

This item is likely protected under Title 17 of the U.S. Copyright Law. Unless on a Creative Commons license, for uses protected by Copyright Law, contact the copyright holder or the author.

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.

Automatic Diagnosis of Quantum Software Bug-Fix Motifs

Krishn V. Kher¹, M. Bharat Chandra¹, Ishan Joshi², Lei Zhang², and M. V. Panduranga Rao¹

¹Department of Computer Science and Engineering, IIT Hyderabad, India

²Department of Information Systems, University of Maryland, Baltimore County, USA

Abstract

*Bug-fix pattern detection has been investigated in the past in the context of classical software. However, while quantum software is developing rapidly, the literature is still lacking automated methods and tools to identify, analyze, and detect bug-fix patterns. To the best of our knowledge, our work is the first to leverage classical techniques to detect bug-fix patterns in quantum code. In this paper, we propose an automated framework, called *Q-Diff*, for detecting bug-fix patterns in IBM Qiskit quantum code. In the framework, we develop a proof-of-concept tool based on Abstract Syntax Trees. To validate our method, we test *Q-Diff* with a variety of quantum bug-fix patterns using examples. We hope our work will attract the attention of the quantum software engineering community to improve the quality of quantum software.*

1 Introduction

With the increasing size and complexity of quantum programs being written, it is natural to expect an increased number of bugs and more complicated bugs to creep into quantum source code. Indeed, this phenomenon is folklore in classical software [3]. Therefore, a significant body of research, tools, and techniques exist in the detection and elimination of bugs in classical software [8]. These techniques range from static code analysis [2] to run-time detection [28].

The effort to understand and classify commonly occurring bugs yields rich dividends. Steps for this approach include the identification and classification of bug patterns, the design of bug-fixes, and the detection of bug-fix patterns. Correct identification of bug-fix patterns is of immense use in statistical analysis of bugs, their prevalence, and fixes. This helps in streamlining and developing tools for automatic bug detection, fixing, and manpower training.

In this paper, we are particularly interested in automated approaches to detect bug-fix patterns in quantum programs.

In classical software engineering, common bug-fix patterns are well studied, and both manual and automated approaches have been proposed [4, 12, 18, 20]. In quantum software engineering [19], preliminary studies exist in testing of quantum programs [14, 15, 16], and bug patterns of quantum programs [6, 29, 30].

In this paper, we propose a new framework, called *Q-Diff*, for detecting quantum bug-fix patterns. We translate the framework into a tool, as a proof-of-concept. The tool uses Abstract Syntax Trees (AST) to compare the quantum buggy code and patches by extracting required information specific to bug-fix patterns (see Section 3). The detectors in the tool then positively identify or reject the bug-fix pattern. We validate this tool with three quantum bug-fix patterns in Qiskit [23] (in Section 4). In accordance with the framework, this in-house tool will be further developed, incorporating more quantum bug-fix patterns and supporting more quantum programming frameworks.

Our contributions are: 1) we develop an AST-based tool to detect bug-fix patterns in Qiskit,¹ which is, to the best of our knowledge, the first work in the literature, and 2) we show that our tool can detect at least three bug-fix patterns.

2 Related Work

In the interest of space, we assume a working knowledge of quantum computing [17]. We provide a brief review of existing bugs, fixes, and bug-fix patterns in both classical and quantum programs.

2.1 Classical Bug and Fix Patterns

Bug-fix patterns for classical programs are widely studied in the literature. Pan et al. [18] identify 27 bug-fix patterns based on an analysis of historical bug-fix pairs. Cam-

¹Our source code is publicly available at <https://github.com/KrishnKher/Q-AutoDiaBFM>.

pos and Maia [4] conduct an empirical analysis to characterize bug-fix patterns in Java open-source repositories. Soto et al. [21] leverage lessons learned in C projects and perform a large-scale study of bug-fix patterns of Java projects on GitHub.

Automated approaches for detecting certain bug-fix patterns are considered more efficient compared to manual approaches in general. Madeiral et al. [12] manually analyze hundreds of bug-fixes and propose an automated tool based on AST to detect repair patterns in bug-fixes using the GumTree algorithm. The AST method is an effective solution in terms of detecting code differences, other AST-based auto detection tools include [7, 13].

2.2 Quantum Bug and Fix Patterns

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 [14]. Software engineering principles are being applied to quantum program testing and debugging [14, 15]; see [29] 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 [1, 6, 9, 10] or, in some cases, introducing debugging tricks, such as extracting classical information [15]. We can also adapt classical fuzzy testing techniques [24] or perform property-based testing [5]. The identification of bug patterns in quantum programs can assist in defect analysis and categorization [11, 30]. Zhao et al. [31] propose a benchmark to evaluate testing and debugging methods for Qiskit programs. As the research of quantum software engineering is still in its infancy, the literature lacks automated solutions to detect bug-fix patterns in quantum programs.

3 Our Approach

Code-diffing is a technique used to compare two versions of code to identify differences or changes. This is typically done by comparing the ASTs of the buggy and the fixed code, line-by-line, and identifying added, deleted, or modified lines. Figure 1 shows the architecture of our tool *Q-Diff*. As can be seen, *Q-Diff* reads both buggy and fixed code, then determines the bug-fix pattern based on a sequence of steps, including 1) obtaining the AST, which abstracts the structured code information, 2) matching against syntactic checks using regular expression (Regex) formulas, which searches for the quantum-related syntax, and 3) performing semantic checks, to identify bug-fix patterns based on quantum code semantics.

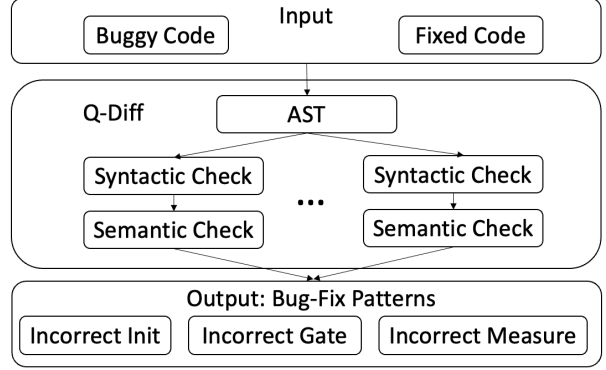


Figure 1: The architecture of *Q-Diff*. There is one detector for each of the three bug-fix patterns. This structure can be extended by adding more detectors with specific syntactic and semantic checks.

As an initial study, *Q-Diff* can detect three bug-fix patterns, i.e., incorrect initialization of qubits, incorrect gate operations, and incorrect measurements, as can be seen in Figure 1. Note that each bug-fix pattern has its own detector, and each detector has its own logic that determines whether a bug-fix code pair falls into this category or not. In other words, three detectors, corresponding to the three bug-fix patterns, have been implemented in *Q-Diff* at present. However, the sequence of logic in each detector is the same, as depicted in Figure 1. This code structure can be extended to add more detectors and to change or augment the logic of various detectors if necessary.

There are two advantages of the current implementation of multiple detectors. First, each detector can run in parallel, in principle. If a large number of bug-fix code pairs are waiting to be fed into *Q-Diff*, we do not need to input each pair sequentially. Second, there may exist hybrid bug-fix patterns where a bug-fix pair can be categorized into multiple patterns. In such a case, the detectors will not interfere with each other. For future work, we propose to adopt a hierarchical approach. Starting with a highly coarse-grained categorization like Python-related bug-fix patterns and Quantum-related bug-fix patterns in Qiskit, we descend to finer grains of categorization, ending in detectors for specific bug-fix patterns at the “leaf” level. We believe that this approach has the potential to improve search efficiency substantially whenever large categories can be pruned out.

We now discuss how our framework operates in detail.

3.1 Creation of Bug-Fix Pattern List

We identify bug-fix patterns from various sources, including the studies of [11, 31], StackOverflow, and GitHub

repositories.² Based on our initial study, we create representative examples of commonly occurring buggy and fixed code for each of the bug-fix patterns. These examples are derived from real-world examples. We then patch them up manually and pass the code pair to *Q-Diff*. For testing the detector of each bug-fix pattern, we create a group of test cases with an equal number of positive test cases (matching the pattern) and negative test cases (not matching the pattern). The detectors should correctly identify the positive cases and reject the negative cases.

3.2 AST Extraction

We first extract the ASTs for the buggy and fixed code, respectively. These ASTs will subsequently provide information to the detectors for syntactic and semantic analysis. Example information that ASTs provide includes identifiers and a number of quantum circuit objects.

3.3 Bug-Fix Pattern Detectors

The implementation of a detector varies from one bug-fix pattern to another. However, they all have an initial syntactic filter and a semantic check module.

3.3.1 Syntactic Checks with RegEx Formulations

For each bug-fix pattern, we formulate a RegEx. The RegEx is used to identify the lines of code that are relevant to a particular bug-fix pattern. If a match is found on a line of code, then we move on to the semantic check phase. When all lines of code are exhausted without finding a match, we declare that the buggy-fix code pair under investigation does not belong to the current pattern.

3.3.2 Semantic Checks

After RegEx matching, we perform semantic checks. These checks would be specific to the bug-fix pattern detector. For example, in the context of incorrect gates, we are still not sure if a line of code matched by the RegEx contains an object of quantum circuits. Thus, we analyze the information extracted from the AST in the context of Qiskit semantics to decide this.

4 Implementation of *Q-Diff*

In this section, we discuss examples of bug-fix patterns and their detection. Note that the examples described here

²Some bug-fix patterns that we manually detect can be found at <https://github.com/KrishnKher/Q-AutoDialBFM/blob/main/QSEBugFindings.xlsx>

are selected to illustrate the working of *Q-diff*. We provide more examples on our GitHub repository.

As discussed in Section 3, the bug-fix patterns that we describe here are 1) incorrect initialization, 2) incorrect gates, and 3) incorrect measurements. These three patterns correspond to the three key elements in quantum computation, i.e., qubits, operations, and measurements. We emphasize that we only consider single-line errors instead of multi-line errors. For most simple code of this nature, multi-line errors do not happen. However, *Q-Diff* can be extended in the future to identify multiple independent single-line bug-fix patterns in a buggy-fixed code pair. We now look at detectors for the three bug-fix patterns with examples.

4.1 Incorrect Initialization

Q-Diff detects two cases of incorrect initialization: 1) A gate operation is applied on a wrong qubit, i.e., it should be applied on a different qubit, as an `IncorrectInitialization` error; 2) The number of qubits that the `QuantumCircuit` works with is different in the buggy and fixed code—the so-called `IncorrectQubitCount` error.

4.1.1 Syntactic Checks

We use RegEx `.*` and `.*QuantumCircuit.*` to retrieve the lines of the code where a Qiskit gate has been used.

4.1.2 Semantic Checks

We conduct a two-step semantic check. First, we identify the lines where the same kind of gate is being used, to separate a case from a `IncorrectGate` error (here, quantum gates are used for qubit initialization). Next, we check each of these valid gates if there is any difference in the qubit indices in the bug-fix code pair. If there is any difference then we flag it as an `IncorrectInitialization` error.

4.1.3 Examples

```
1 Buggy Code:
2     qc = QuantumCircuit(2)
3     - qc.h(0)
4     ...
5 Fixed Code:
6     qc = QuantumCircuit(2)
7     + qc.h(1)
8     ...
```

Listing 1: Incorrect initialization

Listing 1 simulates a scenario where an incorrect qubit is initialized—the developer wants to initialize the second qubit instead of the first one, which is semantically different, although syntactically correct. The only error here is

in the argument to the Hadamard gate being applied. Our semantic checks identify the bug-fix and correctly classify it as an `IncorrectInitialization` error.

4.1.4 IncorrectQubitCount

The case for `IncorrectQubitCount` is very similar—there is a mismatch in the argument for the `QuantumCircuit` object, i.e. in the number of qubits with which the `QuantumCircuit` object is instantiated. This is dealt with in the same way as above; thus, we skip the details.

4.2 Incorrect Gates

Here, we show instances where an incorrect gate is applied on a certain specified qubit of a `QuantumCircuit`.

4.2.1 Syntactic Checks

The RegEx for this type of bug is `[.+.]*`, which is the same as the first one that we use for `IncorrectInit`. However, the semantic check will be different, and we will explain the details in Section 4.2.2. The RegEx abstracts lines of code involving a quantum gate. After identifying the gate operations, we will identify the name of the quantum circuits and gates for comparison.

4.2.2 Semantic Checks

After the syntax checks from RegEx, we perform additional semantic checks in two steps. First, we check 1) if the identifiers are not equal, and 2) if they both actually belong to the inbuilt gates available in Qiskit. If either of these checks fails, we declare the bug-fix pair is not of an `IncorrectGate` type. If both conditions are satisfied, *Q-Diff* classifies the bug-fix pair in this category (as shown in Listing 2). Next, we identify `QuantumCircuit` objects independent of their actual name in the code and detect errors, using the AST. Listing 3 illustrates this case.

4.2.3 Examples

```

1 Buggy Code:
2   qc = QuantumCircuit(2)
3   circuit.h(0)
4   - qc.h(1)
5   ...
6 Fixed Code:
7   qc = QuantumCircuit(2)
8   circuit.h(0)
9   + qc.x(1)
10  ...

```

Listing 2: First example of incorrect gate

Listing 2 (derived from Stack Overflow [22]) illustrates a scenario where the gate operation is incorrect in `QuantumCircuit`. The identifiers are `h` and `x` in this example, both of which correspond to valid inbuilt gates in Qiskit, namely, the Hadamard gate and the X-gate. Obviously, the identifier names, `h` and `x` are not equal. Hence, it is classified as an `IncorrectGate` kind of bug.

```

1 Buggy Code:
2 - a = QuantumCircuit(2)
3 - a.sdg(1)
4   ...
5 Fixed Code:
6 + qc = QuantumCircuit(2)
7 + qc.tdg(1)
8   ...

```

Listing 3: Second example of incorrect gate

Listing 3 is very similar to the first example, except that now the `QuantumCircuit` object has different names, i.e., `a` and `qc`. In the second example, we first check if the underlying gate is being accessed by a `QuantumCircuit` object. We do this using the data extracted from the AST of the buggy and the fixed code. This removes the dependency on the identifier name to determine if a bug-fix pair is in the `IncorrectGate` or not.

4.3 Incorrect Measurements

Here, we describe the bug-fix pattern of incorrect measurements of qubits, i.e., `IncorrectMeasurement` in *Q-Diff*.

4.3.1 Syntactic Checks

We use a different RegEx expression compared with the first two patterns, i.e., `[.+.measure.*]`. This RegEx helps us to retrieve the code where a Qiskit `measure()` function is used for further semantic checks.

4.3.2 Semantic Checks

We categorize a bug-fix pair in this class as follows.

First, we check if there are any measure functions in the bug-fix pair. If not, we declare that the bug is not of an `IncorrectMeasurement` type; otherwise, *Q-Diff* goes to the next step.

Second, there are three commonly used measure functions in Qiskit, namely, `measure()`, `measure_all()`, and `measure_inactive()`. The *Q-Diff* first computes the numbers of all variants of measure functions used in the buggy and fixed code, respectively. If the numbers are different, the bug-fix pair then falls into this category; otherwise, *Q-Diff* will go to the next step. Note that we assume multiple measure functions are in one-to-one correspondence in the buggy and fixed code.

Third, if `measure` functions are different in the buggy and fixed code, this bug-fix pair falls into this category; otherwise, *Q-Diff* will go to the next step.

Fourth, if the `measure` functions are the same, we check if the arguments passed to the `measure` functions are the same in a bug-fix code pair. Since arguments are usually qubit lists, we use Python `numpy` arrays to check the difference. If the arguments are different, then it is in the category of `IncorrectMeasurement`; otherwise, *Q-Diff* will go to the last step of the semantic check.

Finally, if both the `measure` functions and the arguments are the same, we check the positions of the measurements. If the positions are different in the buggy and fixed codes, this code pair also falls into the `IncorrectMeasurement` category.

4.3.3 Examples

```
1 Buggy Code:
2   qr = QuantumRegister(2, name='qreg')
3   cr = ClassicalRegister(2, name='creg')
4   qc = QuantumCircuit(qr, cr)
5   qc.h(qr)
6 -  qc.measure_all()
7 Fixed Code:
8   qr = QuantumRegister(2, name='qreg')
9   cr = ClassicalRegister(2, name='creg')
10  qc = QuantumCircuit(qr, cr)
11  qc.h(qr)
12 +  qc.measure(qc.qubits, qc.clbits)
```

Listing 4: First example of incorrect measurement

We show two examples here. In Listing 4 (derived from `qiskit-terra` issue report #6751), though valid `measure` functions are used in both bug and fix, `measure_all()` function shows in “bug” while `measure()` function is its replacement in “fix”. The outputs of the `measure_all()` function and the `measure()` function are not expected to be the same, hence an `IncorrectMeasurement` error.

```
1 Buggy Code:
2   qc = QuantumCircuit(3,3)
3   qc.x(0)
4   qc.barrier()
5 -  qc.measure([0,1,2],[0,1,2])
6 Fixed Code:
7   qc = QuantumCircuit(3,3)
8   qc.x(0)
9   qc.barrier()
10 +  qc.measure([1,0,2],[1,0,2])
```

Listing 5: Second example of incorrect measurement

Similar to Listing 4, although valid `measure` functions are being used in Listing 5 (issue report #664 in `qiskit-aer`), the outputs written to the classical bits are in a wrong order in the buggy code. Hence, this is an `IncorrectMeasurement` error, and *Q-Diff* identifies it successfully.

4.4 A “Counter Example” Epilogue

How would *Q-Diff* behave if there is a syntactic (or textual) difference between the buggy and the fixed code, but the difference has no semantic consequence?

```
1 Buggy Code:
2   qc = QuantumCircuit(2)
3 -  qc.h(0+1)
4   ...
5 Fixed Code:
6   qc = QuantumCircuit(2)
7 +  qc.h(1)
8   ...
```

Listing 6: Code identified as a correct implementation

In Listing 6, we notice that the underlying logic in both the code is exactly the same, but for the representation, wherein the buggy code, qubit 0 is denoted by `qubit 0 + 1`. A manual check determines this is not a bug-fix pattern and our tool does the same—it is capable of recognizing the semantics as opposed to just reporting text differences.

5 Threats to Validity

Validity threats are classified according to [26, 27]. **Internal and construct validity:** In this initial study, we test *Q-Diff* with bug-fix patterns where only one line of code is modified. We develop both positive and negative test cases to verify our implementation as proof-of-concept and will test our framework with real bug-fix code. **External and conclusion validity:** Software engineering studies suffer from the generalization problem, which can only be solved partially [25]. Although we cover the three most significant bug-fix patterns of Qiskit in quantum computations, our findings may not generalize to other projects. As a pilot study of quantum bug-fix patterns, we plan to extend our research with more bug-fix patterns and quantum frameworks. We also hope the quantum software engineering community will expand methods and tools to improve the quality of quantum software.

6 Conclusions and Future Work

In this paper, we conduct the first analysis of bug-fix patterns in quantum programs. We propose an AST-based framework called *Q-Diff* to automatically detect Qiskit bug-fix patterns and prove the concept with various examples. An obvious future direction is to enhance the tool with detectors for an exhaustive list of bug-fix patterns. Other bug-fix patterns that can be potentially detected by *Q-Diff* include but are not limited to 1) incorrect definition or application of custom gates in Qiskit and 2) unhandled exceptions in Qiskit. A second direction to pursue is the detection of more complex bugs and fixes, especially composite ones, which involve multiple bug-fix patterns and fixes

that involve multiple lines of code that are not necessarily co-located in the code. An example is the “computation in the wrong basis” bug-fix pattern. Several quantum algorithms first change the basis (say, from computational to Fourier), apply unitary gates U_i and measurements M_i , and revert (to computational basis) for further computation—the buggy code could then be $U_1 M_1 \dots$ and the fixed code is $H U_1 M_1 \dots H^{-1}$, where H is the Fourier transform. Finally, a multi-pronged approach that uses ASTs, parse trees, and lexical analysis to analyze bug-fix patterns semantically, would be a promising direction to pursue.

References

- [1] 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)*, pages 13–23. IEEE, 2021.
- [2] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
- [3] A. Barr. *Find the Bug: A Book of Incorrect Programs*. Addison-Wesley Professional, 2004.
- [4] E. C. Campos and M. de Almeida Maia. Common bug-fix patterns: A large-scale observational study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 404–413. IEEE, 2017.
- [5] S. Honarvar, M. R. Mousavi, and R. Nagarajan. Property-based testing of quantum programs in q#. In *Proc. of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 430–435, 2020.
- [6] Y. Huang and M. Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proc. of the 46th International Symposium on Computer Architecture, ISCA’19*, page 541–553. Association for Computing Machinery, 2019.
- [7] M. R. Islam and M. F. Zibran. How bugs are fixed: Exposing bug-fix patterns with edits and nesting levels. In *Proc. of the 35th annual ACM symposium on applied computing*, pages 1523–1531, 2020.
- [8] Y. Lee and J. Yang. Analysis of bug types of textbook code with open-source software. In H. R. Arabnia, L. Deligiannidis, F. G. Tinetti, and Q.-N. Tran, editors, *Advances in Software Engineering, Education, and e-Learning*, pages 629–639, Cham, 2021. Springer International Publishing.
- [9] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proc. of the ACM on Programming Languages*, 4(OOPSLA):150:1–150:29, 2020.
- [10] J. Liu, G. T. Byrd, and H. Zhou. Quantum circuits for dynamic runtime assertions in quantum computation. In *Proc. of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS’20*, page 1017–1030. Association for Computing Machinery, 2020.
- [11] J. Luo, P. Zhao, Z. Miao, S. Lan, and J. Zhao. A comprehensive study of bug fixes in quantum programs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1239–1246. IEEE, 2022.
- [12] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia. Towards an automated approach for bug fix pattern detection. In *Proc. of the VI Workshop on Software Visualization, Evolution and Maintenance (VEM)*, 2018.
- [13] M. Martinez, L. Duchien, and M. Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *2013 IEEE international conference on software maintenance*, pages 388–391. IEEE, 2013.
- [14] A. Miranskyy and L. Zhang. On testing quantum programs. In *Proc. of the 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 57–60. IEEE, 2019.
- [15] A. Miranskyy, L. Zhang, and J. Doliskani. Is your quantum program bug-free? In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER ’20*, page 29–32. ACM, 2020.
- [16] A. Miranskyy, L. Zhang, and J. Doliskani. On testing and debugging quantum software. *arXiv preprint arXiv:2103.09172*, 2021.
- [17] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge Univ. Press, 2010.
- [18] K. Pan, S. Kim, and E. J. Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14:286–315, 2009.
- [19] M. Piattini et al. The talavera manifesto for quantum software engineering and programming. In *Proc. of the 1st International Workshop on the QuANtum SoftWare Engineering & pROgramming, Talavera de la Reina, Spain, 2020*, volume 2561 of *CEUR Workshop Proceedings*, pages 1–5. CEUR-WS.org, 2020.
- [20] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *SANER 2018*, 03 2018.
- [21] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proc. of the 13th International Conference on Mining Software Repositories*, pages 512–515, 2016.
- [22] Stack Overflow. 2 entangled qubit gives all states with 25 %. <https://stackoverflow.com/questions/62661255/2-entangled-qubit-gives-all-states-with-25>, 2022.
- [23] M. Treinish, J. Gambetta, et al. Qiskit/qiskit: Qiskit 0.39.5, Jan. 2023.
- [24] J. Wang, M. Gao, Y. Jiang, J. Lou, Y. Gao, D. Zhang, and J. Sun. Qunafuzz: Fuzz testing of quantum program. *arXiv preprint arXiv:1810.10310*, 2018.
- [25] R. J. Wieringa and M. Daneva. Six strategies for generalizing software engineering theories. *Science of computer programming*, 101:136–152, 4 2015.
- [26] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer Berlin Heidelberg, 2012.
- [27] R. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 2009.
- [28] M. Young and M. Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.
- [29] J. Zhao. Quantum software engineering: Landscapes and horizons. *arXiv preprint arXiv:2007.07047*, 2020.
- [30] P. Zhao, J. Zhao, and L. Ma. Identifying bug patterns in quantum programs. In *Proc. of the 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pages 16–21. IEEE, 2021.
- [31] P. Zhao, J. Zhao, Z. Miao, and S. Lan. Bugs4q: A benchmark of real bugs for quantum programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1373–1376. IEEE, 2021.