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

***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 code, where the test's outcome depends on the state of that component. For example, one prominent category of flaky tests are order-dependent (OD) tests, whose outcomes depend on the order in which they are run (where the order is not guaranteed), 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 execution of the flaky test and comparing them against those captured during a failing execution, 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 between passing and failing runs for the OD tests. The invariants that hold true in a passing order but not in a failing order indicate the "clean" value of the shared state that makes the test pass. We further illustrate how these invariants can be used to even repair OD tests by developing automated approaches that use the invariants as inputs to then search for methods that can reset the shared state back to the desired "clean" state. Takuan's ability to analyze polluted shared state that is external to the program (e.g., in the file system) allows it to handle cases that prior work could not. We conduct a preliminary study of Takuan on existing OD tests and find that our approach has promising results.

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, since their failures still occur regardless of the changes. Companies like Apple [1], Ericsson [2], [3], Facebook [4], [5], Google [6]–[9], Huawei [10], Microsoft [11]–[15], and Mozilla [16], [17] have reported development problems due to flaky tests. A flaky test can both pass and fail when it depends on some component (e.g., a file in the file system) being correctly controlled, but incorrect control of this component leads to the nondeterministic outcomes. Conceptually, the state of this component when the test is passing is different from its state when it is failing, and understanding these state differences can help developers both debug and repair flaky tests.

We propose using dynamic invariants to track these types of state differences between passing and failing executions. A dynamic invariant is an invariant that is observed to always hold true based on dynamic executions. For example, Daikon [18] finds likely dynamic invariants by executing code using different inputs to observe what relations between data holds true across all executions. We want to capture invariants during a passing execution and compare against invariants captured during a failing execution. We are interested in invariants that hold true when the test passes but not when the test fails. We term invariants relevant to the differing test results from two test executions as *problem invariants*, which can help developers better debug and fix the flaky test.

We demonstrate our idea using our technique *Takuan*, which identifies problem invariants specifically for *order-dependent (OD) flaky tests*. OD tests are a prominent type of flaky tests whose pass and fail outcomes depend on the order in which they are run [19]–[23], where the order is not guaranteed. An OD test fails when another test "pollutes" some shared global state that the OD test depends upon. The passing and failing executions for an OD test are tied to specific orders, so Takuan can capture and compare dynamic invariants when running the OD test in the different passing and failing orders.

In this setting, dynamic invariants are truths made about data values at specific program points (e.g., a specific method or class). A problem invariant consists of two sets of invariants involving the same data source (e.g., field) at the same program point but holding different truths: one set for invariants that hold true during passing and the other for failing. Developers can then use these Takuan-generated problem invariants to debug and ultimately repair the OD test. Our analysis of differences to debug flaky tests is similar to prior work that compared execution traces for flaky tests [13], [24]–[26], neural networks [27], and failure explanation [28].

We implement Takuan [29] for Java projects running JUnit [30], using Daikon [18] to obtain dynamic invariants. We evaluate Takuan on 22 projects containing OD tests, on one OD test per project. Takuan generates correct problem invariants for six of the 13 tests that successfully compiled and ran under instrumentation. We further demonstrate how problem invariants can be utilized by developing an automated approach that uses them to repair OD tests. Our approach successfully repairs five of these six tests. Another prior work [31] that similarly searches for the information needed to repair OD tests only works for three of the six OD tests while taking longer to execute. Our preliminary results indicate that generation of problem invariants can help repair more OD tests and repair them faster than prior work. Using invariants represents a promising direction towards a general solution to debugging and fixing flaky tests.

II. Takuan Approach

We introduce *Takuan*, an approach to generate problem invariants for OD tests. We provide some background and related work on OD tests and then explain how to generate problem invariants specifically for these tests.

A. Background and Related Work

Shi et al. [32] previously termed a test that "pollutes" the shared state for an OD test, making it fail, as a *polluter*, while the corresponding OD test that fails is a *victim*. The victim fails in an order where it runs after the polluter; we term this order as the *polluter-victim* order, representing the failing execution for the flaky 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 flaky 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. [22] previously found in their study on 96 OD tests that 39% were due to internal pollution.

In addition, Shi et al. observed the presence of *cleaner test*s, which are tests that, when run between a polluter and victim, "clean" shared polluted state, allowing the victim to pass. Cleaner tests work by calling some *cleaner method*s that directly clean the polluted state (e.g., setting a polluted static field to the correct initial value). Shi et al. developed iFixFlakies [32] to use these cleaner tests to automatically repair OD tests. However, iFixFlakies's requires there being a cleaner test in the existing test suite, which is not generally guaranteed. Li et al. later proposed ODRepair [31], which generates the cleaner tests by first finding the static fields that lead to internal pollution and then finding the cleaner methods for those static fields. ODRepair then generates the cleaner tests by leveraging test generation technique Randoop [33] to target the found cleaner methods. ODRepair could only repair 43% of the evaluated OD tests, in part because it could only handle internal pollution [31].

B. Problem Invariants for OD Tests

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

Figure 1 shows a high-level illustration of the Takuan approach. First, we instrument the code to collect information about the test execution for an order. We build this instrumentation on top of Daikon [18], a tool for collecting likely dynamic invariants. At every program point, Daikon allows us to collect the properties pertaining to the point's relevant values. For example, at each program point in a class with a static field, we capture the value of that static field and output several "statements" relating to that field's value, such as the dynamic type of that field, whether it is `null`, how it compares to other field values, and more. Daikon generates a wide variety of such statements at each program point, but we only consider those related to static field values or method return values, as these are most likely to be related to either internal or external polluted state, respectively. We then create two list of invariants: one for the passing victim-only run, and the other for the failing polluter-victim run.

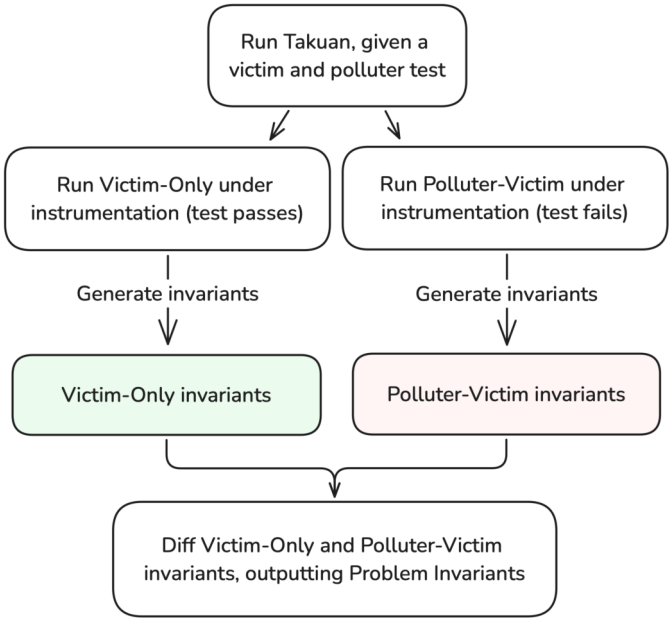

Fig. 1. Overview of the Takuan algorithm.

For each program point executed in both the victim-only and polluter-victim runs, Takuan first removes any invariants that exist at that program point in both executions, as invariants that held true regardless of test pass/fail status cannot be related to why the test is flaky. Next, Takuan removes all invariants that do not have a matching source (field or method return value) in the other execution, eliminating invariants that cannot give us enough information to have both a "cleaned" (from victim-only) and "polluted" (from polluter-victim) value. The remaining invariants at that program point therefore only exist in one execution but not the other, so Takuan groups them by source and constructs a problem invariant consisting of two lists: one containing invariants that only exist in the passing execution and the other containing invariants that only exist in the failing execution. Once Takuan finishes iterating through all program points, it then sorts the problem invariants per program point to bring those with the most information to the top, i.e., the problem invariants with the largest invariant lists in both executions. Takuan reports the top problem invariants from this sorting (we choose the top five for evaluation). Figures 4 and 7 show examples of problem invariant output.

```
1 def find_polluted_field_cleaner(p_invs, polluter):
2     run_polluter()
3     instrument(execute(polluter, get_all_tests()))
4 def on_method_start(method, fields):
5     for field in fields:
6         start_vals[method+field.name] = field.value
7 def on_method_end(method, fields):
8     for field in fields:
9         start_val = start_vals[method+field.name]
10        if victim_invs_match(field.value)
11           and polluter_invs_match(start_val):
12            cleaners.add(method)
```

Fig. 2. Static field cleaner detection algorithm.

```
1 @Test public void
2 createDirectoryManagerNoConstructor() {
3   DirectoryManagerFactory.setDirectoryManagerClass(
4     TestDirectoryManager.class); /*...*/
5 }
6 @Test public void
7   createDefaultDirectoryManagerPath() {/*...*/
8   DirectoryManager dm = DirectoryManagerFactory
9     .createDirectoryManager(path, true); /*...*/
10 }
```

Fig. 3. Shortened polluter and victim test from Wikidata-Toolkit [34]

III. Using Problem Invariants

To demonstrate the use of Takuan's problem invariants, we develop new approaches for finding cleaners that can leverage given problem invariants to repair OD tests.

A. Cleaners from Polluted Static Fields

Given problem invariants related to static fields, we first run the polluter and then the other tests under instrumentation to collect more information (Figure 2). At the start of every method execution, for any static field referenced in the given problem invariants, we record the field's value into a map (`start_values` on Line 6). At the end of every method execution, 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 *uniquely* in the victim-only run, and 2) if the field's value at the start of the method does not match the victim-only invariants (Lines 10-11). This check ensures that the current method correctly cleaned the invariant value. Unlike iFixFlakies that only outputs cleaner tests, this process additionally finds exact cleaner methods.

Example. Figure 3 illustrates an example of internal pollution from static fields, as seen in the Wikidata-Toolkit project [34]. The polluter is `createDirectoryManagerNoConstructor`. When the victim, `createDefaultDirectoryManagerPath`, runs, it implicitly assumes the static field `dmClass` is an instance of `DirectoryManagerImpl`. However, the polluter changes `dmClass` to be an instance of `TestDirectoryManager` instead.

When run on this polluter-victim pair, Takuan outputs the problem invariants shown in Figure 4. The invariants indicate the problem immediately; while the static field `dmClass` was always an instance of `DirectoryManagerImpl` in the victim-only order, the static field is instead an instance of `TestDirectoryManager` in the polluter-victim order. Afterwards, our automated approach uses this problem invariant to identify both a cleaner test and cleaner method that 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 output by Takuan for the Wikidata-Toolkit polluter-victim pair.

```
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.

Because the cleaner method requires a user-defined type, ODRepair [31] is unable to generate a cleaner test for this example. While iFixFlakies [32] finds a cleaner test for this polluter-victim pair, our approach finds the same cleaner test 2.96x faster than iFixFlakies in the average case, and 9.23x faster in the best case.

B. Cleaners from Return Values

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 modifier*s to generate a cleaner method by focusing on the problem invariants related to method return values. As shown in Figure 5, we first run the polluter and then run all other tests under instrumentation. For each polluted method in the given problem invariants, our instrumentation adds a call to the polluted method at the start and end of any methods in the polluted method's class. When the instrumentation finds that the return value of a polluted method changed from the method's start to its end (Line 6), that method is marked as a return value modifier. After collecting the return value modifiers, we then use Randoop [33] to generate potential cleaner tests, targeting those return value modifiers, in the same manner as ODRepair [31]. Unlike ODRepair, this approach can detect external pollution by checking the return values of methods that interact with this external state.

Example. Figure 6 shows an example of an OD test caused by external pollution. The `UserQuery` class contains three methods to modify a database: `createUser`, `deleteUser`, and `countUsers`. The polluter, `testCreateUser`, creates

```
// 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 `UserQuery` and `UserQueryTest` classes.

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

Fig. 7. Problem invariants output by Takuan for `UserQueryTest`'s polluter-victim pair.

a user account in the database, asserting that the new number of users is 1. The victim, `testDeleteUser`, creates and deletes a user account in the `users` table and then checks for success by asserting the number of accounts to be 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 only analyze heap memory cannot apply. Meanwhile, iFixFlakies is also unable to 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 `countUsers()` was always 0 in the victim-only order, while it was always 1 in the polluter-victim order. By tracking invariants centered around API call return values, we can indirectly identify the polluted shared state. We the apply the automated approach at finding the cleaner method to repair the OD test. Our approach finds that the method `deleteUser` can remove a created user from the table if called with the right input, allowing the victim to pass.

IV. Preliminary Results

We randomly selected one victim from each of the 22 projects from Wei et al.'s prior work on OD tests [35], [36]. Due to bugs in Daikon's instrumentation implementation, we cannot run Daikon on five victims, and we cannot run on three victims due to running out of memory when using Daikon. Finally, one victim come from a project that can no longer compile. Takuan detects a valid problem invariant for six of the remaining 13 victims. For these six victims, we utilize our automated approach that leverages problem invariants to find cleaners and construct a patch for five of them.

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

TABLE I
Runtime in seconds when finding cleaners

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 x 5	**9.4**	**3.2**	**12.6**	**105.7**	**84.1**

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 84.1%. Takuan also resolves two additional OD tests that ODRepair cannot, as a result of previously-discussed limitations to ODRepair's approach.

V. Conclusion

We proposed Takuan to help debug and repair OD tests by identifying problem invariants, which are invariants that differ between passing and failing executions of the OD tests at the same program point. Our preliminary results support our intuition that these 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 automated approaches that can use problem invariants to detect and create cleaner tests for OD tests, allowing for automatic repair and an improved debugging experience for developers.

VI. Future Plans

We find that current 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 improving runtime performance, exploring the use of other invariant generation tools beyond Daikon or even other types of dynamic specifications, and utilizing further invariant filtering strategies. We also plan to improve the ability for invariants to capture implicit state.

Further, we believe the high-level ideas behind Takuan are extensible to other categories of flaky tests beyond OD tests [19], [26], [37]–[39], as long as we can extract passing and failing executions. 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. Following our extensions, we plan to conduct a large-scale evaluation of the technique as a basis for future research.

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."

[3] M. H. U. Rehman and P. C. Rigby, "Quantifying no-fault-found test failures to prioritize inspection of flaky tests at Ericsson."

[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: 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 2022 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 2018 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] 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.

[20] 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.

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

[22] 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.

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

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

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

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

[27] 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.

[28] 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.

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

[30] "JUnit," 2024, https://junit.org.

[31] 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.

[32] 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 2019 13th Joint Meeting on Foundations of Software Engineering*, 2019.

[33] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," 2007.

[34] "Wikidata/Wikidata-Toolkit on GitHub," 2024, https://github.com/Wikidata/Wikidata-Toolkit.

[35] 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.

[36] "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.

[37] 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.

[38] 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.

[39] 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.