Identifying and Detecting Bugs in Quantum Programs

趙, 鵬展

Identifying and Detecting Bugs in Quantum Programs

Pengzhan Zhao

Graduate School of Information Science and Electrical Engineering

Kyushu University

A thesis submitted for the degree of

Doctor of Philosophy in Engineering

July 2023

Abstract

Quantum programming involves designing and constructing executable quantum programs to achieve specific computational results. Several quantum programming languages, such as Scaffold, Quipper, Qiskit, Q#, and Cirq, enable researchers and developers to implement and experiment with quantum computing techniques efficiently. However, ensuring the correctness of quantum programs is challenging due to unique features like superposition, entanglement, and no-cloning, which make bug detection difficult.

Although some approaches for debugging and testing quantum software have been proposed, the research in identifying and detecting bugs in quantum programs is still in its early stages. Existing research has shortcomings in the following areas: (1) Redundancy in Bug Patterns: Many publicly reported bugs follow similar patterns, necessitating proper bug pattern classification metrics to assist developers in understanding and avoiding these bugs effectively. (2) Lack of Suitable Bug Benchmark Suites: The absence of comprehensive bug benchmark suites restricts the systematic evaluation of debugging and testing methods for quantum programs, hindering the progress of quantum software development. (3) Challenges of Dynamic Techniques and Classic Static Analysis: Dynamic techniques relying on the execution of quantum programs can be cumbersome and expensive, while classic static analysis tools designed for traditional programs are unsuitable for analyzing quantum programs due to their unique mechanisms.

To address these challenges and improve the quality assurance of quantum software, this thesis primarily focuses on identifying and detecting bugs in quantum programs. The work is divided into three phases: (1) Identification and Categorization of Bug Patterns: The thesis identifies and categorizes bug patterns in the Qiskit quantum programming language. This provides researchers and programmers with a clear understanding of the types of bugs that can occur in quantum programs and how to detect them. The bug patterns cover quantum-related constructs, focusing on symptoms, root causes, cures, and preventions. (2) Bugs4Q Benchmark Suite: As an initial step towards evaluating debugging and testing tools for quantum software, the thesis introduces Bugs4Q. This benchmark suite consists of forty-two real Qiskit bugs obtained from popular platforms like GitHub, StackOverflow, and Stack Exchange. Bugs4Q includes a bug database and test cases to reproduce faulty behaviors. Researchers have access to the actual bugs and corresponding fixes for further study. The

Bugs4Q framework modularizes bugs and offers a user-friendly interface for easy extension. The buggy programs are also evaluated with two test case generation tools, demonstrating the benchmark's effectiveness in evaluating quantum program testing techniques. (3) QChecker: To efficiently detect bugs in quantum programs, the thesis presents QChecker, a static analysis tool specifically designed for Qiskit. QChecker comprises two modules: one for extracting program information based on the abstract syntax tree (AST) and another for detecting bugs based on patterns. The performance of QChecker is evaluated using the Bugs4Q benchmark suite, demonstrating its ability to detect various bugs in quantum programs effectively.

In summary, this thesis fills the gaps in identifying and detecting bugs in quantum programs by (1) providing bug patterns as a foundation for further research in debugging and testing quantum programs, (2) introducing Bugs4Q, a comprehensive benchmark suite that offers a comprehensive view of quantum bugs and facilitates the evaluation of quantum debugging and testing techniques, and (3) developing QChecker, a static analysis tool that efficiently detects bugs in quantum programs.

Acknowledgments

I am grateful to Kyushu University and the Graduate School of Information Science and Electrical Engineering for funding and supporting me throughout this work.

I would like to express my sincere thanks to the members of the defense committee: Dr. Ashir Ahmed, Dr. Jianjun Zhao, and Dr. Shaoying Liu. Your expertise, time, and constructive feedback have been invaluable to me. Thank you all for your dedication and guidance.

I am deeply indebted to my supervisor, Dr. Jianjun Zhao. My first glimpse of the research was enlightened by his enthusiasm, inspiration, and tremendous research efforts. His thoughtful guidance, stimulating suggestions, and warm encouragement helped me in all of the research during my Ph.D.

Extraordinary gratitude goes to all my friends and colleagues in Lab 714 and Lab 726 for their support, all the good times we had together, and our collaborative exchanges over the years. I would also like to express my gratitude to Dr. Lei Ma, Dr. Fuyuan Zhang, and Dr. Zhenya Zhang for their support and guidance.

In addition, I would like to express my special thanks to the Japanese Government (MEXT) Scholarship for financial support.

Lastly, and most importantly, I am forever grateful to my family for their understanding, endless patience, and encouragement when it was most required.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Quantum programming is designing and constructing executable quantum programs to achieve a specific computational result. It has been applied to many cutting-edge areas, such as quantum machine learning [7, 14], big data analysis [45], and molecular simulations [20] due to its unique promising advantage over classical computing. Several quantum programming approaches have recently been available for writing quantum programs, for instance, Qiskit [47], Q# [52], ProjectQ [22], and Scaffold [1]. Given the importance and wide application of quantum programming, ensuring the correctness of quantum programs is crucial for quantum software development. Along with the emergence and advancement of quantum programming languages, debugging and testing quantum programs are gaining more attention.

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 [5], 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 up-to-date and real-world bug benchmark suites so potential users can know how well they work.

The specific features of superposition, entanglement, and no-cloning introduced in quantum programming make it difficult to find bugs in quantum programs [41]. Several approaches have been proposed for debugging and testing quantum software [2, 4, 23, 26, 32, 40] recently, but the debugging and testing remain challenging issues for quantum software [40, 64]. In addition, recent empirical studies [10, 43, 65, 66] have shown that the current quantum program development process is still error-prone. While debugging and testing quantum programs has gained significant attention [4, 16, 23, 26, 32, 34], the existing debugging and testing techniques often require dynamic execution of the underlying quantum programs. Considering that most current quantum programs are executed on quantum computers and simulators available on the cloud, existing techniques can be cumbersome and expensive. In summary, there is a lack of systematic identifications of quantum program bugs and efficient detecting techniques:

- There is still a lack of research on identifying bug patterns in quantum programs, even though a few approaches have been proposed for testing and debugging quantum software [4, 23, 26, 32, 33, 40, 41, 55]. As more publicly reported bugs emerge, many are redundant because they follow a similar pattern. In this case, we may not know what types of bugs are unique or commonly happen to quantum programs without a proper bug pattern classification, which poses several restrictions on the research and development of programs in the language.

- More and more methods and tools have appeared for quantum program testing or debugging [18, 39, 44, 55, 58, 60], which have made some progress in the field of quantum software, while there are few bug benchmark suits for 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.
- In classical software development practice, static analysis techniques have been widely used to detect various types of bugs in classical programs due to their advantages in speed and cost [15, 24, 48, 49, 56]. However, detecting bugs in quantum programs via static analysis can be challenging. Since quantum computation logic is expressed in quantum circuits, and the states of quantum registers are measured probabilistically, static analysis tools designed for classical programs struggle to detect mistakes in quantum programs.

1.2 Contribution

To verify the correctness of quantum programs more efficiently and to improve the quality assurance of quantum software, this doctoral thesis mainly focuses on implementing the identification and detection of quantum program bugs. As the first step toward evaluating quantum software debugging and testing tools, we identify and categorize some bug patterns in the quantum programming language Qiskit and briefly discuss how to eliminate or prevent those bug patterns. This is the first step in providing a fundamental basis for detecting quantum program bugs. Then we introduce 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. Finally, we present QChecker, a static analysis tool designed to detect quantum program bugs, especially for Qiskit.

The contribution of our work is as follows:

- **Bug Pattern.** We identify and categorize some bug patterns in the quantum programming language Qiskit to provide both researchers and programmers with a clear view of what kind of bugs may happen in quantum programs and how to detect them. We also provide an example for each bug pattern to illustrate the pattern's symptoms. Furthermore, information on bug patterns provides a basis for further research on debugging quantum programs. It also provides insight into the possible consequences of different types of bugs and summarizes the common behaviors among similar ones. Finally, these bug patterns can be used to recognize faults that have already existed and prevent potential bugs.

- **Bugs4Q.** We thoroughly collected quantum programs on common quantum programming languages to enrich the Bugs4Q Repository. All the bugs verified from 10069 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.

- **QChecker.** We present the first bug detection tool dedicated to quantum programs in Qiskit. Using static analysis techniques, QChecker can generate diagnostic messages that assist developers in pinpointing potential bugs in their programs quickly. We implement QChecker and evaluate its effectiveness and performance in a real-world Bugs4Q benchmark. The results show that QChecker can effectively detect various types of bugs in quantum programs.

1.3 Thesis Structure

The rest of this Thesis is organized as follows:

- Chapter 2 introduces some background information about quantum programming and presents related works in the field of bug identification and detection.
- Chapter 3 describes our identification of bug patterns in quantum programs.
- Chapter 4 introduces the Bugs4Q benchmark suite. Including the collection and verification of real-world quantum program bugs. As well as the construction and evaluation of bugs4Q.
- Chapter 5 describes QChecker, a bug detection tool dedicated to quantum programs, and presents the performance of QChecker on Bugs4Q.
- Chapter 6 is the concluding remarks of this thesis, followed by our future work.

Chapter 2

Background and Related Work

2.1 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.

2.1.1 Qiskit

Several open-source programming frameworks, such as Qiskit [47], Q# [52], Scaffold [1] and ProjectQ [22], 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 [30]. As a Python package, it provides tools to create and manipulate quantum programs running on prototype quantum devices and simulators [3]. 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 experiment sequences. The object could be used to get status information and retrieve results [38]. Figure 2.1 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 completing the experimental design, the instructions are run through the `execute` method. The `shots` of the simulation 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.

2.1.2 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 0 with probability $|\alpha|^2$ or the 1 with probability $|\beta|^2$.

```
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)
```

Figure 2.1: A sample quantum program in Qiskit.

Quantum gates are used for quantum computation and the manipulation of 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 and target qubits. 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 [42] explains quantum computation more deeply.

2.1.3 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:

- **I**nitialization: The initial stage is initializing the quantum registers to store the qubits that need manipulation. Then the classical registers are initialized to store the values of the measured qubits. As shown in Figure 2.1, `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 $|000\rangle$.

- **G**ate Operation: The core of quantum computing is to operate on qubits. Qiskit provides almost all the gates to implement algorithms in quantum programs [46]. 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.

- **M**easurement: 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 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 Figure 2.1 is `circuit.measure([0,1,2], [0,1,2])`.

- **D**eallocation: 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 thesis uses these four aspects to measure whether a quantum program is qualified. 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 helpful for manual verification in our work. This criterion is also helpful in static analysis and bug localization.

2.2 Related Work

2.2.1 Bug Pattern

The previous research on bug patterns is mainly focused on classical programming languages. Allen [5] summarizes more than 14 bug pattern categories in Java. Following Allen's work, Hovemeyer and Pugh [24] present a novel syntactic pattern-matching approach to detecting the bug patterns in Java and implement a bug-finding tool called FindBugs [6]. Zhang and Zhao [63] and Shen [49] present some bug patterns for AspectJ and develop a tool called *XFindBugs* to detect bug patterns in AspectJ. Our work extends the bug patterns research and identification to quantum programming languages with the Qiskit language. The bug patterns presented in this paper are different in nature from the existing bug patterns in classical programming languages because they explicitly involve quantum programming language features such as superposition, entanglement, and no-cloning.

Huang and Martonosi [25, 26] study the bugs for special quantum programs to support quantum software debugging. Based on the experiences of implementing several quantum algorithms, several types of bugs specific to quantum computing are identified. These bugs include incorrect quantum initial values, incorrect operations and transformations, the incorrect composition of operations using iteration, recursion, mirroring, incorrect classical input parameters, and incorrect deallocation of qubits. The defense strategy for each bug type is also proposed, mainly using some assertions to detect the bugs in runtime. While Huang and Matonosi's work targets quantum software debugging, which involves primarily the runtime execution of a program, our work targets identifying the bug patterns to support bug detection through static analysis, which may not need to execute the program and, therefore, could be more efficient.

2.2.2 Bug Benchmark Suite

Many benchmark suites have been proposed to evaluate debugging and testing methods for classical software. The Siemens test suite [27] 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) [13], 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 [29] and iBug [11] for Java, BenchBug [35], ManyBug (and InterClass) [31], and BUGSJS [21] 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.

Perhaps, the most related work to ours is QBugs proposed by [10], 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.

2.2.3 Static Analysis Technique

A large number of error detection techniques based on static analysis [15, 24, 48, 49, 56] have been developed in classical software development. However, static analysis techniques for classical programs are difficult to apply directly to quantum programs due to the essential differences between quantum programs and classical programs.

Currently, static analysis techniques for quantum programs have emerged. Yu and Palsberg [62] proposed an abstract interpretation technique for quantum programs and used this technique to detect assertions to find errors in the programs. Xia and Zhao [59] proposed a practical static entanglement analysis technique to accurately analyze the entanglement information within and between modules in Q# quantum programs, which can help find errors related to entanglement in the programs. ScaffCC [28] is a scalable compiler framework for the quantum programming language Scaffold [1], supporting entanglement analysis. ScaffCC explores data-flow analysis techniques to automatically track the entanglements within the code by annotating the output of the QASM-HL program, an intermediate representation of ScaffCC, to denote possibly entangled qubits. The analysis is conservative because it assumes that if two qubits interact, they are likely to have become entangled with each other. In contrast to these analysis methods, QChecker aims to find bugs in quantum programs based on bug patterns through static analysis.

Researchers have also extended Hoare logic to the quantum domain to support the formal verification of quantum programs [9, 12, 54, 61, 68]. Among them, Li *et al.* [68] introduced applied quantum Hoare logic (aQHL), which is a simplified version of quantum Hoare logic (QHL) [61], with particular emphasis on supporting debugging and testing of the quantum programs. aQHL simplified QHL through binding QHL to a specific class of pre- and postconditions (assertions), that is, projections, to reduce the complexity of quantum program verification and provide a convenient way to debug and test quantum software. However, as we know, formal verification can be costly and difficult to scale up.

Chapter 3

Identifying Bug Patterns in Quantum Programs

3.1 Overview

Bug patterns are erroneous code idioms or bad coding practices that have been proven to fail repeatedly. These are usually caused by misunderstanding a programming language's features, using erroneous design patterns, or simple mistakes sharing common behaviors. There are several limitations without proper bug pattern identification:

- Developers may not know what type of bugs are most likely to happen in a program and therefore do not know how to prevent them. In other words, a programmer would lack fundamental knowledge on how to write bug-free code.
- Testers do not know how to write adequate test cases to cover the most common potential errors effectively. The tester can only set up criteria for better addressing the specific bugs when understanding how the common bugs happened in programs.

- Software maintenance staff do not know which language features are more likely to result in the incorrect code, so they cannot clearly view the current system when doing the maintenance tasks.

This chapter chooses the widely used quantum programming language Qiskit as our target language and identifies the common bug patterns in Qiskit programs. The bug patterns presented in this chapter may help solve the abovementioned problems. Identifying such patterns in a quantum programming language can help programmers improve their productivity in finding bugs and reduce software maintenance costs. Bug pattern identification can also help language designers and tool developers develop efficient bug-finding techniques to locate bugs in quantum programs' source code through program analysis.

The rest of this chapter is organized as follows. Section 3.2 describes the identified bug patterns in Qiskit in detail. A catalog of bug patterns in Qiskit is described in section 3.3. Section 3.4 introduces the threats to the validity of this work. And concluding remarks are given in Section 3.5.

3.2 Bug Patterns in Quantum Programs

We next introduce six bug patterns in Qiskit as examples. When introducing each bug pattern, we also show an example that contains this specific pattern. Since most bug patterns have some representation variants and alternatives, we choose the one that appears to be the most generally applicable.

```
qreg = QuantumRegister(3)
creg = ClassicalRegister(2)
circuit = QuantumCircuit(qreg, creg)

circuit.h(0)
circuit.cx(0, 1)
circuit.cx(1, 2)
circuit.measure([0,1,2], [0,1,2])
```

Figure 3.1: Unequal classical bits and qubits

3.2.1 Unequal Classical Bits and Qubits

In Qiskit, each classical bit in the classical register stores a measured qubit value. Therefore, it is better to initialize quantum registers of the same size as classical registers. Otherwise, the bug pattern of "*Unequal Classical Bits and Qubits*" may occur, especially when the number of qubits in the quantum register is greater than that of classical bits in the classical register. From the point of view of program integrity, every used qubit should be measured. As shown in Figure 3.1, when we want to measure the third qubit, we receive an error message *CircuitError: 'Index out of range.'* If we do not measure one of the qubits, a qubit will not get reset.

Another case is that the number of bits in the classic register is larger than the qubit. Unless we encounter the need to use multiple classical bits to store a qubit measurement, otherwise, this is not a good habit. On the one hand, resources are wasted when the program is actually developed, and on the other hand, outputting all classic bits will cause messy results. Therefore we do not recommend this operation.

```
qc = QuantumCircuit(3,3)
gt = Gate('my_custom_gate', 3, [])

qc.h(0)
qc.sdg(0)
qc.y(1)

qc.append(gt, [0,1,2])
qc.add_calibration(gt, [0,1,2], schedule)
qc = transpile(qc, backend,
               basis_gates=['u1','u2','cx', gt])
qc.measure([0,1,2], [0,1,2])
```

Figure 3.2: Custom gates not recognised by Qiskit

3.2.2 Custom Gates not Recognised

When defining a custom gate in a program, some programmers will want to define a basic gate that directly controls more than two qubits; the bug pattern *Custom gates not recognized by Qiskit* may occur. This pattern refers to a custom gate that does not use the gate class provided by Qiskit correctly. Alternatively, the gate is not recognized by Qiskit. An example of an error code is shown in Figure 3.2, which is a program that tends to define a three-qubit controlled gate. The user defines a gate named `my_custom_gate` using the `Gate` method and controls the number of qubits to three. When we call this gate, the program will have an error. Because in `basic_gates`, the custom gate `gt` is not the same as other Qiskit-based gates.

This bug pattern is mainly caused by programmers who do not really understand quantum gates. Quantum gates can only control a maximum of 2 qubits and are known

```
num_qubits = 1
tl_circuit = TwoLocal(num_qubits, ['h', 'rx'], 'cz',
                      entanglement='full', reps=3,
                      parameter_prefix = 'y')
tl_circuit.draw(output = 'mpl')
```

Figure 3.3: Insufficient initial qubits

as basic gates. The compound gates we usually use, such as the double-controlled gate CCX (Toffoli) [47], do not directly control three qubits. Instead, multiple single-qubit and controlled gates are combined, resulting in a dual-controlled gate effect. The correct custom gate should be a composite gate combining the basic gates provided by Qiskit and applied to the circuit.

3.2.3 Insufficient Initial Qubits

When the `TwoLocal` method is used, a dual-local parametric circuit consisting of alternating rotating and entangled layers can be formed. The two-local circuit is a parameterized circuit consisting of alternating rotation layers and entanglement layers. Suppose the number of qubits of the variational form does not match. The bug Pattern "*Insufficient Initial Qubits*" may occur. As shown in Figure 3.3, which is part of the code for the Variational Quantum Eigensolver (VQE) algorithm. When defining the VQE solver, method `TwoLocal` is used. As the `num_qubits` is set to `1`. In addition, the value of `num_qubits` is replaced by any other value that does not match, and the desired result is not obtained. So it is important to initialize the values supported by the parameter `num_quibits` when using parametric circuits or methods involving quantum entanglement.

3.2.4 Over Repeated Measurement

Some simulator backends cannot execute the circuit when the measurement operation performed on the qubit is repeated too many times. Alternatively, when some methods, such as `c_if`, are called but do not give the correct result. This situation may lead to the bug pattern of "*Over Repeated Measurement*."

```
def get_circuit(n):

    qreg = QuantumRegister(1)
    creg = ClassicalRegister(n)
    mreg = QuantumRegister(1)
    dreg = ClassicalRegister(1)
    circ = QuantumCircuit(qreg, mreg, creg, dreg)

    for i in range(n):
        circ.measure(qreg[0], creg[i])

    circ.x(mreg[0]).c_if(creg, 0)
    circ.measure(mreg[0], dreg[0])
    return circ

b_aer = BasicAer.get_backend('qasm_simulator')
aer = Aer.get_backend('qasm_simulator')
circ65 = get_circuit(65)
print("65clbits(Aer):",execute(circ65, aer).result().get_counts())
print("65clbits(Basic_Aer):",execute(circ65, b_aer).result().get_counts())
```

Figure 3.4: Over repeated measurement

To show this bug pattern, consider the piece of code in Figure 3.4. This test repeatedly measures quantum characteristics in a computing backend simulator. We use the Qiskit "Aer" simulator backend and the Python-based quantum simulator module "BasicAer" to simulate the circuit `qasm_simulator`. The same qubit is used multiple times here. When we call `BasicAer`, the system may report an error that the number of qubits is greater than the maximum (24) for `qasm_simulator`. Also, the `c_if` method we called did not get the desired result on the "Aer" backend simulator. The qubits of the `mreg` register did not achieve flipping. While the code `circ.x(mreg[0]).c_if(creg,0)` did not achieve. And if `n=63` in the classic register `creg`, the system will hang.

In summary, we do not recommend excessive measurement operations on qubits. The measured qubit is placed in the first position of the quantum register, and then the measurement is placed in the second position. Such repeated operations are equivalent to operating "N" multiple qubits. As a result, it can make the system extremely unstable.

3.2.5 Incorrect Operations after Measurement

When the measurement is completed, we cannot use the measured qubit for entanglement. Otherwise, we will not get the desired result. The measurement result can be treated as a classical value that no longer has the properties that the qubit has. If the measured value continues to be entangled with other qubits, which is used to change the target qubit state, it will be the bug pattern of "*Incorrect Operations after Measurement*." Considering the code snippet in Figure 3.5 taken from GitHub document [36], which realizes a quantum teleportation protocol. The last qubit's state in the code should be changed according to the first two bits' measurement results. The wrong instructions in the example are `teleport.cx(tq[1],tq[2])` and `teleport.cz (tq[0],tq[2])`, which entangles the measured qubit with the un-

```
tq = QuantumRegister(3)
tc0 = ClassicalRegister(1)
tc1 = ClassicalRegister(1)
tc2 = ClassicalRegister(1)
teleport = QuantumCircuit(tq, tc0,tc1,tc2)

teleport.h(tq[1])
teleport.cx(tq[1], tq[2])
teleport.ry(np.pi/4,tq[0])
teleport.cx(tq[0], tq[1])
teleport.h(tq[0])
teleport.barrier()
teleport.measure(tq[0], tc0[0])
teleport.measure(tq[1], tc1[0])
teleport.cx(tq[1], tq[2])
teleport.cz(tq[0], tq[2])
teleport.measure(tq[2], tc2[0])

backend = Aer.get_backend('qasm_simulator')
job = execute(teleport, backend, shots=1, memory=True).result()
result = job.get_memory()[0]
print(job.get_memory()[0])
```

Figure 3.5: Incorrect operations after measurement

measured qubit and therefore affects the result of the last qubit. This mistake is quite common, and many programmers inadvertently use measured qubits. In this program, the correct code should be `teleport.z(tq[2]).c_if(tc0,1)` as well as `teleport.x (tq[2]).c_if(tc1,1)`.

Although these erroneous operations follow quantum measurements, the reason for this is a poor understanding of the effect of measurement operations on qubit states.

3.2.6 Unsafely Uncomputation

Qiskit is a compelling framework because it supports the automatic management of qubits, i.e., there is no need to do the work of unallocated qubits manually. However, different program languages (e.g., Q#) have their own implementations, which can lead to exceptions in different backends and the need to manually unallocated qubits.

3.3 A Catalog of Bug Patterns in Qiskit

Quantum programming introduces new quantum-aware bug patterns that differ from classical ones. These quantum bug patterns should be identified and presented with a catalog. After introducing some bug patterns, in conjunction with section 2.1.3, we summarized some error-prone features in quantum Programs:

- **Initialization**: A quantum program is a series of operations on qubits. The initial stage is to initialize the quantum registers to store the qubits that need manipulation. Then the classical registers are initialized to store the values of the measured qubits. This stage does not include setting the quantum state, as the quantum state setting needs to be implemented by a gate operation. Quantum and classical registers do not have to be of equal initial size. We need to initialize as many classical bits as possible when we use multiple classical bits to store the same qubit measurements. However, another case is that the initialized qubit is larger than the classical bit. Since the programmer does not intend to measure some qubits, it is assumed there is no need to initialize the classical bits equal to the qubits.

Nevertheless, this is also the reason why most programs go wrong. So a hasty initialization can cause some problems for subsequent programs.

- **Gate Operation**: The core of quantum computing is to operate on qubits. Qiskit provides almost all the gates to implement algorithms in quantum programs [46]. To achieve the superposition of qubits, it must pass through the H (Hadamard) Gate. To achieve ”entanglement” in the case of multiple qubits, it must pass through the CNOT (controlled-NOT) gate. Complex gate operations are decomposed into basic gates in quantum language and gradually realized. Controlled gates are parameterized by two qubits, and double-controlled gates require three qubits. However, this does not mean that the double-controlled gate operates on three qubits at the same time. Many errors may occur when inappropriately using gates that operate on the qubits multiple times.

- **Measurement**: When we want 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 measurement is simple, the program executing a measurement statement is very complicated. It requires thousands of projection measurements of the qubits. Finally, it outputs all its possible results. Moreover, the number of occurrences of the result is used to obtain the size of the probability of outputting the correct value. Many errors start with the measurement statement because programmers do not really understand the effect of measurements on the state of qubits.

- **Deallocation**: Resetting and releasing the qubits safely is crucial; otherwise, the auxiliary qubits in the entangled state will affect the output. Deallocation is not considered a specific operation due to the power of Qiskit. We do not need to reset the qubits manually. However, In some backends, not releasing all qubits can be problematic. In other quantum programming languages, not handling all the qubits in the entangled state can cause problems in the program or output unsatisfactory values.

And in Table 3.1, we list the bug patterns in Qiskit we identified, including those detail described in this section. To classify the bug patterns listed, we summarize the description for each pattern by pattern name, category, symptoms, causes, and cures & preventions. Note that this is just a preliminary list of bug patterns in Qiskit, and more bug patterns will be added to the list as we get some new progress. In the current bug pattern catalog in Table 3.1, we classify these bug patterns by initialization (1), gate operation (2), measurement (3), and deallocation (4).

3.4 Threats to Validity

We next discuss the threats to the validity of this work. There are some external and internal validities in this work. And the bug patterns identified by us are publicly available for replicating.

Table 3.1: A catalog of bug patterns in Qiskit

Bug Patterns	Category	Symptoms	Causes	Cures and Preventions
Unequal Classical Bits and Qubits	1	Classical registers are not large enough to store the measured qubits	The initialized classical bits are smaller than the qubits used or to be measured	Try to initialize quantum and classical registers of the same size
Custom gates not recognized	2	The program is unable to customize the gate function and will often report errors	Creating gates that directly control more than three qubits does not follow the principle of two-qubit entanglement	Try to use the gates provided by Qiskit for the implementation of the algorithm
Over Repeated Measurement	3	Output error or program error when measuring the same quantum bit multiple times with a `for` loop	a number of measurements repeated several times	Reduction of meaningless measurements.
Incorrect Operations after Measurement	3	Unable to get the desired post-measurement result	Continued manipulation of the qubit being measured, such as changing its state or re-entangling with other qubits	The measured result cannot be used as a condition unless it is re-operated and measured as the initial qubit after reset
Unsafely Uncomputation	4	The program reports an error or does not achieve the desired result	Auxiliary qubits are not reset and remain entangled with the target qubit, which can affect the results of the target qubit measurement	Correctly reset or release all qubits to ensure they are in their initial or post-measurement states
Insufficient Initial Qubits	1	Causes VQE not to respect the form of input variables and outputs the wrong circuit	When the `TwoLocal` method is used, a dual-local parametric circuit consisting of alternating rotating and entangled layers can be formed	When using parametric circuits or methods involving quantum ”entanglement,” initialize the values supported by the parameter `num_qubits`.
Inappropriately Modification of Register Size	1	Changing the register size may cause the program to report an error. Especially for building complex circuits	Changing the size of a register may change the hash value of the register and its bits, thus prohibiting it from being used as a key for structures such as sets	It is possible to reinitialize the registers. Otherwise, it is not recommended to modify the values of the registers without changing the variable names
Method `measure_all`	3	The program typically outputs the results of all measured qubits. However, it also outputs the classical register values	When the `measure_all` method is used, the program automatically creates a classical register to store all the qubits being measured	If we want to call the `measure_all` method to measure all qubits, we do not need to initialize the classical registers

3.4.1 External Threats

Since there are not enough bugs in other common quantum programming languages, we only use Qiskit as our target language for defining bugs. Besides, the quantum programming language is rapidly evolving, and some bug patterns might be fixed after this work is presented. The main reason is that programmers writing quantum programs are limited by the language itself. Such as the maximum number of qubits supported varies depending on the design of the backend simulator for each language. As the version of the quantum frameworks is updated, we believe that more qubits will be supported for calls.

3.4.2 Internal Threats

We only identified bug patterns in Qiskit programming languages. Therefore, some specific bug patterns regarding their applicability to other quantum programming languages have not been verified. However, the error-prone features we identified may also manifest in other popular quantum programming languages prioritizing quantum programming features over traditional programming language conventions. In future work, we would like to validate our bug patterns in other quantum programming languages.

3.4.3 Verifiability

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

3.5 Conclusion

This chapter has identified some bug patterns in the quantum programming language Qiskit to provide both researchers and programmers with a clear view of what kind of bugs may happen in quantum programs and how to detect them. The study of bug patterns mainly focuses on bug pattern symptoms, root causes, and cures and preventions. These bug patterns are the first result of our research and do not use every possible quantum-related construct or cover all characteristics of a quantum programming language. New research should cover other remaining quantum-related constructs, as well as the interactions between them.

Chapter 4

Bugs4Q: A Benchmark of Existing Bugs to Enable Controlled Testing and Debugging Studies for Quantum Programs

4.1 Overview

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 testing and debugging depends on high-quality bug benchmark suites.

As the first step toward evaluating quantum software debugging and testing tools, this Chapter 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.

The rest of this chapter is organized as follows. Section 4.2 describes Bugs4Q, a bug benchmark suite for Qiskit. Section 4.3 evaluates and discusses existing quantum program testing tools. Section 4.4 is the threats to the validity of our work. And concluding remarks are given in Section 4.5.

4.2 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 4.1 shows all the issues and questions

Table 4.1: 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

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 would be ignored, such as Qiskit's internal documentation and its explanatory documentation, changes to the internal structure of Qiskit, and test files.

- **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 thoroughly. It may lead to bugs that are difficult to reproduce in a controlled environment.

- **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.** 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.

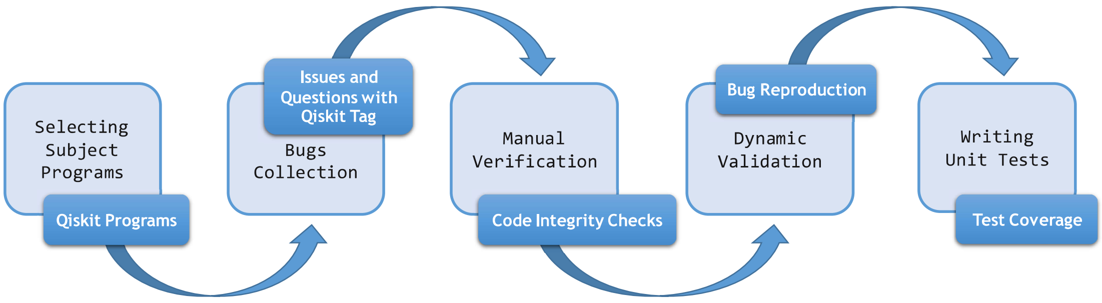

Figure 4.1: The overview of the building process of Bugs4Q benchmark.

Figure 4.1 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 source code versions 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 4.1. 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.

4.2.1 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`[1], `Aer`[2], `Ignis`[3], and `Aqua`[4]). 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 sub-project 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.

[1]https://github.com/Qiskit/qiskit-terra
[2]https://github.com/Qiskit/qiskit-aer
[3]https://github.com/Qiskit/qiskit-ignis
[4]https://github.com/Qiskit/qiskit-aqua

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.

4.2.2 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.

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 [43]). 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[1].) The *link* to each passed report is copied to the Bugs4Q benchmark database with a new ID added for further filtering.

[1]https://github.com/Qiskit/qiskit-terra/issues/3799

Collecting Bugs from Other Platforms

We searched for bugs on Stack Overflow from the page of "*Questions tagged [qiskit]*[1]," 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[2]. All the questions were checked individually, and our recording method was the same as collecting bugs from GitHub.

Unit Test

The unit tests for Qiskit only exist in the official Qiskit library[3] 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.

Collecting Results

After going through all the questions and issues on these three platforms, we have collected 346 bugs from 10069 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.

[1]https://stackoverflow.com/questions/tagged/qiskit?tab=Newest
[2]https://stackexchange.com/search?q=qiskit
[3]https://github.com/Qiskit/qiskit/tree/master/test, etc.

Table 4.2: 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

4.2.3 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 4.2. 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 2.1.3 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 [46], 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 4.3. 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*

```
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([0,1,2,3], [0,1,2,3])
job = execute(qc, backend = Aer.get_backend('qasm_simulator'), shots=1024)
result = job.result()
count = result.get_counts()
print(count)
```

Figure 4.2: The source code of a buggy program.

and the *fixed* versions. As an example, Figures 4.2 and 4.3 show the *buggy* and *fixed* versions, respectively, of one program[1] in our database for dynamic verification.

4.2.4 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 4.2. 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.

[1]https://quantumcomputing.stackexchange.com/questions/18448/how-to-perform-a-plot-histogram-for-a-circuit

```
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([0,1,2,3], [0,1,2,3])
job = execute(qc, backend = Aer.get_backend('qasm_simulator'), shots=1024)
result = job.result()
count = result.get_counts()
print(count)
```

Figure 4.3: The source code of a fixed program by manual completion.

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 Figure 2.1. 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 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.

Table 4.3: 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**

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 can not pass through dynamic validation, shown in Table 4.3. 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.

4.2.5 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 Figure 4.2 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 Figure 4.3, 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 2.1.1, `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 Figure 4.4. *Output wrong* is mainly caused by the fact that the program's output does not match the results expected by the programmer.

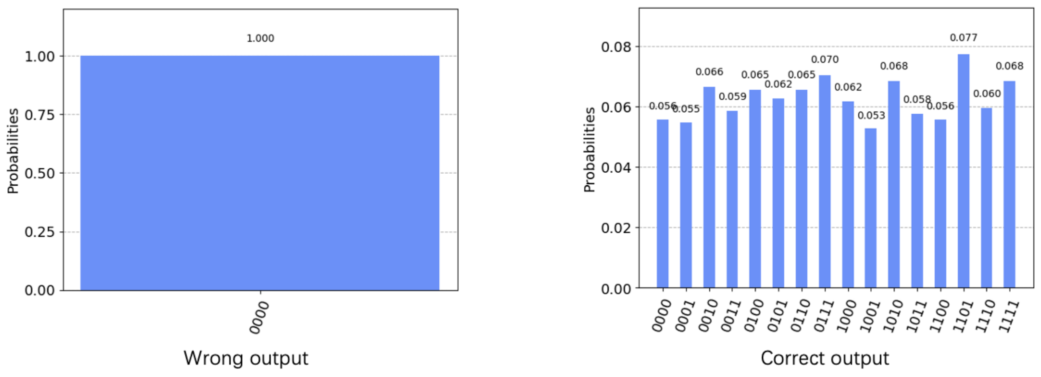

Figure 4.4: The two outputs correspond to one program's buggy and fixed versions, respectively.

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.

Unit Test

```
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([0,1,2,3], [0,1,2,3])
        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)
```

Figure 4.5: A unit test for output wrong (buggy version).

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 specifica-

```
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([0,1,2,3], [0,1,2,3])
        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)
```

Figure 4.6: A unit test for output wrong (fixed version).

tion 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 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.

```
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')
```

Figure 4.7: A unit test for throw exceptions (buggy version).

The unit tests for buggy and fixed versions about *output wrong* can be seen in Figures 4.5 and 4.6, 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 Figures 4.7 and 4.8. In Qiskit, some exceptions do not exist in the assertions contained in

```
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')
```

Figure 4.8: A unit test for throw exceptions (fixed version).

`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 Figures 4.7 and 4.8, 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 4.3. 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.

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 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 4.4. 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 Figure 4.5, 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.

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 definitive statements, and we could not screen them. For example, the unit test for bug `No.1` has nine 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.

Table 4.4: The coverage of source code and unit tests.

	Source Code Coverage			**Unit Test Coverage**		
BugID	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%

4.2.6 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 Figure 4.9.

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 4.5. According to the source of the bug information (i.e., Github, StackOverflow, and Stack Exchange), we divided the bugs into three groups, respectively. 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 directly access the information of *Buggy, Fixed, Modify, and Test*.

Table 4.5: 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

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. The Bugs4Q database also includes bugs of other quantum programming languages, publicly available at https://github.com/Z-928/Bugs4Q.

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 Figure 4.9, 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:

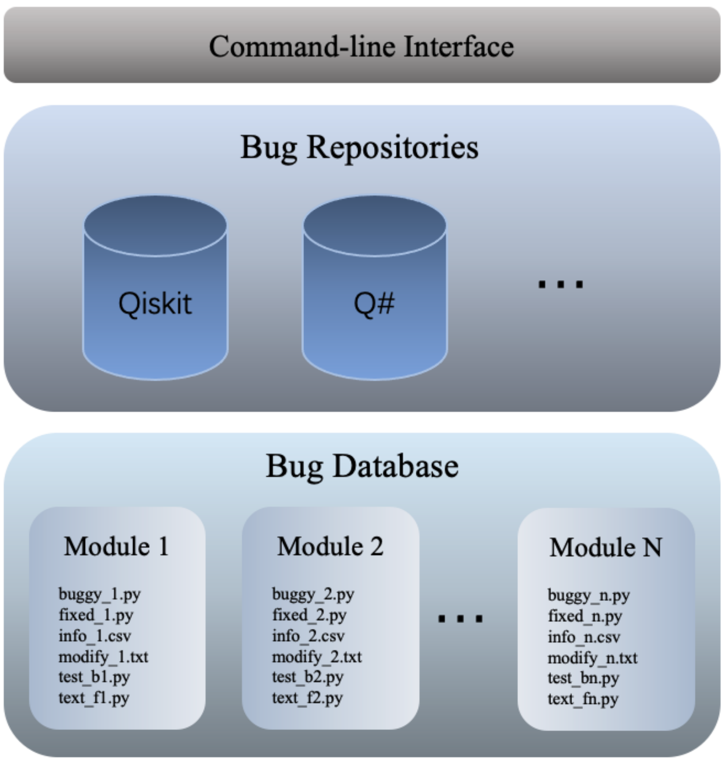

Figure 4.9: The overall structure of Bugs4Q framework.

- **`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 and fixed programs, which are generated 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.

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.

4.3 Evaluation and Discussion

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 potential applications of Bugs4Q.

4.3.1 Evaluation of Testing Tools

Several test case generation methods for quantum programs have been proposed. For example, Quito [60] provides three coverage criteria for quantum programs and their test generation strategies. QuSBT [58] 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, 128G RAM, running on Ubuntu 20.04 with Python 3.10 installed. We set the parameters 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.

Table 4.6: 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

The experiment results are shown in Table 4.6. Firstly, `IC`, `OC`, and `IOC` represent the three coverage criteria in Quito, respectively. Next, in simple terms, `fail`$_{WOO}$ and `fail`$_{uof}$ represent the quantum programs failed by the wrong output value of the program while `fail`$_{OPO}$ and `fail`$_{wodf}$ represent the wrong probability distribution of output values. Finally, the number of tests generated by QuSBT is set manually. As a result, both testing tools have 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 more efficient. In addition, QuSBT would give the optimal solution for each program in the generated test cases.

4.3.2 The Combination between Existing Tools and Bugs4Q

In addition to Quito and QuSBT, several other works have contributed to advancing quantum program testing. QuCAT [57] provides two schemes for generating combinatorial test suites and argues that the more intense combinatorial tests are more effective than the less intense ones. Muskit [39] is a quantum mutation analysis tool for the Qiskit language, focusing on mutation operators for quantum gates. QMutPy [17] can generate effective mutation programs for *measurement* calls and many quantum gates. Fortunato, Campos, and Abreu [16], 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 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 more 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.

4.3.3 Possible Uses of Bugs4Q

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

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. Most bugs we collected are related to quantum properties, so efficient testing methods are urgently needed to detect them.

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.

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.

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 valuable 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 [16, 17, 39] and assertion-based techniques [26, 32, 32, 33, 50, 67] 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.

4.4 Threats to Validity

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

4.4.1 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 [8, 19, 51, 52] 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 4.7 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 widely used. And we will continue to collect new bugs and update Bugs4Q in our future work.

Table 4.7: 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

4.4.2 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.

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* [29]. In addition, almost all the quantum programs we studied were executed based on the simulator. We hope to collect bugs from programs executed with quantum computers in future work.

4.4.3 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`.

4.5 Conclusion

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.

Chapter 5

QChecker: Finding Bugs by Bug Pattern Detectors via Static Analysis

5.1 Overview

This Chapter presents QChecker, a static analysis tool designed for detecting bugs in quantum programs, especially for Qiskit. The approach addresses the challenge above by first distilling a set of common bug patterns summarized from real quantum bugs in previous works [65, 66] and then constructing eight detectors for detecting these bug patterns in quantum programs. The whole process is non-trivial since the distilled bug patterns must carefully consider the domain-specific constraints of quantum computing to be accurate and effective. QChecker consists of two main modules: a module for extracting program information based on abstract syntax tree (AST) and a module for detecting bugs based on patterns. We evaluate QChecker on Bugs4Q [66], a realistic benchmark of 42 real-world buggy quantum programs. Experimental results show that QChecker can efficiently detect bugs in quantum programs. Furthermore, we discuss the extendability of QChercker for other Python-based quantum programming languages.

In summary, this work makes the following contributions:

- We present the first bug detection tool dedicated to quantum programs in Qiskit. Using static analysis techniques, QChecker can generate diagnostic messages that assist developers in pinpointing potential bugs in their programs quickly.
- We implement QChecker and evaluate its effectiveness and performance in a real-world Bugs4Q benchmark. The results show that QChecker can effectively detect various types of bugs in quantum programs.

The rest of this chapter is organized as follows. Section 5.2 describes our QChecker approach for static analysis of quantum programs. Section 5.3 presents the performance of QChecker on Bugs4Q. Section 5.4 reviews our threats of validity. Section 5.5 finally concludes this work.

5.2 The QChecker Tool

In this section, we introduce the construction of QChecker, developed based on Python. As illustrated in Figure 5.1, QChecker first performs a thorough information extraction of the quantum programs based on their ASTs. The corresponding operations are in the module `Ast_Operator`. The information mainly includes the variable assign operations and function calls, which will be stored in `QP_Attribute` and `QP_Operation`. Then QChecker transmits the extracted information to the bug detectors. The bug detectors can detect various bug patterns, as shown in Table 5.1. Finally, QChecker generates bug detection reports, including the buggy programs, line numbers, and bug descriptions.

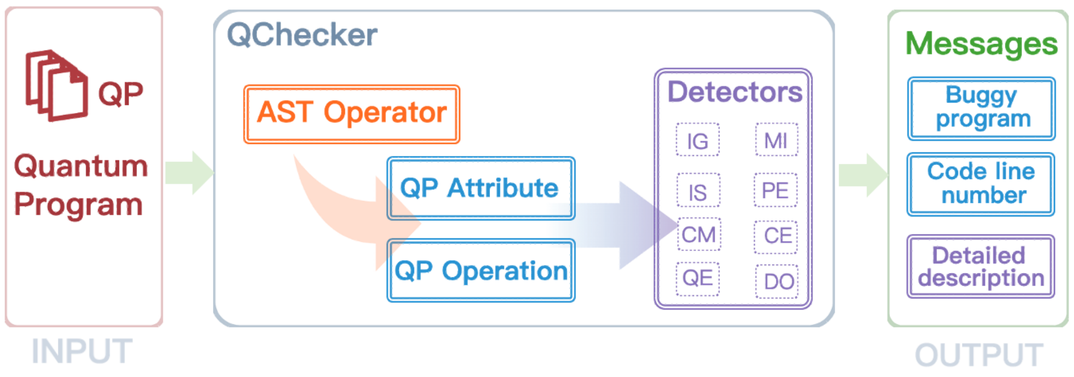

Figure 5.1: The structure of QChecker.

5.2.1 Information Extraction

The previous static analysis tools inspire us (e.g., PyLint [53]) that using AST for program information extraction is effective and efficient. However, different from classical static analysis tools, the `AST_Operator` in QChecker can extract information specific to the semantics and the function of quantum programs. Taking the program shown in Figure 2.1 as an example, we apply a structured parsing to each quantum program file, i.e., generating the AST. We adopt two modules named `QP_Attribute` and `QP_Operation` to store the AST information of all the variables and function calls, respectively. In addition, QChecker also supports handling complex syntax and data structures such as dictionaries, lists, function definitions, loops, and conditional branches. The purpose of this design is that the structured AST-based information extraction can help QChecker trace the relationship between each variable and function call. For example, a variable may be modified multiple times, or its name may be changed when passed as an argument inside a function. Nevertheless, we can still trace back the initial value of the variables in the program. We plot instances of `QP_Attribute` and `QP_Operation` in Figures 5.2 and 5.3, respectively.

- **QP_Attribute**: The AST structure of a variable includes its variable name, variable value (which can come from a constant, another variable, or the result of a function calculation), type, and location (line of code). As shown in Figure 5.2, The `QP_Attribute` module is designed in a key-value manner. The keys are variable names that can be the variable value indices.
- **QP_Operation**: The AST structure of a function call contains a list of its arguments, the type and value of each argument, its position, and other information. As shown in Figure 5.3, The `QP_Operation` module is a list that contains all the function calls in the quantum program file. In detail, each function call can be further divided into function call names, arguments, and values. This information is stored in a more comprehensive table from QChecker, which uses the function call strings in `QP_Operation` as the index.

These two modules contain all the information of a quantum program, making it more straightforward for further bug detection. Moreover, users can directly obtain the above information through QChecker based on the API we released. It is worth mentioning that those programs containing basic syntax errors (e.g., python indentation errors, unrecognized operators, undefined variables and functions, etc.) will not be processed by QChecker. Instead, they will be prompted as syntax errors and thus be excluded from the static checking.

5.2.2 Bug Pattern Detectors

Bug patterns are erroneous code idioms or bad coding practices that have been proven to fail repeatedly. These are usually caused by misunderstanding a programming language's features, using erroneous design patterns, or simple mistakes sharing common behaviors. Previous work has identified some bug patterns for the Qiskit programming

```
('simulator', 'Aer.get_backend("qasm_simulator")')
('qreg', 'QuantumRegister(3)')
('creg', 'ClassicalRegister(3)')
('circuit', 'QuantumCircuit(qreg,creg)')
('job', 'execute(circuit,simulator,shots=1000)')
('result', 'job.result()')
('counts', 'result.get_counts(circuit)')
```

Figure 5.2: Program information extracted by `QP_Attribute`.

language [65, 66]. In this work, we refined these bug patterns and built eight detectors to detect them. Table 5.1 shows the name of detectors and descriptions of bug patterns. We briefly describe each detector as follows.

Incorrect uses of quantum gates (IG)

This detector mainly checks if quantum gates are called correctly. It determines whether Qiskit recognizes a gate and whether it has been defined. In addition, the compliance of a custom gate and a three-qubit gate with the specification would also be checked.

Measurement related issue (MI)

Incorrect measurement means the improper use of measure operation that cause bugs and the wrong operation after measurement. As shown in Figure 5.4, the user wants to achieve a quantum teleportation program. The measured qubits are used as control qubits to entangle with other qubits. This detector mainly acts after the `measure` method is called. It iterates through the operations following the `measure` statement and determines whether the measured qubit appears as a control qubit in the double-qubit

```
'Aer.get_backend("qasm_simulator")'
'QuantumRegister(3)'
'ClassicalRegister(3)'
'QuantumCircuit(qreg,creg)'
'circuit.h(0)'
'circuit.h(2)'
'circuit.cx(0,1)'
'circuit.measure([0,1,2],[0,1,2])'
'execute(circuit,simulator,shots=1000)'
'job.result()'
'result.get_counts(circuit)'
'print(counts)'
```

Figure 5.3: Program information extracted by `QP_Operation`.

gate operations.

Incorrect initial state (IS)

This detector does not simply check whether the definitions of `QuantumRegister` and `ClassicalRegister` conform to the specification. It determines whether the initialization satisfies the entire quantum program's operation on qubits. Sometimes, the Qiskit program limits the number of qubits used when simulating quantum programs. For example, the `Aer.get_backend('qasm_simulator')` backend supports less than 30 qubits for measure operation, while `BasicAer.get_backend('qasm_simulator')` supports less than 24 qubits. In this case, the detector first checks the backend chosen by users and Identifies if the initialized qubits are out of limits. When the number of initialized qubits is set to `n`, the checker will keep track of the number of qubits called in the

```
qc = QuantumCircuit(3, 3)
qc.x(0)
qc.h(1)
qc.cx(1, 2)
qc.cx(0, 1)
qc.h(0)
qc.measure(0, 0)
qc.measure(1, 1)
qc.cx(1, 2)        <- Problematic operation
qc.cz(0, 2)        <- Problematic operation
```

Figure 5.4: Example of Incorrect Mearsurement

program.

Parameter error (PE)

After a quantum gate is invoked, this detector is responsible for determining whether the parameters in the gate are correct, including the parameters that do not exist in multiple-qubit gates and the wrong use of numeric types. However, some bug patterns are not easy to find. From Figure 5.5, we can see that the user wants to assign the qubits in the register to the physical qubits, both `qreg[0]` are `qreg[5]` assigned to the physical qubits `12`. Therefore, the detector goes through the parameter values and checks for duplicate physical qubit occupancy.

Command misuse (CM)

This detector could detect the wrong or improper use of commands. Sometimes, parameters are not recognized because the method name is miswritten. On the other

```
qreg = qk.QuantumRegister(7)

layout = {qreg[0]: 12,     <- Problematic operation
          qreg[1]: 11,
          qreg[2]: 13,
          qreg[3]: 17,
          qreg[4]: 14,
          qreg[5]: 12,     <- Problematic operation
          qreg[6]: 6}
```

Figure 5.5: Example of parameters error

hand, some methods can not recognize parameters and raise errors. As shown in Figure 5.6, attribute `pulse.shiftphase()` is not in module `qiskit.phase`. Some of the commands in Qiskit are difficult to detect. For example, there are more than two quantum circuits while the user wants to nest one circuit with the others: 1) Command `to_gate()` could be used to change the circuit into a combination of gates embedded in other circuits. 2) Command `decompose()` could be used to decompose the circuit for embedded operation automatically.

Call error (CE)

This detector is responsible for call errors, including Python package calls, backend simulator calls, and translator calls. Besides, the detector can check for problems with parameter declarations. As shown in Figure 5.7, this error is not a duplicate call to `PauliMeasurementBasis()`. After running the code, we found an error of *invalid qubits for basis*. The call of `PauliMeasurementBasis()` is invalid for `preparation_basis`. These types of bugs are hard to detect by QChecker; the detector can only judge one

```
phase = Parameter('phase')

with pulse.build(FakeAlmaden())as phase_test_sched:

        pulse.shiftphase(           <- Unrecognized
        phase, pulse.drive_channel(0))

phase_test_sched.instructions
```

Figure 5.6: Example of command misuse.

scenario now.

QASM error (QE)

This detector detects problems with `qasm_simulator` as the backend or when building *qasm* programs with the Qiskit programming language.

Discarded orders (DO)

This detector determines if a deprecated method is being called, and it comes into play when an old operation or variable type is discarded due to a version update.

5.2.3 Bug Detection

The bug patterns shown in Table 5.1 represent the general bugs in quantum programs. In addition to syntactic bugs, it also contains some faulty logic in some quantum-related operations. Based on the program information extraction modules (`QP_Attribute` and `QP_Operation`), as well as the detectors, we can perform a thorough static analysis

```
circ = QuantumCircuit(1,1)
circ.x(0)

tomo = ProcessTomography(
        circuit=circ,
        measurement_basis=PauliMeasurementBasis(),
        measurement_qubits=None,
        preparation_basis=PauliMeasurementBasis(),  <-
        preparation_qubits=None,
        basis_indices=None,
        qubits=None)
```

Figure 5.7: Example of call error

for the quantum program files. To enable comprehensive and useful bug detection and benefit the debug process, we report the details of the buggy programs (code lines, content, etc.), bug types (patterns), and specific descriptions. We hope such information may help users improve the quality of quantum programs.

5.3 Evaluation

This section presents the empirical performance of QChecker. We evaluate QChecker on Bugs4Q [66], which contains 42 real-world buggy quantum programs in Qiskit. The experiments were conducted on a server with an Intel i9-10940X CPU, 128G RAM, running on Ubuntu 20.04 with Python 3.10 installed.

Table 5.1: Bug patterns that each detector is responsible for.

Detector Name	Bug patterns Descriptions
IG	- Gates are not among the backend's basis gates. - Handle custom multi-qubit gates. - Random gate is not defined.
MI	- Ignoring the effects of measurement.
IS	- Number of qubits larger than the registers defined. - The insufficient number of qubits. - Insufficient length of classical registers.
PE	- Instruction not in basis gates. - Incorrect parameters in gates. - Using classical bits for entanglement. - Same physical qubit used in one operation. - Not giving lists for coupling_map.
CM	- Unrecognized parameters. - Quantum circuit interaction error. - Create redundant classical registers. - The wrong command was used.
CE	- Object call error. - Import error. - Backend error. - Translating error.
QE	- Issue with new from_qasm_str() method.
DO	- Method has been deprecated.

5.3.1 Metrics

We adopt *Precision*, *Recall*, and *F1-score* to evaluate the performance of QChecker. Specifically, for each bug b in Bugs4Q, we use the bug type in Bugs4Q as ground truth and apply QChecker to its source quantum program. If the detection result of QChecker matches the corresponding bug type in Bugs4Q, we call this b as *True Positive* (TP). Otherwise, this b is a *False Positive* (FP). *False Negative* (FN) is a ground-truth bug that QChecker can not detect. The *Precision* is calculated as $TP/(TP+$

FP), *Recall* as $TP/(TP+FN)$ and *F1-score* as $2 \times Precision \times Recall/(Precision + Recall)$.

5.3.2 Performance on Real-World Qiskit Programs

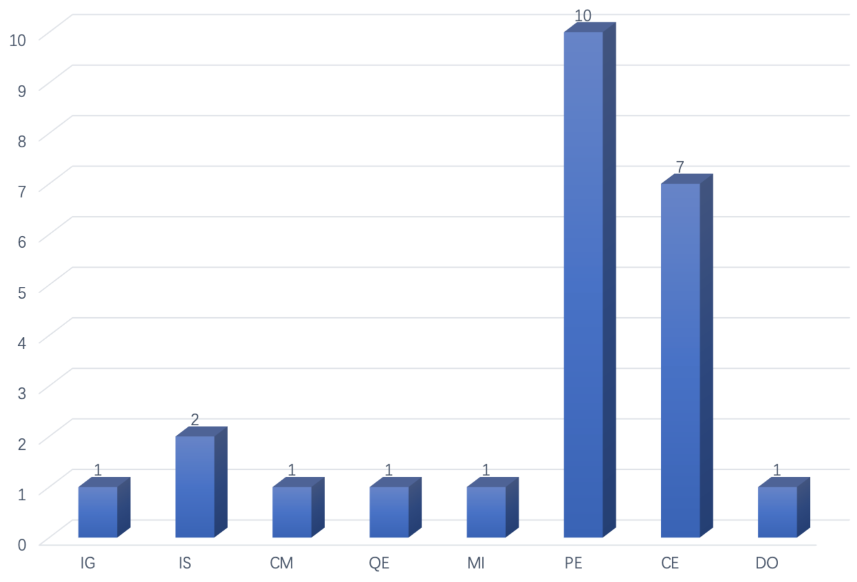

Figure 5.8: Distribution of bugs found by each detector.

First of all, there are 42 Qiskit programs detected by QChecker, and 24 bugs were found. Figure 5.8 shows the empirical results of applying QChecker on Bugs4Q. Detector `PE` found ten bugs, while `CE` and `IS` found 7 and 2 bugs, respectively. And other detectors found one bug in each. As we know, out of the 42 bugs in Bugs4Q, 22 are output errors, i.e., the program's output does not match the user's expectations. This condition makes Qchercker unable to detect these bugs. So we consider that the remaining number of bugs in Bugs4Q found by Qchecker is in line with expectations. Combining Table 5.1 with the results in Figure 5.8, we analyzed and derived two points about the performance of each detector:

- *The more cases the detectors can cover, the more bugs can be found.* As we made the detectors according to the relevance of the bug patterns, the number of cases covered by each detector may vary. Detector `PE` and `CE` can detect more cases than other detectors. And in fact, the two detectors made a better performance.
- *The performance of detectors also depends on the type of errors the programmers make*. Considering the limited number of bugs in Bugs4Q, we believe that some detectors perform poorly because they cover cases that rarely occur. For example, detector `IG`, `CM`, and `IS` can detect more than one case while the results are barely satisfactory.

In addition, a detector that solves only one case does not indicate poor performance. Instead, with their increased functionality, these specialized detectors will realize their potential to detect bugs better.

The efficiency of QChecker on 42 Qiskit programs provided by Bugs4Q is shown in Table 5.2. As illustrated in [37], executing a quantum program on simulators can easily consume more than 10^3 ms. As a result, QChecker demonstrates high efficiency by taking an average time of only 48.2 ms to complete detection on a single quantum program. To better represent this, we investigated the 42 quantum programs in Bugs4Q. The average execution time for these programs was 2.14 seconds, while the average amount of code per program was 31 LOC. From Section 2.1, we already know that obtaining the state of a qubit requires a large number of iterations of the output to obtain its probability distribution, which we consider to be the main cost of executing one program.

In summary, QChecker has the advantage of being efficient and relatively effective in execution, while the disadvantage is QChecker does not avoid the problem of false positives. As the number of qubits in future quantum programs increases, we believe it

Table 5.2: Performance of QChecker on Qiskit programs.

Performance	*Prec.*	*Recall*	*F1-score*	*Avg. Time*
QChecker	0.625	0.882	0.731	48.2ms

is necessary and effective to find bugs before program execution using static analysis.

5.3.3 Extendability of QChecker

```
qubit = cirq.NamedQubit("myqubit")
circuit = cirq.Circuit(cirq.H(qubit))
for i in range(10):
    result2 = cirq.measure(qubit, key='myqubit')
    print(result2)
print(circuit)
result = cirq.Simulator().simulate(circuit)
print(result2)
```

Figure 5.9: An exapmle of Cirq program.

We next discuss the extendability of QChecker. The example of a Cirq program and the corresponding detection result of QChecker are shown in Figures 5.9 and 5.10, respectively. After careful inspection of the results, we find that the information extraction part of QChecker can still function on other Python-based quantum languages (e.g., Cirq). The syntax difference between these quantum languages may cause the detectors designed for Qiskit to fail, leading to the lack of guaranteed performance.

In summary, the experimental results show that QChecker can successfully detect bugs in real-world Qiskit quantum programs, exhibiting the effectiveness of applying

```
('qubit', 'cirq.NamedQubit("myqubit")')
('circuit', 'cirq.Circuit(cirq.H(qubit))')
('result', 'cirq.Simulator().simulate(circuit)')
('result2', 'cirq.measure(qubit,key="myqubit")')
==========================================
'cirq.NamedQubit("myqubit")'
'cirq.Circuit(cirq.H(qubit))'
'range(10)'
'print(circuit)'
'cirq.Simulator().simulate(circuit)'
'print("result:")'
'print(result2)'
'cirq.H(qubit)'
'cirq.measure(qubit,key="myqubit")',
'print(result2)',
'cirq.Simulator()'
```

Figure 5.10: A Cirq program detected by QChecker

static analysis to quantum programs. Besides, the intermediate results indicate that the QChecker can correctly extract the `QP_Attribute` and `QP_Operation` information from the underlying Cirq program, indicating that the QChecker can be easily extended to common Python-based quantum programming languages.

```
eng = MainEngine()
qubits = eng.allocate_qureg(3)
H | qubits[0]
CX | (qubits[0], qubits[2])
eng.flush()
amplitudes = np.array(eng.backend.cheat()[1])
amplitudes = np.abs(amplitudes)
All(Measure) | qubits
```

Figure 5.11: An exapmle of ProjectQ program.

5.4 Treats to Validity

5.4.1 External Threats

Even for the most widely used Qiskit quantum programming language, there are still not enough buggy programs. Moreover, existing quantum programs are usually run on simulators rather than on actual quantum computers, which leads to the small size of current quantum programs. As a result, the number of bug patterns can threaten validity. We have put much effort into collecting bugs from quantum programs and extracting as many bug patterns as possible from these collected bugs. However, due to the limitation of the current scale of development and application of quantum programs, we cannot include more bug patterns in QChecker. Therefore, we will continue to collect quantum programs and their bugs, enrich QChecker's detection capabilities, and continuously update the tool.

5.4.2 Internal Threats

QChecker is designed for Qiskit and can be extended to other Python-based quantum languages (e.g., Cirq) with slight modifications. However, it also has limitations. For instance, ProjectQ has overloaded the | operator, which will cause the information extractor fails to work. As shown in Figures 5.11 and 5.12, QChecker could not extract information from the underlying ProjectQ program, such as the `H` gate and `CX` gate. These limitations will be resolved with the extension of QChecker.

```
('eng', 'MainEngine()')
('qubits', 'eng.allocate_qureg(3)')
('amplitudes', 'np.array([1])')
('amplitudes', 'np.abs(amplitudes)')
==========================================
'MainEngine()'
'eng.allocate_qureg(3)'
'eng.flush()'
'np.array([1])'
'np.abs(amplitudes)'
'All(Measure)'
'eng.backend.cheat()']
```

Figure 5.12: A ProjectQ program detected by QChecker.

5.4.3 Verifiability

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

5.5 Conclusion

This Chapter presents QChecker, a static analysis tool for quantum programs to enable effective and efficient potential bug detection of quantum programs. QChecker involves two AST-based program information extraction modules and comprehensive bug detectors which can detect various bug patterns. We applied QChecker to the Bugs4Q benchmark suite and evaluated its effectiveness. The results show that QChecker can detect multiple types of bugs in quantum programs.

Chapter 6

Conclusion and Future Work

6.1 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. Although several approaches have been proposed for debugging and testing quantum software recently, it remains at an early stage in the research about identifying and detecting bugs in quantum programs. To verify the correctness of quantum programs more efficiently and to improve the quality assurance of quantum software, this doctoral thesis mainly focuses on implementing the identification and detection of quantum program bugs.

In Chapter 3, we identified some bug patterns in the quantum programming language Qiskit to provide both researchers and programmers a clear view of what kind of bugs may happen in quantum programs and how to detect them. The study of bug patterns mainly focuses on bug pattern symptoms, root causes, and cures and preventions. These bug patterns are the first result of our research and do not use every possible quantum-related construct or cover all characteristics of a quantum programming language. New research should cover other remaining quantum-related constructs, as well as the interactions between them.

In Chapter 4 of this thesis, we propose 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 Chapter 5, we presented QChecker, a static analysis tool for quantum programs to enable effective and efficient potential bug detection of quantum programs. QChecker involves two AST-based program information extraction modules and comprehensive bug detectors which can detect various bug patterns. We applied QChecker to the Bugs4Q benchmark suite and evaluated its effectiveness. The results show that QChecker can detect multiple types of bugs in quantum programs.

As a result, this doctoral thesis fills the gaps in identifying and detecting bugs in quantum programs. The bug patterns we identified provide a better understanding of quantum program bugs and help developers avoid these bugs. Through bugs4Q, we have a comprehensive view of quantum bugs and provide a benchmark to evaluate the techniques for quantum programming. The static analysis tool QChecker can efficiently detect bugs in quantum programs.

6.2 Future Work

In our future work, we would like to continue effectively validating the correctness of quantum programs and further improve the quality assurance of quantum software. In detail, we plan to develop our approach to investigating more bug patterns in quantum programs. We would also like to develop more bug-detecting tools based on the identified bug patterns in this thesis to support finding more bugs in quantum programs. In addition, 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. Finally, we plan to extend QChecker to detect more bug patterns of Qiskit programs and support bug detection of other common quantum programming languages such as Cirq and ProjectQ.

Bibliography

[1] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, and F. Chong. Scaffold: Quantum programming language. Technical report, Department of Computer Science, Princeton University, 2012.

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

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

[4] S. Ali, P. Arcaini, X. Wang, and T. Yue. Assessing the effectiveness of input and output coverage criteria for testing quantum programs. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 13–23. IEEE, 2021.

[5] E. Allen. *Bug patterns in Java*. APress LP, 2002.

[6] N. Ayewah and W. Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252, 2010.

[7] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[8] B. Bichsel, M. Baader, T. Gehr, and M. Vechev. 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*, pages 286–300, 2020.

[9] O. Brunet and P. Jorrand. Dynamic quantum logic for quantum programs. *International Journal of Quantum Information*, 2(01):45–54, 2004.

[10] J. Campos and A. 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.

[11] V. Dallmeier and T. 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.

[12] E. D'hondt and P. Panangaden. Quantum weakest preconditions. *Mathematical Structures in Computer Science*, 16(3):429–451, 2006.

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

[14] V. Dunjko, J. M. Taylor, and H. J. Briegel. Quantum-enhanced machine learning. *Physical review letters*, 117(13):130501, 2016.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.

[16] D. Fortunato, J. Campos, and R. Abreu. Mutation testing of quantum programs: A case study with Qiskit. *IEEE Transactions on Quantum Engineering*, 3:1–17, 2022.

[17] Fortunato Daniel and Campos José and Abreu Rui. Mutation testing of quantum programs written in Qiskit. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 358–359, 2022.

[18] M. F. Gely and G. A. Steele. QuCAT: quantum circuit analyzer tool in python. *New Journal of Physics*, 22(1):013025, 2020.

[19] Google AI Quantum Team. Cirq, 2018.

[20] H. R. Grimsley, S. E. Economou, E. Barnes, and N. J. Mayhall. An adaptive variational algorithm for exact molecular simulations on a quantum computer. *Nature communications*, 10(1):1–9, 2019.

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

[22] T. Häner, D. S. Steiger, M. Smelyanskiy, and M. Troyer. High performance emulation of quantum circuits. In *SC'16: Proceedings of the International Confer-*

ence for High Performance Computing, Networking, Storage and Analysis, pages 866–874. IEEE, 2016.

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

[24] D. Hovemeyer and W. Pugh. Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

[25] Y. Huang and M. Martonosi. Qdb: From quantum algorithms towards correct quantum programs. *arXiv preprint arXiv:1811.05447*, 2018.

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

[27] M. Hutchins, H. Foster, T. Goradia, and T. 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.

[28] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi. Scaffcc: Scalable compilation and analysis of quantum programs. *Parallel Computing*, 45:2–17, 2015.

[29] R. Just, D. Jalali, and M. 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.

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

[31] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[32] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[33] J. Liu, G. T. Byrd, and H. 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.

[34] P. Long and J. Zhao. Testing quantum programs with multiple subroutines. *arXiv preprint arXiv:2208.09206*, 2022.

[35] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[36] M. Marques. Qiskit community tutorials. *Accessed on: April, 2020.*

[37] P. Matteo and P. Michael. MorphQ: Metamorphic testing of quantum computing platforms, 2022.

[38] D. C. McKay, T. Alexander, L. Bello, M. J. Biercuk, L. Bishop, J. Chen, J. M. Chow, A. D. Córcoles, D. Egger, S. Filipp, et al. Qiskit backend specifications

for OpenQASM and OpenPulse experiments. *arXiv preprint arXiv:1809.03452*, 2018.

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

[40] A. Miranskyy and L. 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.

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

[42] M. A. Nielsen and I. Chuang. Quantum computation and quantum information, 2002.

[43] M. Paltenghi and M. Pradel. Bugs in quantum computing platforms: An empirical study. *arXiv preprint arXiv:2110.14560*, 2021.

[44] T. Patel and D. Tiwari. 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*, pages 443–455, 2021.

[45] P. Rebentrost, M. Mohseni, and S. Lloyd. Quantum support vector machine for big data classification. *Physical review letters*, 113(13):130503, 2014.

[46] I. Research. Ibm quantum experience. *Accessed on: April, 2020.*

[47] I. Research. Qiskit. *Accessed on: June, 2021.*

[48] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *15th International symposium on software reliability engineering*, pages 245–256. IEEE, 2004.

[49] H. Shen, S. Zhang, J. Zhao, J. Fang, and S. Yao. XFindbugs: extended findbugs for AspectJ. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 70–76, 2008.

[50] K. Singhal. Hoare types for quantum programming languages. 2019.

[51] D. S. Steiger, T. Häner, and M. Troyer. ProjectQ: an open source software framework for quantum computing. *Quantum*, 2:49, 2018.

[52] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. 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.

[53] S. Thénault et al. Pylint. *Code analysis for Python*, 2001.

[54] D. Unruh. Quantum relational hoare logic. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

[55] J. Wang, M. Gao, Y. Jiang, J. Lou, Y. Gao, D. Zhang, and J. Sun. Quanfuzz: Fuzz testing of quantum program. *arXiv preprint arXiv:1810.10310*, 2018.

[56] J. Wang, Y. Huang, S. Wang, and Q. Wang. Find bugs in static bug finders. In *Proc. 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 516–527, New York, NY, USA, 2022. Association for Computing Machinery.

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

[58] X. Wang, P. Arcaini, T. Yue, and S. Ali. Generating failing test suites for quantum programs with search. In *International Symposium on Search Based Software Engineering*, pages 9–25. Springer, 2021.

[59] S. Xia and J. Zhao. Static entanglement analysis of quantum programs. In *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE 2023)*. IEEE, 2023.

[60] W. Xinyi, A. Paolo, Y. Tao, and A. Shaukat. Quito: a coverage-guided test generator for quantum programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1237–1241. IEEE, 2021.

[61] M. Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):1–49, 2012.

[62] N. Yu and J. Palsberg. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 542–558, 2021.

[63] S. Zhang and J. Zhao. On identifying bug patterns in aspect-oriented programs. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 431–438. IEEE, 2007.

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

[65] P. Zhao, J. Zhao, and L. Ma. Identifying bug patterns in quantum programs. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pages 16–21. IEEE, 2021.

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

[67] H. Zhou and G. T. Byrd. Quantum circuits for dynamic runtime assertions in quantum computation. *IEEE Computer Architecture Letters*, 18(2):111–114, 2019.

[68] L. Zhou, N. Yu, and M. Ying. An applied quantum hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1149–1162, 2019.

Published Papers

1 **Pengzhan Zhao**, Jianjun Zhao, Lei Ma. Identifying bug patterns in quantum programs. In *the 2nd IEEE/ACM International Workshop on Quantum Software Engineering (Q-SE 2021)*, pp. 16-21, June 2021.

2 **Pengzhan Zhao**, Jianjun Zhao, Zhongtao Miao, Shuhan Lan. Bugs4Q: A benchmark of real bugs for quantum programs. In *the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*, pp. 1373-1376, November 2021.

3 Junjie Luo, **Pengzhan Zhao**, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. A Comprehensive Study of Bug Fixes in Quantum Programs. In *the 1st International Workshop on Quantum Software Analysis, Evolution and Reengineering (Q-SANER 2022)*, pp. 1239-1246, Honolulu, HI, USA, March 2022.

4 **Pengzhan Zhao**, Xiongfei Wu, Zhuo Li, Jianjun Zhao. QChecker: Detecting Bugs in Quantum Programs via Static Analysis. In *the 4th International Workshop on Quantum Software Engineering (Q-SE 2023)*, pp.50-57, Melbourne, Australia, May 2023.

5 **Pengzhan Zhao**, Xiongfei Wu, Junjie Luo, Zhuo Li, Jianjun Zhao. An Empirical Study of Bugs in Quantum Machine Learning Frameworks. In *the IEEE Interna-*

tional Conference on Quantum Software (QSW 2023), pp.68-75, Chicago, Illinois, USA, July 2023.

6 **Pengzhan Zhao**, Zhongtao Miao, Shuhan Lan, Jianjun Zhao. Bugs4Q: A Benchmark of Existing Bugs to Enable Controlled Testing and Debugging Studies for Quantum Programs. Accepted by *the Journal of Systems and Software*, July 2023.