to source code	Fatima et al. [12]	S7
PEELER	Machine learning in detection	Qin et al. [40]	S8
MDFlaker	Differential coverage	Ahmad et al. [1]	S9

4.1.1 Tools and techniques to detect flaky tests. We identified nine tools for detecting flaky tests in the primary studies we analyzed, as indicated in Table 5. They are next summarized.

(1) **DeFlaker (S1):** Employs the differential coverage technique to detect flaky tests. This method involves examining the modified code based on the version control system. If the test outcome changes without covering any altered code following a new software version or commit, it flags the test as flaky.
(2) **MDFlaker (S9):** Similarly to the previous one, this tool also utilizes the differential coverage technique but with a distinguishing feature. Unlike DeFlaker, MDFlaker does not demand re-running the tests, saving testing time and costs.
(3) **FlakeFlagger (S2):** It adopts a machine learning approach for flaky test detection. It gathers dynamic and static features, such as line coverage and source code, and leverages these to predict tests likely to exhibit flakiness. This prediction is based on similarities in behavior observed across the collected features.
(4) **PEELER (S8):** It also utilizes machine learning for detection but takes a fully static approach. It collects test failure logs and compares them with the values involved in the test assertion statements to identify potential flaky tests.
(5) **FLAST (S6):** It employs a machine learning model to classify tests as flaky or not, relying solely on static resources. Its objective is to detect flaky tests before execution, emphasizing proactive identification.
(6) **FLASH (S4):** It combines test re-execution with machine learning. This tool executes tests multiple times with different random inputs, resulting in varying sequences of results. By comparing the expected values with the outcomes of these random test executions, FLASH identifies flaky tests.
(7) **NonDex (S5, S32):** It employs a unique approach to detection by exploiting non-deterministic implementations. It achieves this by randomly exploring various behaviors of underdetermined APIs during test execution. When a test fails during this exploration, NonDex identifies the specific API invocation instance responsible for the failure. For instance, consider a scenario where the iteration order of a sorting algorithm is underdetermined, and the test code assumes a specific implementation-specific iteration order. NonDex systematically explores different iteration orders, and if a test fails during this exploration, it indicates that the code relies on incorrect assumptions. This failure pinpointed by NonDex highlights potential issues stemming from reliance on implementation-specific behaviors.
(8) **Flakify (S7):** Offers a novel approach to predicting flaky test cases. It uses a black box solution, analyzing only the source code of the test cases rather than the system under test. Moreover, Flakify does not require multiple re-executions of test cases. This approach simplifies the detection process by focusing solely on the test code itself.
(9) **Shaker (S3, S31):** This tool aims to enhance the efficiency of test re-execution. It achieves this by introducing stress tasks that compete for CPU or memory resources while the test suite is re-executed. This strategy aims to increase the likelihood of revealing flaky tests caused by asynchronous waiting and concurrency issues. The rationale behind Shaker's approach lies in the observation that simultaneity is a significant source of instability in tests. By introducing stress into the testing environment, Shaker disrupts the ordering of events, potentially influencing test outcomes and revealing flakiness that might otherwise remain hidden.

4.1.2 Tools and techniques to detect flaky tests in order-dependent tests. The tools identified in Table 6 focus on detecting flaky tests in order-dependent scenarios, where factors like shared datasets or states can lead to test instability. This subsection outlines their objectives, techniques, and approaches, with all tools utilizing test re-execution as a fundamental method.

Table 6: Flaky tests detection tools in order-dependent tests

Tool	Techniques / Approach	Study	ID
iDFlakies	Repeating failed tests Re-execution	Lam et al. [26]	S10
PraDet	Dependency validation Re-execution	Gambi et al. [14]	S11
ElectricTest	Dependency validation Re-execution	Bell et al. [4]	S12
DTDetector	Dependency validation Re-execution	Zhang et al. [49]	S13
PolDet	Test pollution Re-execution	Gyori et al. [19]	S14
FLAKE16	Machine Learning Applications Re-execution	Parry et al. [35]	S15
FlakeScanner	Repeating failed tests Re-execution	Dong et al. [9]	S16
OrcalePolish	Weak assertions Re-execution	Huo and Clause [21]	S17
iPFlakies	Repeating failed tests Re-execution	Wang et al. [46]	S18
TEDD	Dependency validation Re-execution	Biagiola et al. [6]	S19
IncIDFlakies	Repeating failed tests Re-execution	Li and Shi [29]	S20

(1) **OraclePolish (S17):** It uses an approach focused on detecting fragile assertions, which are defined as assertions that depend on values derived from inputs that are not initially controlled and inputs provided by the test that are not verified by an assertion. Therefore, weak assertions can create an opportunity for order-dependent testing to occur, as they cause a test result to depend on inputs that it does not control but which can be previously defined by another test.
(2) **PraDet (S11):** It uses the same approach as the other two tools mentioned above but combines the accuracy of DTDetector and the speed of ElectricTest. It monitors the access patterns of objects in memory between test executions to identify instances of possible test order dependencies. In other words, it manages to identify the tests involved in this order dependency. Then it runs them out of order to try to discover the dependencies that are manifest, reducing the number of test runs needed to expose the manifest dependencies.
(3) **iDFlakies (S10):** It classifies tests into order-dependent or not, according to the comparison of the results of re-executions of failed tests in an order of tests that has been modified from the original order. The tool re-executes the original order of tests several times to check if the result of any test changes. This way, if any test passes and fails on the same code version in the same test order, it is considered a non-order-dependent flaky test. Failed tests are re-executed and if they fail again in the failed order and pass again in the original order, they will be considered as an order-dependent flaky test.

(4) **iPFlakies (S18):** It aims to classify and categorize order-dependent tests by repeating failed tests in projects developed in Python, unlike iDFlakies which focuses on Java projects. iPFlakies is categorized as a tool focused on detecting flaky tests in order-dependent tests and the context of repairing flaky tests since it is composed of the functionalities of the iDFlakies and iFixFlakies tools. iFixFlakies tool will be explained in subsection 4.1.4.
(5) **FlakeScanner (S16):** It uses the technique of repeating failed tests to identify concurrency flaky tests in Android applications. The technique explores possible event execution orders in failed asynchronous tests, so each test run explores a different event execution order. This way, the tool aims to detect flaky tests in a few test executions.
(6) **IncIDFlakies (S20):** It uses a technique to detect order-dependent flaky tests that have been newly introduced after a code change. Built on top of iDFlakies by detecting newly introduced order-dependent flaky tests after code changes. However, IncIDFlakies has also considered code changes since the last time IncIDFlakies was run, executing test orders that belong only to the subset of tests that can become order-dependent flaky tests after the code change.
(7) **TEDD (S19):** It aims to detect flaky tests by validating test dependencies present in interface test suites. The tool detects order-dependent tests in web applications by considering dependencies facilitated by persistent data stored on the server side, rather than internal dependencies such as PraDet, which monitors the access patterns of objects in memory.
(8) **FLAKE16 (S15):** It presents a detection technique that uses machine learning to categorize the flaky tests detected. The tool installs the project in a virtual environment and creates an instance for each run of the test suite. If the tests show inconsistent results during executions in a consistent order, they are considered non-order-dependent flaky tests. Otherwise, they are considered order-dependent flaky tests.

Table 7: Flaky tests mitigation tools

Tool	Techniques / Approach	Study	ID
FlakiMe	Automatic test generation Re-execution	Cordy et al. [8]	S21
VmVm	Automatic test generaion Order dependent tests Re-execution	Bell and Kaiser [3]	S22
FlakyLoc	UI testing Re-execution	Morán Barbón et al. [32]	S23

4.1.3 Tools and techniques to mitigate flaky tests. In addition to detection, some tools focus on mitigating flaky tests' costs and negative impacts (see Table 7).

(1) **VmVm (S22):** It presents an approach to mitigating order-dependent testing. It executes test cases in isolation, each within its own process. In this way, it manages to prevent any side-effects based on the state of each test case executed from affecting subsequent tests, eliminating test order dependencies that usually occur due to shared resources in memory. The tool reinitializes classes containing static fields that can facilitate side effects based on each test case's state. In short, the tool generates a new state for the tests at runtime, reducing the chances of a possible flaky test occurring.
(2) **FlakiMe (S21):** It aims to provide developers with ways to simulate a set of scenarios and conditions for the occurrence of flaky tests. The tool allows exceptions to be generated during the compilation of test cases so that it is possible to predict whether the test in question will fail or not, helping to predict whether the test might trigger a flaky test.
(3) **FlakyLoc (S23):** It aims to identify the root cause of failures in web applications. To do this, it re-executes tests focused on the user interface differently, based on combinatorial tests, and analyzes the test execution using different classification metrics. The tests are performed by changing the environment, such as CPU cores, amount of RAM, web browser and screen resolution.

Table 8: Flaky tests repair tools

Tool	Techniques / Approach	Study	ID
MOKA	Automatic approaches to generating or improving repairs Re-execution	Fazzini et al. [13]	S24
RootFinder	Identifying root causes to aid repair Re-execution	Lam et al. [24]	S25
FaTB	Automatic approaches to generating or improving repairs Re-execution	Lam et al. [25]	S26
iFixFlakies	Automatic approaches to generating or improving repairs Re-execution	Shi et al. [42]	S27
DexFix	Automatic approaches to generating or improving repairs Re-execution	Zhang et al. [48]	S28
iPFlakies	Automatic approaches to generating or improving repairs Re-execution	Wang et al. [46]	S18
FlakeRepro	Identifying root causes to aid repair Re-execution	Leesatapornwongsa et al. [28]	S29
ODRepair	Automatic approaches to generating or improving repairs Re-execution	Li et al. [30]	S30

4.1.4 Tools and techniques to repair flaky tests. Instead of using techniques to help mitigate flaky tests, some tools have been developed that use techniques to support developers in removing flaky tests from their project's test suite. We identified eight tools focused on repairing flaky tests, offering techniques and approaches to assist developers in removing flakiness from their project's test suite. They are listed in Table 8 and summarized next.

(1) **MOKA (S24):** It is a technique that provides automatic generation of test mocks in mobile applications. The tool collects data used in test executions. Then, it generates test simulations from this data, replacing the real interactions between the application and its environment, such as the camera, microphone, and GPS. With each re-execution, the tool adds new simulated data, helping, for example, to run the test with new inputs, which can reduce or indicate the flaky tests in the test suite.
(2) **FaTB (S26):** Aims to alleviate the negative impact of asynchronous wait tests by providing automatic fixes for developers. During test re-execution, the tool identifies the method calls in the code that are related to timeouts or waits and then calculates how often the inconsistent test might fail. Based on this frequency, FaTB will try to repair the test with

several different time values and then provide developers with the minimum time they should use.

(3) **DexFix (S28):** Intends to automatically provide corrections for implementation-dependen flaky tests. The tool was created to be used with the NonDex tool, as explained earlier in subsection 4.1.1. The input to DexFix consists of identifying the project's source code and receiving the test that has inconsistencies, which NonDex previously detected. After this process, DexFix is applied to fix the source code, and then the test must pass when NonDex is run again.

(4) **iFixFlakies (S27):** It automatically indicates to developers corrections for an order-dependent test based on the declarations of other tests in the test suite. After running the tests in isolation, they are classified according to the result obtained, the first being tests that pass even when run in isolation but fail when run with some other tests, and the second being tests that fail when run in isolation but pass when run with other tests. There is a third type of test, called auxiliary tests, where the logic of these tests can define or redefine the state of the environment in such a way that order-dependent tests pass. The tool then indicates the correction to be implemented, resulting in the tests passing even in previously failed orders.

(5) **iPFlakies (S18):** Considered a tool that acts in the context of repair, it is also composed of the functionalities of iFixFlakies, in addition to the order-dependent test detection techniques of iDFlakies. The difference is that iPFlakies is applied to Python projects, unlike iFixFlakies on Java projects.

(6) **ODRepair (S30):** Aims to repair order-dependent tests automatically, but without requiring the logic that defines or redefines the state of the environment to be implemented. Firstly, ODRepair identifies the polluted state of the environment between test executions and then automatically generates a fix that calls the methods that modify the state, redefining the environment for order-dependent tests.

(7) **RootFinder (S25):** Aims to identify the possible causes of flaky tests and then help the developer repair them. The tool analyzes the execution logs of passed and failed tests of the same test to suggest which method calls are possibly responsible for the failure. The tool takes as input the name of a method that is probably the cause of the failure and observes the method's behavior at runtime.

(8) **FlakeRepro (S29):** It also aims to help the developer repair by identifying the root cause. FlakeRepro seeks to analyze the logs with detailed messages about the error location and then reproduce the test in the same way as it failed, without relying on randomness, helping the developer to fix the error within that scenario.

Implications of RQ.1: Our findings indicate a growing interest and development of flaky testing tools, reflected by the significant increase in publications, especially in 2022. The technique of re-executing tests is predominant for the detection, mitigation and repair of flaky tests, highlighting that most tools seek to identify inconsistent tests that fail intermittently. These tools are mainly focused on detecting and repairing flaky tests already present in projects, with the aim of resolving inconsistencies that impact current quality. Therefore, the future of test tool development needs to integrate techniques that not only deal with flaky tests after they occur, but also include preventive practices to reduce the incidence of these problems from the outset.

4.2 *RQ_2: Characteristics and causes of flaky testing tools*

RQ_2 aims to provide a catalog with the highest level characteristics of flaky test detection, mitigation, and repair tools collected after analyzing the primary studies. Table 10 shows the catalog with all the collected tools and their respective characteristics. In addition to the objectives and techniques of each flaky test detection tool explained during RQ_1, there is other pertinent information that the work proposed here will provide to centralize the data on the tools collected. While analyzing the tools, we organized their characteristics into the topics of each column of Table 10, which are detailed next.

- **Tool**: The names of the collected tools.
- **Implementation Language**: This column provides the list of one or more programming languages in which the tool has been or can be implemented.
- **Analysis Language**: This column provides the programming language in which the tool was implemented and analyzed during the study.
- **Framework**: This column lists one or more test development frameworks for the tool in question.
- **Context**: This column provides the context in which the tool acts, whether detection, mitigation, or repair. Detection tools. OD was abbreviated.
- **Interface**: This column indicates how developers can interact with the tool. The options were CLI, pipeline and plugin.
- **Documentation**: This column indicates whether any external documentation about the tool is available online, be it a guide, application repository, plugin page, etc.
- **URL**: This column provides one or more links to the types of documentation for the tool if it has one.

If any column in Table 10 was not discovered during the analysis, it will be filled with UNK, indicating that it is unknown. Out of the 30 tools collected, 70% of them target Java implementations. Consequently, the most used support frameworks are JUnit[7] and Maven [8] since they were specifically built to integrate with Java. Next, the most analyzed languages were Python, with 20%, and C#, with 6.6%. Only the FlakyLoc tool did not have information about its acquired analysis language.

Twenty-two tools have some form of documentation available, often linking to external URLs such as the tool's repository. Some tools provide users with a guide, while others do not, and primary studies frequently serve as overviews of the tools. Among the collected tools, 20 focus on detecting flaky tests, with 11 specifically targeting order-dependent tests. In contrast, only three tools are designed for mitigation, and eight are geared towards repair. It is worth noting that the iPFlakies tool is categorized as both an order-dependent test detection and repair tool.

[7]https://junit.org
[8]https://maven.apache.org

Table 9 shows the causes of flaky tests each tool addresses. While tools operate within specific contexts, such as detection, order-dependent test detection, mitigation, or repair, not all tools address a single cause. For instance, detection tools generally focus on identifying flaky tests by recognizing patterns of inconsistencies. However, there are exceptions, such as the Shaker tools (S3) and NonDex (S5), which specifically tackle issues related to asynchronous wait, concurrency, and implementation dependency.

Table 9: Causes of flaky tests addressed by tools

Causes	Studies
Async Wait	S3, S16, S26
Concurrency	S3, S11, S16, S17, S22
Test Order Dependency	S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S22, S27, S30
External Data Dependency	S23, S24
Implementation Dependency	S5, S27, S28

All tools that target flaky tests in order-dependent scenarios implicitly address *Test Order Dependency*, as previously discussed, given its direct correlation with flakiness. For instance, issues arise when tests share data or states, leading to unpredictable outcomes. Out of the 30 tools analyzed, 14, or 46%, specifically tackle test order dependency. Mitigation and repair tools share a common goal: resolving flakiness by minimizing its effects, providing guidance to developers on how to resolve it, or automatically fixing the issue. While not every tool targets a specific cause, collectively, all causes are addressed by at least one tool within these contexts.

Implications of RQ.2: Our findings indicate that the Java language, with its JUnit and Maven frameworks, dominates the development of tools for detecting, mitigating and repairing flaky tests, reflecting greater support and maturity in the Java ecosystem. This implies that other languages and their development communities need to invest more in flaky testing tools to achieve the same level of support and maturity seen in Java. Furthermore, the prevalence of tools that deal with Test Order Dependency suggests that this type of cause of flaky tests is a significant concern for developers, influencing the focus of tools developed in this context.

5 DISCUSSION

The 30 flaky test detection tools extracted were categorized based on their operational context: nine detection tools, eleven order-dependent test detection tools, three mitigation tools, and eight repair tools. Some tools can operate in both order-dependent test detection and repair. However, treating flaky tests after they have already caused problems does not proactively solve the underlying problem. Therefore, the future of tool development needs to integrate techniques that include preventive practices to avoid flaky tests before they compromise software quality.

Upon analyzing these tools, we could observe that 83% of them employ the re-execution technique to aid in flaky test detection, often combined with other techniques. Re-execution emerges as a prevalent method, constituting the most commonly used technique among the sampled tools. Additionally, 70% of the tools were predominantly analyzed in projects utilizing Java, which indicates greater support for detecting and repairing flaky tests in this language. This concentration highlights a potential gap in other languages that may not have the same level of support.

Despite operating within distinct contexts — detection, order-dependent test detection, mitigation, or repair — not all tools explicitly target specific causes of flakiness. For instance, many detection tools focus on identifying flaky tests broadly without specifically addressing underlying causes. However, it is noteworthy that 46% of the tools specifically tackle the issue of test order dependency. Moreover, when considering mitigation and repair tools collectively, all causes of flakiness are addressed by at least one tool.

Out of the 30 tools examined, 18 were referenced in the related work by Parry et al. [34], emphasizing the ongoing development and exploration of tools addressing flaky tests. Notably, 10 of the 12 newly identified tools were created or published after Parry et al.'s study [34], highlighting the evolving landscape of flaky test management tools. Furthermore, our study offers detailed insights into these tools to assist developers in making informed decisions during tool selection.

6 THREATS TO VALIDITY

Threats to study selection validity: Regarding threats related to study selection, we selected studies with different strings due to the limitations of some libraries, such as Science Direct. We also filtered the article by title, abstract, and the area of Computer Science. To mitigate these threats, we used control articles for the libraries so that all strings used in the libraries returned control articles. To mitigate, we also use the backward snowballing technique.

Threats to data validity: In threats related to data extraction, we obtained a reasonable number of tools for the study. To mitigate this threat, we are based on the systematic review of Parry et al. [34], which also identifies flaky test detection tools in addition to characterizing flaky tests. However, it does not detail these tools. In our study, we identified 12 more tools than Parry et al. [34]'s study. However, some extracted tools were not available for use.

Threats to research process validity: To mitigate the threat to the research process, we use the systematic mapping guidelines proposed by Petersen et al. [37]. Three experts in software testing also participated in the mapping process, aiming to mitigate data selection and extraction bias. Furthermore, we used other systematic reviews in the area of flaky tests as a basis for identifying some flaky test detection tools [34]; however, they did not delve into the characterization of these tools.

7 CONCLUSIONS

In this study, we systematically mapped flaky test detection, mitigation, and correction tools, analyzing 37 studies and identifying 30 flaky testing tools. A notable observation is the increasing trend in tool publications over the years, with 2022 witnessing the highest number of tool releases. We found that the Re-execution technique was the most commonly used to support the detection of flaky tests. Additionally, most of the tools were analyzed in projects utilizing Java languages. While not all tools explicitly specify the type of flaky test detected, it was observed that a majority of them addressed test order dependency.

This work provides a comprehensive overview of flaky tests and the available detection tools, shedding light on their characteristics and offering insights for future studies. Moving forward, we propose several avenues for future research: (i) Implement and evaluate the

Table 10: Characteristics of flaky test tools

Tool	Implementation	Analysis	Framework	Context	Interface	Documentation	URL
MOKA	UNK	Java	JUnit Mockito	Repair	UNK	No	-
DeFlaker	Java	Java	JUnit TestNG	Detection	CLI	Yes	https://github.com/gmu-swe/deflaker
iDFlakies	Java	Java	JUnit Surefire Plugin	Detec. OD	CLI	Yes	https://github.com/UT-SE-Research/iDFlakies https://mvnrepository.com/artifact/edu.illinois.cs/idflakies
FlakeFlagger	UNK	Java	UNK	Detection	UNK	Yes	https://github.com/AlshammariA/FlakeFlagger
RootFinder	C#	C#	MsTest Maven	Repair	UNK	Yes	https://github.com/winglam/RootFinder
FaTB	UNK	Python	UNK	Repair	UNK	Yes	https://github.com/winglam/flaky-test-lifecycle-data
Shaker	Java Python	Java	Maven Pytest	Detection	CLI Pipeline	Yes	https://github.com/STAR-RG/shaker-artifacts-icsme https://star-rg.github.io/shaker/
FLASH	Python	Python	UnitTest Pytest	Detection	CLI	Yes	https://github.com/uiuc-arc/flash
PraDet	Java	Java	JUnit	Detec. OD	CLI	Yes	https://github.com/gmu-swe/pradet-replication
iFixFlakies	Java	Java	JUnit Maven	Repair	CLI Plugin	Yes	https://github.com/TestingResearchIllinois/iFixFlakies
NonDex	Java	Java	JUnit Maven	Detection	CLI Plugin	Yes	https://github.com/TestingResearchIllinois/NonDex https://mvnrepository.com/artifact/edu.illinois/nondex-maven-plugin
FlakiMe	Java	Java	JUnit Maven	Mitigation	CLI Plugin	Yes	https://github.com/serval-uni-lu/flakime https://mvnrepository.com/artifact/lu.uni.serval/flakime-maven-plugin
DexFix	Java	Java	JUnit Maven	Repair	UNK	No	-
ElectricTest	Java	Java	JUnit	Detec. OD	CLI	No	-
DTDetector	Java	Java	JUnit	Detec. OD	CLI	Yes	https://github.com/winglam/dtdetector
VmVm	Java	Java	JUnit Maven Ant	Mitigation	CLI	Yes	https://github.com/Programming-Systems-Lab/vmvm
PolDet	Java	Java	JUnit	Detec. OD	UNK	No	-
FLAKE16	Python	Python	Pytest	Detec. OD	CLI	Yes	https://github.com/flake-it/flake16-framework
FLAST	All	Python	IDE Pipeline	Detection	UNK	Yes	https://github.com/FlakinessStaticDetection/FLAST
Flakify	All	Java	Pipeline	Detection	UNK	Yes	https://github.com/uOttawa-Nanda-Lab/Flakify
FlakeScanner	Scala	Java	JUnit	Detec. OD	UNK	Yes	https://github.com/AndroidFlakyTest
FlakyLoc	UNK	UNK	UNK	Mitigation	UNK	No	-
OrcalePolish	Java	Java	JUnit	Detec. OD	CLI	No	-
iPFlakies	Python	Python	Pytest	Detec. OD Repair	CLI Plugin	Sim	https://github.com/ailen-wrx/python-ipflakies https://sites.google.com/view/ipflakies
PEELER	UNK	Java	UNK	Detection	UNK	No	https://github.com/IntHelloWorld/Peeler
TEDD	Java	Java	JUnit	Detec. OD	CLI	Yes	https://github.com/matteobiagiola/FSE19-submission-material-TEDD
FlakeRepro	C#	C#	UNK	Repair	UNK	No	-
IncIDFlakies	Java	Java	UNK	Detec. OD	UNK	No	-
ODRepair	Java	Java	JUnit	Repair	CLI	Yes	https://github.com/UT-SE-Research/ODRepair
MDFlaker	Python	Python	UNK	Detection	UNK	No	-

tools to assess their applicability in real-world projects; (ii) Conduct a more detailed technical comparison between tools operating in the same context to gather specific data and identify the most suitable tools for various scenarios; (iii) Investigate CI tools for detecting flaky tests and compare them with the tools discussed in this study; and (iv) Develop a flaky test detection tool based on one of the tools proposed in this work. These future efforts aim to further enhance our understanding of flaky tests and contribute to advancing effective detection and management strategies.

ARTIFACT AVAILABILITY

We provide our data and artifacts under open licenses at: https://zenodo.org/records/11403894.

ACKNOWLEDGEMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; CNPq grants 315840/2023-4 and 403361/2023-0; and FAPESB grantPIE0002/2022.

REFERENCES

[1] Azeem Ahmad, Francisco Gomes de Oliveira Neto, Zhixiang Shi, Kristian Sandahl, and Ola Leifler. 2021. A Multi-factor Approach for Flaky Test Detection and Automated Root Cause Analysis. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 338–348.

[2] 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)*. 1572–1584. https://doi.org/10.1109/ICSE43902.2021.00140

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

[4] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 770–781.

[5] 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)*. 433–444. https://doi.org/10.1145/3180155.3180164

[6] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web test dependency detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 154–164.

[7] Marcello Cordeiro, Denini Silva, Leopoldo Teixeira, Breno Miranda, and Marcelo d'Amorim. 2021. Shaker: a tool for detecting more flaky tests faster. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1281–1285.

[8] Maxime Cordy, Renaud Rwemalika, Adriano Franci, Mike Papadakis, and Mark Harman. 2022. Flakime: laboratory-controlled test flakiness impact assessment. In *Proceedings of the 44th International Conference on Software Engineering*. 982–994.

[9] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2021. 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*. 367–378.

[10] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning

applications. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 211–224.

[11] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 830–840. https://doi.org/10.1145/3338906.3338945

[12] Sakina Fatima, Taher A Ghaleb, and Lionel Briand. 2022. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering* (2022).

[13] Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. 2020. A framework for automated test mocking of mobile apps. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1204–1208.

[14] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–11.

[15] Martin Gruber and Gordon Fraser. 2022. A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 82–92. https://doi.org/10.1109/ICST53961.2022.00020

[16] Martin Gruber and Gordon Fraser. 2022. A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 82–92. https://doi.org/10.1109/ICST53961.2022.00020

[17] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 148–158. https://doi.org/10.1109/ICST49551.2021.00026

[18] 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.

[19] 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.

[20] Sarra Habchi, Guillaume Haben, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2022. A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 244–255. https://doi.org/10.1109/ICST53961.2022.00034

[21] Chen Huo and James Clause. 2014. 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* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 621–631. https://doi.org/10.1145/2635868.2635917

[22] Samireh Jalali and Claes Wohlin. 2012. Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. 29–38.

[23] Staffs Keele et al. 2007. Guidelines for performing systematic literature reviews in software engineering.

[24] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 101–111. https://doi.org/10.1145/3293882.3330570

[25] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1471–1482. https://doi.org/10.1145/3377811.3381749

[26] 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.

[27] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A Large-Scale Longitudinal Study of Flaky Tests. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 202 (nov 2020). https://doi.org/10.1145/3428270

[28] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. 2022. FlakeRepro: automated and efficient reproduction of concurrency-related flaky tests. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1509–1520.

[29] Chengpeng Li and August Shi. 2022. Evolution-aware detection of order-dependent flaky tests. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 114–125.

[30] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing order-dependent flaky tests via test generation. In *Proceedings of the 44th International Conference on Software Engineering*. 1881–1892.

[31] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[32] Jesús Morán Barbón, Cristian Augusto Alonso, Antonia Bertolino, Claudio A Riva Álvarez, Pablo Javier Tuya González, et al. 2020. Flakyloc: flakiness localization for reliable test suites in web applications. *Journal of Web Engineering, 2* (2020).

[33] G.J. Myers, C. Sandler, and T. Badgett. 2011. *The Art of Software Testing*. Wiley. https://books.google.com.br/books?id=GjyEFPkMCwcC

[34] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (oct 2021), 74 pages. https://doi.org/10.1145/3476105

[35] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2022. Evaluating Features for Machine Learning Detection of Order-and Non-Order-Dependent Flaky Tests. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 93–104.

[36] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. Surveying the developer experience of flaky tests. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/3510457.3513037

[37] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and software technology* 64 (2015), 1–18.

[38] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the Vocabulary of Flaky Tests?. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 492–502. https://doi.org/10.1145/3379597.3387482

[39] Valeria Pontillo. [n.d.]. Static test flakiness prediction. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 325–327.

[40] Yihao Qin, Shangwen Wang, Kui Liu, Bo Lin, Hongjun Wu, Li Li, Xiaoguang Mao, and Tegawendé F Bissyandé. 2022. PEELER: Learning to Effectively Predict Flakiness without Running Tests. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 257–268.

[41] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 80–90.

[42] 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.

[43] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–311. https://doi.org/10.1109/ICSME46990.2020.00037

[44] Amjed Tahir, Shawn Rasheed, Jens Dietrich, Negar Hashemi, and Lu Zhang. 2023. Test flakiness' causes, detection, impact and responses: A multivocal review. *Journal of Systems and Software* 206 (2023), 111837. https://doi.org/10.1016/j.jss.2023.111837

[45] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. 2021. Know you neighbor: Fast static prediction of test flakiness. *IEEE Access* 9 (2021), 76119–76134.

[46] Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: a framework for detecting and fixing python order-dependent flaky tests. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 120–124.

[47] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. Association for Computing Machinery, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[48] 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.

[49] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 385–396.

[50] 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.