The Journal of Systems & Software 205 (2023) 111805

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Bugs4Q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs☆

Pengzhan Zhao[1], Zhongtao Miao[2], Shuhan Lan[3], Jianjun Zhao *

Kyshu University, Fukuoka, Japan

ARTICLE INFO

Article history:
Received 16 May 2022
Received in revised form 15 May 2023
Accepted 15 July 2023
Available online 20 July 2023

Dataset link: https://github.com/Z-928/Bugs4Q-Framework

Keywords:
Quantum software testing
Quantum program debugging
Bug benchmark suite
Bugs4Q

ABSTRACT

Realistic benchmarks of reproducible bugs and fixes are vital to good experimental evaluation of debugging and testing approaches. However, there is no suitable bug benchmark suite that can systematically evaluate the debugging and testing methods of quantum programs until now. This paper proposes Bugs4Q, a benchmark of forty-two real, manually validated Qiskit bugs from three popular platforms (GitHub, StackOverflow, and Stack Exchange) in programming, supplemented with test cases to reproduce buggy behaviors. Bugs4Q also provides interfaces for accessing the buggy and fixed versions of the Qiskit programs and executing the corresponding source code and unit tests, facilitating the reproducible empirical studies and comparisons of Qiskit program debugging and testing tools. Bugs4Q is publicly available at https://github.com/Z-928/Bugs4Q-Framework.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Quantum programming is designing and constructing executable quantum programs to achieve a specific computational result. Several quantum programming approaches have been available recently to write quantum programs, for instance, Qiskit (Research, 2021), Q# (Svore et al., 2018), ProjectQ (Häner et al., 2016), and Scaffold (Abhari et al., 2012). Along with the emergence and advancement of quantum programming languages, debugging and testing quantum programs are gaining more attention. The specific features of superposition, entanglement, and no-cloning introduced in quantum programming make it difficult to find bugs in quantum programs (Miranskyy et al., 2020). Several approaches have been proposed for debugging and testing quantum software (Li et al., 2020; Huang and Martonosi, 2019; Honarvar et al., 2020; Miranskyy and Zhang, 2019; Ali et al., 2021; Abreu et al., 2022) recently, but the debugging and testing remain challenging issues for quantum software (Zhao, 2020; Miranskyy and Zhang, 2019).

Software bugs significantly impact the economy, security, and quality of life. A software bug is considered an abnormal program behavior that deviates from its specification (Allen, 2002), including poor performance when a threshold level of performance is included in the specification. The diagnosis and repair of software bugs consume significant time and money. An appropriate bug-finding method 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. Existing methods 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 which includes a test set that exposes failures, and a passed version which includes changes about fixing 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 testing and debugging depends on high-quality bug benchmark suites.

☆ Editor: Antonia Bertolino.

* Correspondence to: Faculty of Information Science and Electrical Engineering, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan.
E-mail addresses: zhao.pengzhan.813@s.kyushu-u.ac.jp (P. Zhao), miao.zhongtao.915@s.kyushu-u.ac.jp (Z. Miao), lan.shuhan.197@s.kyushu-u.ac.jp (S. Lan), zhao@ait.kyushu-u.ac.jp (J. Zhao).

[1] A third-year Ph.D. student at the Graduate School of Information Science and Electrical Engineering of Kyushu University. His research focuses on testing and debugging quantum programs.

[2] A research student at the Graduate School of Information Science and Electrical Engineering of Kyushu University. His research interests include software testing and program debugging.

[3] A master student in the Graduate School of Information Science and Electrical Engineering at Kyushu University, Japan. He obtained B.E. in Computer Science and Technology from Nankai University, China, in 2020. His research interests are mainly in automatic quantum programming.

https://doi.org/10.1016/j.jss.2023.111805
0164-1212/

With the development of traditional software engineering techniques, research on bug benchmark suites for classical software has been studied extensively (Do et al., 2005; Dallmeier and Zimmermann, 2007; Just et al., 2014; Lu et al., 2005; Gyimesi et al., 2019; Hutchins et al., 1994; Le Goues et al., 2015). However, there are few bug benchmark suits for quantum software. On the other hand, more and more methods and tools have appeared for quantum program testing or debugging (Mendiluze et al., 2021; Wang et al., 2018, 2021b,a; Gely and Steele, 2020; Patel and Tiwari, 2021), which have made some progress in the field of quantum software. In this case, we may not know which debugging or testing methods are suitable for quantum software without a unified bug benchmark suite to evaluate these tools. This may restrict the research and development of quantum software debugging and testing techniques.

As the first step toward evaluating quantum software debugging and testing tools, this paper proposes Bugs4Q, a benchmark of forty-two real, manually validated bugs in Qiskit programming from GitHub and two other popular Q&A sites, StackOverflow and Stack Exchange, supplemented with the test cases for reproducing buggy behaviors. Bugs4Q has made the following contributions:

- We conducted a thorough collection of quantum programs on common quantum programming languages to enrich the Bugs4Q Repository. All the bugs verified from 10,069 items are realistic.
- The buggy and fixed programs in Bugs4Q are reproducible. Each actual bug and the corresponding fix are publicly available for research. Besides, each program is equipped with a manually generated unit test.
- Bugs4Q has a bug database containing the bug information. It provides a user-friendly execution framework, which is easy to extend and supports calling program source code files and unit test files.
- The combination of Bugs4Q with existing quantum program testing tools was discussed, and we applied the buggy programs to two test case generation tools for evaluation.

This paper is an extended version of our previous work, "*Bugs4Q: A Benchmark of Real Bugs for quantum programs*" (Zhao et al., 2021), published at the 36th IEEE/ACM International Conference on Automated Software Engineering (NIER Track). The improved version has the following significant changes. Section 2 was added, which details the introduction of quantum programs and quantum programming languages. Besides, we increased the bugs in Qiskit and other common quantum programming languages. We also provided a detailed description of collecting and filtering bugs, making the Bugs4Q benchmark database even more comprehensive. Moreover, the new extensions of the Bugs4Q benchmark suite include modularizing each bug, providing an executable framework, and offering unit tests. In addition, we added a new Section 4, which evaluated the performance of two test case generation tools and mainly discussed the combination between Bugs4Q and other works.

The rest of the paper is organized as follows. Section 2 introduces some basics of quantum programming. Section 3 describes Bugs4Q, a bug benchmark suite for Qiskit. Section 4 evaluates and discusses existing quantum program testing tools. Section 5 is the threats to validity of our work. Related work is discussed in Section 6, and concluding remarks are given in Section 7.

2. Background

In this section, we briefly introduce Qiskit with a simple example, followed by some basic concepts to understand quantum programming better. Finally, is our conclusion of quantum program features.

```
simulator = Aer.get_backend('qasm_simulator')

qreg = QuantumRegister(3)
creg = ClassicalRegister(3)
circuit = QuantumCircuit(qreg, creg)

circuit.h(0)
circuit.h(2)
circuit.cx(0, 1)
circuit.measure([0,1,2], [0,1,2])
job = execute(circuit, simulator, shots=1000)
result = job.result()
counts = result.get_counts(circuit)
print(counts)
```

Fig. 1. A sample quantum program in Qiskit.

2.1. Qiskit

Several open-source programming frameworks, such as Qiskit (Research, 2021), Q# (Svore et al., 2018), Scaffold (Abhari et al., 2012) and ProjectQ (Häner et al., 2016), have been proposed for supporting quantum programming recently, which are used further to advance the implementation and application of quantum algorithms. This paper chooses Qiskit, one of the most widely used quantum programming languages, as the first target language for conducting our work.

Qiskit is one of the most widely used open-source frameworks for quantum computing, allowing us to create algorithms for quantum computers (Koch et al., 2019). As a Python package, it provides tools to create and manipulate quantum programs running on prototype quantum devices and simulators (Aleksandrowicz et al., 2019). In addition, it offers built-in modules for noise characterization and circuit optimization to reduce the impact of noise. It also provides a library of quantum algorithms for machine learning, optimization, and chemistry. In Qiskit, an experiment is defined by a quantum object data structure that contains configuration information and the experiment sequences. The object could be used to get status information and retrieve results (McKay et al., 2018). Fig. 1 shows a simple Qiskit program that illustrates the entire workflow of a quantum program. The function `Aer.get_backend('qasm_simulator')` returns a backend object for the given backend name (`qasm_simulator`). The `backend` class is an interface to the simulator and the actual name of `Aer` for this class is `AerProvider`. After the experimental design is completed, the instructions are run through the `execute` method. The `shots` of the simulation, which means that the number of times the circuit is run, is set to 1000 while the default is 1024. When outputting the results of a measurement, the method `job.result()` is used to retrieve the measurement results. We can access the counts via the method `get_counts(circuit)`, which gives the experiment's aggregate outcomes.

2.2. Basic concepts

A quantum bit (qubit) is the analog of one classical bit but has many different properties. A classical bit, like a coin, has only two states, 0 and 1, while a qubit can be in a continuum of states between $|0\rangle$ and $|1\rangle$ in which the $|\rangle$ notation is called Dirac notation. We can represent a qubit mathematically as $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ where $|\alpha|^2 + |\beta|^2 = 1$ and the numbers α and β are complex numbers. The states $|0\rangle$ and $|1\rangle$ are called computational basis states. Unlike classical bits, we cannot examine a qubit directly to get the values of α and β. Instead, we measure a qubit to obtain either the result 0 with probability $|\alpha|^2$ or the result 1 with probability $|\beta|^2$.

Quantum gates are used for quantum computation and manipulating quantum information. Some basic quantum gates are as follows:

- Quantum NOT gate takes the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ into the state $|\psi\rangle = \alpha|1\rangle + \beta|0\rangle$. We can use a matrix to represent this operation:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- The Z gate can be expressed as

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

  From the matrix, we know the Z gate leaves the $|0\rangle$ unchanged and changes the sign of $|1\rangle$.

- The Hadamard gate turns the $|0\rangle$ into $(|0\rangle + |1\rangle)/\sqrt{2}$ and turns the $|1\rangle$ into $(|0\rangle - |1\rangle)/\sqrt{2}$. The matrix form of the Hadamard gate is

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

All the matrices are unitary ones. Instead of these single-qubit gates, there are multiple qubit gates, such as the Controlled-NOT (CNOT) gate. This gate has two input qubits, the control qubit and the target qubit. If the control qubit is 0, then the target qubit remains unchanged. If the control qubit is 1, then the target qubit is flipped. We can express the behavior of the CNOT gate as $|A, B\rangle \rightarrow |A, B \oplus A\rangle$.

Quantum circuits are models of all kinds of quantum processes. We can build quantum circuits with quantum gates and use wires to connect the components in quantum circuits. These wires can represent the passage of time or a physical particle moving from one position to another. Another essential operation in quantum circuits is measurement. Measurement operation observes a single qubit and obtains a classic bit with a certain probability. Nielsen's book (Nielsen and Chuang, 2002) has a more detailed explanation of quantum computation.

2.3. Quantum program features

A quantum program is a series of operations on qubits. By focusing on the quantum program language features, we can classify a complete quantum program into the following four steps:

- **Initialization**: The initial stage is to initialize the quantum registers to store the qubits that need to be manipulated. Then the classical registers are initialized to store the values of the measured qubits. As shown in Fig. 1, `qreg = QuantumRegister(3)` means assigning a quantum register of three qubits, and the value of each qubit is $|0\rangle$ by default. So the initial value of these three qubits is $|000\rangle$.
- **Gate Operation**: The core of quantum computing is to operate on qubits. Qiskit provides almost all the gates to implement algorithms in quantum programs (Research, 2020). Such as, to achieve the superposition of qubits, it must pass through the H (Hadamard) Gate (e.g., `circuit.h(0)` & `circuit.h(2)`). And to achieve entanglements in the case of multiple qubits, the CNOT (controlled-NOT) gate is necessary (e.g., `circuit.cx(0,1)`). Complex gate operations are decomposed into basic gates in quantum language and gradually realized. Two qubits parameterize controlled gates, and double-controlled gates require three qubits.
- **Measurement**: To obtain the output, we must perform a measurement operation on the target qubit. The measured qubit is returned as the classical state's value, which no longer has superposition properties. So the qubit that has been measured cannot be used as a control qubit to entangle with other qubits. Although the measurement operation is simple, the program executing a measurement statement is very complicated. Obtaining a relatively accurate probability distribution requires thousands of projection measurements on qubits. In this way, the number of occurrences of the result is used to obtain the size of the probability of outputting the value. The measurement statement of qubits shown in Fig. 1 is `circuit.measure([0,1,2], [0,1,2])`.
- **Deallocation**: It is critical to reset and release qubits safely; otherwise, ancilla qubits in an entangled state may affect the output, i.e., by the measurement of the target qubit. For some backends of Qiskit or other quantum programming languages, failure to reset the temporary value to zero and release it safely may result in program bugs.

This paper uses these four aspects to measure whether a quantum program is qualified. On the one hand, if a bug occurs in one of these steps, we pinpoint it as a quantum-related bug. On the other hand, as a standard, it is useful for manual verification in our work. This criterion is also helpful for static analysis and bug localization.

3. Bugs4Q benchmark

This section details the process of building the Bugs4Q benchmark suite. We collect the existing bugs in the version control history and the real fixes provided on GitHub, StackOverflow, and Stack Exchange. Table 1 shows all the issues and questions presented to each platform, as well as the final numbers of corresponding real bugs that are available in the bug database of Bugs4Q. Besides, to achieve the benchmark rigor, each real bug must have its original bug version and a fixed version, which 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:

- **Written by the users**. We only collect buggy programs written by the users of Quantum platforms. For example, a user submits one program written in the Qiskit language, and an error is caused by one or more lines of code that cause the program to turn out differently than expected. In this case, the bugs caused by Qiskit platform components are not part of our collection.
- **Bugs from source code.** We only focus on bugs in the source code that cause the quantum program to fail. The fix of each bug only concerns the source code program itself. The rest of the fixes, such as Qiskit's internal documentation and its explanatory documentation, changes to the internal structure of Qiskit, and test files, would be ignored.
- **Quantum-specific bugs.** We only collect bugs that are related to quantum programs. These bugs should affect the operation or the result of the quantum program. It is important to note that although some bugs occur in quantum programs, this does not mean that the bugs themselves are quantum related.
- **Reproducible.** As the bugs must be reproducible under certain conditions, we have to perform multiple executions of one buggy program. Depending on the nature of a quantum program, for example, the presence of probabilistic output causes the program to be unable to reproduce the results completely. It may lead to bugs that are difficult to reproduce in a controlled environment.

Table 1
The number of reproducible real-world bugs in Bugs4Q.

Platform	Source	Objectives	Number of items	Reproducible bugs	LOC
GitHub	IBM Qiskit	Issues in qiskit-related repositories	4621	20	510
StackOverflow	IBM Qiskit	Questions tagged with *qiskit*	465	7	263
Stack Exchange	IBM Qiskit	Questions with *qiskit* as keyword	4984	15	535
Total	IBM Qiskit	Issues and questions	10069	42	1308

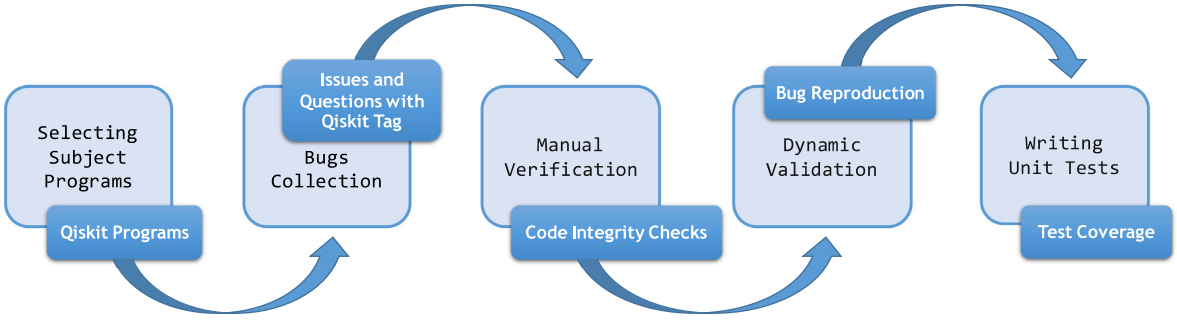

Fig. 2. The overview of the building process of Bugs4Q benchmark.

- **Isolated.** Bug fixes should be related to the source files. Irrelevant changes need to be removed. Such as code refactoring due to version changes, fixes unrelated to the current bug, and bug fixes based on other irrelevant fixes. Overly complex changes to source files would be incorporated into our database after careful verification of isolation.
- **Valid pre- and post-fixing codes.** In order to write test cases, The programs are required to have complete pre- and post-fixing code. During the collection process, we only use the developer's feedback to confirm whether the changes were successful. However, the validity of the changes can be thoroughly checked when writing the tests.

Fig. 2 depicts the primary process of building our benchmark. We first choose Qiskit as our target quantum programming language and search programs from three popular platforms: GitHub, StackOverflow, and Stack Exchange. Next, we collect the issues and questions with *qiskit* tags together. We manually check the source code and sift through the bugs without fixes. We execute the buggy and fixed versions of the source code for dynamic validation to confirm that the bugs are reproducible and successfully fixed. Finally, we manually write unit tests for the buggy and fixed versions and provide the test coverage for each bug. At this point, the Bugs4Q benchmark database can be successfully constructed.

The bugs in our database are classified by the place of collection (i.e., platform), as shown in Table 1. The final number of bugs in Bugs4Q is 42. Moreover, the LOC shows that quantum programs are generally small, as conventional computers cannot simulate large-scale qubits manipulation. Besides, each bug corresponds to two unit tests, one for the buggy version and the other for the fixed version.

3.1. Selecting subject programs

We first choose Qiskit as our target quantum programming language and try to select large-scale quantum programs on GitHub. Besides, we target the official projects of Qiskit, proposed by IBM (e.g., `Terra`,[4] `Aer`,[5] `Ignis`,[6] and `Aqua`[7]). As quantum programming is still preliminary, many developers write programs based on the algorithmic procedures presented in the official IBM documentation. In addition, most of the bug reports are raised to the official repository. Also, the Qiskit platform has several issues that need to be addressed and is constantly updated. So the need for feedback on many issues has led to many bug reports being submitted. As a result, we examined all of Qiskit's subproject systems on GitHub and selected the *issues* tag as our target item. Since most of the reports with *bug* tags refer to the bugs of the Qiskit platform rather than the program bugs written by developers, we checked all issue reports in case some bugs were missed.

StackOverflow and Stack Exchange are popular Q&A sites in programming, including many bugs raised by programmers that come from programs written by themselves. We entered "qiskit" as the search keyword in StackOverflow and Stack Exchange and chose questions with the *qiskit* tag as our target.

The selected project programs cover almost all the bugs on the three platforms. We believe that the buggy programs found and filtered on this basis are somewhat universal and convincing.

3.2. Bugs collection

We next detail the different ways to collect bugs from GitHub, StackOverflow, and Stack Exchange. In addition, we discuss the unit tests for quantum programs and our collecting results.

3.2.1. Collecting bugs from GitHub

For each subject we select, all the bugs with source code have been collected, no matter whether the bugs are in *open* or *closed* status. Bugs in *closed* status mean that they have been resolved or disappeared due to version change. For bugs in *open* status, sometimes we could find bugs that meet the requirements to be resolved simply because the issue has not been closed in time. During the process of our bug collection, there were many program bugs proposed by developers. However, due to the preliminary stage of quantum programming, there are also many problems with quantum programming languages, which lead to lots of bug reports submitted on GitHub related to Qiskit itself (i.e., platform related (Paltenghi and Pradel, 2021)). Especially reports with the *bug* tag are mainly platform bugs. So all the items in the *issue* tag should be checked in this step. We collect bugs and fixes according to their IDs (e.g., #3799.[8]) The *link* to each passed report is copied to the Bugs4Q benchmark database with a new ID added for further filtering.

[4] https://github.com/Qiskit/qiskit-terra
[5] https://github.com/Qiskit/qiskit-aer
[6] https://github.com/Qiskit/qiskit-ignis
[7] https://github.com/Qiskit/qiskit-aqua
[8] https://github.com/Qiskit/qiskit-terra/issues/3799

Table 2
Criteria for fixing quantum bugs.

Criteria	Description
Isolation	Each fix submission can only address one bug, and that bug cannot exist on top of any other bugs
Reconfiguration	Fixed commits are file rewrites caused by refactoring or version changes
Dependencies	The fixed commit introduces a new library
Platform Irrelevant	The fixes do not involve changes to the internal files of the Qiskit framework

3.2.2. Collecting bugs from other platforms

We searched for bugs on Stack Overflow from the page of "*Questions tagged [qiskit]*,[9]" and found more than 150 questions. For Stack Exchange, we searched for *qiskit* as a keyword and found over 3500 questions which are proposed by developers.[10] All the questions were checked individually, and our recording method was the same as collecting bugs from GitHub.

3.2.3. Unit test

The unit tests for Qiskit only exist in the official Qiskit library[11] for testing each function. And all these unit tests serve only the Qiskit language and some example programs. There are no unit tests for defective programs provided by Qiskit users. As the first step in this work, we manually write unit tests for *buggy* and *fixed* versions of each bug. Qiskit provides the `QiskitTestCase` class inheriting from Python's `unittest` that can be used to write unit tests for the buggy and fixed programs in Bugs4Q.

3.2.4. Collecting results

After going through all the questions and issues on these three platforms, we have collected 346 bugs from 10,069 items and put them into the Bugs4Q database. These bugs include fixes and source code. Many questions and issue reports are mainly for environment configurations, such as errors in installing and importing packages and version changes, which are not our targeted bugs. Besides, only the source code of the buggy programs and their fixes are collated in the Bugs4Q database. We have filtered out any description files and test files in this step.

3.3. Manual verification and code completion

After collecting all the bug reports with source code in our database, we manually inspect each bug with its submitted fixes. For bug fixing, we propose several criteria as shown in Table 2. We only consider bugs that have been fixed and are fully reproducible. So the bugs caused by the Qiskit programming language and unrelated to quantum programs should be filtered out. We also discard the case of having multiple fixes for bugs, i.e., having various fix links. Besides, bugs that disappear due to version changes are also not considered.

We first examine the source code of each bug manually. The ultimate indicator is whether the source code is quantum-relevant (i.e., whether it operates on qubits). The specific operations on qubits are described in detail in Section 2.3 of the quantum program features. On the other hand, verifying whether the program supports the entire run is necessary since many submissions are incomplete in the source code. In addition, some of the program code is pseudo-code or QASM code (Research, 2020), which is not supported to run in the Qiskit environment. Next, we copy the code to our local repository if the source code meets the requirements and create a new *.py* file to be placed in the corresponding *bugID*. With this comes the collection of bug commit information, specifically the current version, commit date, fix status, and bug type. Afterward, we verify the fixes in bug reports. The main focus is finding the correct bug fixes from the various comments. Besides, as most of the fixes are only for buggy lines in a program, and there is no automated bug-fixing tool for quantum programs, we need to manually patch the repaired program to make it a complete program that can run successfully. A fixed program is saved to a local *.py* file, and another file will be created with the modified part. The three authors divided all the bugs equally and filtered the assigned bugs to complete the manual validation. After the above analysis steps, each author marks the uncertain buggy programs and discusses them together until they reach a consensus. All bugs that pass the validation procedure have remained in the Bugs4Q database.

The results of our manual validation are shown in Table 3. The initial number of bug reports which have source code is 346. There are 206 bugs (over 25k LOC, including fixes) related to the Qiskit language itself, which leads to only 146 (about 3172 LOC) quantum program bugs remaining. And the final number of bugs after a manual verification is 84. The filtration of platform bugs is based on whether the committed fix is a Qiskit internal file. In addition, 27 bugs have no fixed code, which prevents us from determining exactly where or how to fix them. Moreover, 35 bugs do not support execution due to incomplete or no source code. In this process, We found that only the source files have been modified for almost all the bugs we collected, and there are two main reasons for this situation. Firstly, the difficulty of simulating large quantum programs by classical computers has limited quantum programs to more straightforward functions. Therefore, no other documents are needed to constitute the project. Secondly, many programs written by programmers at this stage are designed to learn and explore quantum languages. For example, some programs attempt to incorporate QFT circuits into the code to reproduce existing algorithmic procedures.

Due to the need for dynamic validation, we have to manually restore both the *buggy* and the *fixed* versions. As an example, Figs. 3 and 4 show the *buggy* and *fixed* versions, respectively, of one program[12] in our database for dynamic verification.

3.4. Dynamic validation for reproduction

This section describes the process of dynamic validation as well as the way we reproduce bugs. Most bugs we would like to reproduce depend on the programs executed by Qiskit simulators. Therefore, the recurrence process is implemented manually on our PC side. We also try to reproduce the operations performed in the IBM cloud backend as much as possible. The specific rules are the same for manual verification, as shown in Table 2. We separate each bug, clean up irrelevant changes in advance, ignore some description files, and keep only the source code related to the bug and the fix.

Considering the initialized part as the input of a quantum program, we can see that any qubits have the value of $|0\rangle$ by default from Fig. 1. Moreover, adding a phase gate to the program is necessary if the value needs to be changed. Such as, we can add an X (**NOT**) gate to flip the value of the initial qubit from $|0\rangle$ to $|1\rangle$. Therefore, we consider that modifying the input or adding

[9] https://stackoverflow.com/questions/tagged/qiskit?tab=Newest
[10] https://stackexchange.com/search?q=qiskit
[11] https://github.com/Qiskit/qiskit/tree/master/test, etc.
[12] https://quantumcomputing.stackexchange.com/questions/18448/how-to-perform-a-plot-histogram-for-a-circuit

```
qc = QuantumCircuit(4, 4)
qc.cx(3, 1)
qc.cx(1, 0)
qc.cx(0, 1)
qc.ccx(3, 2, 1)
qc.cx(1, 2)
qc.cx(3, 2)
qc.measure(0, 0)
qc.measure(1, 1)
qc.measure(2, 2)
qc.measure(3, 3)
job = execute(qc, backend = Aer.get_backend('
                            qasm_simulator'),
                            shots=1024)
result = job.result()
count = result.get_counts()
print(count)
```

Fig. 3. The source code of a buggy program.

```
qc = QuantumCircuit(4, 4)
for i in range(4):              <- Modify(Mod)
    qc.h(i)                     <- Modify(Mod)
qc.cx(3, 1)
qc.cx(3, 1)
qc.cx(1, 0)
qc.cx(0, 1)
qc.ccx(3, 2, 1)
qc.cx(1, 2)
qc.cx(3, 2)
qc.measure(0, 0)
qc.measure(1, 1)
qc.measure(2, 2)
qc.measure(3, 3)
job = execute(qc, backend = Aer.get_backend('
                            qasm_simulator'),
                            shots=1024)
result = job.result()
count = result.get_counts()
print(count)
```

Fig. 4. The source code of a fixed program by manual completion.

or removing the number of qubits can result in modifications to the program itself. So the way we verify and reproduce quantum programs is different from the traditional way because we cannot modify the program itself, i.e., we cannot change the input values of the program. Instead, the only way is to run the source program multiple times and see if the results are the same as described in the bug report. The reason is that the output of a quantum program is not constant. We also need to get the output of each quantum program and check the probability of getting the result after the measurement. In the dynamic validation process for one bug, we first configure the environment based on the version information submitted by the program raiser. After executing the *buggy* program in the configured environment, the only result we could get is consistent with the description of the bug submission message, which proves that the bug has been successfully reproduced. Next, the *fixed* program version replaces the *buggy* program in the environment. If the bug disappears, the program runs successfully and is consistent with the description of the fix, and the test passes.

After dynamic validation, there are 42 bugs retained in the bugs4Q database. Four main reasons lead to bugs that cannot pass through dynamic validation, shown in Table 3. Firstly, three bugs were filtered out since their fixes did not work. Secondly, ten actual program bugs do not match the description in their bug reports. For example, the wrong output of one quantum program is very different from the value provided by the programmer. In addition, 19 buggy programs cannot be executed smoothly, which differs from the second cause. This include four cases: `ImportError`, `NameError`, `AttributeError` and `ModuleNotFoundError`. `ImportError` means the package in Qiskit could not be imported, which is an environmental problem. `NameError` is a variable name not defined in the program. `AttributeError` is the case that an object in Qiskit does not have this property. Moreover, `ModuleNotFoundError` means no modules can be found in Qiskit. Such bugs affect the program's execution and are not fixed accordingly, nor can we resolve them during the reproduction process. Finally, the source code of 4 buggy version programs runs smoothly as the bug has disappeared due to a version change. The bugs in these cases are filtered out.

In summary, we have carefully examined 391 bug reports, and 42 reproducible bugs were extracted. The three authors were jointly involved in resolving the disagreements in the labeled programs. For the entire manual and dynamic validation process, Cohen's Kappa coefficient was 0.82, which implies approximate agreement.

3.5. Unit tests and coverage

Before writing unit tests, we need to figure out the characteristics of quantum program bugs, particularly the problem of the probabilistic output of quantum programs. The bugs that passed validation are classified into two broad categories: *output wrong* and *throw exceptions*. The wrong output of quantum programs can be divided into wrong output values and wrong probability distributions. The program in Fig. 3 is a typical wrong output caused by a bug, reflected in its probability distribution. The output of this program is always `0000`, which is not the correct result that the programmer expects. Consider the source code in Fig. 4, where there are four qubits in the quantum register with a value of 0. Theoretically, the output value of this program is between 0 and 2^4, with a probability of 6.25% to obtain respectively. As explained in Section 2.1, `shots=1024` while the result is the output `count`. The number of occurrences of each value between 0 and 16 should theoretically be evenly distributed, i.e., 64 times per value. However, in practice, the actual outputs of each probability would not be accurate. This program's correct and incorrect output is shown in Fig. 5. *Output wrong* is mainly caused by the fact that the program's output does not match the results expected by the programmer.

Another type of bug is *throw exceptions*, which can be caused by problems such as *Command Wrong*, *SyntaxError*, and *Command misuse*. These bugs have a common manifestation, i.e., the program does not execute smoothly but throws an exception instead.

3.5.1. Unit test

Since the existing quantum programs are small in size and most have only one bug, unit tests are sufficient to target a complete quantum program of Bugs4Q. On the other hand, it is hard to find out the test file written by the users of Qiskit. So we considered writing unit tests for programs in Bugs4Q. Firstly, we determine the writing specification following the assertions in the unit test files provided by the Qiskit library, which are used to test Qiskit compilers. Next, three authors wrote unit tests independently in a uniform format. These tests focus on *output wrong* and *throw exceptions*. Both types of unit tests are implemented as writing assertions. Each bug has two tests, one for the *buggy* program and the other for the *fixed* program, while both tests have the same function. Finally, we captured the program's

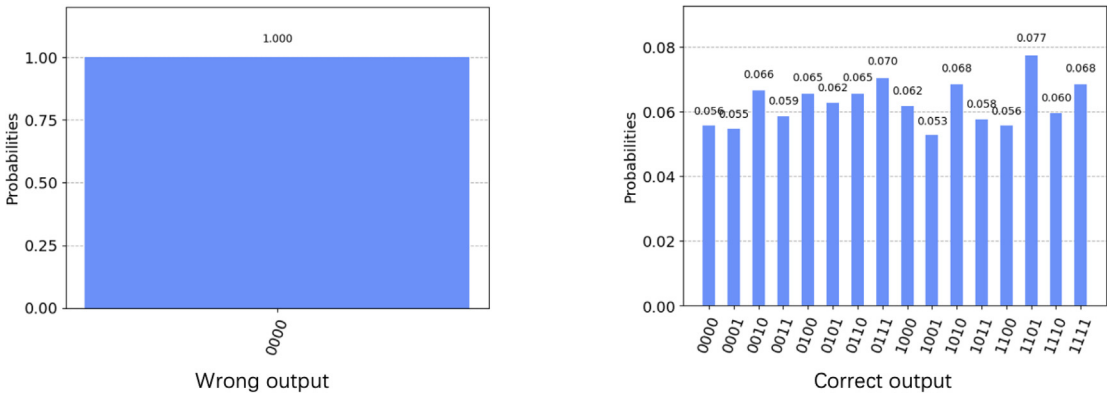

Fig. 5. The two outputs correspond to one program's buggy and fixed versions, respectively.

```
class Test(QiskitTestCase,unittest.TestCase):
    def test_b39(self):
        qc = QuantumCircuit(4, 4)
        qc.cx(3, 1)
        qc.cx(3, 1)
        qc.cx(1, 0)
        qc.cx(0, 1)
        qc.ccx(3, 2, 1)
        qc.cx(1, 2)
        qc.cx(3, 2)
        qc.measure(0, 0)
        qc.measure(1, 1)
        qc.measure(2, 2)
        qc.measure(3, 3)
        job = execute(qc,
                      backend =  Aer.get_backend('
                      qasm_simulator'), shots=1024)
        result = job.result()
        count = result.get_counts()
        print(count)
        self.assertDictAlmostEqual('0000':64,
        '0001':64, '0010':64, '0011':64, '0100':64,
        '0101':64, '0110':64, '0111':64, '1000':64,
        '1001':64, '1010':64, '1011':64, '1100':64,
        '1101':64, '1110':64, '1111':64
        ,count,delta=30)
if __name__ == '__main__':
    unittest.main(argv=[''])
```

Fig. 6. A unit test for output wrong (buggy version).

```
class Test(QiskitTestCase,unittest.TestCase):
    def test_f39(self):
        qc = QuantumCircuit(4, 4)
        for i in range(4):                    ->Mod
            qc.h(i)                           ->Mod
        qc.cx(3, 1)
        qc.cx(3, 1)
        qc.cx(1, 0)
        qc.cx(0, 1)
        qc.ccx(3, 2, 1)
        qc.cx(1, 2)
        qc.cx(3, 2)
        qc.measure(0, 0)
        qc.measure(1, 1)
        qc.measure(2, 2)
        qc.measure(3, 3)
        job = execute(qc,
                      backend =  Aer.get_backend('
                      qasm_simulator'), shots=1024)
        result = job.result()
        count = result.get_counts()
        print(count)
        self.assertDictAlmostEqual('0000':64,
        '0001':64, '0010':64, '0011':64, '0100':64,
        '0101':64, '0110':64, '0111':64, '1000':64,
        '1001':64, '1010':64, '1011':64, '1100':64,
        '1101':64, '1110':64, '1111':64
        ,count,delta=30)
if __name__ == '__main__':
    unittest.main(argv=[''])
```

Fig. 7. A unit test for output wrong (fixed version).

abnormal behavior and compared it with the bug description to check their correspondence. Moreover, we also executed the test file of the fixed program to verify whether the bugs had disappeared.

The unit tests for buggy and fixed versions about *output wrong* can be seen in Figs. 6 and 7, respectively. For one bug, the unit tests of the buggy and fixed versions have the same assertions, which can visually compare the test results of the two program versions. In this example, we use `assertDictAlmostEqual` as an assert method to check the probability distributions and report failures. The parameter `delta` indicates the upward and downward fluctuations concerning the specified number of output counts. This work specifies that the test is passed if the fluctuation value is within 30. A test result for a *buggy* program would fail while a *fixed* version would pass the test. The examples of unit tests for *throw exceptions* can be seen in Figs. 8 and 9. In Qiskit, some exceptions do not exist in the assertions contained in `unittest.testcase`, such as `QiskitError`. In this case, we can only detect the presence of an exception and cannot use the `assertEqual` method. As shown in Figs. 8 and 9, the test passes if an exception is caught for programs that throw exceptions. Otherwise, the test fails if no exception is caught.

The final number of bugs with unit tests is shown in Table 3. Some reasons lead to us being unable to write tests successfully. Two bugs threw exceptions that were already caught and handled internally by the Qiskit platform, so they prevented our unit tests from catching the exceptions. Four programs had an output that was an image generated using the *matplotlib* package of Python, which prevented us from writing assertions in our test cases. In addition, two programs had no output, and three programs had output but were too complex to generate unit tests. As a result, of the 42 reproducible bugs in the Bugs4Q benchmark, 30 bugs and their fixes have unit tests.

3.5.2. Coverage

In this work, we used `Coverage.py`, a tool for measuring code coverage of Python programs to measure our unit test coverage. The validity of existing coverage criteria for real-world quantum

```
class Test(unittest.TestCase):
    def test_b19(self):
        try:
            qc = QuantumCircuit(2)
            qc.h(0)
            qc.cx(0,1)
            qc.draw('mpl')
            qi.Operator.from_label('HI')
            qi.Operator.from_label('CX')
        except Exception as e:
            print('Reason:', e)
        else:
            self.fail('There is no error raised')
if __name__ == '__main__':
    unittest.main(argv=[''])
```

Fig. 8. A unit test for throw exceptions (buggy version).

```
class Test(unittest.TestCase):
    def test_f19(self):
        try:
            qc = QuantumCircuit(2)
            qc.h(0)
            qc.cx(0,1)
            qc.draw('mpl')
            qi.Operator.from_label('H')     ->Mod
            qi.Operator.from_label('X')     ->Mod
        except Exception as e:
            print('Reason:', e)
        else:
            self.fail('There is no error raised')
if __name__ == '__main__':
    unittest.main(argv=[''])
```

Fig. 9. A unit test for throw exceptions (fixed version).

Table 3
Statistics for manual validation, dynamic validation, and writing unit tests.

Description		Count
Initial number		**346**
Manual validation	Platforms bug	206
	Incomplete source code	35
	No fix code	27
After manual validation		**84**
Dynamic validation	Fixes not work	4
	Bug not as described	12
	Source code can not run	21
	The buggy version runs smoothly	5
After dynamic validation		**42**
Unit tests	Can not catch exceptions	3
	Output is `matplotlib` diagrams	4
	No output	2
	Output too complex	3
Final number		**30**

program bugs is unclear. In this case, we first tried to apply the most intuitive statement coverage to the bugs4Q benchmark. We selected a representative sample of 14 programs for validation. The statement coverage of the buggy programs and their unit tests are shown in Table 4. From the data of coverage, we can conclude three kinds of information:

- *The coverage of both source code and unit tests is 100%.* The program executed successfully and got the error output. From Fig. 6, we can see that there is no branching in the assertions on the output of the program. Therefore, the bugs must fall into the *wrong output* category.
- *Only the source code has been fully covered.* The program has multiple outputs resulting in the need for multiple assertion validation. Or the program throws an exception on the last line.
- *Neither the source code nor the unit tests are 100% covered.* The program runs interrupted and throws an exception.

Table 4
The coverage of source code and unit tests.

BugID	Source code coverage			Unit test coverage		
	Stmts	Miss	Cover	Stmts	Miss	Cover
No. 01	4	0	100%	13	1	92%
No. 07	28	0	100%	36	1	97%
No. 08	16	0	100%	23	1	96%
No. 10	4	0	100%	10	0	100%
No. 12	9	0	100%	20	7	65%
No. 17	11	0	100%	17	0	100%
No. 18	19	2	89%	19	2	89%
No. 20	15	2	87%	24	3	88%
No. 24	8	0	100%	14	2	86%
No. 25	24	0	100%	27	7	74%
No. 26	13	0	100%	20	0	100%
No. 28	4	1	75%	14	2	86%
No. 31	13	0	100%	19	0	100%
No. 39	17	0	100%	25	5	80%

Considering only the statements of the program, the effect of statement coverage in a quantum program is not much different from that of a traditional program. However, the source code and the unit tests contain traditional statements, and we could not screen them. For example, the unit test for bug `No. 1` has 9 traditional statement lines than the source code while the coverage is almost the same. Therefore, proposing a new statement coverage for quantum programs is necessary and remains challenging.

3.6. Bugs4Q benchmark framework

The construction of the Bugs4Q benchmark framework can be divided into three main steps: building the database, storing programs as modules, and implementing the user interface. As a result, the structure of the Bugs4Q framework is shown in Fig. 10.

3.6.1. Bugs4Q database

At first, we made our buggy programs public via the GitHub repository. The example bugs were added to our database as shown in Table 5. According to the source of the bug information (i.e., Github, StackOverflow, and Stack Exchange), we divided the bugs into three groups, respectively. In order to document each bug in detail, *Issue No* links to the original report. And we described each bug and linked it to a local file in our organization to make it easy for users to access the information of *Buggy, Fixed, Modify, and Test* directly.

The Bugs4Q database allows users to access the bugs we collected directly without downloading the framework. In addition, this makes building our underlying data store easier in the form of modules. Bugs4Q database also includes bugs of other quantum programming languages, which is publicly available at https://github.com/Z-928/Bugs4Q.

3.6.2. The construction of repositories of programs

We construct the Bugs4Q framework by constructing repositories to store all programs. Here we describe the design of these repositories. As shown in Fig. 10, each repository corresponds to a specific quantum programming framework. For example, the first repository of *Bug Repositories* contains all programs written in Qiskit.

In each repository, we use `Bug_ID`, a unique number, as the identifier for every bug in Bugs4Q. Each bug is encapsulated into a corresponding module for easy expansion. All the files related to a specific bug are put into one module. And the module number is the same as the `Bug_ID`. A module contains six parts:

Table 5
An example of the benchmark database for Bugs4Q.

Bug ID	Issue No	Buggy	Fixed	Modify	Status	Version	Type	Issue Registered	Issue Resolved
1	#5908	Buggy	Fixed	Mod	Resolved	0.17.0	Bug	Feb 26, 2021	Mar 1, 2021
2	#664	Buggy	Fixed	Mod	Resolved	0.4.1	Bug	Mar 19, 2020	Mar 25, 2020

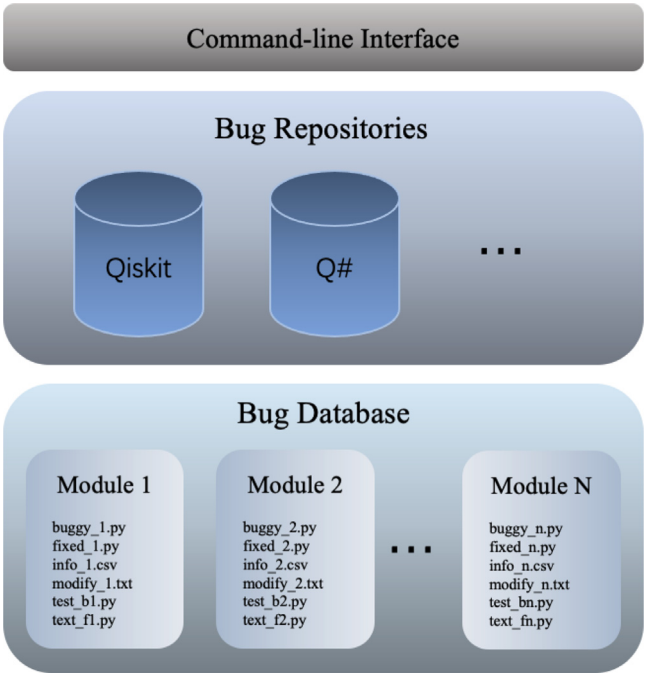

Fig. 10. The overall structure of Bugs4Q framework.

- `buggy`. The name of one buggy program. Such as, `buggy_1.py` represents the buggy version of the first program.
- `fixed`. The name of one fixed program. Such as `fixed_1.py` represents the fixed version of the program.
- `info`. The name of a CSV file which means the file contains information about the program. Such as `info_1.csv`.
- `modify`. The name of a text file. The file contains the result of comparing the buggy program and the fixed program, which is generated by using the diff command in Unix-like systems.
- `test_b`. The name of a Python file, which means the file is used to verify the bug's existence in the buggy version of the program.
- `test_f`. The name of a Python file, which means the file is used to verify the success of fixing the program bug.

There are 42 modules in the Bugs4Q framework. And all of them belong to the *qiskit* repository. We would like to collect bugs written in other quantum programming languages to enrich our repositories.

3.6.3. Command-line interface

After the benchmark program repository has been constructed, we need a convenient way to run these programs of the repositories. Therefore we developed a command-line interface for users of this benchmark. Our command-line interface program (`main.py`) is implemented with Python. We use the package `argparse` to deal with all operations related to command-line processing. To use the command-line interface, ensure the environment is set up in which Python 3.6 or above and the corresponding package are installed. So far, we have only created one repository for Qiskit. Repositories for other quantum programming frameworks are being constructed, and we will make them publicly available in the future.

Next, we briefly introduce the commands of Bugs4Q. Commands `python main.py -h` and `python main.py -help` can be used to get help. The command-line interface has four functions:

- **Info Command.** This command can be used to show information about the benchmark. The detailed description of all arguments of the info command is displayed if we type the following commands into the computer.
- **Checkout Command.** This command generates the source files for a given bug within a directory that one can specify.
- **Run Command.** The `run` command is used to run the buggy or fixed version of a given bug.
- **Test Command.** Test cases related to a given bug can be executed using the test command.

Bugs4Q framework simplifies the implementation of experimental tools in quantum software testing research. It has a uniform interface for checking out buggy and fixed program versions and provides uniform access to program information and source code. Besides, the Bugs4Q implementation framework has few requirements for the environment and is easy to use. It is also extensible because a bug's information can be integrated into a module so that new bugs can easily be added to the database as modules. The source code can be executed directly to support new testing and repair tools. The unit tests in Bugs4Q can provide direction for improving the unit testing schemes for quantum programs.

4. Evaluation and discussions

Next, we present the results of our evaluation of the performance of some existing testing tools based on Bugs4Q. Based on this, we discuss possible combinations between existing tools and the Bugs4Q framework and possible applications of Bugs4Q.

4.1. Evaluation of testing tools

Several test case generation methods for quantum programs have been proposed. For example, Quito (Wang et al., 2021b) provides three coverage criteria for quantum programs and their test generation strategies. QuSBT (Wang et al., 2021a) is a search-based testing method that designs 30 buggy versions for six quantum programs to demonstrate their effectiveness. In this section, we execute these two testing tools on programs in Bugs4Q. The experiments were conducted on a server with an Intel i9-10940X CPU, 128 G RAM, running on Ubuntu 20.04 with Python 3.10 installed. We set the parameters of each of the two tools to be the same for all the under-testing programs. The selected programs in Bugs4Q must meet two criteria: (1) The program needs to be fully executed, and the output exists. (2) The program meets the requirements to support the execution of Quito and QuSBT.

The experiment results are shown in Table 6. Firstly, `IC`, `OC`, and `IOC` represent the three coverage criteria in Quito, respectively. Next, in simple terms, $\texttt{fail}_{WOO}$ and $\texttt{fail}_{uof}$ represent the quantum programs failed by the wrong output value of the program while $\texttt{fail}_{OPO}$ and $\texttt{fail}_{wodf}$ represent the wrong probability

Table 6
Evaluation of test cases generation tools.

Bugs4Q	$Quito_{IC}$			$Quito_{OC}$			$Quito_{IOC}$			QuSBT		
Bug ID	tests	$fail_{OPO}$	$fail_{WOO}$	tests	$fail_{OPO}$	$fail_{WOO}$	tests	$fail_{OPO}$	$fail_{WOO}$	tests	$fail_{wodf}$	$fail_{uof}$
buggy_10	400	0	0	400	0	0	400	0	0	500	0	0
buggy_12	800	16	0	4000	16	0	4000	16	0	500	500	0
buggy_17	1	0	1	1	0	1	1	0	1	500	0	500
buggy_21	1	0	1	1	0	1	1	0	1	500	1	446
buggy_25	1	0	1	1	0	1	1	0	1	500	0	500
buggy_26	2	0	1	5	0	1	1	0	1	500	0	500
buggy_31	1	0	1	1	0	1	1	0	1	500	0	391
buggy_39	3200	256	0	4000	256	0	3200	256	0	500	500	0

distribution of output values. Finally, the number of tests generated by QuSBT is set manually. As a result, both of the testing tools have well found the test cases that lead to program failures. From `buggy_10`, we can find the bugs would not affect the output values of the program. It may only lead to circuit diagram generation errors. The results of `buggy_12` and `buggy_39` show that QuSBT performs better in finding the wrong probability distribution. And for other defective programs, Quito gives a more intuitive result for value errors, which is better in terms of efficiency. In addition, QuSBT would give the optimal solution for each program in the generated test cases.

4.2. The combination between existing tools and Bugs4Q

In addition to Quito and QuSBT, several other works have contributed to the advancement of quantum program testing. QuCAT (Wang et al., 2021) provides two schemes for generating combinatorial test suites and argues that the more intense combinatorial tests are superior in effectiveness to the less intense ones. Muskit (Mendiluze et al., 2021) is a quantum mutation analysis tool for the Qiskit language, focusing on mutation operators for quantum gates. QMutPy (Fortunato et al., 2022b) can generate effective mutation programs for *measurement* calls and a large number of quantum gates. Fortunato et al. in their case study demonstrated the validity of QMutpy and indicated that mutants in QMutPy matching real-world bugs would be available to other quantum languages.

Given the practical benefits of these tools, we would like to combine them with Bugs4Q:

- It is possible to further apply Quito, QuCAT, and QuSBT to Bugs4Q programs, thus adding more possibilities for testing real-world Qiskit programs. And the only major challenge is to extend these tools to support quantum programs with different coding styles.
- QMutPy gives mutation scores from program source files and unit test files, which Bugs4Q can provide. And the only challenge is to modify the programs and unit tests in a way that MutPy can support. On the other hand, Muskit defines selection criteria to reduce the number of mutants generated and simplify test analysis. Therefore, if the mutation tool can match all programs of Bugs4Q, it will further facilitate the mutation testing of real-world quantum program bugs.

As an increasing number of tools and methods for testing quantum programs become available, there is an urgent need for a benchmark of real bugs generated by programmers during the practical development of quantum software to enable the evaluation and integration of these tools and methods. The Bugs4Q framework currently supports and requires integrating test methods and coverage criteria. We are also keen to apply these testing tools to Bugs4Q.

4.3. Possible uses of Bugs4Q

We next discuss some possible uses of Bugs4Q in quantum software engineering activities.

4.3.1. Testing of quantum programs

In quantum program testing, challenges remain in generating valid test cases for real-world bugs. Bugs4Q can assist with research related to quantum program testing. In detail, the source code of buggy programs can be used to help test tools measure their effectiveness against real-world bugs. The Bugs4Q framework facilitates researchers to provide APIs for testing, code coverage, etc. Modular handling of bugs makes the Bugs4Q framework well-extensible. The unit tests and statement coverage for each bug make comparing different testing methods and coverage criteria easy. As most bugs we collected are related to quantum properties, efficient testing methods are urgently needed to detect them.

4.3.2. Quantum program analysis

During the execution of quantum programs, we cannot read the internal state of qubits due to the non-cloning principle, and the measurement of qubits will destroy the state of qubits, so the running cost of dynamic techniques will be relatively high. On the other hand, due to the unique nature of quantum programs, the existing static analysis tools for classical programs are insufficient to support the analysis of quantum programs, and we need to develop new methods for the analysis of quantum programs. Using the bug information provided in Bugs4Q, we can identify and summarize the bug patterns in quantum programs, and this information can be used to develop practical analysis tools to detect and prevent bugs in quantum programs. Researchers can also use Bugs4Q as a benchmark to evaluate the effectiveness of static analysis tools for quantum programs. We have started to conduct preliminary research in this area (Zhao et al., 2023).

4.3.3. Bug localization of quantum programs

The Bugs4Q benchmark is an essential resource for developing bug localization tools for quantum software. Its diverse and standardized set of bugs enables researchers to evaluate the effectiveness of various bug localization techniques. Bugs4Q allows researchers to compare and contrast different methods, identify their strengths and weaknesses, and improve upon them by providing a testbed for new bug localization methods. Bugs4Q also encourages the development of new approaches better suited to the unique challenges of quantum programming. We believe that the Bugs4Q framework for evaluating and improving bug localization techniques may potentially accelerate the development of high-quality quantum software.

4.3.4. Automatic repair of quantum programs

The Bugs4Q benchmark contains a diverse set of quantum-specific bugs that can be used to evaluate the effectiveness of repair techniques for quantum software. It includes a range of bug types, such as initialization errors, measurement errors, and incorrect use of quantum gates, making it a useful tool for testing the resilience of repair techniques in the face of multiple types of bugs. The benchmark also provides the source code of both the buggy and fixed programs, allowing developers to verify repaired code and compare repair techniques. Currently, there is a lack of automatic repair techniques for quantum programs. However, recent work on mutation analysis (Fortunato et al., 2022b; Mendiluze et al., 2021; Fortunato et al., 2022a) and assertion-based techniques (Huang and Martonosi, 2019; Li et al., 2020, 2019; Zhou and Byrd, 2019; Liu et al., 2020; Singhal, 2019) offers promising approaches for developing such techniques. By providing a framework for evaluating these state-of-the-art methods, Bugs4Q can guide and support the development of repair technologies for quantum software.

5. Threats to validity

In this section, we consider the threats to the validity of our work from both external and internal perspectives. Accompanied by the verifiability of bugs4Q.

5.1. External threats

External threats are mainly in the form of limits on the number of bugs. Constructing the Bugs4Q benchmark, we found that the most significant limitation currently is the need for more quantum software projects (programs). Although some quantum programming languages (Svore et al., 2018; Steiger et al., 2018; Google A.I. Quantum team, 2018; Bichsel et al., 2020) have emerged, we tried to collect as many bugs as possible from projects developed in these languages, and we found that many of them are not filtered enough to create a benchmark. In addition to the Qiskit language, we found similar bugs in other languages as in Qiskit. Table 7 shows the number of bugs in programs developed in several common quantum programming languages. All bugs are accompanied by their corresponding source code, and the corresponding fixes have been submitted. Among them, the *Initial number* refers to the number of buggy programs with fixes before verification, and the *Final number* refers to the number of buggy programs that can be reproduced after manual and dynamic verification.

In summary, even for the most widely used Qiskit quantum programming language, there are still not enough bugs for research. Moreover, existing quantum programs are usually run on simulators rather than on real quantum computers, which leads to the small size of current quantum programs. As a result, the collection of 42 buggy programs for Qiskit that we have discussed in this paper is already the largest and most typical of the buggy programs. This limitation will gradually be lifted as quantum programming languages become more widely used. And we will continue to collect new bugs and update Bugs4Q in our future work.

5.2. Internal threats

Although we have tried our best to collect as many bugs as possible, due to the current number and size of quantum programs, we have not comprehensively collected all possible types of bugs in quantum programs, which requires our continuous attention in future research. In addition, although we have successfully reproduced the bugs we have collected, it is difficult to write corresponding unit tests for some bugs. For example, Qiskit has internally caught *Exceptions*, which leads to our unit tests being unable to catch the bug.

Table 7
The bug numbers in common quantum programming languages.

Language name	Initial number	Final number
Cirq	20	7
Q#	21	2
ProjectQ	3	0
ScaffCC	1	0
Total	44	9

Regarding the bugs themselves, we have yet to determine if some bugs in Bugs4Q are related to quantum features. For example, for the bug type *output wrong*, some programs do not use the simulator for execution but only draw a complete circuit diagram as output. However, the behavior of the output (for example, in Qiskit) is to call a Python method that draws the circuit diagram by string. We consider it a bug if the output circuit diagram does not match the developer's expectations. Furthermore, the output should be a complete measurement of the quantum program. Such a bug is necessarily related to quantum, and we combine Qiskit's `QiskitTestCase` class with Python's `unittest` module to write unit tests. Throwing exceptions is another bug type that we conclude cannot be detected by applying existing quantum program testing techniques.

Currently, the Bugs4Q framework only provides full support for bugs in Qiskit, while bugs in other quantum programming languages are only supported for their storage in the Bugs4Q database. To cover other quantum programming languages with our Bugs4Q framework, we need to collect enough information about the bugs associated with these languages. In addition, we need to integrate the language environment required to execute these buggy programs into the Bugs4Q framework. In addition, our benchmark framework has not been able to come up with more efficient API interfaces to implement more features like *test suite operations*, *test generation*, *variation analysis*, and *code coverage analysis* (Just et al., 2014). In addition, almost all the quantum programs we studied were executed based on the simulator. In future work, we hope to collect bugs from programs executed with quantum computers.

5.3. Verifiability

This threat concerns the possibility of replicating this research. We try to provide all the necessary details to help researchers replicate this work. The replication package is made publicly available at https://github.com/Z-928/Bugs4Q-Framework.

6. Related work

We next present some related work in the field of bug benchmarks.

6.1. Bug benchmark suite for classical software

Many benchmark suites have been proposed to evaluate debugging and testing methods for classical software. The Siemens test suite (Hutchins et al., 1994) 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 actual bugs and fixes is SIR (Software Artifact Infrastructure Repository) (Do et al., 2005), which enables reproducibility in software testing research. SIR contains multiple versions of Java, C, C++, and C# programs which consist of test

suites, data of bugs, and scripts. Other benchmark suites include Defects4J (Just et al., 2014) and iBug (Dallmeier and Zimmermann, 2007) for Java, BenchBug (Lu et al., 2005), ManyBug (and InterClass) (Le Goues et al., 2015), and BugsJS (Gyimesi et al., 2019) for JavaScirpt projects. However, all the benchmarks mentioned above focus on classical software systems and, therefore, cannot be used for evaluating and comparing quantum software debugging and testing methods.

6.2. Bug benchmark suite for quantum software

Perhaps, the most related work to ours is QBugs proposed by Campos and Souto (2021), a collection of reproducible bugs in quantum algorithms for supporting controlled experiments for quantum software debugging and testing. QBugs offers some initial ideas on building a benchmark for an experimental infrastructure to support the evaluation and comparison of new research and the reproducibility of published results on quantum software engineering. It also discusses some challenges and opportunities in the development of QBugs. Bugs4Q benchmark suite, on the other hand, aims to construct a bug benchmark suite of actual bugs derived from real-world quantum programs for quantum software debugging and testing, with real-world test cases for reproducing the buggy behaviors of identified bugs.

7. Concluding remarks

As quantum computers gradually come into the limelight, quantum programs have intensified, with analysis and testing techniques becoming an essential part of the process. This paper proposes Bugs4Q, a benchmark of forty-two real, manually validated Qiskit bugs supplemented with tests to reproduce buggy behaviors. Bugs4Q also provides a user-friendly and scalable implementation framework for accessing the buggy and fixed versions of the Qiskit programs and executing the corresponding unit tests, facilitating the reproducible empirical studies and comparisons of Qiskit analysis and testing tools.

In our future work, we would like to keep updating the Bugs4Q benchmark and improve the tests to reproduce more bugs in Qiskit. Our benchmark will be continuously maintained on an ongoing basis. With the version update of quantum platforms and new test methods proposed, we will continue updating our database and extending our framework. In addition, we would like to extend Bugs4Q Framework to other common quantum programming languages to support more bugs and propose new methods for testing and debugging quantum programs.

CRediT authorship contribution statement

Pengzhan Zhao: Project administration, Writing – original draft, Writing – review, Data curation, Validation, Investigation. **Zhongtao Miao:** Software, Data curation. **Shuhan Lan:** Validation, Investigation. **Jianjun Zhao:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have made my data/code published at:https://github.com/Z-928/Bugs4Q-Framework.

References

Abhari, A.J., Faruque, A., Dousti, M.J., Svec, L., Catu, O., Chakrabati, A., Chiang, C.-F., Vanderwilt, S., Black, J., Chong, F., 2012. Scaffold: Quantum Programming Language. Technical Report, Department of Computer Science, Princeton University.

Abreu, R., Fernandes, J.P., Llana, L., Tavares, G., 2022. Metamorphic testing of oracle quantum programs. In: 2022 IEEE/ACM 3rd International Workshop on Quantum Software Engineering. (Q-SE), IEEE, pp. 16–23.

Aleksandrowicz, G., Alexander, T., Barkoutsos, P., Bello, L., Ben-Haim, Y., Bucher, D., Cabrera-Hernández, F., Carballo-Franquis, J., Chen, A., Chen, C., et al., 2019. Qiskit: An open-source framework for quantum computing. 16, Accessed on: Mar.

Ali, S., Arcaini, P., Wang, X., Yue, T., 2021. 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, pp. 13–23.

Allen, E., 2002. Bug Patterns in Java. APress L. P..

Bichsel, B., Baader, M., Gehr, T., Vechev, M., 2020. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 286–300.

Campos, J., Souto, A., 2021. 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.

Dallmeier, V., Zimmermann, T., 2007. Extraction of bug localization benchmarks from history. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. pp. 433–436.

Do, H., Elbaum, S., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. 10 (4), 405–435.

Fortunato, D., Campos, J., Abreu, R., 2022a. Mutation testing of quantum programs: A case study with qiskit. IEEE Trans. Quantum Eng. 3, 1–17.

Fortunato, D., Campos, J., Abreu, R., 2022b. Mutation testing of quantum programs written in QISKit. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. pp. 358–359.

Gely, M.F., Steele, G.A., 2020. QuCAT: quantum circuit analyzer tool in python. New J. Phys. 22 (1), 013025.

Google A.I. Quantum team, 2018. Cirq. URL https://github.com/quantumlib/Cirq.

Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, A., Ferenc, R., Mesbah, A., 2019. Bugsjs: a benchmark of JavaScript bugs. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification. ICST, IEEE, pp. 90–101.

Häner, T., Steiger, D.S., Smelyanskiy, M., Troyer, M., 2016. High performance emulation of quantum circuits. In: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, pp. 866–874.

Honarvar, S., Mousavi, M., Nagarajan, R., 2020. Property-based testing of quantum programs in q#. In: First International Workshop on Quantum Software Engineering. (Q-SE 2020).

Huang, Y., Martonosi, M., 2019. Statistical assertions for validating patterns and finding bugs in quantum programs. In: Proceedings of the 46th International Symposium on Computer Architecture. pp. 541–553.

Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In: Proceedings of 16th International Conference on Software Engineering. IEEE, pp. 191–200.

Just, R., Jalali, D., Ernst, M.D., 2014. 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. pp. 437–440.

Koch, D., Wessing, L., Alsing, P.M., 2019. Introduction to coding quantum algorithms: A tutorial series using pyquil. arXiv preprint arXiv:1903.05195.

Le Goues, C., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W., 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Trans. Softw. Eng. 41 (12), 1236–1256.

Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y., 2019. Poq: projection-based runtime assertions for debugging on a quantum computer. arXiv preprint arXiv:1911.12855.

Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y., 2020. Projection-based runtime assertions for testing and debugging quantum programs. Proc. ACM Programm. Lang. 4 (OOPSLA), 1–29.

Liu, J., Byrd, G.T., Zhou, H., 2020. 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. pp. 1017–1030.

Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., Zhou, Y., 2005. Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the Evaluation of Software Defect Detection Tools, Vol. 5.

McKay, D.C., Alexander, T., Bello, L., Biercuk, M.J., Bishop, L., Chen, J., Chow, J.M., Córcoles, A.D., Egger, D., Filipp, S., et al., 2018. Qiskit backend specifications for openqasm and OpenPulse experiments. arXiv preprint arXiv:1809.03452.

Mendiluze, E., Ali, S., Arcaini, P., Yue, T., 2021. Muskit: A mutation analysis tool for quantum software testing. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1266–1270.

Miranskyy, A., Zhang, L., 2019. On testing quantum programs. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results. (ICSE-NIER), IEEE, pp. 57–60.

Miranskyy, A., Zhang, L., Doliskani, J., 2020. Is your quantum program bug-free? arXiv preprint arXiv:2001.10870.

Nielsen, M.A., Chuang, I., 2002. Quantum Computation and Quantum Information. American Association of Physics Teachers.

Paltenghi, M., Pradel, M., 2021. Bugs in quantum computing platforms: An empirical study. arXiv preprint arXiv:2110.14560.

Patel, T., Tiwari, D., 2021. Qraft: reverse your quantum circuit and know the correct program output. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 443–455.

Research, I., 2020. IBM quantum experience. Accessed on: April, URL https://quantum-computing.ibm.com/docs/.

Research, I., 2021. Qiskit. Accessed on: June, URL https://qiskit.org.

Singhal, K., 2019. Hoare types for quantum programming languages.

Steiger, D.S., Häner, T., Troyer, M., 2018. ProjectQ: An open source software framework for quantum computing. Quantum 2, 49.

Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M., 2018. Q#: enabling scalable quantum computing and development with a high-level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. pp. 1–10.

Wang, X., Arcaini, P., Yue, T., Ali, S., 2021. Application of combinatorial testing to quantum programs. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 179–188.

Wang, X., Arcaini, P., Yue, T., Ali, S., 2021a. Generating failing test suites for quantum programs with search. In: International Symposium on Search Based Software Engineering. Springer, pp. 9–25.

Wang, X., Arcaini, P., Yue, T., Ali, S., 2021b. Quito: A coverage-guided test generator for quantum programs. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1237–1241.

Wang, J., Gao, M., Jiang, Y., Lou, J., Gao, Y., Zhang, D., Sun, J., 2018. QuanFuzz: Fuzz testing of quantum program. arXiv preprint arXiv:1810.10310.

Zhao, J., 2020. Quantum software engineering: Landscapes and horizons. arXiv preprint arXiv:2007.07047.

Zhao, P., Wu, X., Li, Z., Zhao, J., 2023. Qchecker: Detecting bugs in quantum programs via static analysis. In: 2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering. (Q-SE 2023), IEEE.

Zhao, P., Zhao, J., Miao, Z., Lan, S., 2021. Bugs4Q: A benchmark of real bugs for quantum programs. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021), New Ideas and Emerging Results Track. IEEE, pp. 1373–1376.

Zhou, H., Byrd, G.T., 2019. Quantum circuits for dynamic runtime assertions in quantum computation. IEEE Comput. Archit. Lett. 18 (2), 111–114.