Science of Computer Programming 252 (2026) 103462

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original Software Publication

JS-TOD: Detecting order-dependent flaky tests in Jest

Negar Hashemi [a,*], Amjed Tahir [a,*], Shawn Rasheed [b], August Shi [c], Rachel Blagojevic [a]

[a] Massey University, New Zealand
[b] Universal College of Learning, New Zealand
[c] The University of Texas at Austin, United States

ARTICLE INFO

Keywords:
Flaky tests
Test order dependency
JavaScript
Jest

ABSTRACT

We present JS-TOD (JavaScript Test Order-dependency Detector), a tool that can extract, reorder, and rerun Jest tests to reveal possible order-dependent test flakiness. Test order dependency is one of the leading causes of test flakiness. Ideally, each test should operate in isolation and yield consistent results no matter the sequence in which tests are run. However, in practice, test outcomes can vary depending on their execution order. JS-TOD employed a systematic approach to randomising tests, test suites, and `describe` blocks. The tool is highly customisable, as one can set the number of orders and reruns required (the default setting is 10 reorder and 10 reruns).

1. Motivation

Test flakiness is a significant issue in software testing. Flaky tests are known to impact product correctness and quality negatively [1–3]. Among the many causes of test flakiness, test order dependency is widely acknowledged as a common cause of test flakiness across multiple languages and application domains [4,5].

Listing 1 shows an example of order-dependent tests in Jest. The tests check how many times the `logger.log` function was called. The first test (‘`logger has not been called yet`’) checks that no calls have been made to pass. While the second test (‘`calls logger once`’) makes a call and expects it to be logged once. Since the mock keeps its state between tests, running the first test after the second causes the first one to fail. The mock should be cleared before each test.

There is some tooling support to automatically detect possible test order-dependent tests for Java (JUnit) [6,7] and Python (pytest) [8,9]. To the best of our knowledge, there are no similar tools for JavaScript (Jest).

Below we present our Jest test-order dependency detection approach, implemented in our tool JS-TOD.

2. Approach

2.1. Jest overview

Jest[1] is one of the most used testing frameworks in JavaScript [10,11]. In Jest, test files (test suites) are structured using blocks with the following keywords: `describe`, `test` (or `it`). Test files also utilize test hooks, such as `beforeEach` and `afterAll`, which

* Corresponding authors.
E-mail addresses: n.hashemi@massey.ac.nz (N. Hashemi), a.tahir@massey.ac.nz (A. Tahir), s.rasheed@ucol.co.nz (S. Rasheed), august@utexas.edu (A. Shi), R.V.Blagojevic@massey.ac.nz (R. Blagojevic).

[1] https://jestjs.io/

https://doi.org/10.1016/j.scico.2026.103462
Received 27 August 2025; Received in revised form 24 January 2026; Accepted 12 February 2026
Available online 18 February 2026
0167-6423/© 2026 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

```
1 test('logger has not been called yet', () => {
2   expect(logger.log).not.toHaveBeenCalled();
3 });
4
5 test('calls logger once', () => {
6   logger.log(‘Test Log‘);
7   expect(logger.log).toHaveBeenCalledTimes(1);
8 });
```

Listing 1. Order-dependent tests in Jest.

define when test fixtures and cleaning of test state should occur. These blocks help organise tests and manage setup/teardown logic. The `describe` block is used to group related tests together in the same block. By default, Jest runs test suites in parallel. It runs test blocks in the order they appear in the file (first one first). However, it also offers multiple options to change the running order of tests or only run a subset of tests. For example, the `randomize`[2] option randomises the running order of tests within a test file.

2.2. JS-TOD Implementation

Similar to tools such as iDFlakies [6], iFixFlakies [12], and FlaPy [8], JS-TOD detects order-dependent behaviour by reordering and rerunning test suites and recording the outcomes. However, unlike these existing tools, which target Java or Python and often involve additional steps such as automated classification or repair, JS-TOD is specifically designed for the JavaScript ecosystem and the Jest testing framework. It provides a lightweight, configurable, and framework-integrated solution, addressing a gap where no prior detection tool exists for JavaScript or Jest.

JS-TOD reveals order-dependent behaviour by extracting, reordering, and rerunning tests according to a user-defined number of permutations and reruns. It does not perform automated classification or repair but is intended as a diagnostic aid for developers. When failures occur in certain execution orders, JS-TOD provides the corresponding failing permutations and error logs, enabling developers to investigate and pinpoint the underlying causes manually. In practice, it is most effective when integrated into regression testing or continuous integration pipelines to expose latent order dependencies that may not appear under default configurations. Users can specify the level of reordering, the number of permutations to generate, and the number of reruns. JS-TOD also saves the newly generated test orders as separate test suites for deeper analysis. Unlike Jest's `randomize` option, which shuffles test execution order within a suite using a seed value, JS-TOD performs systematic, reproducible reordering across three levels: test suites (files), `describe` blocks, and individual test blocks. The tool is publicly available on GitHub,[3] and is also distributed as a pre-built Docker image to support reproducible execution across platforms.

Fig. 1 illustrates JS-TOD approach. For a given project with Jest test files, JS-TOD first extracts the test data of a project based on the specified reordering level. It then reorders and reruns the newly added test files for the given numbers. The result of each rerun is saved in a JSON file.

2.2.1. Extracting test data

For automatically extracting the test suites' paths, we used Jest's `-listTests` option,[4] which allows us to list all test files that Jest will run. To enable reordering of `describe` blocks or individual tests, JS-TOD utilises the Babel toolchain,[5] a JavaScript compiler and source code transformer, to parse each test suite. Babel constructs an Abstract Syntax Tree (AST) for each suite, capturing its structure, parameters, and node types. Babel defines a range of node types, and by analysing the AST, we identify all `describe` blocks and tests present in each test suite of a given project. These blocks are detected by locating AST nodes where the type is "identifier" and the name is "describe" for `describe` blocks or either "it" or "test" for `tests`.

2.2.2. Reordering tests

Using the extracted test data for a given project, JS-TOD can reorder tests based on the specified level. At the test level, JS-TOD reorders the tests within each `describe` block a given number of times and creates a new version of the test file in the same directory. The new test file retains the original filename with the reorder number appended to it.

Similarly, at the describe level, JS-TOD reorders the `describe` blocks within a file for the specified number of reorders. For each reorder, it generates a new test file named after the original, with the term describe and the reorder number appended.

At the test suite level, JS-TOD reorders the execution order of test files a given number of times and passes each permutation to `customSequencer.js`, a custom sequencer that extends Jest's built-in testSequencer.

[2] https://jestjs.io/docs/cli#--randomize
[3] https://github.com/Negar-Hashemi/JS-TOD
[4] https://jestjs.io/docs/cli#--listtests
[5] https://babel.dev

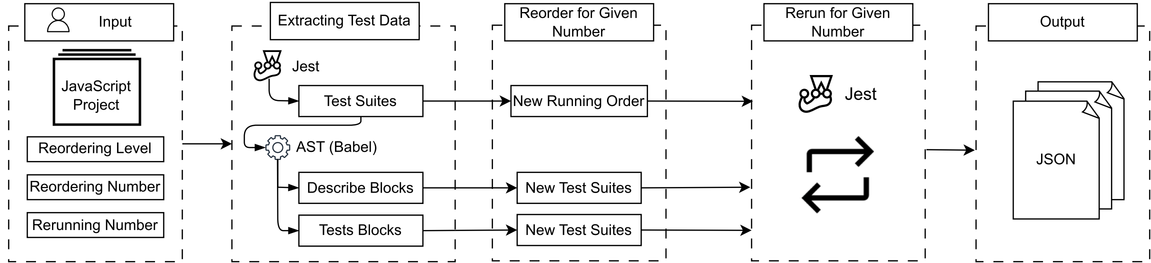

Fig. 1. JS-TOD approach.

2.2.3. Rerunning tests

After reordering the tests, `JS-TOD` reruns the tests for the specified number of times. It then saves the results in a directory under the project folder, depending on the reordering level: `_extracted results_` for tests, `_extracted results describes_` for `describe` blocks, and `_extracted results test files_` for test suites.

For test and `describe` block level reruns, the result files are named starting with *testOutput*, followed by the name of the test suite and the rerun count (e.g., for a test suite named Foo with one rerun, the output file will be `testOutputFoo1`). For test suite-level reruns, the result files are named starting with *testOutput*, followed by the reorder number and rerun count.

3. Using JS-TOD in CI/CD pipelines

To integrate `JS-TOD` into a CI/CD workflow, first navigate to the directory containing the tool:

```
cd /path/to/JS-TOD
```

Ensure that the required dependencies are installed and that the environment uses `Node.js (version` ≥ `18.17.0)` or higher.

```
npm install
```

Then, determine the desired reordering level, `test`, `describe`, `suite`, or `all` (which executes all three levels sequentially), and provide the path to the project under analysis. JS-TOD can be invoked using its main entry point, `reorderRunner.js`, as follows:

```
node reorderRunner.js \
  --project_path="/path/to/project" \
  --level="<test|describe|suite|all>" \
  --reorder=<number_of_permutations> \
  --rerun=<number_of_reruns>
```

- `project_path` defines the root directory of the target project containing the Jest test suites.
- `level` specifies the granularity of reordering. The options are `test` (individual tests), `describe` (groups of tests), `suite` (entire test files), or `all` (runs all levels in sequence). The default is `all`.
- `reorder` sets how many distinct permutations of test order should be generated and executed. The default is 10.
- `rerun` determines how many times each permutation should be executed to confirm test stability. The default is 10.

4. Evaluation

In our evaluation of `JS-TOD` using 81 open-source JavaScript projects [13], the tool's accuracy rate for returning the correct test paths was 90% (i.e., 73 out of 81 projects returned the correct test paths) and 85% for correctly reordering tests (i.e., 57 out of 67 projects were correctly reordered). By correctly reordering, we mean that `JS-TOD` recognises test blocks (including nested and individual tests) and reorders them without omitting or modifying any other parts of the test file. All reordering and rerunning steps were executed entirely within `JS-TOD`'s automated workflow.

5. Limitations

`JS-TOD` extracts test file paths using Jest's `--listTests` option, which is available starting from Jest version 20.0.0[6]. This option allows `JS-TOD` to systematically discover and process all test files in a project, regardless of their structure. However, this introduces a limitation: projects using Jest versions older than 20.0.0 do not support the `--listTests` option and, therefore, cannot use `JS-TOD` in its current form.

[6] https://github.com/jestjs/jest/releases/tag/v20.0.0

Additionally, JS-TOD depends on modern ECMAScript features and Jest's programmatic APIs (e.g., `testSequencer`), which may differ across major releases. Although this is unlikely in controlled environments, differences in Node.js or Jest versions may lead to reduced functionality or occasional execution failures. This is especially true in CI pipelines that use different base images or cached dependencies. These issues can be easily resolved by using an environment similar to the one validated in our evaluation (Node.js 18.16.1, npm 9.5.1, and Jest version 27 or higher).

Another limitation of JS-TOD is that reruns are executed sequentially. If a developer configures JS-TOD to perform many reorderings or reruns, the process may become time-consuming, particularly for large-scale projects with extensive test suites. Furthermore, the current version of JS-TOD randomizes a subset of tests, `describe` blocks or test suites when generating new execution orders. While this approach increases variability and helps uncover many order-dependent behaviors, it does not guarantee comprehensive coverage of all possible order interactions. More systematic ordering strategies, such as those based on Tuscan squares [14] or NDOD/NDOI sampling methods [15], can be used to improve the representativeness of selected test orders. Integrating such combinatorial designs in future versions of JS-TOD could enhance the detection efficiency and reduce redundant executions.

CRediT authorship contribution statement

Negar Hashemi: Writing – review & editing, Writing – original draft, Software, Methodology, Formal analysis, Data curation, Conceptualization; **Amjed Tahir:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Funding acquisition, Data curation, Conceptualization; **Shawn Rasheed:** Writing – review & editing, Validation, Investigation; **August Shi:** Writing – review & editing, Validation, Methodology; **Rachel Blagojevic:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We acknowledge the support of the New Zealand SfTI National Science Challenge grant no. MAUX2004, the US National Science Foundation grant no. CCF-2145774 and CCF-2217696, and the Jarmon Innovation Fund.

References

[1] Q. Luo, F. Hariri, L. Eloussi, D. Marinov, An empirical analysis of flaky tests, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014.
[2] A. Tahir, S. Rasheed, J. Dietrich, N. Hashemi, L. Zhang, Test flakiness' causes, detection, impact and responses: a multivocal review, J. Syst. Software 206 (2023).
[3] K. Costa, R. Ferreira, G. Pinto, M. d'Amorim, B. Miranda, Test flakiness across programming languages, IEEE Trans. Software Eng. 49, 4, (2022) 2039–2052.
[4] W. Lam, M. Hilton, A. Shi, C. Kästner, Y. Brun, DeFlaker: automatically detecting flaky tests, in: Proceedings of the 40th International Conference on Software Engineering (ICSE), 2019.
[5] M. Gruber, S. Lukasczyk, F. Kroiß, G. Fraser, An empirical study of flaky tests in Python, in: 14th IEEE Conference on Software Testing, Verification and Validation (ICST), 2021.
[6] W. Lam, R. Oei, A. Shi, D. Marinov, T. Xie, IDFlakies: a framework for detecting and partially classifying flaky tests, in: IEEE Conference on Software Testing, Validation and Verification (ICST), 2019.
[7] C. Li, A. Shi, Evolution-aware detection of order-dependent flaky tests, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2022.
[8] M. Gruber, G. Fraser, FlaPy: mining flaky python tests at scale, in: Proceedings of the IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings, 2023.
[9] R. Wang, Y. Chen, W. Lam, 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, 2022.
[10] G.A. Yost, Finding flaky tests in JavaScript applications using stress and test suite reordering, Master's thesis, The University of Texas at Austin, 2023.
[11] M. Taleb, JavaScript unit testing frameworks in 2024: a comparison · Raygun Blog, 2023, (https://raygun.com/blog/javascript-unit-testing-frameworks/). (Accessed on 09/02/2024).
[12] A. Shi, W. Lam, R. Oei, T. Xie, D. Marinov, FixFlakies: a framework for automatically fixing order-dependent flaky tests, in: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019.
[13] N. Hashemi, A. Tahir, S. Rasheed, A. Shi, R. Blagojevic, Detecting and evaluating order-Dependent flaky tests in JavaScript, in: IEEE Conference on Software Testing, Verification and Validation (ICST), 2025.
[14] C. Li, M.M. Khosravi, W. Lam, A. Shi, Systematically producing test orders to detect order-dependent flaky tests, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2023.
[15] W. Lam, S. Winter, A. Astorga, V. Stodden, D. Marinov, Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects, in: 2020IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2020.