

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Zhang, Lei, Mahsa Radnejad, and Andriy Miranskyy. "Identifying Flakiness in Quantum Programs". In 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 1–7, 2023. <https://doi.org/10.1109/ESEM56168.2023.10304850>.

<http://doi.org/10.1109/ESEM56168.2023.10304850>

Access to this work was provided by the University of Maryland, Baltimore County (UMBC) ScholarWorks@UMBC digital repository on the Maryland Shared Open Access (MD-SOAR) platform.

Please provide feedback

Please support the ScholarWorks@UMBC repository by emailing scholarworks-group@umbc.edu and telling us what having access to this work means to you and why it's important to you. Thank you.

Identifying Flakiness in Quantum Programs

Lei Zhang

Department of Information Systems
University of Maryland, Baltimore County
Baltimore, USA
leizhang@umbc.edu

Mahsa Radnejad

Department of Computer Engineering
Isfahan Branch, Islamic Azad University
Isfahan, Iran
radnejad@khuisf.ac.ir

Andriy Miranskyy

Department of Computer Science
Toronto Metropolitan University
Toronto, Canada
avm@torontomu.ca

Abstract—In recent years, software engineers have explored ways to assist quantum software programmers. Our goal in this paper is to continue this exploration and see if quantum software programmers deal with some problems plaguing classical programs. Specifically, we examine whether intermittently failing tests, i.e., flaky tests, affect quantum software development.

To explore flakiness, we conduct a preliminary analysis of 14 quantum software repositories. Then, we identify flaky tests and categorize their causes and methods of fixing them.

We found flaky tests in 12 out of 14 quantum software repositories. In these 12 repositories, the lower boundary of the percentage of flaky tests ranges between 0.26% and 1.85% per repository. We identified 38 distinct flaky tests with 10 groups of causes and 7 common solutions. Further, we notice that quantum programmers are not using some of the recent flaky test countermeasures developed by software engineers.

This work may interest practitioners, as it provides useful insight into the resolution of flaky tests in quantum programs. Researchers may also find the paper helpful as it offers quantitative data on flaky tests in quantum software and points to new research opportunities.

I. INTRODUCTION

A test running on the same code sometimes produces different results, i.e., it shows “passed” sometimes and “failed” other times. Such tests are called flaky tests. Flaky tests can negatively impact developers by providing misleading signals. One can view flaky tests as bugs of testing that produce a non-deterministic result. Such tests consume significant resources. At Google, in 2014, 73K out of 1.6M (4.56%) of test failures were caused by flaky tests [1]; in 2017, 1.5% of 4.2M tests were flaky [2], [3].

A flaky test can be caused by two factors: either non-determinism in the source code or the test itself [1], [4]–[7]. Quantum programs are inherently non-deterministic. The randomness comes from a variety of sources. For example, it can be caused by the physical properties of quantum systems (e.g., quantum indeterminacy [8, Ch. 1]) or hardware issues (e.g., measurement errors or networking problems) or both (e.g., quantum decoherence [9, Ch. 7]). In a simulation of a quantum computer on a classical computer, pseudo-random number generators (PNGs) are used to emulate these sources of randomness. Randomness from all these sources leads to a distribution of output values, which may result in flaky tests.

To the best of our knowledge, no one has assessed the prevalence and type of flaky tests in quantum software. Thus, we aim to do an initial analysis of flakiness and its root causes

in quantum programs. To achieve this, we will seek answers to the following three **research questions** (RQs).

RQ1. How prevalent are flaky tests in quantum programs?

RQ2. What causes flakiness in quantum software?

RQ3. How do quantum programmers fix flaky tests?

The paper makes two major **contributions**. *First*, we perform an initial study of flaky tests in quantum programs. Specifically, we investigate code and bug tracking repositories of 14 quantum software; 12 out of 14 software have at least one flaky test (38 unique flaky tests in total). We estimate that at least 0.26% to 1.85% of issue reports in these 12 software are related to flaky tests. *Second*, we identify and categorize 10 groups of causes for flakiness in quantum software and 7 common fixes. We find that the most common cause of flakiness is assert statements in testing, and the most common fix is to increase the tolerance of assertions. Moreover, quantum programmers do not use some recent countermeasures [6], [7] developed by software engineers to deal with flaky tests.

II. EMPIRICAL STUDY: DATA GATHERING

To answer our RQs, we perform an empirical study focusing on open-source quantum programs on GitHub because open-source projects have transparent development histories and bug reports. We follow three steps to collect flaky test reports from open-source quantum projects.

First, we choose four popular quantum platforms (i.e., IBM Qiskit [10], Microsoft Quantum Development Kit [11], TensorFlow Quantum [12], and NetKet [13]) and identify 14 repositories with active contributions and bug reports. Despite not being an exhaustive list of all quantum software, these platforms represent some of the most active and popular open-source quantum ecosystems.

Second, we search for six keywords: “flaky”, “flakiness”, “flakey”, “occasion”, “occasional”, and “intermit”, in all the closed reports of those repositories. We focus only on closed reports because they have been verified by developers.

Finally, the first two authors examine the closed reports containing those six keywords and their corresponding commits and pull requests to determine if it is a real report of a flaky test. We then filter out two repositories without a verified flaky test (namely, `qiskit-finance` and `qiskit-optimization`) and end up with 12 repositories.

Based on our manual examination, the common case of a flaky test report consists of an issue report with a pull request

TABLE I: Statistics of quantum software repositories with flaky tests. Three right-most columns are as follows: count of closed issue reports (Column T), count of closed flaky test reports (Column F), and percentage of reports related to flaky tests (Column P) computed as $F/T \times 100\%$.

Platform	Repository	Language	T	F	P
Qiskit	qiskit-terra	Python	2,810	19	0.68%
Qiskit	qiskit-aer	Python	558	2	0.36%
Qiskit	qiskit-nature	Python	287	2	0.70%
Qiskit	qiskit-experiments	Python	255	1	0.39%
Qiskit	qiskit-ibm-runtime	Python	217	2	0.92%
Qiskit	qiskit-ibm-provider	Python	184	2	1.09%
Qiskit	qiskit-machine-learning	Python	378	1	0.26%
Microsoft	qdk-python	Python	64	1	1.56%
Microsoft	QuantumLibraries	Q#	136	1	0.74%
Microsoft	Quantum	Q#	108	2	1.85%
TensorFlow	quantum	Python	192	1	0.52%
NetKet	netket	Python	295	4	1.36%
Total			5,484	38	

that fixes the flakiness. However, there are three other cases: 1) multiple pull requests are related to a single report of flaky tests, e.g., a backport pull request, which we consider as one flaky test report; 2) a pull request that resolves flakiness without an issue report; and 3) a closed flakiness issue report without an associated pull request, e.g., a flaky test that is resolved for unknown reasons. For simplicity, we call all four cases above “flaky test reports”.

III. ANALYSIS AND RESULTS

In this section, we seek answers to our three RQs by analyzing the obtained flaky test reports.

A. RQ1: How prevalent are flaky tests in quantum programs?

Table I shows the statistics of the quantum program repositories and the flaky test reports that we detect. The data was last updated on January 12, 2023; thus, the statistics may change in the future.

As shown in Table I, we detect 38 flaky tests in the 12 repositories. Among all the repositories, the core Qiskit component `qiskit-terra` has the most flakiness reports (i.e., 19). The average percentage of flakiness varies from 0.26% (`qiskit-machine-learning`) to 1.85% (Microsoft/Quantum). Since our list of keywords is not exhaustive, the flakiness percentages represent a lower bound on the number of flaky tests (see further discussion in Section IV). In other words, there could be more flaky tests than what we have observed.

B. RQ2: What causes flakiness in quantum software?

RQ2 is answered by manually analyzing all flaky test reports, categorizing them, and listing the summary in Table II. The table shows that two of the most common causes of flaky tests are “assertions” and “random number seeds”. We detect two flakiness reports containing issues and commits related simultaneously to assertions and random number seeds. Therefore, the total number of flaky test reports in Table II is

TABLE II: Cause categories of flaky tests.

Cause Category	Count of Flaky Test Reports	Percentages
Assertion	9	22.5%
Random Number Seed	8	20.0%
Software Environment	6	15.0%
Multi-Threading	3	7.5%
Visualization	2	5.0%
Unhandled Exception	2	5.0%
Network	1	2.5%
Unordered Collection	1	2.5%
Others	6	15.0%
Unknown	2	5.0%
Total	40	100%

40 instead of 38 (the total number of flaky test reports in Table I).

Luo et al. [1] identified¹ 10 causes of flakiness in classical computer software: `async wait`, `concurrency`, `test order dependency`, `resource leak`, `network`, `time`, `IO`, `randomness`, `floating point operations`, and `unordered collections`. We have four categories in common with theirs: `concurrency`, `network`, `randomness`, and `unordered collections`.

We will now examine the details of the 10 categories of causes listed in Table II. Due to space constraints, we provide complete examples for two of the most common causes and brief summaries for the remaining eight groups.

1) *Assertion*: An assert statement is an expression that encapsulates some testable logic specified about a target (e.g., a variable) under test. Due to the non-deterministic nature of quantum programs, we observe that there is always a non-zero probability that an assertion may not cover all the outputs, even when testing a simple quantum circuit with two qubits (see Listing 1). Hence, developers need to adopt an assert statement that takes the random nature into account. Given the difficulty for developers to consider all the possibilities, it is not surprising that assertion is the major cause of flaky tests. In our study, we observe nine (22.5%) assertion-related flaky test reports.

```

1 self.ansatz = RealAmplitudes(num_qubits=2, reps
2 =2)
3 ...
4 def test_run_with_shots_option(self, backend):
5     est = BackendEstimator(backend=backend)
6     result = est.run([self.ansatz],
7 ...
8 ).result()
9     np.testing.assert_allclose(result.values,
    [-1.307397243478641], rtol=0.05)

```

Listing 1: An example of an assertion-related flaky test.

For example, Listing 1 shows a testing code snippet in `qiskit-terra` (pull request #8820). The backend estimator is tested with a two-qubit quantum circuit. Line 9 will raise an `AssertionError` if the two objects are not equal up to desired

¹Most flaky tests in [1] are related to Java projects and distributed systems (e.g., Apache HBase and Hadoop). Concurrency and `async waits` are the two main causes of flakiness in their study.

tolerance, which is set by a relative tolerance `rtol`. However, due to the indeterminacy, the difference between the two objects is sometimes greater than the tolerance, and developers observe a flaky test. We will see how this flaky test can be fixed in Section III-C1.

2) *Random Number Seed*: We define a flaky test as random-number-seed-related if a change in a random seed value for PNGs changes the test output.

For example, Listing 2, based on issue report #5217 in `qiskit-terra`, shows such a defect. The logic of the test `test_append_circuit`, verifying if appending quantum circuits works fine, is correct. However, Lines 3 and 6 cause flakiness in the test because the function `random_circuit` generates a random quantum circuit using a randomly selected seed (by default). Thus, the test fails occasionally because of randomness. We will discuss how to fix it in Section III-C2.

```
1 def test_append_circuit(self, num_qubits):
2     ...
3     first_circuit = random_circuit(num_qubits[0],
4                                   depth)
5     ...
6     for num in num_qubits[1:]:
7         circuit = random_circuit(num, depth)
8     ...
```

Listing 2: An example of a random-number-seed-related flaky test.

3) *Software Environment*: This category includes flaky tests caused by specific software or library dependency issues. For example, pull request #47 in `Microsoft/Quantum` discusses a flaky test caused by a timeout in a continuous integration pipeline.

4) *Multi-Threading*: This category of flaky tests is caused by multi-threading issues, e.g., concurrency and overload. As an example, issue report #5904 in `qiskit-terra` describes a flaky test caused by address collisions due to parallel builds.

5) *Visualization*: This group of flaky tests is related to image generation. For example, `qiskit-terra` has a test manager that schedules visual tests sequentially and allows these tests to communicate with one another. Issue #3283 reports a flaky test in the visualizer of the test manager due to out-of-date reference indexes.

6) *Unhandled Exception*: This group of flaky tests is caused by the code that does not appropriately handle exceptions. For example, `Microsoft/QuantumLibraries` issue report #398 illustrates a flaky test that is triggered by unexpected negative coefficient values that are chosen randomly.

7) *Network*: A flaky test in this category occurs as a result of network-related issues, such as an unstable network or server. For example, issue #584 of `qiskit-ibm-runtime` reports a flaky test due to timeouts and socket connection problems.

8) *Unordered Collection*: This is a category of flaky tests in Python. Dictionaries (hash maps or hash tables) are implemented as an unordered collection from Python 3.3 to 3.7 [5]. When tests have order dependencies (e.g., pull request

TABLE III: Common fix patterns of flaky tests.

Cause Category	Fix Category
Assertion	Increase Tolerance
Random Number Seed	Fixed Seed
Software Environment	Alter Software Environment
Multi-Threading	Single Thread
Unhandled Exception	Add Exception Handler
Network	Synchronization
Unordered Collection	Use Keys for Order

#8627 in `qiskit-terra`), test outcomes can become non-deterministic, which leads to flakiness.

9) *Others*: Flaky test reports are included in this category if there is only one observation of the cause or the cause has not been classified in previous studies (e.g., [1]). Examples include 1) typos in tests (issue report #62 in `Microsoft/Quantum`) or 2) keeping local outputs in Jupyter Notebook and preventing flaky tests from being executed in the continuous integration pipeline (pull request #453 in `TensorFlow/quantum`).

10) *Unknown*: A flaky test cause is classified as unknown if there is insufficient information about its cause or fix. For example, a flaky test stack trace is provided in issue report #185 of `qiskit-machine-learning`. The report is later closed because of insufficient information.

C. RQ3: How do quantum programmers fix flaky tests?

Table III shows seven fix patterns for the majority of flaky tests that we found (based on the flakiness-related commits and pull requests). Due to the page limit, we provide two concrete examples (corresponding to the two examples in Section III-B) and brief summaries for the remaining five patterns. For the “visualization” category, we do not have a fix pattern because we only observe flaky tests in `qiskit-terra`, and they are dependent on visualization software, such as `Graphviz`.

1) *Increase Tolerance*: Loosening the thresholds for assert statements helps to avoid assertion-related flaky tests. Listing 3 shows a fix where developers increase the relative tolerance `rtol` from 0.05 to 0.1 to remove the flakiness in Listing 1. In case an assertion has no tolerance arguments, e.g., `assertEqual` in pull request #9023 in `qiskit-terra`, it can be replaced with an approximate assert statement (e.g., `assertAlmostEqual`). Listing 3 uses hard-coded increased tolerance to avoid flakiness, which is not optimal. Dynamic tolerance is a better solution: e.g., pull request #1147 uses an average error as a tolerance.

```
1 - np.testing.assert_allclose(result.values,
2   [-1.307397243478641], rtol=0.05)
3 + np.testing.assert_allclose(result.values,
4   [-1.307397243478641], rtol=0.1)
```

Listing 3: An example of tolerance increase for Listing 1.

2) *Fixed Seed*: It is easier to compare a variable with a constant when a PNG’s seed is fixed. To avoid flakiness in Listing 2, Listing 4 describes a solution to replace the default random seed with a fixed seed (pull request #5599 in `qiskit-terra`).

```

1 - first_circuit = random_circuit(num_qubits[0],
   depth)
2 + first_circuit = random_circuit(num_qubits[0],
   depth, seed=4200)
3 ...
4 - circuit = random_circuit(num, depth)
5 + circuit = random_circuit(num, depth, seed=4200)

```

Listing 4: An example of fixed seed for Listing 2.

While flakiness can be controlled by fixing seeds for the PNGs, this approach can potentially make testing less effective because it limits the possible executions that can expose real bugs [7]. Further, a change in the PNG algorithm may make the test case flaky again.

We have not seen robust fixes to this issue that involve running the test case multiple times to compute the distribution of successful and failed executions, then performing distribution analysis to assess whether new changes to the code alter the distribution (see [7] for details). This could be due to higher computation and execution time requirements. Additionally, programmers may not want to build such a test framework from scratch and do not realize that test frameworks like this, e.g., FLEX [7], already exist.

3) *Alter Software Environment*: Flaky tests can be removed by updating or changing the development environment. In the six observed flaky test reports related to software environment (see Table II), three of them are fixed by upgrading or changing the dependencies, one (i.e., issue #319 in Microsoft/qdk-python) is fixed by changing the configuration of the continuous integration pipeline, and one (issue #1466 in qiskit-aer) is closed because of dependency changes. The remaining pull request #1369 in netket is fixed not by altering the environment but by simplifying the test case.

4) *Single Thread*: Multi-threading-related flaky tests can be resolved by limiting the number of threads to one. Two of the three multi-threading-related flaky tests that we observe were resolved by setting a single thread. For example, in pull request #780 of qiskit-experiments, developers observe occasional timeout issues when running multiple tests because multi-threading is disabled in certain environments. Though the flakiness may be resolved by disabling multi-threading, performance overheads may arise.

5) *Add Exception Handler*: Flaky tests caused by unhandled exceptions can be mitigated by adding an exception handler or removing the exception. For example, developers add an if condition to remove any unexpected negative coefficients in pull request #399 of Microsoft/QuantumLibraries.

6) *Synchronization*: We observe one network-related flaky test report, i.e., issue #584 and pull request #588 in qiskit-ibm-runtime. Here, tests for callback functions are dependent on socket tests. However, sometimes those callback tests finish before the socket tests, which causes flakiness. Developers increase callback test iterations so that socket tests have sufficient time to finish first. This may be a suboptimal solution as the number of iterations is hard-coded.

The ideal solution would be to synchronize callback tests with the completion of socket tests.

7) *Use Keys for Order*: For flaky tests caused by unordered Python dictionaries, developers can use key values for ordering instead of using the insertion order. As for Python 3.8, dictionaries are order-preserved, so an upgrade of the Python environment can solve the problem. Python upgrades can, however, cause dependency issues.

IV. THREATS TO VALIDITY

Validity threats are classified according to [14], [15].

Internal and construct validity. Data harvesting and cleaning are error-prone processes. The data were collected manually. Several co-authors independently examined the search results of flaky tests, then jointly reviewed and discussed the findings to finalize the list. The six keywords used to select the issue reports (and shown in Section II) are not exhaustive². Therefore, we underestimate the number of reports related to flaky tests. Yet, even this limited set of keywords proves that flaky tests exist in quantum computer programs.

External and conclusion validity. Generally, software engineering studies suffer from real-world variability, and the generalization problem can only be solved partially [16]. We need to generalize a theoretical population and understand its architectural similarity relation to build a theory [16]. Although 14 open-source projects are used in this study, our findings may not generalize to other projects. We intended to perform a pilot study of flaky tests in programs for quantum computers and find out what causes flakiness and how to fix it. The same empirical examination can be conducted on other software products using well-designed and controlled experiments. Through such future research, we hope the community will expand our taxonomy of causes and identify patterns that will eventually lead to a general theory of flaky tests in quantum computing.

V. RELATED WORK

The literature on testing and debugging quantum programs is growing. A quantum program is challenging to test because of the underlying principles of quantum mechanics [17]. Software engineering principles are being applied to quantum program testing and debugging [17], [18]; see [19] for a comprehensive overview of quantum software engineering research work.

The community takes multiple approaches to tackle the challenge. Testing quantum programs may be simplified by adding assertion checks to the code [20]–[23] or, in some cases, introducing debugging tricks, such as extracting classical information [18]. The identification of bug patterns in quantum programs can assist in defect analysis and categorization [24]. We can also adapt classical fuzzy testing techniques [25] or perform property-based testing [26]. We

²In the future, we will expand the list of keywords. For example, the search for “random” results in 479 closed reports in qiskit-terra, which will be inspected manually to determine whether the reports are related to flakiness.

can also debug quantum programs in a simulator on a classical computer (frameworks, such as Qiskit and Q# readily provide such an option), but they can be used only for small problems [17], [27].

As far as we know, there has been no study of flaky tests in quantum programs. As discussed in Section III-B, Luo et al. [1] summarize 10 common causes of flakiness in software for classical computers; four of them overlap with ours. Similar taxonomies have been presented by [4]–[6], which are complementary to ours.

VI. CONCLUSIONS

This paper examines flakiness in 14 quantum programs. In 12 of these programs, at least 0.26% to 1.85% of issue reports relate to flaky tests. Additionally, we identify 10 groups of causes for flaky tests and 7 common fixes. The final observation is that quantum programmers are not using recent software engineering techniques developed to deal with flaky tests. We hope that these findings will assist researchers and developers in mitigating the risk of flakiness when designing and testing quantum programs.

In the future, the number of keywords mapped to flaky issue reports will be expanded, making our estimates less conservative. Additionally, we will study more quantum programs.

REFERENCES

- [1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 643–653.
- [2] J. Micco, “The state of continuous integration testing @google,” 2017. [Online]. Available: <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45880.pdf>
- [3] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, “Taming google-scale continuous testing,” in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 233–242.
- [4] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, “A survey of flaky tests,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–74, 2021.
- [5] M. Gruber, S. Lukaszczuk, F. Kroiß, and G. Fraser, “An empirical study of flaky tests in python,” in *Proceedings of the 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 148–158.
- [6] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, “Detecting flaky tests in probabilistic and machine learning applications,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 211–224.
- [7] S. Dutta, A. Shi, and S. Misailovic, “Flex: fixing flaky tests in machine learning projects by updating assertion bounds,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 603–614.
- [8] D. C. Marinescu, *Classical and quantum information*. Academic Press, 2011.
- [9] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge Univ. Press, 2010.
- [10] M. Treinish, J. Gambetta et al., “Qiskit/qiskit: Qiskit 0.39.5,” Jan. 2023. [Online]. Available: <https://doi.org/10.5281/zenodo.7545230>
- [11] Microsoft, “Q# and the quantum development kit,” 2023. [Online]. Available: <https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing>
- [12] TensorFlow, “TensorFlow quantum,” 2023. [Online]. Available: <https://www.tensorflow.org/quantum>
- [13] NetKet, “NetKet - the machine learning toolbox for quantum physics,” 2023. [Online]. Available: <https://www.netket.org/>
- [14] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, ser. Computer Science. Springer Berlin Heidelberg, 2012.
- [15] R. Yin, *Case Study Research: Design and Methods*, ser. Applied Social Research Methods. SAGE Publications, 2009.
- [16] R. J. Wieringa and M. Daneva, “Six strategies for generalizing software engineering theories,” *Science of computer programming*, vol. 101, pp. 136–152, 4 2015.
- [17] A. Miranskyy and L. Zhang, “On testing quantum programs,” in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2019, pp. 57–60.
- [18] A. Miranskyy, L. Zhang, and J. Doliskani, “Is your quantum program bug-free?” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER ’20. ACM, 2020, p. 29–32.
- [19] J. Zhao, “Quantum software engineering: Landscapes and horizons,” *arXiv preprint arXiv:2007.07047*, 2020.
- [20] Y. Huang and M. Martonosi, “Statistical assertions for validating patterns and finding bugs in quantum programs,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA’19. Association for Computing Machinery, 2019, p. 541–553.
- [21] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, “Projection-based runtime assertions for testing and debugging quantum programs,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 150:1–150:29, 2020.
- [22] J. Liu, G. T. Byrd, and H. Zhou, “Quantum circuits for dynamic runtime assertions in quantum computation,” in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS’20. Association for Computing Machinery, 2020, p. 1017–1030.
- [23] S. Ali, P. Arcaini, X. Wang, and T. Yue, “Assessing the effectiveness of input and output coverage criteria for testing quantum programs,” in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 13–23.
- [24] P. Zhao, J. Zhao, and L. Ma, “Identifying bug patterns in quantum programs,” in *Proceedings of the 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. IEEE, 2021, pp. 16–21.
- [25] J. Wang, M. Gao, Y. Jiang, J. Lou, Y. Gao, D. Zhang, and J. Sun, “Quanfuzz: Fuzz testing of quantum program,” *arXiv preprint arXiv:1810.10310*, 2018.
- [26] S. Honarvar, M. R. Mousavi, and R. Nagarajan, “Property-based testing of quantum programs in q#,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 430–435.
- [27] M. P. Usaola, “Quantum software testing,” in *Short Papers Proceedings of the 1st International Workshop on the QuANtum SoftWare Engineering & pRogramming*, ser. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 57–63.