Takuan: Using Dynamic Invariants to Debug Order-Dependent Flaky Tests

Nate Levin[1], Chengpeng Li[2], Yule Zhang[3], August Shi[2], Wing Lam[4]
[1] Yorktown High School, Arlington VA, USA
natelevindev@gmail.com
[2] The University of Texas at Austin, Austin TX, USA
{chengpengli,august}@utexas.edu
[3] University of California San Diego, La Jolla CA, USA
yuz300@ucsd.edu
[4] George Mason University, Fairfax VA, USA
winglam@gmu.edu

***Abstract*—Automated regression testing is critical to effective software development, but it suffers from flaky tests, i.e., tests that can nondeterministically pass or fail when run on the same version of code. Conceptually, a flaky test depends on a component not controlled by the test, and the test's outcome depends on the state of the component. For example, a prominent type of flaky tests is order-dependent (OD) tests, whose results depend on the unspecified order in which they are run, as a result of some other test "polluting" shared state. We propose the use of dynamic invariants to help debug flaky tests. By capturing the dynamic invariants that hold true during a passing run and comparing them against those captured during a failing run, we can isolate the reason for the flaky behavior.**

To illustrate the potential of using dynamic invariants for this task, we implement Takuan, a technique for debugging OD tests by analyzing differences in dynamic invariants collected in the passing and failing runs of OD tests. Invariants that hold true in a passing order but not in a failing order can indicate the "clean" value of the shared state that makes the test pass. To demonstrate how these invariants can be used to debug and repair OD tests, we develop automated approaches that use the invariants to search for methods that can reset the shared state back to the "clean" state. Takuan's ability to analyze polluted external shared state (e.g., in the file system) allows it to handle cases that prior work cannot. We conduct a preliminary study of Takuan on existing OD tests; Takuan provides an average runtime improvement of 88.1% over prior work while handling more OD tests.

I. Introduction

Automated regression testing helps ensure quality in software, but real-world regression tests often include *flaky tests*, i.e., tests that can pass and fail on the same version of code. Flaky test failures mislead developers on the correctness of their code changes, as their failures can occur regardless of the changes. Companies have reported development problems due to flaky tests [1]–[17]. A flaky test can both pass and fail when it depends on some component (e.g., a file in the file system) that the test does not control. Conceptually, the state of this component when the test is passing is different from its state when it is failing, and understanding these differences can help developers debug and repair flaky tests.

We propose using dynamic invariants to track state differences between passing and failing executions. Dynamic invariants are truths about data values (e.g., fields or method return values) at specific program points (e.g., a method or class) based on dynamic executions. Prior work Daikon [18] finds likely dynamic invariants by executing code with different inputs to observe what relations hold true across executions.

To help debug flaky tests, we aim to capture and compare passing run invariants to failing run invariants. We are looking for two sets of invariants for the same data source at the same program point where one set holds true only during passing runs while the other set holds true only during failing runs. We refer to such invariants as *problem invariants*.

We implement *Takuan* [19], a tool that identifies problem invariants for *order-dependent (OD) flaky tests* in Java projects. OD tests are a prominent type of flaky tests whose pass/fail results depend on the order in which they are run [20]–[24], where the order is not controlled by the test. An OD test fails when another test runs before and "pollutes" shared state that the OD test depends upon. The passing and failing runs for an OD test are tied to specific orders, so Takuan captures and compares passing-run and failing-run dynamic invariants by running these specific orders.

We evaluate Takuan on 13 OD tests from 13 projects. Takuan generates correct problem invariants for six of the 13 tests. We further demonstrate how to use problem invariants by developing an automated approach to discover or generate cleaners that can be used to repair OD tests, successfully finding cleaners for five of the six tests. Prior work [25] that similarly searches for the information needed to repair OD tests works only for three of the six OD tests, while taking longer to execute. Our preliminary results indicate that problem invariants can help debug and repair more OD tests while being faster than prior work. The use of invariants represents a promising direction towards a general solution to debugging and fixing flaky tests.

II. Takuan Approach

In this section, we provide some background and related work on OD tests and then explain how *Takuan* [19] generates problem invariants for such tests.

A. Background and Related Work

Shi et al. [26] previously referred to a test as a *polluter*, if that test "pollutes" the shared state and makes an OD test fail. The corresponding OD test that fails is referred to as a *victim*. The victim fails in an order where it runs after the polluter; we refer to this order as the *polluter-victim* order, representing the failing execution for the OD test. Meanwhile, the victim passes in the order where it runs on its own, termed the *victim-only* order, representing the passing execution for the OD test. We classify the pollution caused by the polluter as either: 1) *internal pollution*, where the polluted state is within a test's runtime environment, i.e., heap memory reached from static field(s), or 2) *external pollution*, where the polluted state is outside a test's runtime environment, e.g., a polluted file or database. Zhang et al. [23] previously found in their study on 96 OD tests that 39% of them were due to internal pollution.

In addition, Shi et al. [26] observed the presence of *cleaner*s, which are tests that, when run between a polluter and victim, will "clean" the shared polluted state, thereby allowing the victim to pass. Cleaners work by calling some *cleaner method*s that directly clean the polluted state (e.g., setting a polluted static field to the correct value).

Shi et al. developed iFixFlakies [26] to automatically repair the two types of OD tests. iFixFlakies is guaranteed to repair all *brittles* (tests that fail in isolation but pass when run with some other test), though such tests represented only 10% of OD tests in Shi et al.'s dataset. For the other type of tests (victims), iFixFlakies requires that there is a cleaner in the existing test suite, which may not always exist. Additionally, iFixFlakies depends on randomized execution orderings of the test suite to find cleaners, which can lead to suboptimal variable performance. Li et al. later proposed ODRepair [25], which generates cleaners by first searching for the static fields that lead to internal pollution and then finding potential cleaner methods of these fields statically using heuristics. ODRepair uses a test generation technique, Randoop [27], to generate cleaners. ODRepair could repair only 43% of the evaluated OD tests, because it can handle only internal pollution.

Our analysis of differences to debug flaky tests is similar to prior work that compared execution traces for flaky tests [13], [28]–[30], neural networks [31], and failure explanation [32]. Unlike prior work, Takuan uses dynamic and problem invariants from just one passing and one failing execution to output information that can be used to debug and repair OD tests.

B. Problem Invariants for OD Tests

Intuitively, one can find the source of a victim's pollution by finding differences in the state of the passing and failing executions, i.e., running in victim-only and polluter-victim orders, respectively. We can model the state during the executions by using dynamic invariants collected during each execution. A dynamic invariant is an invariant observed to always hold true during executions, expressed with values of the state captured during executions. We develop Takuan to capture and compare the dynamic invariants of passing and failing executions.

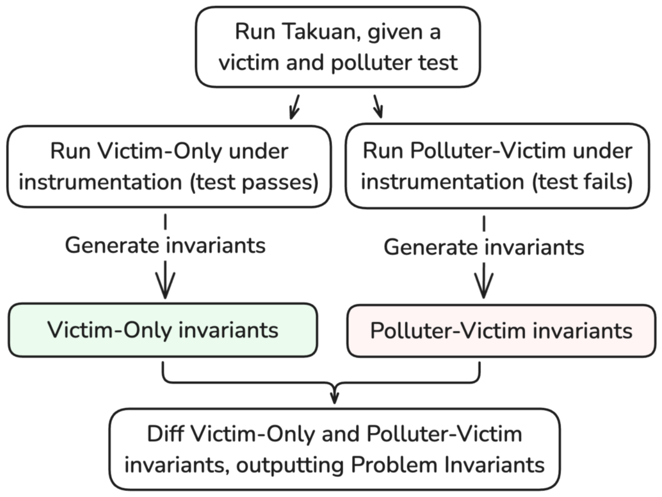
Run Takuan, given a victim and polluter test
Run Victim-Only under instrumentation (test passes)
Run Polluter-Victim under instrumentation (test fails)
Generate invariants
Generate invariants
Victim-Only invariants
Polluter-Victim invariants
Diff Victim-Only and Polluter-Victim invariants, outputting Problem Invariants

Fig. 1. Overview of the Takuan algorithm.

Figure 1 shows a high-level illustration of Takuan. First, Takuan instruments the code to collect information about the test passing and failing. We build this instrumentation using Daikon [18], a tool for collecting likely dynamic invariants. At every program point, Daikon can collect the properties pertaining to the point's relevant values. For example, at each program point referencing a static field, Daikon captures the value of that static field and outputs "statements" relating to that field's value, e.g., whether its dynamic type is `null` and how its value compares to other fields. Takuan uses Daikon to generate such statements related to static field values or method return values. This process creates two invariant lists, which we then diff: one for the passing victim-only run and the other for the failing polluter-victim run.

For each program point executed in both the victim-only and polluter-victim runs, Takuan first removes any invariants at that point that are true in both executions, as such invariants did not depend on the result of the test and cannot be used to determine why the test is flaky. Next, Takuan groups invariants by their source (the field or method return values), then removes any groups with invariants originating from only the passing or only the failing run. Each remaining group is referred to as a problem invariant, which consists of a list of invariants uniquely true only in the passing execution and a list of invariants uniquely true only in the failing execution. Daikon may generate multiple invariants for a particular program point (e.g., `[(a == m), (a != null)]`, where `m` is an arbitrary variable) from a single program execution, but the invariants should not overlap. After processing all groups, Takuan sorts the problem invariants to bring those with the largest combined lists of invariants to the top. Takuan then reports the top problem invariants (we choose the top five for our evaluation).

III. Using Problem Invariants

To demonstrate the use of Takuan's problem invariants, we develop new approaches for finding problem invariants to debug or repair OD tests. We show the use of polluted static

```
1 def on_method_start(method, fields):
2   for field in fields:
3     start_vals[method+field.name] = field.value
4 def on_method_end(method, fields):
5   for field in fields:
6     start_val = start_vals[method+field.name]
7     if victim_invs_match(field.value)
8        and polluter_invs_match(start_val):
9       cleaners.add(method)
```

Fig. 2. Pseudocode for how Takuan detects static field cleaners.

```
1 @Test public void
2 createDirectoryManagerNoConstructor() {
3   DirectoryManagerFactory.dmClass
4     = TestDirectoryManager.class; /*...*/
5 }
6 @Test
7 public void createDefaultDirectoryManagerPath() {
8   Path path=Path.of(System.getProperty("user.dir"));
9   DirectoryManager dm = DirectoryManagerFactory
10    .createDirectoryManager(path, true); /*...*/
11 }
```

Fig. 3. Simplified polluter and victim test from Wikidata-Toolkit [33]

field problem invariants to discover existing cleaners and the use of return value problem invariants to generate cleaners.

A. *Cleaners from Polluted Static Field Problem Invariants*

Given problem invariants related to static fields, we first run the polluter and then the other tests under instrumentation to collect more information. As shown in Figure 2, at the start of every method execution (`on_method_start`), for any static field referenced in the given problem invariants, we record the field's value into a map (Line 3). At the end of every method execution (`on_method_end`), for any static field referenced in the problem invariants, we detect a cleaner if 1) the field's value at method exit matches the invariants seen only in the victim-only run, and 2) the field's value at the start of the method matches the invariants seen only in the polluter-victim run (lines 7-8). This check ensures that the current method correctly cleaned the invariant value. Unlike iFixFlakies, which finds only cleaner tests that contain some lines to clean the pollution, our process finds both cleaner tests and the exact method that cleans the pollution.

Example. Figure 3 illustrates an example of internal pollution from static fields in the Wikidata-Toolkit project [33]. The polluter is `createDirectoryManagerNoConstructor`. When the victim, `createDefaultDirectoryManager-Path`, runs, it implicitly assumes the static field `dmClass` is an instance of `DirectoryManagerImpl`. However, the polluter changes `dmClass` to be an instance of `Test-DirectoryManager` instead.

When run on this polluter-victim pair, Takuan outputs the problem invariants shown in Figure 4. We observe that, while the static field `dmClass` was always an instance of `DirectoryManagerImpl` in the victim-only order (lines 5 and 6), this field can be an instance of `TestDirectory-Manager` in the polluter-victim order. Our automated approach uses this problem invariant to identify both a cleaner test and cleaner method that directly resets this shared state.

```
1 DirectoryManagerFactory:::OBJECT
2   pv> dmClass.getType() one of {
3  "DirectoryManagerFactoryTest$TestDirectoryManager",
4  "DirectoryManagerImpl" }
5   .v> dmClass dmClass.getType()
6       == "DirectoryManagerImpl"
```

Fig. 4. Problem invariants outputted by Takuan for the polluter-victim pair in Figure 3. `pv>` shows the invariants for the polluter-victim run, while `v>` shows the invariants for the victim-only run.

```
1 method_to_ret = {}
2 def on_method_enter(self, method, polutd_method):
3   method_to_ret[method] = self.call(polutd_method)
4 def on_method_exit(self, method, polutd_method):
5   exit_value = self.call(polutd_method)
6   if method_to_ret[method] != exit_value:
7     use_randoop_to_find_if_cleaner(method)
```

Fig. 5. Return value cleaner detection instrumentation.

ODRepair [25] does not generate a cleaner test (or method) for this example, because the cleaner requires a user-defined type. While iFixFlakies [26] finds a cleaner test for this polluter-victim pair, our approach finds the same test 8.79x faster in the average case and 13.49x faster in the best case.

B. *Cleaners from Return Value Problem Invariants*

When the problem invariant indicates a polluted return value, i.e., a method returns different values between the polluter-victim and victim-only runs, we call the method a *polluted method*. We aim to find *return value modifiers*, which are methods that can modify the return value of a polluted method, to generate a cleaner test with these methods. To find return value modifiers, we instrument each polluted method in the given problem invariants by adding a call to the polluted method at the start and end of any method in the polluted method's class as shown in Figure 5. Specifically, at the start of every method execution (`on_method_start`), we record what the return value of the polluted method is for the method that is about to be executed (Line 3). At the end of every method execution (`on_method_end`), we again record the return value of the polluted method for the method that just executed (Line 5) and check if the return value of a polluted method changed from the start of the method (Line 6). If the polluted method's return value has changed, then we mark the method that just executed as a return value modifier. After collecting the return value modifiers, we use Randoop [27] to generate potential cleaner tests, targeting those return value modifiers. As ODRepair [25] generates potential cleaner tests using static fields, which deals with only internal pollution related OD tests, our approach can help deal with internal and external pollution related OD tests that are affected by return values of polluted methods.

Example. We illustrate using an artificial example representing standard database operations, shown in Figure 6. The `UserQuery` class contains three methods to modify a database: `createUser`, `deleteUser`, and `countUsers`. The polluter, `testCreateUser`, creates a user account in the database, asserting that the new number of users is 1. The victim, `testDeleteUser`, creates and deletes a user

```
// UserQuery.createUser(String name): Add user
// UserQuery.deleteUser(String name): Delete user
// UserQuery.countUsers(): Count number of users
public class UserQueryTest {
  @Test public void testCreateUser() { // polluter
    UserQuery.createUser("BobPolluter");
    assertEquals(UserQuery.countUsers(), 1);
  }
  @Test public void testDeleteUser() { // victim
    UserQuery.createUser("AliceVictim");
    UserQuery.deleteUser("AliceVictim");
    assertEquals(UserQuery.countUsers(), 0);
  }
}
```

Fig. 6. The `UserQueryTest` class.

```
com.example.UserQuery.countUsers():::EXIT
    pv> return == 1
    .v> return == 0
```

Fig. 7. Problem invariants outputted by Takuan for the polluter-victim pair in Figure 6. `pv>` shows the invariants for the polluter-victim run, while `v>` shows the invariants for the victim-only run.

account and then checks whether the number of accounts is zero. The victim therefore implicitly assumes that the database has no users before starting. However, the polluter breaks this assumption by adding a user without clearing the table. This database table is external to Java heap memory, meaning other approaches, like ODRepair, that analyze only heap memory do not work. Meanwhile, iFixFlakies cannot find cleaner tests, simply because the test suite does not contain any. Figure 7 shows the output of running Takuan to find problem invariants in this situation. It shows that the return value of the polluted method, `countUsers()`, was always 0 in the victim-only order, while it was always 1 in the polluter-victim order. We then apply an automated test generation tool, Randoop [27], to find return value modifiers that can repair the OD test. Takuan finds that `deleteUser` is a return value modifier, and it can change the polluted method, `countUsers()`, to return 0 in the polluter-victim order, thereby allowing the victim to pass.

IV. Preliminary Results

We randomly selected one victim from each of the 22 projects used in Wei et al.'s prior work [34], [35]. Due to compatibility issues with Daikon, we are unable to run five victims. Three other victims had out of memory issues. Finally, one victim comes from a project that no longer compiles. Takuan detects at least one valid problem invariant for six of the remaining 13 victims. To reduce Takuan's performance overhead, we configure it to instrument only the module in which the victim is in. Our manual inspection finds that Takuan may not detect valid problem invariants for all 13 victims, in part because some victims' pollution is in a different module than the victim. For the six victims with problem invariants, we then use Takuan to find cleaners for five of them, following the approach in Section III-A.

Table I shows the preliminary results of using Takuan and the automated cleaner finding approaches on our evaluation OD tests. The "Takuan" column shows the average time Takuan takes to generate invariants and find problem invariants

TABLE I
Runtime in seconds when finding cleaners on a M1 MacBook Pro with 16GB RAM and 8 cores.

Project	Takuan	Cleaner-F	Total	ODRepair	Imp. %
http-request	10.1	2.4	12.6	107.9	88.4
marine-api	24.6	2.4	27.0	101.8	73.5
wikidata-toolkit	1.4	1.2	2.6	-	-
cukes	8.9	1.2	10.1	-	-
openpojo	2.0	9.0	10.9	107.5	89.8
Average	**9.4**	**3.2**	**12.6**	**105.7**	**88.1**

for a given victim and corresponding polluter. The "Cleaner-F" column represents the time to find the cleaners given the problem invariants, while "Total" shows the combined, total time. The column "ODRepair" shows the average time for ODRepair to run for each victim. The last column shows the percentage improvement of Takuan's runtime over ODRepair.

The average total runtime of Takuan is 12.6 seconds. The average time to find the first cleaner using ODRepair is 105.7 seconds, giving Takuan an average performance improvement of 88.1%. Takuan also resolves two additional OD tests that ODRepair did not, due to the high time cost that ODRepair needs and a deserialization issue with ODRepair.

V. Conclusion and Future Work

We proposed Takuan [19] to help debug and repair OD tests by identifying problem invariants. Our preliminary results support our intuition that problem invariants can be used to discover the source of pollution, regardless of whether it is internal (e.g., static fields) or external (e.g., a database). We also show how automated approaches can use problem invariants to detect and create cleaners for OD tests, allowing for automatic repair and an improved debugging experience.

We find that the limitations of Takuan include the performance cost of dynamic invariant generation, project compatibility with Daikon, and the existence of noise in the output of likely invariants. We plan to address these limitations in future work by optimizing invariant generation, exploring the use of other invariant generation tools or other types of dynamic specifications, and utilizing different invariant filtering strategies. Further, we believe the high-level ideas behind Takuan are extensible to other categories of flaky tests, beyond OD tests [20], [30], [36]–[38], as long as we can extract information from passing and failing runs. Problem invariants appear likely to be effective in addressing problems of other flaky tests that have yet to receive as much attention in the community. In future research, we plan to extend the Takuan approach to work for additional types of flaky tests, including extensions to other programming languages (e.g. C++). Following these extensions, we plan to conduct a large-scale evaluation of the technique as a basis for future work.

Acknowledgements

Nate Levin's and Yule Zhang's work was carried out while they were at George Mason University. We acknowledge support from NSF grants nos. CCF-2145774, CCF-2217696, and CCF-2338287, as well as Dragon Testing and the Jarmon Innovation Fund.

REFERENCES

[1] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, “Modeling and ranking flaky tests at Apple,” in *ICSE SEIP 2020: Proceedings of the 42nd International Conference on Software Engineering, Software Engineering in Practice Track*, 2020.

[2] J. Malm, A. Causevic, B. Lisper, and S. Eldh, “Automated analysis of flakiness-mitigating delays,” in *AST 2020: Proceedings of the 1st Workshop on Automation of Software Test*, 2020.

[3] M. H. U. Rehman and P. C. Rigby, “Quantifying no-fault-found test failures to prioritize inspection of flaky tests at Ericsson,” in *ESEC/FSE Industry 2021: Proceedings of the 15th Joint Meeting on Foundations of Software Engineering, Industry Track*, 2021.

[4] “Facebook testing and verification request for proposals 2019,” 2024, https://research.fb.com/programs/research-awards/proposals/facebook-testing-and-verification-request-for-proposals-2019.

[5] M. Harman and P. O’Hearn, “From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis,” in *SCAM 2018: Proceedings of the 18th International Working Conference on Source Code Analysis and Manipulation*, 2018.

[6] Google, “TotT: Avoiding flakey tests,” 2024, http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html.

[7] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, “Taming Google-scale continuous testing,” in *ICSE SEIP 2017: Proceedings of the 39th International Conference on Software Engineering, Software Engineering in Practice Track*, 2017.

[8] J. Micco, “The state of continuous integration testing at Google,” in *ICST 2017: Proceedings of the 10th International Conference on Software Testing, Verification and Validation*, 2017.

[9] C. Ziftci and J. Reardon, “Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale,” in *ICSE SEIP 2017: Proceedings of the 39th International Conference on Software Engineering, Software Engineering in Practice Track*, 2017.

[10] H. Jiang, X. Li, Z. Yang, and J. Xuan, “What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing,” in *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering*, 2017.

[11] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, “The art of testing less without sacrificing quality,” in *ICSE 2015: Proceedings of the 37th International Conference on Software Engineering*, 2015.

[12] K. Herzig and N. Nagappan, “Empirically detecting false test alarms using association rules,” in *ICSE 2015: Proceedings of the 37th International Conference on Software Engineering*, 2015.

[13] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, “Root causing flaky tests in a large-scale industrial setting,” in *ISSTA 2019: Proceedings of the 2019 International Symposium on Software Testing and Analysis*, 2019.

[14] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, “A study on the lifecycle of flaky tests,” in *ICSE 2020: Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[15] T. Leesatapornwongsa, X. Ren, and S. Nath, “FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests,” in *ESEC/FSE 2022: Proceedings of the 16th Joint Meeting on Foundations of Software Engineering*, 2022.

[16] “Test verification,” 2024, https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification.

[17] 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 *ESEC/FSE 2018: Proceedings of the 12th Joint Meeting on Foundations of Software Engineering*, 2018.

[18] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants,” *Science of Computer Programming*, 2007.

[19] “Takuan: Using dynamic invariants to debug order-dependent flaky tests,” 2024, https://sites.google.com/view/takuan-od.

[20] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, 2014.

[21] M. Barboni, A. Bertolino, and G. D. Angelis, “What we talk about when we talk about software test flakiness,” *Communications in Computer and Information Science*, 2021.

[22] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, “iDFlakies: A framework for detecting and partially classifying flaky tests,” in *ICST 2019: Proceedings of the 12th International Conference on Software Testing, Verification and Validation*, 2019.

[23] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, “Empirically revisiting the test independence assumption,” in *ISSTA 2014: Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[24] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, “An empirical study of flaky tests in Python,” in *ICST 2021: Proceedings of the 14th International Conference on Software Testing, Verification and Validation*, 2021.

[25] C. Li, C. Zhu, W. Wang, and A. Shi, “Repairing order-dependent flaky tests via test generation,” in *ICSE 2022: Proceedings of the 44th International Conference on Software Engineering*, 2022.

[26] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, “iFixFlakies: A framework for automatically fixing order-dependent flaky tests,” in *ESEC/FSE 2019: Proceedings of the 13th Joint Meeting on Foundations of Software Engineering*, 2019.

[27] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *ICSE 2007: Proceedings of the 29th International Conference on Software Engineering*, 2007.

[28] A. Ahmad, E. N. Held, O. Leifler, and K. Sandahl, “Identifying randomness related flaky tests through divergence and execution tracing,” in *ICSTW 2022: Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2022.

[29] C. Ziftci and D. Cavalcanti, “De-Flake your tests : Automatically locating root causes of flaky tests in code at Google,” in *ICSME 2020: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution*, 2020.

[30] J. Morán, C. Augusto, A. Bertolino, C. de la Riva, and J. Tuya, “Debugging flaky tests on web applications,” in *WEBIST 2019: Proceedings of the International Conference on Web Information Systems and Technologies*, 2019.

[31] G. Tao, S. Ma, Y. Liu, Q. Xu, and X. Zhang, “TRADER: Trace divergence analysis and embedding regulation for debugging recurrent neural networks,” in *ICSE 2020: Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[32] W. N. Sumner and X. Zhang, “Comparative causality: Explaining the differences between executions,” in *ICSE 2013: Proceedings of the 35th International Conference on Software Engineering*, 2013.

[33] “Wikidata/Wikidata-Toolkit on GitHub,” 2024, https://github.com/Wikidata/Wikidata-Toolkit.

[34] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, “Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests,” in *TACAS 2021: Tools and Algorithms for the Construction and Analysis of Systems*, 2021.

[35] “Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests - tools and dataset,” 2024, https://sites.google.com/view/tuscan-squares-probabilities.

[36] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, “An empirical analysis of UI-based flaky tests,” in *ICSE 2021: Proceedings of the 43rd International Conference on Software Engineering*, 2021.

[37] A. M. Memon and M. B. Cohen, “Automated testing of GUI applications: Models, tools, and controlling flakiness,” in *ICSE 2013: Proceedings of the 35th International Conference on Software Engineering*, 2013.

[38] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, “Making system user interactive tests repeatable: When and what should we control?” in *ICSE 2015: Proceedings of the 37th International Conference on Software Engineering*, 2015.