IPSJ SIG Technical Report

Vol.2021-QS-3 No.2
2021/7/1

Bugs4Q: A Benchmark of Real Bugs for Qiskit Programs

Pengzhan Zhao[1,a) Jianjun Zhao[1,b) Zhongtao Miao[1,c) Shuhan Lan[1,d)

Abstract: Realistic benchmarks of reproducible bugs and fixes are vital to good experimental evaluation of testing and analysis approaches. Unfortunately, until now, there is no suitable benchmark suite that can be used to evaluate testing and debugging tools for quantum programs systematically. This paper proposes Bugs4Q, a benchmark of thirty real, manually validated Qiskit bugs from four popular Qiskit elements (`Terra`, `Aer`, `Ignis`, and `Aqua`), supplemented with the test cases for reproducing buggy behaviors. Bugs4Q also provides interfaces for accessing the buggy and fixed versions of the Qiskit programs and executing the corresponding test cases, facilitating the reproducible empirical studies and comparisons of Qiskit analysis and testing tools.

Keywords: Quantum software testing, quantum program debugging, bug benchmark suite, Qiskit, Bugs4Q

1. Introduction

A software bug is considered as abnormal program behavior that deviates from its specification [3], including poor performance when a threshold level of performance is included in the specification. Software bugs have a significant impact on the economy, security, and quality of life. The diagnosis and repair of software bugs consume a significant amount of time and money. An appropriate method of bug finding can quickly help developers locate and fix bugs. Many software engineering tasks, such as program analysis, debugging, and software testing, are dedicated to developing techniques and tools to find and fix bugs. Software bugs can also be handled more effectively or avoided by studying past bugs and their fixes. In general, these techniques and tools should be evaluated on real-world, up-to-date bug benchmark suites so that potential users can know how well they work. Such a benchmark suite should contain fail-pass pairs, consisting of a failed version, including a test set that exposes failures, and a passed version, which includes changes that fix failures. Based on this, researchers can evaluate the effectiveness of techniques and tools for performing bug detection, localization, or repair. As a result, research progress in this field is closely dependent on high-quality bug benchmark suites.

Quantum programming is the process of designing and constructing executable quantum programs to achieve a specific computational result. A number of quantum programming approaches are available recently to write quantum programs, for instance, Qiskit [19], Q# [20], ProjectQ [21], Scaffold [1], and Quipper[7]. The current research in quantum programming focuses mainly on problem analysis, language design, and implementation. Despite their importance, program debugging and software testing have received little attention in the quantum programming paradigm [24]. The specific features of superposition, entanglement, and no-cloning introduced in quantum programming, make it difficult to find the bugs in quantum programs. Recently, several approaches have been proposed for testing and debugging quantum software [14, 15, 10, 9, 17, 18, 2, 23], but the testing and debugging remain challenging issues for quantum software [24].

Researches on bug benchmark suites for classical software have been studied extensively [6, 5, 12, 16, 8, 11, 22, 13], but few have been proposed for quantum software. Recently, Campos and Souto [4] proposed some initial ideas on building a bug benchmark for quantum software testing and debugging experiments, but the details of the benchmark are still unclear.

We may not know which debugging, and testing tools are suitable for quantum software without a suitable bug benchmark suite for evaluating these tools, and this may pose some restrictions on the research and development of quantum software testing and debugging techniques. As the first step towards evaluating quantum software testing and debugging tools, this paper presents Bugs4Q, a benchmark of 30 real, manually validated Qiskit bugs from four popular Qiskit programs, supplemented with the test cases for reproducing buggy behaviors. Bugs4Q has made the following contributions:

- Bugs4Q collects reproducible bugs in quantum programs and supports downloading and running test cases to support quantum software testing. Each actual bug and the corresponding fixes are publicly available for research.
- Bugs4Q collects almost all the existing bugs of Qiskit on GitHub and updates them in real-time, including the four elements of `Terra`, `Aer`, `Ignis`, and `Aqua`. Furthermore, these programs are sorted separately and filtered except for the bugs with originally available test cases and support for reproduction.
- Bugs4Q provides a database that includes an analysis of bug

[1] Kyushu University, Fukuoka, Japan
a) zhao.pengzhan.813@s.kyushu-u.ac.jp
b) zhao@ait.kyushu-u.ac.jp
c) miao.zhongtao.915@s.kyushu-u.ac.jp
d) lan.shuhan.197@s.kyushu-u.ac.jp

Table 1 Programs and number of real bugs available in the initial version of Bugs4Q

Program	Source	Description	Bugs	KLOC	Test KLOC	Tests
Terra	IBM Qiskit	Foundation of Qiskit	21	139	56	467
Aer	IBM Qiskit	Simulators with noise models	3	62	18	149
Ignis	IBM Qiskit	Reveal information about the device quality	2	11	3	59
Auqa	IBM Qiskit	A library of cross-domain algorithms	4	56	17	211

types to classify existing bugs for experimental evaluation of isolated bugs.

The rest of the paper is organized as follows. Section 2 briefly describes Bugs4Q, a bug benchmark suite for Qiskit. Section 3 introduces the process of manually reproducing bugs. Section 4 describes the specific types of bugs in Qiskit with examples. Related work is discussed in Section 5, and concluding remarks are given in Section 6.

2. Bugs4Q Benchmark Suite

To make sure we can build a benchmark of real bugs, we have collected the existing bugs in the version control history and the real fixes provided by the developers.

Table 1 shows all programs and the numbers of corresponding real bugs that are available in the bug database of Bugs4Q. In order to achieve benchmark rigor, each real bug must have its original bug version as well as a fixed version. This requires us to extract the relevant description of the bug and refer to its fixed commit. Moreover, the bugs we collect must comply with the following requirements:

- **Related to source code.** The reason for the bug is on the source files of the build system, but not the test files, or the underlying files that build the Qiskit programs.
- **Related to quantum program.** The bug should have an impact on the operation or the outcome of the quantum program. Problems caused by configuration files required to run quantum programs or classical parts of quantum algorithms are not included in our database.
- **Reproducible.** More than one test case must be used to demonstrate the bug, and the bug must be reproducible under certain requirements. Depending on the nature of the quantum program, for example, the presence of probabilistic output causes the program not to be able to reproduce the results completely. It can lead to bugs that are difficult to reproduce in a controlled environment.
- **Isolated.** Fixes submitted by developers should also be related to the source files. Irrelevant changes need to be removed, and there is no code refactoring due to version changes. Excessive source file changes that are too complex will be incorporated into our database repository later after careful verification of isolation.

Figure 1 depicts the main process of building our benchmark. First, we look for programs on as criteria for our base database classification. After that, we collect the issues with *bug* tags and incorporate them into our first version of the database for manual validation. We then manually reproduce the bugs for further

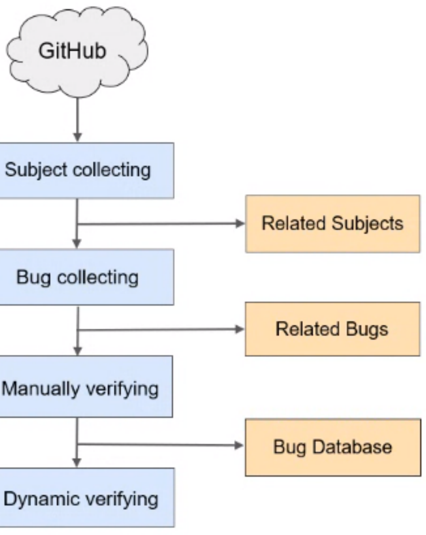


Fig. 1 Overview of the benchmark database build process

filtering and place those that meet the isolation criteria into our version-2.0 database for dynamic validation. Finally, extraneous patches will be cleaned up to complete the final version of the benchmark database.

2.1 Selecting Subject Programs

For project program selection, we only target Qiskit programs that are relatively well used on GitHub. We use the GitHub's *issue* tab to find bugs in the program and collect both the *buggy* version and the *fixed* version. We have collected all the issues with *bug* tags. Besides, Qiskit programs generally generate circuit diagrams, which serve only to represent the process of changing quantum states and do not impact on the program's execution. However, wrong circuit diagrams can also mislead users, and therefore we have collected them into one type.

We base the four elements of `Terra`, `Aer`, `Ignis`, `Aqua` in Qiskit as our list of topics. A brief descriptions of the four elements are as follows, which comes from the official description of Qiskit [19]:

- `Terra` is the foundation on which the rest of Qiskit is built.
- `Aer` provides high-performance quantum computing simulators with realistic noise models.
- `Ignis` provides tools for quantum hardware verification, noise characterization, and error correction.
- `Aqua` provides algorithms for quantum computing applications.

Table 2 Example of a benchmark database for Bugs4Q

Bug ID	Issue No	Buggy	Fixed	Modify	Status	Version	Type	Test	Issue Registered	Issue Resolved
1	#5908	Buggy	Fixed	Mod	Resolved	0.17.0	Bug	Test	Feb 26, 2021	Mar 1, 2021
2	#5914	Buggy	Fixed	Mod	Resolved	0.16.4	Bug	Test	Feb 27, 2021	Feb 28, 2021

2.2 Collecting Bugs

The bug collection process consists of two steps as follows.

2.2.1 Collecting and Filtering Bugs

For each element in Qiskit, we focus on collecting issues from *closed* tags on GitHub. We also collect obvious and important bugs with *open* status and mark them so that they can be put into our database as soon as they are submitted for fixing. For bugs in *closed* status, we collect bugs and submitted fixes according to their IDs (e.g., #1324). Of course, in our benchmark library, we will use our own set ID as the index.

Bugs that are not submitted to fix the close status and bugs that are not related to the quantum program and bug patterns will be filtered out. We still discard the case of having multiple fixes for bugs, i.e., having multiple fix links. Besides, bugs that disappear due to version changes are also not considered. After this work, we built the original bug database, which contains 206 quantum program-related bugs. As more and more bugs are raised, our database will be updated as we go forward.

2.2.2 Screening Test Cases

In order to reproduce the bugs more accurately, we choose the original program proposed by the developer in the bug report as our test case. The program proposed by the developer will be collected and tagged with the test in our library, accompanied by the second screening. There are 115 bugs with original test cases. For the bugs, without test cases, we will write the corresponding test cases according to the bugs in future work. Next is manual verification for the third screening.

2.3 Manually Verifying and Reproducing Bugs

We manually check that our requirements are met for each bug that has a test case. All bugs that can be reproduced and meet the isolation criteria will be placed in our final benchmark suite. For incomplete test cases, we modify the recovery procedure as much as possible to achieve its proper operation. Nevertheless, in addition to errors in the original test cases, it is not uncommon to have too many and too complex revision submissions that are not reproducible. As in `Ignis`, many files have the suffix `hpp` and `cpp` instead of `py`. We chose to forgo collecting them into the current version of the benchmark suite. On the other hand, some of the submitted fixes have no impact on the bug recovery, so we only keep the fixes that impact the bug. The results of the manual verification showed that only 30 bugs were successfully reproduced and isolated. The process of reproduction is still ongoing.

2.4 Sanity Checking through Dynamic Validation

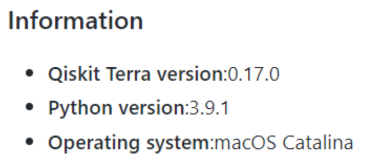
In order to better reproduce the bug, we try to implement an automated approach. We first implement version control, calling the corresponding version of Qiksit for different bugs. After calling the bug indexed by ID, the file of the bug version will replace the corresponding file in Qiskit. Finally, the same process is implemented for the fixed version. As the test suite continues to improve, the version control environment will be ported to more platforms.

3. Available Bugs Reproduced

This section describes the process of manually reproducing bugs. Bugs are complex to reproduce, so first, they need to meet some rules. As shown in Table 3 is the restrictions on reproducing bugs. We separate each bug, clean up irrelevant changes in advance, ignore some description files, and keep only the original files related to the bug and the fixed files. Any bugs or fixes with the above characteristics will be filtered out. Next is the specific process.

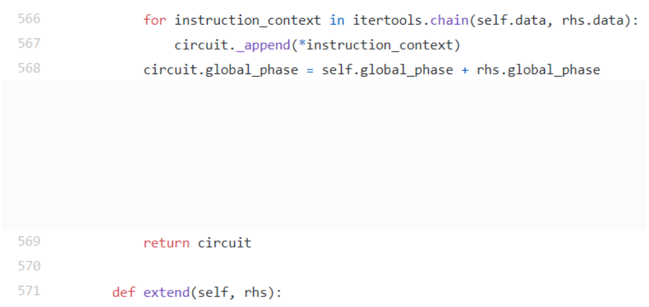
3.1 Restoring Version Environment

We configure the environment based on the version information submitted by the bug raiser in Figure 2. This is error #5908 [*1] as an example, its proposed version is `terra 0.17.0`. After that, we will find the file location and the repaired file code in the fix commit. Then we manually restore the fixed code to the code at the time of the buggy. Then replace the files in the environment with the restored source files. In this sample, the blue code section in Figure 4 is the code added by the repair file. We restore it to the buggy state as represented in Figure 3.


Information

- Qiskit Terra version:0.17.0
- Python version:3.9.1
- Operating system:macOS Catalina

Fig. 2 Version information of bug submission



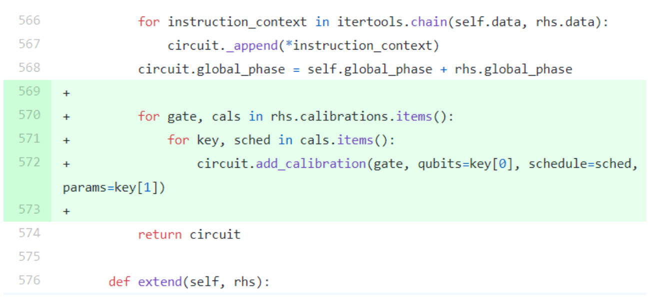
```
        for instruction_context in itertools.chain(self.data, rhs.data):
            circuit._append(*instruction_context)
        circuit.global_phase = self.global_phase + rhs.global_phase




        return circuit

    def extend(self, rhs):
```

Fig. 3 The partial code of buggy file

*1 https://github.com/Qiskit/qiskit-terra/issues/5908

Table 3 Restrictions on reproducing bugs

Restrictions	Description
Isolation	Each fix submission can only address one bug, and that bug cannot exist on top of any other bug
Complication	A bug fix corresponds to multiple fixes submitted, or too many fixes are submitted to make it impossible to determine the location of the bug
Reconfiguration	Fixed commits are file rewrites caused by refactoring or version changes
Dependencies	The fixed commit introduces a new library



```
        for instruction_context in itertools.chain(self.data, rhs.data):
            circuit._append(*instruction_context)
        circuit.global_phase = self.global_phase + rhs.global_phase
+
+       for gate, cals in rhs.calibrations.items():
+           for key, sched in cals.items():
+               circuit.add_calibration(gate, qubits=key[0], schedule=sched,
params=key[1])
+
        return circuit

    def extend(self, rhs):
```

Fig. 4 The partial code of fixed file

```
qc = QuantumCircuit(2)
qc.rzx(0.1,0,1)

pass_ = TemplateOptimization
                (**rzx_templates(['zz2']))
qc_cal = PassManager(pass_).run(qc)
pass_ = RZXCalibrationBuilder(backend)
qc_cal = PassManager(pass_).run(qc_cal)

qc2 = QuantumCircuit(2)
qc2 +=qc_cal

print (schedule(transpile(qc,backend),
                backend).duration)
print (schedule(transpile(qc_cal,backend),
                backend).duration)
print (schedule(transpile(qc2,backend),
                backend).duration)
```

Fig. 5 The code in the problem description

3.2 Running Test Cases

We first select the program code provided by the bug raiser. Figure 5 shows the program code in the error submission message that does not meet the program run criteria. The code will be downloaded, fixed, and placed in our database as original test cases for verification. When we run the program in the configured environment, the result is consistent with the description of the bug submission message, which proves that the bug was successfully reproduced. Next, the fixed file represented in Figure 4 is replaced by the file in the environment. If the bug disappears, the program runs successfully, and is consistent with the description of the fix, the test passes.

3.3 Representation in Benchmark Suite

The reproduced available bugs will be added to our database in the form of Table 2. In order to be able to document each bug in detail, we provide a detailed description of each bug and links to local files of our organization for *Buggy, Fixed, Test*. Furthermore, *Issue No, Mdify* are linked to bugs and fixes committed in GitHub, respectively. After reproducing each bug and populating the benchmark database, we next summarized some bug types.

```
Output before fixed:
            v1 = 32.0
            v2 = 8.0

Output after fixed:
            v1 = 8.0
            v2 = 8.0
```

Fig. 6 An example of output wrong

```
Output before fixed:
        From config : ['id', 'rz', 'sx', 'x', 'cx']
        From the noise model :
                      ['cx', 'id', 'sx', 'u3', 'x']

Output after fixed:
        From config : ['id', 'rz', 'sx', 'x', 'cx']
        From the noise model :
                      ['id', 'rz', 'sx', 'x', 'cx']
```

Fig. 7 An example of noise simulation error

4. Analysis

We summarize the common types of errors in Qiskit. Some errors only occur in specific elements. Others are not related to the main part of the quantum program but can also lead to misunderstanding the quantum programs.

4.1 Output Wrong

Output errors are the bugs we are most concerned about, which are not easily discovered but play a critical role in quantum programs. Erroneous output results can mislead program users. Here we reproduce a simple example in `Aqua`, with the issue number of #1324 [*2].

Considering the code snippet in Figure 6. The program that calls the `CircuitSampler` method and finds that `v1` and `v2` should have output the same result. But they are only the same when `coeff=1`, otherwise they often have different results. The file to fix this bug is `vector_state_fn.py` only, which is the source file in `qikist/aqua`. Such bugs account for a large proportion of the bugs we reproduce.

4.2 Noise Simulation Error

Due to the inherent nature of quantum computer hardware, the presence of noise makes programs that actually run in quantum computers not extraordinarily stable. Qiskit so provides `Aer` can be implemented to simulate noise on a classical computer. This provides great convenience for us to study real quantum programs. Therefore it is also particularly important to target the bugs of quantum noise simulators.

As shown in Figure 7, which is a program that imports the base

[*2] https://github.com/Qiskit/qiskit-aqua/issues/1324

```
Output before fixed:
        QiskitError: "Type error handeling
        [(QuantumRegister(1, 'q1'), 0), 1]
        (<class 'list'>)"

Output after fixed:
        Qubit ordering:
        [Qubit(QuantumRegister(1, 'q2'), 0),
         Qubit(QuantumRegister(1, 'q1'), 0)]
        Classical bit ordering:
        [Clbit(ClassicalRegister(2, 'c'), 0),
         Clbit(ClassicalRegister(2, 'c'), 1)]
```

Fig. 8 An example of throwing exceptions

gate from the quantum simulation backend from issue #1107[*3]. By default, the noise model contains in its usual base gate the `id` and `U3`. The purpose is that circuits could be executed even if the developer did not define noise on all gates. However, a bug in running the program due to a change in the default base gate of the IBM Q backend prevents it from working correctly. The bug here is that the `u3` gate should not appear in the noise model, but rather the `X` gate. The fix for this commit is that the noise model will always have the same base gate as the backend base gate, regardless of whether the instruction has an error in the noise model. This type of error is only found in the `Aer` element and is not common.

4.3 Lost Information

Lost information, i.e., the program does not implement a specific function. Terra is the most used element and the one with the most bugs filed. There are many times the fix for bugs in other elements of the commit is to modify the files in the `terra`. The sample we cite is #5908, as shown in Figure 5. This is a program that combines a conventional `QuantumCircuit` with a calibration circuit. The buggy version of this program uses the default gate circuit, and the calibration circuit information is missing. The error output with `name = "sched4"` indicates that `QuantumCircuit`'s += does not remain calibrated, thus causing the problem. Such a bug is not closely related to traditional quantum circuits and is not common.

4.4 Throwing Exceptions

Throwing an exception is as common and basic as an output error. Program errors and output errors account for almost all of the bugs in our Database. As shown in Figure 8, which is also from `Qiskit Terra`, #2369 [*4]. This is a simple program of using indexes and bits as parameters. Until it is fixed, this bug can be considered as a bug pattern. However, this issue is fixed as a bug here. So we collected it into our Database. In general, this kind of bug can be understood as the parameters of the called method do not support string types, or more precisely, only integer types. The problem here is that the gate parameter could not support a mix of indexes and bits. A commit fix made it possible for the gate parameter to support them.

In addition to the error types described above, we constantly summarize other bug types, such as quantum circuit diagram drawing errors. In our future work, we will add the types of bugs to our benchmark and propose a database with bug types as classification criteria for better use by researchers and developers.

*3 https://github.com/Qiskit/qiskit-aer/issues/1107
*4 https://github.com/Qiskit/qiskit-terra/issues/2369

5. Related Work

5.1 Bug Benchmark Suite for Classical Software

Many bug benchmark suites have been proposed to analyze and evaluate bug detection techniques and tools for classical software. The Siemens test suite [11] is one of the first bug benchmark suites used in testing research. It consists of seven C programs, which contain manually seeded faults. The first widely used benchmark suite of real bugs and fixes is the SIR (Software Artifact Infrastructure Repository) [6], which enables reproducibility in software testing research. SIR contains multiple versions of Java, C, C++, and C# programs which consist of test suites, bug data, and scripts. The benchmark contains both real and seeded faults. Defects4J [12] is a bug database and extensible framework which contains 357 validated bugs from five real-world Java projects. iBug [5] is another benchmark that contains real Java bugs from bug-tracking systems originally proposed for bug localization research. iBug consists of 390 bugs and 197 KLOC, which took from three open-source projects. Other benchmark suites include BenchBug [16], ManyBug (and InterClass) [13], and BUGSJS [8] for JavaScirpt projects.

However, all the benchmarks mentioned above focus on the classical software systems, and therefore cannot be used for the evaluation and comparison of quantum software debugging and testing techniques and tools.

5.2 Bug Benchmark Suite for Quantum Programs

Perhaps, the most related work with ours is QBugs proposed by Campos and Souto [4], which aims to build a collection of reproducible bugs in quantum algorithms for supporting controlled experiments for quantum software testing and debugging. In addition to proposing some initial ideas on building a benchmark for providing an experimental infrastructure to support the evaluation and comparison of new research and the reproducibility of published research results on quantum software engineering, they also point out challenges and opportunities on the development of QBugs. However, they provide no detailed information on the QBugs; the usability and availability are still unclear. Our Bugs4Q, on the other hand, aims to construct a bug benchmark suite of real bugs derived from four real-world IBM Qiskit programs for quantum software testing and debugging, with real-world test cases for reproducing the buggy behaviors of identified bugs.

6. Concluding Remarks

In this paper, we have proposed Bugs4Q, a benchmark of thirty real, manually validated Qiskit bugs from four popular Qiskit programs, supplemented with the test cases for reproducing buggy behaviors. Bugs4Q also provides interfaces for accessing the buggy and fixed versions of the Qiskit programs and executing the corresponding test cases, facilitating the reproducible empirical studies and comparisons of Qiskit analysis and testing tools.

We would like to keep updating the bug reports submitted on

GitHub for future work and continue to improve the tests to reproduce more bugs in Qiskit. On the other hand, we would like to summarize more bug types for the commonality of bugs to be more easily uncovered when our benchmark is extended to more quantum programming languages.

References

[1] Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold: Quantum programming language. Technical report, Department of Computer Science, Princeton University, 2012.

[2] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao 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.

[3] Eric Allen. Bug patterns in java. 2002.

[4] José Campos and André Souto. Qbugs: A collection of reproducible bugs in quantum algorithms and a supporting infrastructure to enable controlled quantum software testing and debugging experiments. *arXiv preprint arXiv:2103.16968*, 2021.

[5] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436, 2007.

[6] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[7] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342, 2013.

[8] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.

[9] Shahin Honarvar, Mohammadreza Mousavi, and Rajagopal Nagarajan. Property-based testing of quantum programs in q#. In *First International Workshop on Quantum Software Engineering (Q-SE 2020)*, 2020.

[10] Yipeng Huang and Margaret Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 541–553, 2019.

[11] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.

[12] René Just, Darioush Jalali, and Michael D Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[13] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[14] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[15] Ji Liu, Gregory T Byrd, and Huiyang Zhou. Quantum circuits for dynamic runtime assertions in quantum computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1017–1030, 2020.

[16] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[17] Andriy Miranskyy and Lei Zhang. On testing quantum programs. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 57–60. IEEE, 2019.

[18] Andriy Miranskyy, Lei Zhang, and Javad Doliskani. Is your quantum program bug-free? *arXiv preprint arXiv:2001.10870*, 2020.

[19] IBM Research. Qiskit. *Accessed on: April, 2020*, 2017.

[20] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, 2018.

[21] ProjectQ Team. ProjectQ. *Accessed on: April, 2020*, 2017.

[22] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349. IEEE, 2019.

[23] Jiyuan Wang, Ming Gao, Yu Jiang, Jianguang Lou, Yue Gao, Dongmei Zhang, and Jiaguang Sun. Quanfuzz: Fuzz testing of quantum program. *arXiv preprint arXiv:1810.10310*, 2018.

[24] Jianjun Zhao. Quantum software engineering: Landscapes and horizons. *arXiv preprint arXiv:2007.07047*, 2020.