The Journal of Systems and Software 211 (2024) 112000

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Equivalence, identity, and unitarity checking in black-box testing of quantum programs☆

Peixun Long [a,b,*], Jianjun Zhao [c]

[a] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing, China
[b] University of Chinese Academy of Science, Beijing, China
[c] Kyushu University, Fukuoka, Japan

ARTICLE INFO

Keywords:
Quantum programs
Software testing
Black-box testing
Equivalence checking
Unitarity checking

ABSTRACT

Quantum programs exhibit inherent non-deterministic behavior, which poses more significant challenges for error discovery compared to classical programs. While several testing methods have been proposed for quantum programs, they often overlook fundamental questions in black-box testing. In this paper, we bridge this gap by presenting three novel algorithms specifically designed to address the challenges of equivalence, identity, and unitarity checking in black-box testing of quantum programs. We also explore optimization techniques for these algorithms, including specialized versions for equivalence and unitarity checking, and provide valuable insights into parameter selection to maximize performance and effectiveness. To evaluate the effectiveness of our proposed methods, we conducted comprehensive experimental evaluations, which demonstrate that our methods can rigorously perform equivalence, identity, and unitarity checking, offering robust support for black-box testing of quantum programs.

1. Introduction

Quantum computing, which utilizes the principles of quantum mechanics to process information and perform computational tasks, is a rapidly growing field with the potential to revolutionize various disciplines (National Academies of Sciences, Engineering and Medicine et al., 2019). It holds promise for advancements in optimization (Farhi et al., 2014), encryption (Mosca, 2018), machine learning (Biamonte et al., 2017), chemistry (McArdle et al., 2018), and materials science (Yang et al., 2017). Quantum algorithms, when compared to classical algorithms, offer the potential to accelerate the solution of specific problems (Deutsch, 1985; Grover, 1996; Shor, 1999). As quantum hardware devices and algorithms continue to develop, the importance of creating high-quality quantum software has become increasingly evident. However, the nature of quantum programs, with their special characteristics such as superposition, entanglement, and non-cloning theorems, makes it challenging to track errors in these programs (Miranskyy et al., 2020; Huang and Martonosi, 2019). Therefore, effective testing of quantum programs is crucial for the advancement of quantum software development.

Black-box testing (Beizer, 1995) is a software testing method that assesses the functionality of a program without examining its internal structure or implementation details. This method has broad applications for identifying software errors and improving software reliability. The inherently non-deterministic behavior of quantum programs makes error detection more challenging than in classical programs. Additionally, due to the potential interference of measurement with quantum states, observing the internal behavior of quantum programs becomes nearly impossible. Consequently, black-box testing assumes a crucial role in testing quantum programs. While several testing methods for quantum programs have been proposed (Abreu et al., 2022; Ali et al., 2021; Fortunato et al., 2022a; Li et al., 2020; Miranskyy and Zhang, 2019; Wang et al., 2018), these methods have paid limited attention to the fundamental questions in black-box testing of quantum programs. Important questions essential to black-box testing for quantum programs have remained largely unexplored.

Black-box testing of quantum programs refers to testing these programs based solely on selecting the appropriate inputs and detecting the corresponding outputs. To effectively address the challenges associated with black-box testing in quantum programs, it is essential to explore and answer the following fundamental research questions (RQs):

(1) **Equivalence Checking:** Given two quantum programs $\mathcal{P}$ and $\mathcal{P}'$, how can we determine whether they are equivalent?

☆ Editor: Raffaela Mirandola.

* Corresponding author at: State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing, China.
E-mail addresses: longpx@ios.ac.cn (P. Long), zhao@ait.kyushu-u.ac.jp (J. Zhao).

https://doi.org/10.1016/j.jss.2024.112000
Received 4 July 2023; Received in revised form 1 December 2023; Accepted 10 February 2024
Available online 15 February 2024

(2) **Identity Checking:** Given a quantum program $\mathcal{P}$, how can we check whether it represents an identity transform?
(3) **Unitarity Checking:** Given a quantum program $\mathcal{P}$, how can we ascertain whether it represents a unitary transform?

In this paper, we aim to address these RQs and lay the foundation for black-box testing of quantum programs. We propose three novel methods that specifically target equivalence, identity, and unitarity checking in quantum programs. For equivalence checking, we introduce a novel algorithm based on the Swap Test, which compares the outputs of two quantum programs on Pauli input states. To simplify identity checking, we present a straightforward algorithm to avoid repeated running. Additionally, we derive a critical theorem that provides a necessary and sufficient condition for a quantum operation to be a unitary transform, enabling us to develop an effective unitarity checking algorithm.

Furthermore, we conduct theoretical analyses and experimental evaluations to demonstrate the effectiveness and efficiency of our proposed algorithms. We also discuss the optimization of our algorithms. The results of our evaluations confirm that our algorithms successfully perform equivalence, identity, and unitarity checking, supporting the black-box testing of quantum programs.

In summary, our paper makes the following contributions:

- **A theorem on unitarity checking:** We proved a critical theorem about how to check whether a quantum operation is a unitary transform, which is a basis for developing our algorithm for unitarity checking.
- **Checking algorithms:** We develop three novel algorithms for checking the equivalence, identity, and unitarity of quantum programs, respectively.
- **Algorithm optimization:** We explore the optimization of our checking algorithms by devising optimized versions that specifically target equivalence and unitarity checking. We also discuss and provide detailed insights into the selection of algorithm parameters to maximize their performance and effectiveness.

Through these contributions, our paper advances the foundation of black-box testing for quantum programs and provides valuable insights for quantum software development.

The organization of this paper is as follows. Section 2 introduces some basic concepts and technologies of quantum computation. We discuss questions and their motivations we want to address in this paper in Section 3. Section 4 presents novel algorithms to solve these questions. Section 5 discusses the optimization of equivalence checking and unitarity checking. Section 6 discusses the experimental evaluation. Section 7 discusses the threats to the validity of our methods. We discuss related work in Section 8, and the conclusion is given in Section 9.

2. Background

We next introduce some background knowledge about quantum computation which is necessary for understanding the content of this paper.

2.1. Basic concepts of quantum computation

A *qubit* is the fundamental unit of quantum computation. Like the classical bit has values 0 and 1, a qubit also has two *basis states* with the form $|0\rangle$ and $|1\rangle$. However, a qubit is allowed to contain a *superposition* between basis states, with the general state being a linear combination of $|0\rangle$ and $|1\rangle$: $a|0\rangle + b|1\rangle$, where a and b are two complex numbers called *amplitudes* that satisfy $|a|^2 + |b|^2 = 1$. The amplitudes describe the proportion of $|0\rangle$ and $|1\rangle$. For multiple qubits, the basis states are analogous to binary strings. For example, a two-qubit system has four basis states: $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$, and the general state is $a_{00}|00\rangle+a_{01}|01\rangle+a_{10}|10\rangle+a_{11}|11\rangle$, where $|a_{00}|^2+|a_{01}|^2+|a_{10}|^2+|a_{11}|^2 = 1$. The state can also be written as a column vector $[a_{00}, a_{01}, a_{10}, a_{11}]^T$, which is called *state vector*. Generally, the quantum state on n qubits can be represented as a state vector $|\psi\rangle$ in *Hilbert Space* of d dimension, where $d = 2^n$.

The states mentioned above are all *pure states*, which means they are not probabilistic. Sometimes a quantum system may have a probability distribution over several pure states rather than a certain state, which we call a *mixed state*. Suppose a quantum system is in state $|\psi_i\rangle$ with probability p_i. The completeness of probability gives that $\sum_i p_i = 1$. We can denote the mixed state as the *ensemble representation*: $\{(p_i, |\psi_i\rangle)\}$.

Besides state vectors, the *density matrix* or *density operator* is another way to express a quantum state, which is convenient for expressing mixed states. Suppose a mixed state has the ensemble representation $\{(p_i, |\psi_i\rangle)\}$, the density matrix of this state is $\rho = \sum_i p_i|\psi_i\rangle\langle\psi_i|$, where $\langle\psi_i|$ is the conjugate transpose of $|\psi_i\rangle$ (thus it is a row vector and $|\psi_i\rangle\langle\psi_i|$ is a $d \times d$ matrix). Owing that the probability $p_i \geq 0$ and $\sum_i p_i = 1$, a lawful and complete density matrix ρ should satisfy: (1) $tr(\rho) = 1$ and (2) ρ is a positive matrix[1]. ρ represents a pure state if and only if $tr(\rho^2) = 1$ (Nielsen and Chuang, 2010). Obviously, The density matrix of a pure state $|\phi\rangle$ can be written as $|\phi\rangle\langle\phi|$. In this paper, we will use both state vector and density matrix representations.

2.2. Evolution of quantum states

Quantum computing is performed by applying proper *quantum gates* on qubits. An n-qubit quantum gate can be represented by a $2^n \times 2^n$ unitary matrix G. Applying gate G on a state vector $|\psi\rangle$ will obtain state vector $G|\psi\rangle$. Moreover, applying G on a density matrix ρ will obtain density matrix $G\rho G^\dagger$, where $G^\dagger$ is the *conjugate transpose* of G. For a unitary transform, $G^\dagger = G^{-1}$, where G^{-1} is the *inverse* of G.

There are several basic quantum gates, such as single-qubit gates X, Z, S, H, and two-qubit gate CNOT. The matrices of them are:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix},$$

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In quantum devices, the information of qubits can only be obtained by *measurement*. Measuring a quantum system will get a classical value with the probability of corresponding amplitude. Then the state of the quantum system will collapse into a basis state according to obtained value. For example, measuring a qubit $a|0\rangle + b|1\rangle$ will obtain result 0 and collapse into state $|0\rangle$ with probability $|a|^2$; and obtain result '1' and collapse into state $|1\rangle$ with probability $|b|^2$. This property brings uncertainty and influences the testing of quantum programs.

A quantum circuit is a popular model to express the process of quantum computing. Every line represents a qubit in a quantum circuit model, and a sequence of operations is applied from left to right. Figs. 1 and 2 are two examples of quantum circuits.

2.3. Quantum operations and quantum programs

Quantum operation is a general mathematical model that can describe the general evolution of a quantum system (Nielsen and Chuang, 2010). It is a linear map on the space of the density matrix. In fact, both quantum gates and measurements can be represented by quantum operations. Suppose we have an input density matrix ρ ($d \times d$ matrix). After an evolution through quantum operation $\mathcal{E}$, the density matrix becomes $\mathcal{E}(\rho)$. $\mathcal{E}(\rho)$ can be represented as the *operator-sum representation* (Nielsen

[1] For any column vector $|\alpha\rangle$, $\langle\alpha|\rho|\alpha\rangle \geq 0$

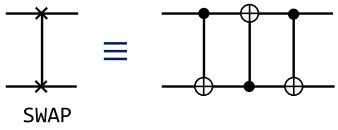


Fig. 1. A SWAP gate that can be implemented by three CNOT gates.

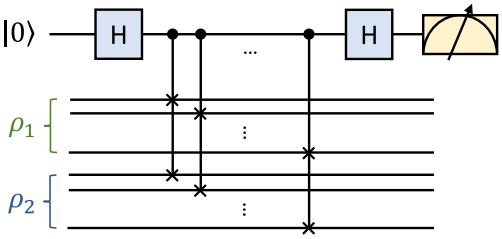


Fig. 2. The quantum circuit of Swap Test.

and Chuang, 2010): $\mathcal{E}(\rho) = \sum_i E_i \rho E_i^\dagger$, where $\{E_i\}$ is a group of $d \times d$ matrices. If $\sum_i E_i^\dagger E_i = I$, then $\mathcal{E}$ is called a *trace-preserving* quantum operation.

In developing quantum programs, programmers use quantum gates, measurements, and control statements to implement some quantum algorithms. A quantum program usually transforms the quantum input state into another state. An important fact is that a quantum program with *if* statements and *while-loop* statements, where the *if* and *while-loop* conditions can contain the result of measuring qubits, can be represented by a quantum operation (Mingsheng, 2016). So, quantum operations formalism is a powerful tool that can be used by our testing methods to model the state transformations of quantum programs under test. In addition, a program will eventually terminate if and only if it corresponds to a trace-preserving quantum operation (Mingsheng, 2016). This paper focuses only on trace-preserving quantum operations, i.e., the quantum programs that always terminate.

2.4. Swap test

The Swap Test (Buhrman et al., 2001; Barenco et al., 1997) is a procedure in quantum computation that allows us to determine how similar two quantum states are. Fig. 2 shows the quantum circuit of the Swap Test. It includes two (single or multi-qubit) registers carrying the states ρ_1 and ρ_2 and a single ancillary qubit initialized in state $|0\rangle$. It contains a series of "Controlled-SWAP" gates on each pair of corresponding qubits of target states ρ_1 and ρ_2, and the ancilla qubit is used for controlling. SWAP gate can be implemented as three CNOT gates as shown in Fig. 1. There are two H gates on the ancilla qubit before and after these Controlled-SWAP gates, respectively. The ancilla qubit is measured at the end, and the probability that result '1' occurs, denoted as p_1, is related to states ρ_1 and ρ_2 as shown in the following formula (1):

$$tr(\rho_1\rho_2) = 1 - 2p_1 \tag{1}$$

Note that the probability p_1 can be estimated by repeating running the Swap Test and counting the proportion of obtaining result '1'. Based on this process, we can estimate parameter $tr(\rho_1\rho_2)$ and then obtain more useful information on states ρ_1 and ρ_2. In particular, if we set $\rho_1 = \rho_2 = \rho$, we can estimate $tr(\rho^2)$ and use it to determine whether ρ is a pure or mixed state. Also, if we let ρ_1 and ρ_2 be two pure states, where $\rho_1 = |\alpha\rangle\langle\alpha|$ and $\rho_2 = |\beta\rangle\langle\beta|$, then we have $tr(\rho_1\rho_2) = |\langle\alpha|\beta\rangle|^2$, where $\langle\alpha|\beta\rangle$ represents the *inner product* of two state $|\alpha\rangle$ and $|\beta\rangle$. We call $|\alpha\rangle$ and $|\beta\rangle$ are *orthogonal*, denoted as $|\alpha\rangle \perp |\beta\rangle$, if $\langle\alpha|\beta\rangle = 0$.

2.5. Quantum tomography

Quantum tomography (D'Ariano et al., 2003) is used for obtaining the details of a quantum state or operation. Owing that measurement may collapse a quantum state, we need many copies of the target quantum state or repeat the target quantum operation many times to reconstruct the target. *State tomography* reconstructs the information of a quantum state, and *process tomography* reconstructs the information of a quantum operation.

In quantum tomography, The following four *Pauli matrices* are important:

$$\sigma_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$
$$\sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

All σ_1, σ_2 and σ_3 have two eigenvalues: -1 and 1. σ_3 has eigenstates $|0\rangle = [1,0]^T$ (for eigenvalue 1) and $|1\rangle = [0,1]^T$ (for eigenvalue -1). σ_1 has eigenstates $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$ (for 1) and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$ (for -1). σ_2 has eigenstates $|+_i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$ (for 1) and $|-_i\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$ (for -1). σ_0 has only one eigenvalue 1, and any single-qubit state is its eigenstate.

An important fact is that the four Pauli matrices form a group of bases of the space of the single-qubit density matrix. Similarly, the tensor product of Pauli matrices $\sigma_{\vec{v}} = \sigma_{v_1} \otimes \sigma_{v_2} \otimes \cdots \otimes \sigma_{v_n}$ form a basis of an n-qubit density matrix, where $v_i \in \{0,1,2,3\}$, $\vec{v} = (v_1, \ldots, v_n)$. So a density matrix ρ can be represented as the linear combination of Pauli basis (Nielsen and Chuang, 2010):

$$\rho = \sum_{\vec{v}} \frac{tr(\sigma_{\vec{v}}\rho)}{2^n}\sigma_{\vec{v}}, \tag{2}$$

where the combination coefficient for basis matrix $\sigma_{\vec{v}}$ is $tr(\sigma_{\vec{v}}\rho)/2^n$. In practice, $tr(\sigma_{\vec{v}}\rho)$ can be estimated by *Pauli measurement* of Pauli matrix $\sigma_{\vec{v}}$. We can conduct experiments to observe each $tr(\sigma_{\vec{v}}\rho)$ on ρ and then use formula (2) to reconstruct the density matrix of ρ.

A quantum operation $\mathcal{E}$ is a linear transformation on the input space, so it depends on the behavior of a group of bases. We can use state tomography to reconstruct the corresponding output density matrices of all basis input states, and then $\mathcal{E}$ can be reconstructed by these output density matrices (Nielsen and Chuang, 2010).

3. Questions and motivations

We next outline the key questions addressed in this paper and provide a comprehensive rationale for each question, supported by relevant scenarios. By doing so, we aim to highlight the motivation and necessity behind investigating these specific aspects.

3.1. Modeling black-box quantum programs

In this paper, we focus on black-box testing, where we do not require prior knowledge of the internal structure of the target program. Instead, we achieve the testing objective by observing the program's outputs when provided with suitable inputs. This approach allows us to evaluate the program's behavior without relying on specific implementation details.

The beginning of our discussion is to model a black-box quantum program. In Ali et al. (2021), Ali et al. gives some definitions of a quantum program, input, output, and program specification to guide the generation of test cases. They define the program specification as the expected probability distribution of the classical output values under the given inputs. As Long and Zhao (2023) points out, this definition is inherently classical, implying that the program ends with measurements to transform quantum information into classical probabilities. It is more reasonable to deal directly with quantum information

rather than transform them into classical probabilities. Accordingly, the quantum input and output of the program can be modeled by quantum states (state vectors or density matrices).

According to Section 2.2, a quantum program with *if* statements and *while-loop* statements can be represented by a quantum operation (Mingsheng, 2016). So in this paper, we use the quantum operation model to represent a black-box quantum program. A black-box quantum program with quantum input variable ρ can be represented by an unknown quantum operation $\mathcal{E}$, and $\mathcal{E}(\rho)$ represents the corresponding output under input ρ. The research on the properties of quantum programs can be transformed into research on quantum operations.

3.2. Questions

We introduce the key questions addressed in this paper, namely *equivalence checking*, *identity checking*, and *unitarity checking*. While a quantum program can involve classical inputs and outputs, our focus in this paper is solely on the quantum aspects. Specifically, we aim to examine the checking of quantum inputs and outputs while keeping other classical parameters fixed. We outline the three questions addressed in this paper as follows:

Question 1. ***Equivalence Checking***

Given two quantum programs $\mathcal{P}_1$ and $\mathcal{P}_2$, how can we determine whether they are equivalent, meaning they produce the same quantum output for the same quantum input?

Question 2. ***Identity Checking*** *Given a quantum program $\mathcal{P}$, how can we check whether it represents an identity transform where the input qubits remain unchanged?*

Question 3. ***Unitarity Checking***

Given a quantum program $\mathcal{P}$, how can we determine whether $\mathcal{P}$ represents a unitary transform?

3.3. Motivations

Next, we discuss the motivation and necessity of each question. We will present several scenarios highlighting the practical application of equivalence, identity, and unitary checking in the context of black-boxing testing of quantum programs. We consider these three questions together because they involve typical relations and share similar problem-solving frameworks. Identity checking can be seen as a special case of equivalence checking, with one program being the target $\mathcal{P}$ and the other being the identity operator I. However, we also examine identity checking independently for two reasons: firstly, identity relations are among the most common, and secondly, it may be possible to develop a more efficient algorithm for identity checking compared to equivalence checking.

The equivalence of two programs holds substantial importance, as it is a prevalent relationship. Even when testing classical programs, checking the equivalence of two programs is a fundamental testing task. For instance, in the following Scenario 1, one of the typical applications of equivalence checking is to guarantee the correctness of an updated program version. While white-box checking for quantum circuit equivalence has been explored in several existing studies (Viamontes et al., 2007; Yamashita and Markov, 2010; Burgholzer and Wille, 2020; Hong et al., 2021, 2022; Wang et al., 2022b), our focus in this paper is on black-box checking.

Scenario 1. *Correctness Assurance in Version Update*

One important application of equivalence checking is to ensure the correctness of an updated version of a quantum program in regression testing. Let us consider a scenario where we have an original program $\mathcal{P}$, and after a few months, we develop an updated version $\mathcal{P}'$ by making improvements such as optimizing the program's structure to enhance its execution speed. In this case, ensuring that these modifications do not alter the program's behavior becomes crucial, meaning that $\mathcal{P}'$ should be equivalent to $\mathcal{P}$. By performing an ***equivalence checking****, we can verify if the updated version $\mathcal{P}'$ maintains the same functionality as the original version $\mathcal{P}$.*

The classical counterpart to identity checking, which involves verifying whether a classical program reproduces its input, lacks practical significance. Identity checking finds significance in the context of quantum programs because creating a *inverse variant* of the original quantum program is a common practice in quantum programming. Many quantum programs inherently involve performing a unitary transformation on the quantum input state. As detailed in Section 2.2, any unitary transformation U possesses an inverse denoted as U^{-1}, and they satisfy the relation $UU^{-1} = I$, where I denotes the identity transformation. Scenario 2 illustrates how identity checking can be employed to test the inverse variant of the original quantum program.

Scenario 2. *Testing the Inverse Variant*

Suppose we have completed a testing task for an original program P. Now, we proceed to implement the reverse version invP of P. While some quantum programming languages, such as Q# (Svore et al., 2018) and isQ (Guo et al., 2023), offer mechanisms for generating invP automatically, these mechanisms come with some restrictions on the target program. For instance, if P contains if statements associated with classical input parameters, the quantum behavior of P is a unitary transform for any fixed classical parameters, implying the existence of the inverse invP. However, these languages may not have the ability to automatically generate invP as their mechanisms are limited to dealing with simple programs. In such cases, manual implementation is required. As a result, effective testing becomes crucial during the manual implementation process.

To test invP, as proposed in Long and Zhao (2023), we only need to check the following relation:

$$\texttt{P} \circ \texttt{invP} = I$$

where '$\circ$' represents the sequential execution (from right to left) of the two subroutines. This task involves ***checking the identity relationship*** *between the composite program and the identity program (I).*

Besides the inverse variant, as mentioned in Long and Zhao (2023), there are also two other variants of an original program P - *controlled variants* `CtrlP` and *power variants* `PowP`. For example, the testing process for power variants can be reduced to check the following relations:

$$\text{For } k > 0, \quad \texttt{Inv}\texttt{P}^{k} \circ \texttt{PowP}(k) = I;$$
$$\text{For } k < 0, \quad \texttt{P}^{|k|} \circ \texttt{PowP}(k) = I.$$

The motivation for unitarity checking is based on the fact that many practical quantum programs are unitary transformations, which means that they do not contain measurements. If the unitarity checking fails for these programs, we can know that there exist some unexpected measurements in them. In fact, unitarity checking reflects the unique properties of quantum programs, and there is no classical counterpart to this type of checking. Scenario 3 gives another case where we need to employ unitarity checking.

Scenario 3. *Checking the Program Specification*

Checking whether the program output conforms to the program specification is a crucial step in testing. However, unlike classical programs, where the output can be directly observed, checking the output of quantum programs under arbitrary inputs can be challenging. Fortunately, as discussed in Long and Zhao (2023), if the intended behavior of the program is to perform a unitary transform, we can simplify the checking process. It involves:

(1) Checking the output under classical input states;
(2) Performing additional ***unitarity checking*** *on the program.*

In practice, it may appear that testing all possible quantum inputs is necessary, yet the number of quantum inputs is infinite. Fortunately, the linearity of quantum operations enables us to examine only a finite subset of all potential inputs. Moreover, our objective is testing rather than rigorous mathematical proof. Consequently, we only need to select a small, appropriate set of test cases to validate.

4. Checking algorithms

We next introduce concrete checking algorithms to solve these questions mentioned in Section 3. We discuss the theoretical foundations for each algorithm first and then give concrete algorithms. We denote n as the number of qubits of target programs. According to the discussion in Section 3.1, we use quantum operation as the equivalent of a quantum program in the following sections.

4.1. Implementation of swap test

First, we discuss the implementation of the Swap Test, as it is the fundamental of checking algorithms. The quantum circuit for the Swap Test is shown in Fig. 2. According to the formula (1), a useful parameter is the probability of returning result '1'. So our implementation returns the number of occurrences of result '1', with a given number of repeat s of the Swap Test. The initial states ρ_1 and ρ_2 are generated by two input subroutines $\mathcal{P}_1$ and $\mathcal{P}_2$.

The implementation of the Swap Test is shown in Algorithm 1. Lines $3 \sim 4$ initialize quantum variables, where $|0\rangle^{\otimes n}$ means n-qubit all-zero state. Line 5 prepare the input states by $\mathcal{P}_1$ and $\mathcal{P}_2$ on quantum variables **qs1** and **qs2**, respectively. Lines $6 \sim 8$ implement a series of controlled-SWAP operations in Fig. 2. Line 10 measures the ancilla qubit, and lines $11 \sim 13$ accumulate the number of results 1. Owing that there are two layers of nested loops (lines 2 and 6) and the iteration numbers are s and n, respectively, the time complexity of Algorithm 1 is $O(ns)$. As line 3, it requires $2n+1$ qubits.

Algorithm 1: SwapTest

```
Input: (n, s, P1, P2)
  n: the number of qubits of one target state;
  s: the number of rounds;
  P1, P2: two subroutines to generate two target states.
Output: s1: Number of occurrences of result '1'.
1  s1 ← 0;
2  for i = 1 to s do
3  |  qanc ← |0⟩; qs1 ← |0⟩^⊗n; qs2 ← |0⟩^⊗n;
4  |  H(qanc);
5  |  P1(qs1); P2(qs2);
6  |  for j = 0 to n − 1 do
7  |  |  Controlled SWAP(qanc, (qs1[j], qs2[j]));
8  |  end
9  |  H(qanc);
10 |  m ← Measure(qanc);
11 |  if m = 1 then
12 |  |  s1 ← s1 + 1;
13 |  end
14 end
15 return s1;
```

4.2. Equivalence checking

4.2.1. Theoretical foundations

A straightforward idea to perform the equivalence checking is to use the quantum process tomography (D'Ariano et al., 2003) (see Section 2.5) for two target programs and compare the reconstruction results of these programs. However, quantum process tomography is costly and may contain much unnecessary, redundant information for equivalence checking. However, we can construct the equivalence checking algorithm based on the idea of quantum process tomography — the behavior of a quantum operation $\mathcal{P}$ depends on its behavior under a group of bases of the input density matrix space.

In practical, a common choice of the bases is the Pauli bases $\{\sigma_{\vec{v}}\}$, where $\sigma_{\vec{v}} = \sigma_{v_1} \otimes \sigma_{v_2} \otimes \cdots \otimes \sigma_{v_n}$ is the tensor product of Pauli matrices, $v_i \in \{0,1,2,3\}$, and $\vec{v} = (v_1, \ldots, v_n)$. In other words, $\mathcal{P}(\rho)$, the output of quantum operation $\mathcal{P}$ under input ρ, depends on every $\mathcal{P}(\sigma_{\vec{v}})$, and $\sigma_{\vec{v}}$ can be further decomposed into the sum of its eigenstates $\sigma_{\vec{v}} = \sum_i \lambda_i^{\vec{v}} |\psi_i^{\vec{v}}\rangle\langle\psi_i^{\vec{v}}|$, where $\lambda_i^{\vec{v}}$ is the ith eigenvalue of $\sigma_{\vec{v}}$ and $|\psi_i^{\vec{v}}\rangle$ is the corresponding eigenstate. So the behavior of a quantum operation depends on the output on every input $|\psi_i^{\vec{v}}\rangle$. Specifically, $|\psi_i^{\vec{v}}\rangle$ is the tensor product of single-qubit Pauli eigenstate[2]:

$$|\psi_i^{\vec{v}}\rangle \in \{|0\rangle, |1\rangle, |+\rangle, |-\rangle, |+_i\rangle, |-_i\rangle\}^{\otimes n} = A \tag{3}$$

where states $|+\rangle$, $|-\rangle$, $|+_i\rangle$ and $|-_i\rangle$ are defined in Section 2.5. Thus, the behavior of a quantum operation depends on the output under every input state in set A. We call the states in set A as *Pauli input states*. Note that no matter what Pauli index vector $\vec{v}$ is, state $|\psi_i^{\vec{v}}\rangle$ has the form of formula (3). It deduces that two quantum operations are equivalent if and only if their outputs are equivalent on all 6^n Pauli input states.[3]

Commonly, the equivalent of two states ρ_1 and ρ_2 can be judged by *trace distance* $D(\rho_1, \rho_2) = \frac{1}{2} tr|\rho_1 - \rho_2|$ or *fidelity* $F(\rho_1, \rho_2) = tr\sqrt{\sqrt{\rho_1}\rho_2\sqrt{\rho_1}}$ (Nielsen and Chuang, 2010). $\rho_1 = \rho_2$ if and only if $D(\rho_1, \rho_2) = 0$ or $F(\rho_1, \rho_2) = 1$. However, these two parameters are complex to estimate in practice. For a testing task, we need a parameter that is experimentally easy to be obtained. Consider the *Cauchy Inequality* of density matrix $tr(\rho_1\rho_2) \le \sqrt{tr(\rho_1^2)tr(\rho_2^2)}$, and the *Mean Value Inequation* $\sqrt{xy} \le \frac{x+y}{2}$, we have

$$tr(\rho_1\rho_2) \le \frac{tr(\rho_1^2) + tr(\rho_2^2)}{2} \tag{4}$$

with equality if and only if $\rho_1 = \rho_2$. We can construct a parameter:

$$E(\rho_1, \rho_2) = \left| \frac{tr(\rho_1^2) + tr(\rho_2^2)}{2} - tr(\rho_1\rho_2) \right| \tag{5}$$

If $\rho_1 = \rho_2$, then $E(\rho_1, \rho_2) = 0$; otherwise $E(\rho_1, \rho_2) > 0$. So given two quantum programs $\mathcal{P}_1$ and $\mathcal{P}_2$, we can estimate $E(\mathcal{P}_1(|\psi_i\rangle\langle\psi_i|), \mathcal{P}_2(|\psi_i\rangle\langle\psi_i|))$, where $|\psi_i\rangle$ is taken over all Pauli eigenstates. To estimate $E(\rho_1, \rho_2)$, we need to estimate $tr(\rho_1\rho_2)$, $tr(\rho_1^2)$ and $tr(\rho_2^2)$, and they can be finished by Swap Test (see Section 2.4). Suppose we repeat s times for input pairs (ρ_1, ρ_1), (ρ_2, ρ_2), (ρ_1, ρ_2), and there exist s_1, s_2, and s_{12} times obtaining result '1', respectively. According to formula (1), $tr(\rho_1^2) \approx 1 - \frac{2s_1}{s}$, $tr(\rho_2^2) \approx 1 - \frac{2s_2}{s}$, and $tr(\rho_1\rho_2) \approx 1 - \frac{2s_{12}}{s}$. By substituting them into the formula (5), we have

$$E(\rho_1, \rho_2) \approx \left| \frac{2s_{12} - s_1 - s_2}{s} \right| = \tilde{E} \tag{6}$$

Denote the experimental estimated value of $E(\rho_1, \rho_2)$ as $\tilde{E}$. It seems that we can simply compare $\tilde{E}$ with 0. However, since there will be some errors in the experiment, a better solution is to give a tolerance error ϵ. If $\tilde{E} \le \epsilon$, which means $\tilde{E}$ is close to 0, we think that ρ_1 and ρ_2 are equivalent.

There are 6^n Pauli eigenstates. Fortunately, in program testing tasks, we can tolerate small errors, so we just need to test only a small fraction of the input pairs instead of all of them, i.e., only k input states, where $k \ll 6^n$. We call the checking process for each input a *test point*. For each input state $|\psi_i\rangle$, we estimate $\tilde{E}$ and return FAIL whenever there is at least one case such that $\tilde{E} > \epsilon$. PASS is returned only when all test points satisfy $\tilde{E} \le \epsilon$.

[2] Any single-qubit state is the eigenstate of σ_0, so we only need to consider the eigenstates of σ_1, σ_2, and σ_3.

[3] Actually, the degree of freedom for an n-qubit density matrix is 4^n, indicating that some Pauli states lack independence. Nevertheless, this is inconsequential for our methods, as they only require sampling a small subset of Pauli input states.

4.2.2. Algorithm

According to the above discussion, given two target quantum programs $\mathcal{P}_1$ and $\mathcal{P}_2$, the idea of equivalence checking algorithm includes the following three steps:

(1) Randomly select and generate a set of Pauli input states.
(2) For each input state $|\psi\rangle$, run $\mathcal{P}_1$ and $\mathcal{P}_2$, and use Swap Test to check the equivalence of their output states.
(3) As long as one output pair is not equivalent, return FAIL; otherwise, if all output pairs are equivalent, return PASS.

In practice, we can represent each Pauli input state using an n-element integer array, where each integer ranges from 0 to 5 and corresponds to the single-qubit states defined in formula (3). For instance, if $n = 3$, the array $K = [1, 2, 3]$ represents the input state $|\psi_K\rangle = |1\rangle|+\rangle|-\rangle$. Constructing the unitary transform G_K to generate state $|\psi_K\rangle$ from the all-zero state $|000\rangle$ is straightforward: it involves applying the gates X, H, and HX[4] to the 1st, 2nd, and 3rd qubits, respectively. In fact, the six single-qubit Pauli eigenstates in formula (3) can be generated by applying the gates I, X, H, HX, SH, and $S^{-1}H$ to the state $|0\rangle$, respectively.

The algorithm for equivalence checking is presented in Algorithm 2. We execute k test points (testing k input states, line 1). In each iteration of the loop, a random integer array K for the input state is generated (line 2), and the corresponding unitary transform to generate Pauli input state $|\psi_K\rangle$ is denoted as G_K. Notably, the `SwapTest` subroutine requires two parameters related to generation subroutines for identifying the target states. $\mathcal{P}_1 \circ G_K$ represents a composite subroutine that executes G_K and $\mathcal{P}_1$ successively, thereby generating the state that results from applying $\mathcal{P}_1$ to the input produced by G_K. The same applies to $\mathcal{P}_2 \circ G_K$. These two composite subroutines, $\mathcal{P}_1 \circ G_K$ and $\mathcal{P}_2 \circ G_K$, serve as input parameters for the `SwapTest` function. Consequently, lines $3 \sim 5$ return the results of the Swap Test for the following three pairs of states:

- Line 3: $\mathcal{P}_1(|\psi_K\rangle\langle\psi_K|)$, $\mathcal{P}_1(|\psi_K\rangle\langle\psi_K|)$
- Line 4: $\mathcal{P}_2(|\psi_K\rangle\langle\psi_K|)$, $\mathcal{P}_2(|\psi_K\rangle\langle\psi_K|)$
- Line 5: $\mathcal{P}_1(|\psi_K\rangle\langle\psi_K|)$, $\mathcal{P}_2(|\psi_K\rangle\langle\psi_K|)$

and the results are assigned to s_1, s_2, and s_{12} denoted in formula (6), respectively. Line 6 calculates parameter $\tilde{E}$ by formula (6). Lines $7 \sim 9$ compare $\tilde{E}$ with ϵ, and return FAIL if $\tilde{E}$ is over the limit of ϵ.

The loop is repeated for k rounds (line 2). In each round, three Swap Tests are executed, each requiring $O(ns)$ time and $2n+1$ qubits. So the time complexity of Algorithm 2 is $O(nks)$ and requires $2n+1$ qubits.

Algorithm 2: `EquivalenceChecking_original`

Input: $(n, k, s, \epsilon, \mathcal{P}_1, \mathcal{P}_2)$
n: the number of qubits of the target program;
k: the number of test points;
s: the number of rounds of Swap Test;
ϵ: tolerance error;
$\mathcal{P}_1, \mathcal{P}_2$: two target programs.
Output: PASS or FAIL.

```
1  for i = 1 to k do
2      K ← Randomly choose a n-element integer array, where each
         integer is in 0, ..., 5;
3      s1 ← SwapTest(n, s, P1∘GK, P1∘GK);
4      s2 ← SwapTest(n, s, P2∘GK, P2∘GK);
5      s12 ← SwapTest(n, s, P1∘GK, P2∘GK);
6      E ← (2s12 − s1 − s2)/s;
7      if |E| > ε then
8          return FAIL;
9      end
10 end
11 return PASS;
```

4.3. Identity checking

It seems that identity checking is a special case of equivalence checking — whether the target program P is equivalent to identity program I. Fortunately, identity checking has some good properties to avoid repeating running the Swap Test. Thus, it has a faster algorithm than equivalence checking.

Just like equivalence checking discussed in Section 4.2, we consider Pauli input state $|\psi_K\rangle$, which is generated by a unitary transform G_K from all-zero state, where K is the integer array to represent the input state. Suppose the target program is $\mathcal{P}$. If $\mathcal{P}$ is identity, it keeps any input state, and thus applying the inverse transform G_K^{-1} will recover the state back into the all-zero state. Then, measuring the state will always obtain a result of 0. In other words, after this process, if there is a non-zero result occurs, the target program $\mathcal{P}$ is not identity. Just like the equivalence checking, we test only a small subset (size k) of input states for identity checking. During the checking, FAIL is returned whenever one non-zero result occurs; otherwise, PASS is returned.

The algorithm for identity checking is shown in Algorithm 3. The algorithm first generates a random integer array K for the input state in line 2 and then initializes the quantum variable **qs** in line 3. After that, it executes the subroutines G_p, $\mathcal{P}$, and G_p^{-1} on **qs** (line 4) and makes a measurement on **qs** (line 5). Finally, the algorithm judges whether the measurement result is not 0 (lines $6 \sim 8$), and if so, returns FAIL immediately. PASS is returned only when all measurement results are 0 (line 10).

The loop is repeated for k rounds (line 2). So the time complexity of Algorithm 3 is $O(nk)$ and requires n qubits. Compared with equivalence checking (Algorithm 2), owing to the Swap Test being avoided in identity checking, it is faster and requires fewer qubits. In fact, this algorithm may be faster for checking a non-identity target program due to the immediate return of non-zero measurement results.

Algorithm 3: `IdentityChecking`

Input: $(n, k, \mathcal{P})$
n: the number of qubits of the target program;
k: the number of test points;
$\mathcal{P}$: the target program.
Output: PASS or FAIL.

```
1  for i = 1 to k do
2      K ← Randomly choose a n-element integer array, where each
         integer is in 0, ..., 5;
3      qs ← |0⟩^⊗n;
4      GK(qs); P(qs); GK^-1(qs);
5      r ← Measure(qs);
6      if r ≠ 0 then
7          return FAIL;
8      end
9  end
10 return PASS;
```

4.4. Unitarity checking

4.4.1. Theoretical foundations

Note that a unitarity transform exhibits two obvious properties: (1) it preserves the inner product of two input states, and (2) it maintains the purity of an input pure state. In the context of black-box testing, specifically for unitarity checking, we present a novel theorem that serves as a guiding principle for performing such checking. The formalization of this theorem is provided below.

Theorem 1. *Let $\mathcal{H}$ be a d-dimensional Hilbert space and*

[4] Applying X gate first and then applying H gate.

$\{|1\rangle, \ldots, |d\rangle\}$ is a group of standard orthogonal basis of $\mathcal{H}$. Let $|+_{mn}\rangle = \frac{1}{\sqrt{2}}(|m\rangle + |n\rangle)$ and $|-_{mn}\rangle = \frac{1}{\sqrt{2}}(|m\rangle - |n\rangle)$, where $m, n = 1, \ldots, d$ and $m \neq n$. A quantum operation $\mathcal{E}$ is a unitary transform if and only if it satisfies:

(1) $\forall m \neq n$, $tr\left[\mathcal{E}(|m\rangle\langle m|)\mathcal{E}(|n\rangle\langle n|)\right] = 0$;

(2) for $d-1$ combinations of $(m, n), m \neq n$ which form the edges of a connected graph with vertices $1, \ldots, d$, $tr\left[\mathcal{E}(|+_{mn}\rangle\langle +_{mn}|)\mathcal{E}(|-_{mn}\rangle\langle -_{mn}|)\right] = 0$.

The proof of Theorem 1 is presented in Appendix A. It is important to note that $|m\rangle \perp |n\rangle$ and $|+_{mn}\rangle \perp |-_{mn}\rangle$ hold true when $m \neq n$. Intuitively, Theorem 1 establishes that a quantum operation qualifies as a unitary transform if and only if it maps two orthogonal states into two orthogonal states. Essentially, the *preservation of orthogonality* emerges as the fundamental characteristic of unitary transforms. For a more comprehensive exploration of these properties and their impact on the unitarity estimation of quantum channels, Chen et al. (2023) have conducted an in-depth study.

According to Theorem 1, determining whether a quantum program $\mathcal{P}$ qualifies as a unitary transform necessitates estimating parameters of the form $tr\left(\mathcal{P}(\rho_1)\mathcal{P}(\rho_2)\right)$, where ρ_1 and ρ_2 denote input states. These input pairs (ρ_1, ρ_2) should encompass two categories: (a) pairs of computational basis states $(|m\rangle, |n\rangle)$, where $m \neq n$, and (b) pairs of superposition states $(|+_{mn}\rangle, |-_{mn}\rangle)$, where $m \neq n$. Similar to equivalence checking and identity checking, we need to test only a small subset of all possible input pairs, ensuring coverage of both type (a) and type (b). In practical implementation, we employ the Swap Test for evaluating $tr\left(\mathcal{P}(\rho_1)\mathcal{P}(\rho_2)\right)$. Let us assume there are a total of s rounds for the Swap Test on each input, with s_1 rounds yielding the result '1'. Based on formula (1), when $tr\left(\mathcal{P}(\rho_1)\mathcal{P}(\rho_2)\right) = 0$, we have $r = 1 - \frac{2s_1}{s} = 0$. In practice, we set a tolerance value ϵ. Our testing approach returns a FAIL outcome if there exists a test point where $|r| > \epsilon$, and it returns a PASS only if all test points satisfy $|r| \leq \epsilon$.

4.4.2. Algorithm

Based on the preceding discussion, the unitarity checking algorithm for a target quantum program $\mathcal{P}$ involves the following three main steps:

(1) Randomly generate two types of input state pairs: (a) pairs of computational basis states $(|m\rangle, |n\rangle)$, where $m \neq n$; (b) pairs of superposition states $(|+_{mn}\rangle, |-_{mn}\rangle)$, where $m \neq n$. Ensure that each type constitutes approximately half of the total number of test points k.
(2) For each input pair (ρ_1, ρ_2) created in step (1), apply $\mathcal{P}$ to both input states, resulting in the output pair $(\mathcal{P}(\rho_1), \mathcal{P}(\rho_2))$. Employ the Swap Test to determine whether $tr(\mathcal{P}(\rho_1)\mathcal{P}(\rho_2))$ equals 0.
(3) If any group of output pairs $(\mathcal{P}(\rho_1), \mathcal{P}(\rho_2))$ yields $tr(\mathcal{P}(\rho_1)\mathcal{P}(\rho_2)) \neq 0$, the algorithm reports FAIL; otherwise, if all output pairs yield $tr(\mathcal{P}(\rho_1)\mathcal{P}(\rho_2)) = 0$, it reports PASS.

The concrete method is presented in Algorithm 4. The "if" statement in line 2 enacts the classification sampling. Lines 3 to 5 deal with input states of type (b), while lines 7 to 8 manage input states of type (a). In our implementation, for type (b), we select n, which is the bitwise negation of m, to ensure the superposition covers all qubits (as per the SCAQ criterion in Long and Zhao (2023)). The generation processes of computational basis states $|m\rangle$ and $|n\rangle$ are denoted as C_m and C_n, respectively, while the generation processes of superposition states $|+_{mn}\rangle$ and $|-_{mn}\rangle$ are denoted as $S^+_{m,n}$ and $S^-_{m,n}$. Similar to Algorithm 2, the notation $\mathcal{P}\circ\mathcal{X}$ signifies the subroutine that executes $\mathcal{P}$ on the input state generated by $\mathcal{X}$. Consequently, lines 5 and 8 implement the required Swap Test on the output pair $(\mathcal{P}(\rho_1), \mathcal{P}(\rho_2))$ to obtain s_1 for two different cases, respectively. Line 10 calculates the parameter $tr(\mathcal{P}(\rho_1)\mathcal{P}(\rho_2))$, and line 11 determines whether it equals 0 with a tolerance ϵ. If it does not, it immediately returns FAIL (line 12).

The loop is repeated for k rounds (line 2). In each round, three Swap Tests are executed, each requiring $O(ns)$ time and $2n+1$ qubits. So the time complexity of Algorithm 4 is $O(nks)$ and requires $2n+1$ qubits.

Algorithm 4: `UnitarityChecking_original`

Input: $(n, k, s, \epsilon, \mathcal{P})$
n: the number of qubits of the target program;
k: the number of test points;
s: the number of rounds of Swap Test;
ϵ: tolerance error;
$\mathcal{P}$: the target program.
Output: PASS or FAIL.

```
1  for i = 1 to k do
2      if i ≤ ⌈k/2⌉ then
3          m ← Randomly choose a number from 0 to 2^n − 1;
4          n ← Bitwise negation of m;
5          s_1 ← SwapTest(n, s, P∘S^+_{m,n}, P∘S^−_{m,n});
6      else
7          m, n ← Randomly choose two different numbers from 0 to
             2^n − 1;
8          s_1 ← SwapTest(n, s, P∘C_m, P∘C_n);
9      end
10     r ← 1 − (2s_1/t);
11     if |r| > ε then
12         return FAIL;
13     end
14 end
15 return PASS;
```

4.5. Parameter selection

In Algorithms 2, 3, and 4, several parameters need to be selected by users — number of test points k, number of rounds of Swap Test s, and the tolerance error ϵ (the latter two parameters are required in equivalence checking and unitarity checking because they are based on Swap Test). In this section, we discuss how these parameters influence the effectiveness of our testing methods and how to select these parameters.

Due to quantum programs' nondeterministic nature, testing algorithms may output wrong results. Consider that the target program has two statuses: *Correct* or *Wrong*, and the output also has two statuses: *PASS* or *FAIL*. So, there are two types of wrong results:

- **Error Type I:** A wrong program passes;
- **Error Type II:** A correct program fails.

The general principle of parameter selection is to balance the accuracy and running time. To increase the accuracy, we should try to control the probabilities of both two types of errors.

In identity checking, the unique parameter is the number of test points k. Because an identity program always returns a result 0 in identity checking, a type II error will not occur. For a non-identity program, the average probability of returning result 0, denoted as p, satisfies $0 < p < 1$. Taking k test points, the probability of type I error is p^k. It means that with the increase of k, the probability of type I error decreases exponentially. Therefore, selecting a moderately sized value for k is sufficient to control the probability of errors.

In the equivalence and unitarity checking, there are three parameters: number of test points k, number of rounds of Swap Test s, and tolerance ϵ. They are based on statistics, and both type I and type II errors can possibly occur due to the estimation errors. Intuitively, larger k (testing more points) and smaller ϵ (more strict judgment condition for the correctness) are helpful to reduce type I error. However, wrong programs are diverse, and we cannot know any information about errors before testing. So, the selection of k and ϵ is more empirical. In Section 6, we will find an appropriate selection of k and ϵ according to some benchmark programs.

If we choose a smaller ϵ, i.e., the judgment condition for the correctness is more strict, we need to improve the accuracy of the Swap Test to avoid type II error, i.e., select a larger s. Fortunately,

Table 1
The choice of s with different k and ϵ under $\alpha_2 = 0.1$.

	Equivalence Check				Unitarity Check			
k \ ϵ	0.05	0.10	0.15	0.20	0.05	0.10	0.15	0.20
1	9587	2397	1066	600	–	–	–	–
2	11722	2931	1303	733	3428	857	381	215
3	12991	3248	1444	812	3886	972	432	243
4	13898	3475	1545	869	4213	1054	469	264
6	15181	3796	1687	949	4676	1169	520	293
10	16804	4202	1868	1051	5261	1316	585	329

owing to the behavior of the correct program is unique, given k and ϵ, the parameter s can be calculated before testing. We need an extra parameter α_2, the required maximum probability of type II error. Then, we have the following proposition:

Proposition 1. *In equivalence checking or unitarity checking, suppose we have selected the number of test points k and tolerance ϵ. Given the allowed probability of type II error α_2. If the number of rounds of Swap Test s satisfies:*

(1) $s \geq \frac{8}{\epsilon^2} \ln \frac{2}{1-(1-\alpha_2)^{\frac{1}{k}}}$ *for equivalence checking, or*

(2) $s \geq \frac{2}{\epsilon^2 \ln 2} \ln \frac{1}{1-(1-\alpha_2)^{\frac{1}{k}}}$ *for unitarity checking,*

then the probability of type II error will not exceed α_2.

As a practical example, Table 1 gives the selection of s on several groups of k and ϵ under $\alpha_2 = 0.1$. The lower bound of s seems complicated, but we can prove:

$$\frac{k}{-\ln(1-\alpha_2)} \leq \frac{1}{1-(1-\alpha_2)^{\frac{1}{k}}} \leq \frac{k}{\alpha_2} \tag{7}$$

where $k \geq 1$ and $0 < \alpha_2 < 1$. It deduces:

$$\frac{1}{1-(1-\alpha_2)^{\frac{1}{k}}} = \Theta(k) \tag{8}$$

The proofs of Proposition 1 and formula (7) are shown in Appendix B. Proposition 1 and formula (8) deduce the following corollary about the asymptotic relations of s and the overall time complexity of Algorithms 2 and 4:

Corollary 1. *In both equivalence checking and unitarity checking, for a fixed α_2, if we choose s according to the lower bound in Proposition 1, then $s = \Theta\left(\frac{\log k}{\epsilon^2}\right)$. Further more, when adopting this choice, the time complexities of Algorithm 2 and 4 are both $O\left(\frac{nk \log k}{\epsilon^2}\right)$.*

In Section 6, we will also provide experimental research on the parameters k, s, and ϵ.

5. Optimization for equivalence and unitarity checking

5.1. Optimization ideas

The original algorithms for equivalence checking and unitarity checking (Algorithms 2 and 4) involve repeated running the Swap Test many times to obtain the indicating parameters of equivalence and unitarity. In contrast, identity checking (Algorithm 3) immediately returns FAIL upon encountering a non-zero result. Thus, it enhances the efficiency of discovering bugs. This observation prompts us to seek similar properties for equivalence and unitarity checking that allow us to confirm the existence of errors immediately upon a certain result rather than waiting for all results.

As discussed in Section 2.4, formula (1) about Swap Test gives $tr(\rho_1\rho_2) = 1 - 2p_1$. So $tr(\rho_1\rho_2) = 1$ if and only if p_1, the probability of obtaining result '1', equals 0. In other words, as long as a result '1' occurs, we can immediately conclude that $tr(\rho_1\rho_2) \neq 1$. Thus, we introduce Algorithm 5, to fast determine whether $tr(\rho_1\rho_2) = 1$. The main body of Algorithm 5 is the same as Algorithm 1 about Swap Test. The difference is that Algorithm 5 returns FALSE immediately when result '1' occurs (lines 10 ~ 12), while Algorithm 1 accumulates the number of results '1'.

Note that both equivalence and unitarity checking are based on the Swap Test. To optimize them using Algorithm 5, it is necessary to find their properties related to whether $tr(\rho_1\rho_2)$ equals 1. Obviously, there are two related properties:

(1) The *purity checking* for a state:
By setting $\rho_1 = \rho_2 = \rho$, Algorithm 5 is able to determine whether $tr(\rho^2) = 1$, thereby assessing whether ρ represents a pure state.
(2) The *equality checking* for two pure states:
Let $\rho_1 = |\alpha\rangle\langle\alpha|$ and $\rho_2 = |\beta\rangle\langle\beta|$ are two pure states, then $tr(\rho_1\rho_2) = |\langle\alpha|\beta\rangle|^2$, and it equals 1 if and only if $|\alpha\rangle = |\beta\rangle$.

In the following part of this section, we will identify and examine the properties of equivalence and unitarity checking and delve into the strategies for optimizing these properties using Algorithm 5.

Algorithm 5: `is_TrAB_equals_1`

Input: $(n, s, \mathcal{P}_1, \mathcal{P}_2)$
n: the number of qubits of one target state;
s: the number of rounds;
$\mathcal{P}_1, \mathcal{P}_2$: subroutines to generate target states ρ_1 and ρ_2.
Output: TRUE or FALSE about whether $tr(\rho_1\rho_2) = 1$.

```
1  for i = 1 to s do
2      qanc ← |0⟩; qs1 ← |0⟩^⊗n; qs2 ← |0⟩^⊗n;
3      H(qanc);
4      P1(qs1); P2(qs2);
5      for j = 0 to n − 1 do
6          Controlled SWAP(qanc, (qs1[j], qs2[j]));
7      end
8      H(qanc);
9      m ← Measure(qanc);
10     if m = 1 then
11         return FALSE;
12     end
13 end
14 return TRUE;
```

5.2. Optimized equivalence checking

As discussed in Section 4.2, equivalence checking relies on the evaluation of the following parameter

$$E(\rho_1, \rho_2) = \left| \frac{tr(\rho_1^2) + tr(\rho_2^2)}{2} - tr(\rho_1\rho_2) \right|$$

and determines whether it equals 0. The values of $tr(\rho_1^2)$, $tr(\rho_2^2)$, and $tr(\rho_1\rho_2)$ can be estimated using the Swap Test. When both ρ_1 and ρ_2 are pure states (i.e., $tr(\rho_1^2) = tr(\rho_2^2) = 1$), we have $E(\rho_1, \rho_2) = 0$ if and only if $tr(\rho_1\rho_2) = 1$. As discussed in Section 5.1, purity checking for a single state and determining whether $tr(\rho_1\rho_2) = 1$ for two pure states can be accomplished fast using Algorithm 5. The idea of the optimization approach involves first checking whether the input states are both pure states. If so, Algorithm 5 is directly employed to assess whether $tr(\rho_1\rho_2) = 1$, bypassing the need for the general Swap Test. Consider that numerous practical quantum programs execute unitary transform, thereby preserving the purity of output states when given pure input states. Consequently, this optimization strategy may be effective for such programs.

The optimized algorithm is presented in Algorithm 6. Lines 1 ~ 2 and 14 ~ 23 are the same as Algorithm 2, while lines 3 ~ 13 are newly introduced for optimization. Similar to the discussion in Section 4.2, for $i = 1, 2$, $\mathcal{P}_i \circ G_K$ generates the state of the output of $\mathcal{P}_i$ under the

input generated by G_K. So lines $3 \sim 4$ invoke Algorithm 5 to verify the purity of the output states from the two target programs, respectively. The additional input parameter t represents the number of rounds in Algorithm 5, while the number of rounds in the Swap Test is denoted as s. If their purities differ (one state is pure, but the other is not, line 5), it can be concluded that the output states are distinct, and consequently, the target programs are not equivalent (line 6). If both states are pure, Algorithm 5 is employed to assess their equivalence (lines $9 \sim 12$), providing a fast determination. Otherwise, the original equivalence checking algorithm must be executed (lines $14 \sim 23$).

Algorithm 6: `EquivalenceChecking_optimized`

Input: $(n, k, t, s, \epsilon, \mathcal{P}_1, \mathcal{P}_2)$
n: the number of qubits of the target program;
k: the number of test points k;
t: the number of rounds of trace-1-checking subroutine;
s: the number of rounds of Swap Test;
ϵ: tolerance error;
$\mathcal{P}_1, \mathcal{P}_2$: two target programs.
Output: PASS or FAIL.

```
1  for i = 1 to k do
2      K ← Randomly choose a n-element integer array, where each
       integer is in 0, ..., 5;
3      isPure1 ← is_TrAB_equals_1(n, t, P1∘G_K, P1∘G_K);
4      isPure2 ← is_TrAB_equals_1(n, t, P2∘G_K, P2∘G_K);
5      if isPure1 ≠ isPure2 then
6          return FAIL;
7      end
8      if isPure1 = TRUE then
9          TrEq ← is_TrAB_equals_1(n, t, P1∘G_K, P2∘G_K);
10         if TrEq = FALSE then
11             return FAIL;
12         end
13     else
14         s1 ← SwapTest(n, s, P1∘G_K, P1∘G_K);
15         s2 ← SwapTest(n, s, P2∘G_K, P2∘G_K);
16         s12 ← SwapTest(n, s, P1∘G_K, P2∘G_K);
17         E ← (2s12 − s1 − s2)/s;
18         if |E| > ϵ then
19             return FAIL;
20         end
21     end
22 end
23 return PASS;
```

5.3. Optimized unitarity checking

The original unitarity checking algorithm is based on checking the orthogonal preservation condition, requiring the evaluation of $tr(\rho_1\rho_2) = 0$ for two states ρ_1 and ρ_2. However, $tr(\rho_1\rho_2) = 0$ if and only if the probability p_1 of obtaining the '1' result equals $1/2$. It implies that we need to check if the '1' result occurs approximately half of the time, unable to return immediately.

The idea of optimization is to focus on another crucial property of unitary transforms: *purity preservation*. A unitary transform must map a pure state into another pure state. If a program maps the input pure state into a mixed state, it can be immediately concluded that the program is not a unitary transform. By employing Algorithm 5, we can perform purity checking. However, purity preservation alone is insufficient to guarantee unitarity. For instance, the `Reset` procedure maps any input state to the *all-zero state* $|0 \dots 0\rangle$. While this transformation maps a pure state to another pure state (since the all-zero state is pure), it is not a unitary transform. Thus, the original checking approach that relies on *orthogonality preservation* remains essential.

The optimized algorithm, presented in Algorithm 7, enhances the original approach. It includes a purity checking step for the output state of the target program $\mathcal{P}$ when subjected to Pauli inputs (generated by G_K with random K) using Algorithm 5 (lines $2 \sim 3$). If the purity checking returns FALSE (line 4), the algorithm immediately fails (line 5). Similar to the equivalence checking, we introduce a new parameter t to represent the number of rounds in Algorithm 5, and s denotes the number of rounds in the Swap Test. Upon successful purity checking, the original unitarity checking should be executed (lines $8 \sim 9$).

Algorithm 7: `UnitarityChecking_optimized`

Input: $(n, k, t, s, \epsilon, \mathcal{P})$
n: the number of qubits of the target program;
k: the number of test points;
t: the number of rounds of trace-1-checking subroutine;
s: the number of rounds of Swap Test;
ϵ: tolerance error;
$\mathcal{P}$: the target program.
Output: PASS or FAIL.

```
1  for i = 1 to k do
2      K ← Randomly choose a n-element integer array, where each
       integer is in 0, ..., 5;
3      isTr1 ←is_TrAB_equals_1(n, t, P∘G_K, P∘G_K);
4      if isTr1 = FALSE then
5          return FAIL;
6      end
7  end
8  r ← UnitarityChecking_original(n, k, s, ϵ, P);
9  return r
```

5.4. Selection of t

In Algorithms 6 and 7, the parameters k, s, and ϵ are selected using the same strategy as the original algorithms, which has been discussed in Section 4.5. However, the optimized algorithms introduce an additional parameter t, which should have a balanced choice. If t is too small, there is a risk of misjudging a mixed state as a pure state since both results, '0' and '1', are possible for mixed states. This can lead to subsequent misjudgments. On the other hand, if t is too large, it may reduce the efficiency. In Section 6, we will conduct experimental research on the parameter t to determine an appropriate value.

6. Experimental evaluation

In this section, we present the experimental evaluation of our algorithms about equivalence (EQ) checking, identity (ID) checking, and unitarity (UN) checking.

6.1. Experimental design

Each of our checking algorithms yields a PASS or FAIL result for the given input programs or pairs. For a comprehensive evaluation, it is essential to prepare two categories of targets: *expected-pass* and *expected-fail*. These are used to assess the algorithms' performance on programs that either meet or do not meet the specified relations. An effective algorithm should return PASS for *expected-pass* targets and FAIL for *expected-fail* targets. Therefore, our benchmark programs include both types of targets for each checking task. Existing benchmarks, such as those in Zhao et al. (2021), are designed for general testing and not for the specific tasks discussed in our paper, making them unsuitable for evaluating our relation-focused testing task. To remedy this, we are creating a specialized program benchmark, the details of which will be outlined in Section 6.2.

As outlined in Sections 4 and 5, our checking algorithms are influenced by several adjustable parameters. A key aspect of our evaluation involves investigating how these parameters affect the algorithms' accuracy. Moreover, we aim to compare the efficacy of the original and optimized versions of our EQ and UN checking algorithms. Additionally, a comprehensive assessment of these algorithms' overall performance, including an exploration of cases where they underperform, is a crucial part of our evaluation. To guide this process, we will focus on the following research questions (RQs):

- **RQ1:** How do the parameters k and ϵ affect the bug detection capability of the original algorithms?
- **RQ2:** Is the selection of the s value, as presented in Proposition 1, effective for practical testing tasks using the original algorithms?
- **RQ3:** What is the performance of the optimized algorithms in terms of efficiency and bug detection accuracy?
- **RQ4:** How well do our checking methods perform on the benchmark programs?
- **RQ5:** What are the characteristics of the cases on which our algorithms do not perform so well?

Our experiments do not include comparisons with existing methods due to two key reasons: Firstly, our approach to EQ and ID checking differs from current methods, which primarily concentrate on quantum information theory (Janzing et al., 2005; Flammia and Liu, 2011; Montanaro and de Wolf, 2013) or white-box checking (Viamontes et al., 2007; Yamashita and Markov, 2010; Burgholzer and Wille, 2020; Hong et al., 2021, 2022; Wang et al., 2022b), as opposed to our focus on black-box testing. Secondly, UN checking represents a novel problem area for which no existing methods have been established.

To facilitate our experiments, we have developed a dedicated tool for implementing equivalence, identity, and unitarity checking of Q# programs. This tool enables us to assess the performance and effectiveness of our algorithms effectively. We conducted our experiments using Q# language (SDK version 0.26.233415) and its simulator.

The experiments were conducted on a personal computer equipped with an Intel Core i7-1280P CPU and 16 GB RAM. To ensure accuracy and minimize the measurement errors resulting from CPU frequency reduction due to cooling limitations, we executed the testing tasks sequentially, one after another.

6.2. Benchmark programs

6.2.1. Original programs

We select several typical quantum programs as original programs to construct the benchmark, as Table 2 shows. The "#Qubits" column indicates the number of used qubits, while the "#QOps" column provides an approximate count of the single- or two-qubit gates for each program,[5] which indicates the program's scale. The "UN" column indicates whether the program implements a unitary transform. Similarly, the "ID" column indicates whether the program is identity.

We have considered the properties of our tasks about EQ, ID, and UN checking in the selection of original programs. First, we select two sets of equivalent circuit pairs as shown in Fig. 3: one set is unitary, while the other is not. `Cir1A` and `Cir1B` (Fig. 3(a)) are from Pointing et al. (2021), and `Cir2A` and `Cir2B` (Fig. 3(b)) are from Garcia-Escartin and Chamorro-Posada (2011). Then, we select two typical identity programs: `Empty` and `TeleportABA`. Considering that `P∘invP` is a common identity relation, we select two unitary programs `QFT`, `QPE` and their inverse programs `invQFT`, `invQPE`. We also select a complicated program `CRot` and a common non-unitary program `Reset`.

6.2.2. Construct the benchmark

We begin by constructing the benchmark based on the original programs, as discussed in Section 6.1. Our benchmark must include both expected-pass and expected-fail targets for EQ checking. Expected-pass pairs consist of an original program and a carefully selected equivalent program, while expected-fail pairs consist of an original program and a mutated program obtained by applying a mutant operator into the original program. In our evaluation, we consider two types of mutant operators: gate mutant (GM) operators, which involve the addition, removal, or modification of gates, and measurement mutant (MM) operators, which encompass the addition or removal of measurement operations (Fortunato et al., 2022a; Long and Zhao, 2023). The concrete benchmark is outlined in Table 3. Programs labeled with the prefix "*error*" are mutant versions of the original programs generated using either GM or MM. "5×(error `Cir1A`, `Cir1B`)" signifies that we have prepared five pairs, where each pair consists of one mutant `Cir1A` and the original `Cir1B`, and different pair features a different mutant operator.

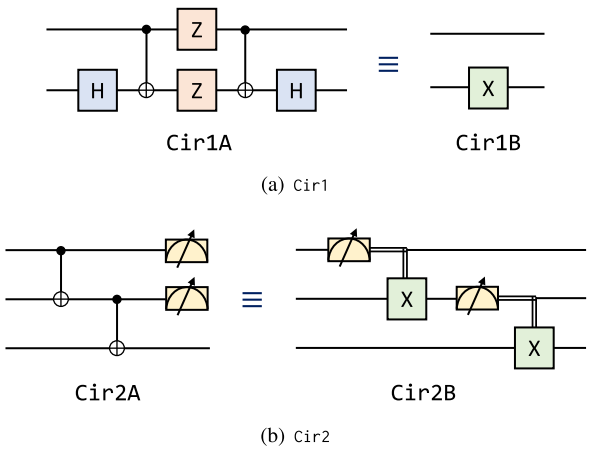


Fig. 3. Two sets of equivalent circuits.

For ID checking evaluation, the expected-pass programs are constructed based on known identity relations, such as `P∘invP`, to assess the effectiveness of testing inverse programs (see Scenario 2 in Section 3). Notably, both No. 4 and No. 11 pertain to the relation `QFT∘invQFT`. No. 4 utilizes EQ checking with `Empty`, while No. 11 employs ID checking. These cases are used to compare ID checking and EQ checking with identity. The expected-fail programs are generated by applying GM or MM mutant operator to each expected-pass program.

For UN checking evaluation, the expected-pass programs comprise those listed in Table 2 that do not contain any measurement operations. Since GM mutations do not affect unitarity while MM mutations do, some expected-pass programs are constructed by applying GM mutant operators, while all expected-fail programs are created by applying MM mutant operators.

As illustrated in Table 3, our benchmark comprises 63 Q# programs or program pairs used to evaluate the effectiveness of our methods.

6.3. Experiment configurations

We describe our experimental configurations for each research question as follows.[6]

- **RQ1:** We ran our algorithms on each expected-fail program, adjusting the values of k and ϵ. The selection of these ranges and values was determined by proper preliminary experiments. We examine ϵ values of 0.05, 0.10, 0.15, and 0.20 for EQ and UN. For EQ, we test k values of 1, 2, 3, 4, 6, and 10. Since the UN algorithm requires two types of inputs, the minimum k value we consider is 2. Thus, we explore k values of 2, 3, 4, 6, and 10 for UN. The corresponding s values are determined as shown in Table 1. For ID, the algorithm does not involve the Swap Test, allowing for larger k values. Therefore, we select k

[5] Q# programs are allowed to include loops. The count of gates is determined by (1) fully expanding all loops and (2) decomposing all multi-qubit gates or subroutines into basic single- or two-qubit gates. Due to the possibility of different decomposition methods, #QOps just represents an approximate count for each program.

[6] The source code of our algorithms and evaluation are available at https://github.com/MgcosA/EvaluationCodeOfQuantumRelationChecking

Table 2
Original programs to construct the benchmark.

Program	Description	#Qubits	#QOps	UN	ID
`Cir1A`	The implementation the circuit in Fig. 3(a) left.	2	6	✓	
`Cir1B`	The implementation of the circuit in Fig. 3(a) right. It is equivalent to `Cir1A`.	2	1	✓	
`Cir2A`	The implementation of the circuit in Fig. 3(b) left.	3	4		
`Cir2B`	The implementation the circuit in Fig. 3(b) right. It is equivalent to `Cir2A`.	3	4		
`Empty`	A program with no statement; thus, it is identity.	6	0	✓	✓
`TeleportABA`	A program using quantum teleportation to teleport quantum state from input variable `A` to intermediate variable `B`, and then teleport `B` back to `A`. So it is the identity on input variable `A`.	3	60	✓	✓
`QFT`	An implementation of the Quantum Fourier Transform (QFT) algorithm.	5	17	✓	
`invQFT`	The inverse program of `QFT`.	5	17	✓	
`QPE`	An implementation of the Quantum Phase Estimation algorithm. It has an input parameter of estimated unitary transform U. In our benchmark, we fix this parameter as $U = SH$.	5	22	✓	
`invQPE`	The inverse program of `QPE`.	5	22	✓	
`CRot`	The key subroutine in *Harrow-Hassidim-Lloyd* (HHL) algorithm (Harrow et al., 2009). It implements the unitary transform: $\|\lambda\rangle\|0\rangle \to \|\lambda\rangle(\sqrt{1-\frac{c^2}{\lambda^2}}\|0\rangle + \frac{c}{\lambda}\|1\rangle)$, where c represents a parameter. In our benchmark, we fix $c = 1$.	5	680	✓	
`Reset`	A subroutine that resets a qubit register to the all-zero state $\|0\ldots0\rangle$. Note that it is not a unitarity transform.	6	12		

Table 3
Benchmark programs.

Task	Expected	No.	Programs/Pairs	#
EQ	PASS	1	(`Cir1A`, `Cir1B`)	4
		2	(`Cir2A`, `Cir2B`)	
		3	(`QFT`, `QFT`)	
		4	(`QFT`∘`invQFT`, `Empty`)	
	FAIL	5	5×(error `Cir1A`, `Cir1B`)	20
		6	2×(error `Cir2A`, `Cir2B`)	
		7	3×(`Cir2A`, error `Cir2B`)	
		8	5×(error `QFT`, `QFT`)	
		9	5×(`QFT`∘error `invQFT`, `Empty`)	
ID	PASS	10	`Empty`	4
		11	`QFT`∘`invQFT`	
		12	`QPE`∘`invQPE`	
		13	`TeleportABA`	
	FAIL	14	2× error `Empty`	17
		15	5× `QFT`∘error `invQFT`	
		16	5× `QPE`∘error `invQPE`	
		17	5× error `TeleportABA`	
UN	PASS	18	`Cir1A`	8
		19	`Cir1B`	
		20	`Empty`	
		21	`QFT`	
		22	`CRot`	
		23	3× GMs of `QFT`	
	FAIL	24	`Cir2A`	10
		25	`Cir2B`	
		26	`Reset`	
		27	5× MMs of `QFT`	
		28	2× MMs of `CRot`	

values of 1, 2, 3, 4, 6, 10, 15, 20, 30, and 50. We repeat every testing process for each program 100 times and record the number of outputs returning PASS. A smaller number indicates higher effectiveness in detecting bugs. Throughout the experiment, we aim to identify a set of k and ϵ values that balance between accuracy and running time.

- **RQ2:** We fix k and ϵ in the EQ and UN algorithms and select different values of s to run the algorithms for each expected-pass program. We calculate the value of s_0 using Proposition 1, and then choose $s = 0.05s_0$, $0.1s_0$, $0.2s_0$, $0.3s_0$, $0.4s_0$, $0.5s_0$, $0.7s_0$, $0.9s_0$, and $1.0s_0$, respectively. We repeat the testing process for each program with different choices of s 100 times respectively, and record the number of outputs that return PASS. A higher number indicates fewer misjudgments by the algorithm.
- **RQ3:** We conducted experiments to evaluate the performance of the optimized algorithms — Algorithm 6 and Algorithm 7. We keep the parameters k, ϵ, and s fixed and run Algorithms 6 and 7 on both the expected-pass and expected-fail programs. The values of k and ϵ are selected based on the balanced configuration determined in the experiments for RQ1, while the value of s is determined using Proposition 1. We explored the impact of different values of t on the algorithm's performance. We selected t values as $t = 1$, 2, 3, 4, 6, 10, 15, 20, 30, and 50, respectively. For each benchmark program and different choice of t, we repeated the testing process 100 times, recording the number of outputs that returned a "PASS" result. Additionally, we kept track of the number of times the optimization rules were triggered during the testing. The trigger condition was defined as Algorithm 6 returning at line 6 or 11, and Algorithm 7 returning at line 5. By comparing the number of "PASS" results obtained from the optimized algorithms with those from the original algorithms and analyzing the number of triggers, we can gain insights into the performance of the optimized algorithms. This evaluation allows us to understand how effectively the optimization rules improve the algorithms' performance.
- **RQ4:** We executed both the original and optimized algorithms for each benchmark program listed in Table 3. To ensure a balanced selection of parameters k, ϵ, and t, we leveraged the findings from the experiments conducted in RQ1 and RQ3. Additionally, we calculated the value of s using Proposition 1. Each program was tested 100 times, and we recorded the number of outputs that returned PASS. If an expected-pass program yielded a number close to 100 and an expected-fail program yielded a number close to 0, we can conclude that the algorithm is effective for them. Moreover, we measured the running time of both the original algorithms (T_0) and the optimized algorithms (T_{opt}) for each program. The ratio T_{opt}/T_0 serves as a measure of the efficiency of the optimization rules, where a smaller ratio indicates higher efficiency.
- **RQ5:** We identify specific cases from RQ1 to RQ4 where our algorithms underperform. Our subsequent analysis aims to extract insights regarding the bug patterns within these particular instances.

6.4. Result analysis

6.4.1. RQ1: About parameter k and ϵ

The testing results for benchmark programs with different parameters k and ϵ for EQ and UN are presented in Tables 4 and 6, while the

Table 4
The number of PASS with different k and ϵ for expected-fail programs in EQ.

Task	Test No. \ k	$\epsilon = 0.05$						$\epsilon = 0.10$						$\epsilon = 0.15$						$\epsilon = 0.20$					
		1	2	3	4	6	10	1	2	3	4	6	10	1	2	3	4	6	10	1	2	3	4	6	10
EQ	5.1	35	10	4	2	0	0	31	10	4	1	0	0	30	9	5	2	0	0	44	18	3	2	0	0
	5.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	5.3	16	4	0	0	0	0	12	7	1	1	0	0	18	7	1	1	0	0	28	5	1	0	0	0
	5.4	23	5	2	2	0	0	19	6	0	0	0	0	17	5	0	0	0	0	23	2	1	0	0	0
	5.5	33	12	6	0	0	0	21	14	4	2	0	0	38	15	3	1	0	0	38	9	3	1	1	0
	6.1	10	1	1	0	0	0	9	4	0	0	0	0	17	1	0	0	0	0	17	4	0	0	0	0
	6.2	58	30	25	11	4	2	53	30	17	14	4	1	81	65	46	31	16	12	78	66	54	58	36	21
	7.1	53	40	12	7	3	0	63	34	24	12	5	0	75	58	48	35	21	10	88	75	59	66	46	22
	7.2	33	11	6	0	0	0	40	12	9	1	0	0	55	34	12	6	0	1	62	47	24	23	7	1
	7.3	25	2	0	0	0	0	25	5	0	0	0	0	20	4	1	0	0	0	20	8	2	1	0	0
	8.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	8.2	21	14	4	0	0	0	26	15	6	0	0	0	30	12	2	1	0	0	44	12	7	2	0	0
	8.3	0	0	0	0	0	0	1	0	0	0	0	0	8	1	0	0	0	0	31	10	2	0	0	0
	8.4	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	8.5	2	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	3	0	0	0	0	0
	9.1	4	0	0	0	0	0	1	0	0	0	0	0	2	0	0	0	0	0	2	0	0	0	0	0
	9.2	2	0	0	0	0	0	4	0	0	0	0	0	3	0	0	0	0	0	3	0	0	0	0	0
	9.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	9.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	9.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0

Table 5
The number of PASS with different k for expected-fail programs in ID.

Task	Test No. \ k	1	2	3	4	6	10	15	20	30	50
ID	14.1	67	46	33	22	5	0	0	0	0	0
	14.2	96	87	76	64	55	28	18	17	6	0
	15.1	26	3	0	0	0	0	0	0	0	0
	15.2	27	5	1	0	0	0	0	0	0	0
	15.3	1	0	0	0	0	0	0	0	0	0
	15.4	3	0	0	0	0	0	0	0	0	0
	15.5	28	5	1	0	0	0	0	0	0	0
	16.1	13	1	1	0	0	0	0	0	0	0
	16.2	22	7	1	0	0	0	0	0	0	0
	16.3	44	11	4	3	0	0	0	0	0	0
	16.4	49	21	8	5	0	0	0	0	0	0
	16.5	4	0	0	0	0	0	0	0	0	0
	17.1	2	0	0	0	0	0	0	0	0	0
	17.2	17	1	0	0	0	0	0	0	0	0
	17.3	18	3	1	0	0	0	0	0	0	0
	17.4	10	3	0	0	0	0	0	0	0	0
	17.5	14	0	0	0	0	0	0	0	0	0

results with different k for ID are shown in Table 5. In Table 4, "Test No. 5.1" means the 1st program/pair of No. 5 in Table 3. Similarly, the same notation is used in Tables 5 and 6. Most of the results regarding the number of PASS are 0, indicating that our algorithms effectively detect failures in the expected-fail programs/pairs corresponding to the selected parameters. Notably, even a single gate or measurement mutation can lead to a significant deviation from the original program. While some programs or pairs have non-zero results, their trend suggests that the number of PASS decreases as k increases and ϵ decreases. This indicates that these programs or pairs are closer to the correct version, and the selected parameters are too lenient to identify their bugs.

According to Corollary 1, the time complexity of EQ and UN is more sensitive to ϵ (which depends on $\frac{1}{\epsilon^2}$) than to k (which depends on $k \log k$). Therefore, it is advisable to avoid choosing very small values for ϵ, but k can be slightly larger. Interestingly, for some programs such as No. 6.2 and No. 27.3, even adopting excessively small ϵ, the number of unrevealed bugs cannot be reduced to a low level. Based on the findings in Tables 4, 5 and 6, we proceed with $k = 4$ and $\epsilon = 0.15$ for EQ and UN, and $k = 50$ for ID in the subsequent experiments.

6.4.2. RQ2: About parameter s

The testing results for benchmark programs with different parameters s for EQ and UN are presented in Fig. 4. It is evident that selecting $s = s_0$ leads to nearly all expected-pass programs returning PASS. This outcome demonstrates the effectiveness of the chosen parameter s based on Proposition 1 in controlling type II errors. Furthermore, it is notable that the bad results (i.e., the rate of PASS is less than $1 - \alpha_2 = 0.9$) is primarily observed in certain programs when $s < 0.5s_0$. Additionally, for Test No. 1, 3, and 4, the number of PASS remains at 100 even when $s = 0.05s_0$. This observation indicates that different programs exhibit varying degrees of sensitivity to the parameter s, and the selection of s in Proposition 1 is conservative to ensure a high PASS rate in the worst-case scenario.

6.4.3. RQ3: The performance of optimized algorithms

The testing results with different t for the optimized algorithms are presented in Table 7. It can be observed that most of the expected-fail programs are killed by the optimization rules, indicating the effectiveness of our optimization rules in accelerating bug detection. In addition,

Table 6
The number of PASS with different k and ϵ for expected-fail programs in UN.

Task	Test No. \ k	$\epsilon = 0.05$						$\epsilon = 0.10$						$\epsilon = 0.15$						$\epsilon = 0.20$					
		1	2	3	4	6	10	1	2	3	4	6	10	1	2	3	4	6	10	1	2	3	4	6	10
UN	24	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0
	25	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0
	26	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0
	27.1	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0
	27.2	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0
	27.3	–	77	61	61	43	26	–	78	66	54	50	28	–	81	64	67	48	30	–	87	83	81	74	53
	27.4	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0
	27.5	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0	–	0	0	0	0	0
	28.1	–	6	2	1	0	0	–	6	2	1	0	0	–	5	1	0	0	0	–	8	3	2	1	0
	28.2	–	4	0	0	1	0	–	7	0	0	0	0	–	11	1	0	0	0	–	5	4	0	0	0

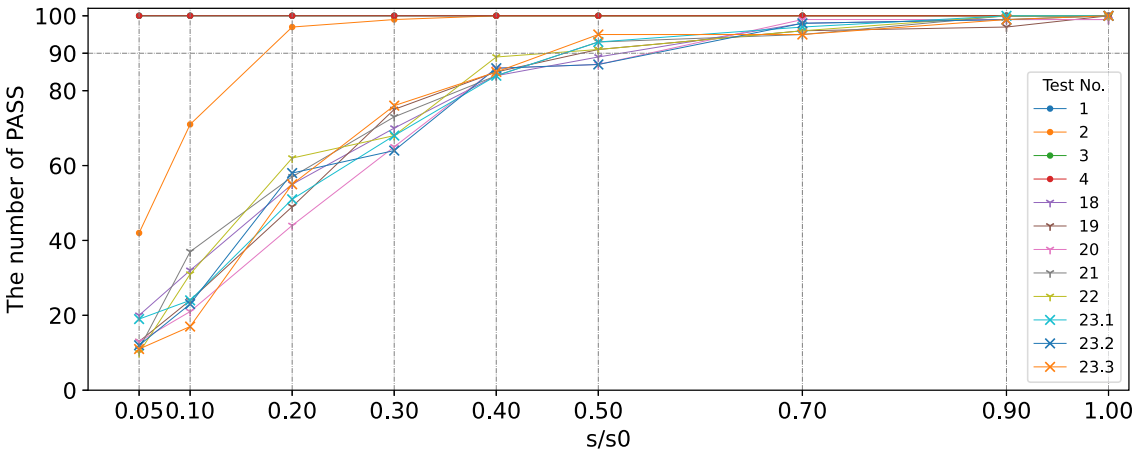


Fig. 4. The number of PASS with different s for expected-pass programs in EQ and UN under $k = 4, \epsilon = 0.15$.

most expected-pass programs are not triggered by the optimization rules. It is because our optimization rules only relate to returning FAIL immediately.

Several interesting cases, namely No. 2, No. 6.2, No. 7.1, No. 26, and No. 27.3, deserve attention. For No. 2, it is an expected-pass case but triggered by optimization rules, which means a misjudgment. As t increases, the number of PASS instances increases while the number of triggers decreases. It is noteworthy that `Cir2A` and `Cir2B` contain measurements (Fig. 3(b)), resulting in expected mixed-state outputs. Therefore, a small value of t may lead to the misjudgment of pure and mixed states, increasing the likelihood of misjudgment of results in this particular test scenario. In the case of No. 6.2 and No. 7.1, it appears that a small t effectively excludes incorrect results. However, it is important to observe that as t increases, the number of triggers decreases. This implies that returning FAIL based on optimization rules is also a misjudgment (it is similar to No. 2). No. 26 (program `Reset`) is an intriguing case as it represents an expected-fail scenario that cannot be triggered by the optimization rule of UN. This is a typical example that shows that purity preservation alone is insufficient for unitarity checking. Lastly, for No. 27.3, a significant observation is that the number of PASS instances in the optimized algorithm is considerably smaller than in the original algorithm (column 'Base'). It should be noted that the original algorithm is based on orthogonal preservation, while the optimization rule is based on purity preservation. This suggests that purity preservation checking is more effective than orthogonal preservation checking in this particular task.

Based on the results presented in Table 7, we find that $t = 20$ provides a balanced choice, which we will further analyze in the subsequent overall evaluation.

6.4.4. RQ4: Overall performance for benchmark programs

The running results are shown in Table 8, including both the original and optimized algorithms. We can see that our methods work well for most programs, i.e., the percentage of PASS is near 100 for expected-pass programs and near 0 for expected-fail programs. It means that our methods are effective for most benchmark programs with the parameters setting in the "Running Parameters" column of Table 8. The "Average Run Time" column gives the average running time in a single test.

The average running time of expected-fail programs is less than that of corresponding expected-pass programs. That is because the FAIL result is returned as long as one test point fails. Conversely, the PASS result is returned only if all test points pass. ID is faster than EQ and UN because ID does not have the repeat of the Swap Test in EQ and UN. For example, consider programs No. 4 and No. 11, which both check the relation $\mathtt{QFT} \circ \mathtt{invQFT} = I$. No. 4 uses EQ and another identity program, while No. 11 uses ID directly to check it. No. 11 requires only about one-thousandth (for the original EQ algorithm) or one-eighth (for optimized EQ algorithm) running time of No. 4. So identity checking can be implemented efficiently. It also tells us that finding more quantum metamorphic relations which can be reduced to identity checking is valuable.

Take note of the column labeled "T_{opt}/T_o", which indicates the efficiency of the optimization rules. We observe that for the equivalence (EQ) checking, the optimized algorithm demonstrates varying degrees of acceleration with the same parameter configuration. The acceleration ranges from dozens of percent (e.g., No. 2, No. 6.2, No. 7.1, No. 7.2, No. 7.3) to several hundred multiples (remaining EQ cases). These results indicate the effectiveness of the optimization rules for EQ. In the case of unitarity (UN) checking, the optimized algorithm is slightly slower than the original algorithm for all expected-pass cases and No. 26. This is due to the additional purity preservation checking in the optimized algorithm, and they cannot be excluded by purity preservation checking. Consequently, the orthogonal checking cannot be bypassed. Nevertheless, the optimized algorithm significantly

Table 7
The number of PASS and triggers by optimization rules with different t for programs in EQ and UN under $k = 4$, $\epsilon = 0.15$ and s selected by Proposition 1.

Task	Expected	Test No. \ t	Number of PASS											Number of triggers by optimization rules									
			Base	1	2	3	4	6	10	15	20	30	50	1	2	3	4	6	10	15	20	30	50
EQ	PASS	1	100	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		2	100	4	3	8	19	38	84	94	99	100	100	96	97	92	81	62	16	6	1	0	0
		3	100	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		4	100	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
	FAIL	5.1	2	23	4	3	1	2	0	2	3	1	1	77	96	97	99	98	100	98	97	99	99
		5.2	0	17	7	1	0	0	0	0	0	0	0	83	93	99	100	100	100	100	100	100	100
		5.3	1	23	7	1	0	0	0	0	1	0	0	77	93	99	100	100	100	100	99	100	100
		5.4	0	24	12	4	2	0	0	1	0	0	0	76	88	96	98	100	100	99	100	100	100
		5.5	1	30	12	4	2	3	1	0	1	4	2	70	88	96	98	97	99	100	99	96	98
		6.1	0	5	0	0	0	0	0	0	0	0	0	95	100	100	100	100	100	100	100	100	100
		6.2	31	3	6	3	11	14	33	42	42	42	33	93	91	83	67	48	18	3	0	0	0
		7.1	35	4	7	5	7	25	30	38	39	43	34	92	84	77	70	42	19	4	0	0	0
		7.2	6	2	2	0	2	6	7	18	8	10	12	91	87	70	66	41	14	5	0	0	0
		7.3	0	4	0	0	0	1	0	0	0	0	0	81	77	69	46	44	19	19	16	18	13
		8.1	0	2	0	0	0	0	0	0	0	0	0	98	100	100	100	100	100	100	100	100	100
		8.2	1	20	14	7	3	1	0	2	2	2	1	80	86	93	97	99	100	98	98	98	99
		8.3	0	14	0	0	0	0	0	0	0	0	0	86	99	100	100	100	100	100	100	100	100
		8.4	0	2	0	0	0	0	0	0	0	0	0	98	100	100	100	100	100	100	100	100	100
		8.5	0	0	0	0	0	0	0	0	0	0	0	100	100	100	100	100	100	100	100	100	100
		9.1	0	12	1	0	0	0	0	0	0	0	0	88	99	100	100	100	100	100	100	100	100
		9.2	0	16	2	0	0	0	0	0	0	0	0	84	98	100	100	100	100	100	100	100	100
		9.3	0	8	0	0	0	0	0	0	0	0	0	92	100	100	100	100	100	100	100	100	100
		9.4	0	6	0	0	0	0	0	0	0	0	0	94	100	100	100	100	100	100	100	100	100
		9.5	0	3	0	0	0	0	0	0	0	0	0	97	100	100	100	100	100	100	100	100	100
UN	PASS	18	99	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		19	100	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		20	100	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		21	99	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		22	99	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		23.1	99	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		23.2	100	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
		23.3	99	100	100	100	100	100	100	100	100	100	100	0	0	0	0	0	0	0	0	0	0
	FAIL	24	0	0	0	0	0	0	0	0	0	0	0	67	93	97	98	100	100	100	100	100	100
		25	0	0	0	0	0	0	0	0	0	0	0	69	87	96	99	100	100	100	100	100	100
		26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		27.1	0	0	0	0	0	0	0	0	0	0	0	87	97	100	100	100	100	100	100	100	100
		27.2	0	0	0	0	0	0	0	0	0	0	0	61	76	86	83	99	97	98	98	98	100
		27.3	67	22	11	4	2	1	0	0	0	0	0	60	84	96	96	99	100	100	100	100	100
		27.4	0	0	0	0	0	0	0	0	0	0	0	91	96	100	100	100	100	100	100	100	100
		27.5	0	0	0	0	0	0	0	0	0	0	0	86	100	100	100	100	100	100	100	100	100
		28.1	0	1	0	0	0	0	0	0	0	0	0	56	79	95	98	100	100	100	100	100	100
		28.2	0	0	0	0	0	0	0	0	0	0	0	67	87	93	96	100	100	100	100	100	100

accelerates several hundred multiples for other UN cases. Notably, the optimization may also enhance the accuracy of bug detection in some cases, such as No. 27.3.

6.4.5. RQ5: about the cases not performed so well

Table 8 also presents the performance of error mutation programs. However, it is observed that some error mutation programs, such as No. 6.2, No. 7.1, and No. 27.3, demonstrate less effectiveness in terms of the number of PASS outcomes. It may be because they are close to the correct one. Here, we further research these cases and try to find some insights about bug patterns.

Table 3 and Table 8 indicate that the troublesome cases predominantly arise from the benchmark programs `Cir2A`, `Cir2B`, and `QFT`. Notably, these programs also include instances where our algorithms perform well, such as in cases No. 6.1, No. 7.2, No. 7.3, No. 27.1, No. 27.2, No. 27.4, and No. 27.5. Therefore, contrasting the successful cases with the troublesome ones within the same benchmark programs could be insightful. Fig. 5 presents the Q# implementation of `Cir2A`, `Cir2B`, and `QFT`, along with their mutant operators used in our evaluation.

Fig. 5(a) illustrates the mutant operators for cases No. 6.1 $\sim$ 6.2 and No. 7.1 $\sim$ 7.3. This comparison reveals varying degrees of impact on equivalence by different mutant operators. For instance, altering an operation (as in No. 6.1, No. 7.2, No. 7.3) appears to have a more significant effect than swapping two operations (as in No. 6.2, No. 7.1). Additionally, it is noteworthy that the troublesome cases for EQ checking predominantly occur in non-unitary programs like `Cir2A` and `Cir2B`. This observation suggests that non-unitary programs may demand more precise EQ checking compared to unitary programs.

Fig. 5(b) illustrates the mutant operators for cases No. 27.1 $\sim$ 27.5. A key observation is that troublesome case 27.3 involves a mutation by adding a measurement at the end of the program, contrasting with the other cases where measurements are introduced at the beginning or within loops. This suggests that adding a measurement to a single qubit towards a program's end disrupts its unitarity slightly. Conversely, measurements placed in other positions, especially before controlled gates or within loops, appear to significantly amplify the measurement's effect, thereby greatly disrupting the program's overall unitarity.

In practice, we can select stricter parameters – specifically, choosing a smaller ϵ and a larger k, as depicted in Tables 4 and 6 – can enhance our ability to detect and address troublesome cases effectively. However, by Corollary 1, this implies that a super-linearly longer running time is required. Therefore, balancing the ability to find bugs and the running time is also an aspect that should be considered. In most cases, choosing too precise parameters for a few troublesome error programs may be unnecessary.

7. Threats to validity

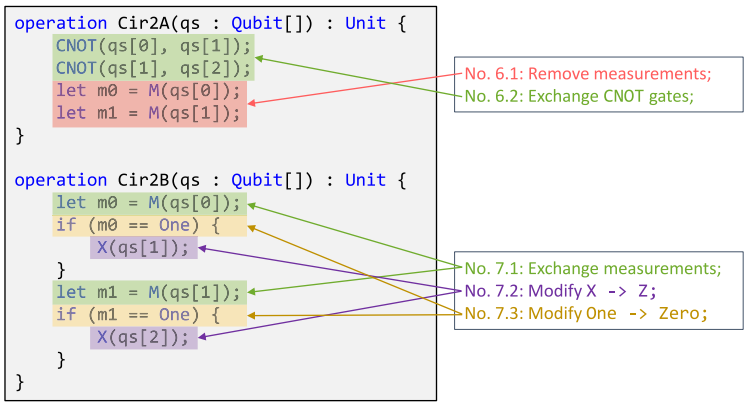
Our theoretical analysis and experimental evaluation have demonstrated the effectiveness of our methods. However, like other test methods, there are still some threats to the validity of our approach.

The first challenge arises from the limitation on the number of qubits. Our equivalence and unitarity checking methods rely on the Swap Test, which requires $2n + 1$ qubits for a program with n qubits. Consequently, our methods can only be applied to programs that utilize less than half of the available qubits. Furthermore, when conducting testing on a classical PC simulator, the running time for large-scale programs becomes impractical.

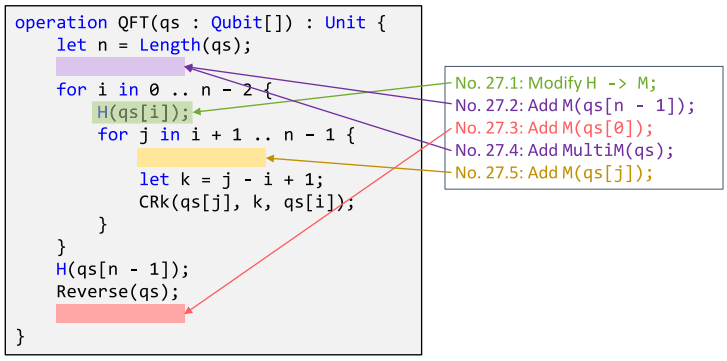
Although we evaluated our methods on a Q# simulator, it is important to note that our algorithms may encounter difficulties when

Table 8
The testing results of benchmark programs with both original algorithms and optimized algorithms.

Task	Running parameters	Expected	Test No.	#Qubits	Original algorithm		Optimized algorithm		T_{opt}/T_0
					% of PASS	Average run time T_0	% of PASS	Average run time T_{opt}	
EQ	$k = 4$ $s = 1545$ $\epsilon = 0.15$ $t = 20$ for optimized algorithm.	PASS	1	2	100	3.0 s	100	42 ms	1.4×10^{-2}
			2	3	100	4.98 s	100	4.2 s	0.84
			3	5	100	21.9 s	100	279 ms	1.3×10^{-2}
			4	5	100	20.5 s	100	283 ms	1.4×10^{-2}
		FAIL	5.1	2	2	1.02 s	0	15 ms	1.5×10^{-2}
			5.2		0	780 ms	0	9.7 ms	1.2×10^{-2}
			5.3	2	0	1.02 s	0	12 ms	1.2×10^{-2}
			5.4		0	1.02 s	0	13 ms	1.3×10^{-2}
			5.5		0	1.2 s	3	11 ms	9.2×10^{-3}
			6.1	3	0	1.2 s	0	9.1 ms	7.6×10^{-3}
			6.2		31	3.36 s	40	3.24 s	0.96
			7.1	3	45	3.72 s	28	2.76 s	0.74
			7.2		14	2.76 s	6	1.86 s	0.67
			7.3		0	1.56 s	0	1.32 s	0.85
			8.1	5	0	5.34 s	0	28 ms	5.2×10^{-3}
			8.2		1	8.22 s	1	70 ms	8.5×10^{-3}
			8.3		0	6.18 s	0	35 ms	5.7×10^{-3}
			8.4		0	5.7 s	0	33 ms	5.8×10^{-3}
			8.5		0	5.88 s	0	33 ms	5.6×10^{-3}
			9.1	5	0	4.92 s	0	49 ms	1.0×10^{-2}
			9.2		0	5.34 s	0	55 ms	1.0×10^{-2}
			9.3		0	5.16 s	0	52 ms	1.0×10^{-2}
			9.4		0	5.22 s	0	53 ms	1.0×10^{-2}
			9.5		0	5.46 s	0	21 ms	9.3×10^{-3}
ID	$k = 50$	PASS	10	6	100	12 ms			
			11	5	100	33 ms			
			12	5	100	48 ms			
			13	3	100	231 ms			
		FAIL	14.1	2	0	1.5 ms			
			14.2		0	2.5 ms			
			15.1	5	0	1.8 ms			
			15.2		0	1.8 ms			
			15.3		0	1.8 ms			
			15.4		0	1.8 ms			
			15.5		0	1.7 ms			
			16.1	5	0	2.2 ms			
			16.2		0	2.2 ms			
			16.3		0	2.2 ms			
			16.4		0	2.3 ms			
			16.5		0	2.2 ms			
			17.1	3	0	6.9 ms			
			17.2		0	8.2 ms			
			17.3		0	8.4 ms			
			17.4		0	7.8 ms			
			17.5		0	8.1 ms			
UN	$k = 4$ $s = 469$ $\epsilon = 0.15$ $t = 20$ for optimized algorithm.	PASS	18	2	100	421 ms	100	434 ms	1.03
			19	2	100	316 ms	98	329 ms	1.04
			20	6	100	3.3 s	99	3.36 s	1.02
			21	5	100	2.4 s	100	2.46 s	1.03
			22	5	100	5.04 s	100	5.52 s	1.10
			23.1	5	99	2.4 s	99	2.46 s	1.03
			23.2		100	2.4 s	99	2.58 s	1.08
			23.3		100	2.46 s	99	2.52 s	1.02
		FAIL	24	3	0	139 ms	0	3.3 ms	2.4×10^{-2}
			25	3	0	131 ms	0	3.1 ms	2.4×10^{-2}
			26	6	0	1.08 s	0	1.14 s	1.06
			27.1	5	0	583 ms	0	5.3 ms	9.1×10^{-3}
			27.2		0	600 ms	0	32 ms	4.8×10^{-2}
			27.3		74	2.6 s	0	7.6 ms	1.0×10^{-2}
			27.4		0	660 ms	0	5.3 ms	2.9×10^{-3}
			27.5		0	660 ms	0	7.2 ms	1.1×10^{-2}
			28.1	5	0	1.68 s	0	18 ms	1.1×10^{-2}
			28.2		0	1.68 s	0	18 ms	1.1×10^{-2}



(a) `Cir2A`, `Cir2B` and their mutant operators in Test No. 6 and 7.



(b) `QFT` program and its mutant operators in Test No. 27.

Fig. 5. The Q# implementation for programs `Cir2A`, `Cir2B`, and `QFT` and their mutant operators in the evaluation.

running on real quantum hardware. For instance, our methods rely on the availability of arbitrary gates, whereas some quantum hardware platforms impose restrictions on the types of quantum gates, such as only supporting the controlled gate (CNOT) on two adjacent qubits. Additionally, the presence of noise in the hardware can potentially diminish the effectiveness of our methods on real quantum hardware.

8. Related work

We provide an overview of the related work in the field of testing quantum programs. Quantum software testing is a nascent research area still in its early stages (Miranskyy and Zhang, 2019; Miranskyy et al., 2021; García de la Barrera et al., 2021; Zhao, 2020). Various methods and techniques have been proposed for testing quantum programs from different perspectives, including quantum assertion (Honarvar et al., 2020; Li et al., 2020), search-based testing (Wang et al., 2021b, 2022a,c), fuzz testing (Wang et al., 2018), combinatorial testing (Wang et al., 2021a), and metamorphic testing (Abreu et al., 2022). These methods aim to adapt well-established classical software testing techniques to the domain of quantum computing. Moreover, Ali et al. (2021) and Wang et al. (2021c) defined input–output coverage criteria for quantum program testing and employed mutation analysis to evaluate the effectiveness of these criteria. However, their coverage criteria have limitations that restrict their applicability to testing complex, multi-subroutine quantum programs. Long and Zhao (Long and Zhao, 2023) presented a framework for testing quantum programs with multiple subroutines. Furthermore, researchers have explored the application of mutation testing and analysis techniques to the field of quantum computing to support the testing of quantum programs (Fortunato et al., 2022b,a,c; Mendiluze et al., 2021). However, black-box testing for quantum programs has not been adequately addressed. In this paper, we propose novel checking algorithms for conducting equivalence, identity, and unitarity checking in black-box testing scenarios.

The field of equivalence checking for quantum circuits is currently active, with a number of tools dedicated to this task (Viamontes et al., 2007; Yamashita and Markov, 2010; Burgholzer and Wille, 2020; Hong et al., 2021, 2022; Wang et al., 2022b). Typically, this task assumes prior knowledge of the structures of two quantum circuits to verify their equivalence, known as white-box checking. Our research, however, pivots towards the equally important but less explored domain of black-box checking for equivalence.

To the best of our knowledge, our work represents the first attempt to adapt quantum information methods to support black-box testing of quantum programs. Various methods have been proposed to address problems related to quantum systems and processes, including quantum distance estimation (Flammia and Liu, 2011; Cerezo et al., 2020), quantum discrimination (Barnett and Croke, 2008; Zhang et al., 2006, 2007), quantum tomography (Vogel and Risken, 1989; Chuang and Nielsen, 1997), and Swap Test (Buhrman et al., 2001; Ekert et al., 2002). By leveraging these methods, it is possible to conduct equivalence checking by estimating the distance between two quantum operations. Additionally, Chen et al. (2023) have conducted a thorough investigation into the unitarity estimation of quantum channels. However, it should be noted that these methods are primarily tailored for quantum information processing, specifically addressing aspects such as quantum noise. Their prerequisites and methodologies are not directly applicable to the black-box testing of quantum programs.

9. Conclusion

This paper presents novel methods for checking the equivalence, identity, and unitarity of quantum programs, which play a crucial role

in facilitating black-box testing of these programs. Through a combination of theoretical analysis and experimental evaluations, we have demonstrated the effectiveness of our methods. The evaluation results clearly indicate that our approaches successfully enable equivalence, identity, and unitarity checking, thereby supporting the black-box testing of quantum programs.

Several areas merit further research. First, exploring additional quantum relations and devising new checking methods for them holds promise for enhancing the scope and applicability of our approach. Second, considering quantum programs that involve both classical and quantum inputs and outputs presents an intriguing research direction, necessitating the development of appropriate modeling techniques.

CRediT authorship contribution statement

Peixun Long: Data curation, Formal analysis, Investigation, Methodology, Project administration, Software, Validation, Visualization, Writing – original draft, Writing – review & editing, Conceptualization. **Jianjun Zhao:** Investigation, Supervision, Writing – review & editing, Validation.

Declaration of competing interest

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

Data availability

We have shared the GitHub link to our code in the article.

Acknowledgments

The authors would like to thank Kean Chen for his valuable discussions regarding the writing of this paper. This work is supported in part by the National Natural Science Foundation of China under Grant 61832015 and JSPS KAKENHI, Japan Grant No. JP23H03372.

Appendix A. Proof of Theorem 1

To prove Theorem 1, we need the following lemma first.

Lemma 1. *Let $\mathcal{E}$ be a quantum operation on d-dimensional Hilbert space. Suppose it has the operator-sum representation $\mathcal{E}(\rho) = \sum_{i=1}^{d^2} E_i \rho E_i^\dagger$. $\mathcal{E}$ represents a unitary transform U if and only if $\forall E_i$, $E_i = k_i U$, where k_i are complex numbers satisfying $\sum_{i=1}^{d^2} |k_i|^2 = 1$.*

Proof. It is obvious that $\{F_i\}$, where $F_1 = U$, $F_k = 0$ for $k > 1$, is an operator-sum representation of U. According to Theorem 8.2 in Nielsen and Chuang (2010), other group of elements $\{E_i\}$ represents U if and only if $E_i = \sum_j w_{ij} F_j = w_{i1} U$, where all w_{ij} constitutes a unitary matrix and thus $\forall j, \sum_i |w_{ij}|^2 = 1$. Let $k_i = w_{i1}$ and then $E_i = k_i U$.

Now we are able to prove Theorem 1.

Proof (*For Theorem 1*).

"⇒". Note that if ρ_1, ρ_2 are both pure states, $tr(\rho_1 \rho_2)$ is the inner product of ρ_1 and ρ_2. $|m\rangle$ and $|n\rangle$ are two different basis vectors, so $\langle m|n\rangle = 0$ and $\langle +_{mn}|-_{mn}\rangle = 0$ for $m \neq n$. A unitary transform keeps purity and inner products.

"⇐". Suppose $\mathcal{E}$ has the operator-sum representation $\mathcal{E}(\rho) = \sum_{i=1} E_i \rho E_i^\dagger$. According to condition (1),

$$
\begin{aligned}
&tr\left[\mathcal{E}(|m\rangle\langle m|)\mathcal{E}(|n\rangle\langle n|)\right] \\
&= tr\left[\sum_{i,j} E_i|m\rangle\langle m|E_i^\dagger E_j|n\rangle\langle n|E_j^\dagger\right] \\
&= \sum_{i,j} tr\left[E_i|m\rangle\langle m|E_i^\dagger E_j|n\rangle\langle n|E_j^\dagger\right] \\
&= \sum_{i,j} \left|\langle m|E_i^\dagger E_j|n\rangle\right|^2 = 0
\end{aligned} \tag{9}
$$

Let $A_{ij} = E_i^\dagger E_j$. So $\forall i, j$ and $m \neq n$, $\langle m|A_{ij}|n\rangle = 0$. It means each A_{ij} is diagonal on the basis $\{|1\rangle \dots |d\rangle\}$. According to condition (2), similarly,

$$
\begin{aligned}
&tr\left[\mathcal{E}(|+_{mn}\rangle\langle +_{mn}|)\mathcal{E}(|-_{mn}\rangle\langle -_{mn}|)\right] \\
&= \sum_{i,j} \left|\langle +_{mn}|A_{ij}|-_{mn}\rangle\right|^2 = 0
\end{aligned} \tag{10}
$$

It means $\forall i, j$,

$$
\langle +_{mn}|A_{ij}|-_{mn}\rangle = \frac{1}{2}(\langle m| + \langle n|)A_{ij}(|m\rangle - |n\rangle) = 0 \tag{11}
$$

so,

$$
\begin{aligned}
&\frac{1}{2}(\langle m|A_{ij}|m\rangle - \langle m|A_{ij}|n\rangle + \langle n|A_{ij}|m\rangle - \langle n|A_{ij}|n\rangle) \\
&= \frac{1}{2}(\langle m|A_{ij}|m\rangle - \langle n|A_{ij}|n\rangle) = 0
\end{aligned} \tag{12}
$$

and thus for $m \neq n$,

$$
\langle m|A_{ij}|m\rangle = \langle n|A_{ij}|n\rangle \tag{13}
$$

Condition (2) selected $d-1$ combinations of $(m, n), m \neq n$ which form the edges of a connected graph with vertices $1, \dots, d$. Then from (13), we can deduce that

$$
\langle 1|A_{ij}|1\rangle = \langle 2|A_{ij}|2\rangle = \cdots = \langle d|A_{ij}|d\rangle \tag{14}
$$

It means the diagonal elements of A_{ij} are equal, i.e., $\forall i, j$, $A_{ij} = E_i^\dagger E_j = k_{ij} I$, where I is identity matrix.

For the case of $i = j$, $E_i^\dagger E_i = k_{ii} I$. Note that $(E_i^\dagger E_i)^\dagger = E_i^\dagger E_i$, which means k_{ii} is a non-negative real number. Let $W_i = \frac{1}{\sqrt{k_{ii}}} E_i$, then $W_i^\dagger W_i = I$. So W_i is a unitary matrix and $E_i = \sqrt{k_{ii}} W_i$. From $\sum_i E_i^\dagger E_i = I$ we can deduce that $\sum_i k_{ii} = 1$.

For the case of $i \neq j$, $E_i^\dagger E_j = \sqrt{k_{ii} k_{jj}} W_i^\dagger W_j = k_{ij} I$. Let $t_{ij} = \frac{k_{ij}}{\sqrt{k_{ii} k_{jj}}}$, then $W_i^\dagger W_j = t_{ij} I$ and thus $W_j = t_{ij} W_i$. Note that $W_i^\dagger W_j$ is a unitary matrix, so $|t_{ij}| = 1$. Let $W = W_1, t_1 = 1$, and for $j \neq 1, t_j = t_{1j}$, then $\forall j, W_j = t_j W$ and $E_j = \sqrt{k_{jj}} t_j W$. In addition, $\sum_j \left|\sqrt{k_{jj}} t_j\right|^2 = \sum_j k_{jj} = 1$. According to Lemma 1, $\mathcal{E}$ is a unitary transform.

Appendix B. Proof of Proposition 1

B.1. s of equivalence checking

Each round of the Swap Test can be seen as a Bernoulli experiment. Algorithm 2 execute three groups of Swap Test with the same number of rounds s. Denote the result of ith round for Swap Test in lines 3, 4, and 5 as x_i, y_i, and z_i respectively. Note that $x_i, y_i, z_i \in \{0, 1\}$ and $s_1 = \sum_{i=1}^s x_i$, $s_2 = \sum_{i=1}^s y_i$, $s_{12} = \sum_{i=1}^s z_i$. Construct a variable w_i:

$$
w_i = 2z_i - x_i - y_i \tag{15}
$$

Obviously $w_i \in [-2, 2]$, and the mean value

$$
\begin{aligned}
\bar{w} = \frac{1}{s}\sum_{i=1}^{s} w_i &= \frac{1}{s}\left(2\sum_{i=1}^{s} z_i - \sum_{i=1}^{s} x_i - \sum_{i=1}^{s} y_i\right) \\
&= \frac{2s_{12} - s_1 - s_2}{s}
\end{aligned} \tag{16}
$$

In Algorithm 2, we estimate variable $r = \frac{2s_{12} - s_1 - s_2}{s} = \bar{w}$. If two target programs are equivalent, whatever the input is, the distribution of every x_i, y_i, and z_i are the same, and the expectation of $\bar{w}$:

$$
E(\bar{w}) = \frac{1}{s}\left(2\sum_{i=1}^{s} E(z_i) - \sum_{i=1}^{s} E(x_i) - \sum_{i=1}^{s} E(y_i)\right) = 0 \tag{17}
$$

According to the *Hoeffding's Inequality* (Hoeffding, 1963),

$$Pr(|\bar{w} - E(\bar{w})| \geq \epsilon) \leq 2e^{\frac{-2\epsilon^2 s^2}{s\cdot(2-(-2))^2}} \tag{18}$$

we obtain

$$Pr(|r| \geq \epsilon) \leq 2e^{-\frac{s\epsilon^2}{8}} \tag{19}$$

$Pr(|r| \geq \epsilon)$ is the probability of failing in one test point. Note if two target programs are equivalent, it is the same for all test points. Let P_2 be the probability of passing in one test point, then

$$P_2 > 1 - 2e^{-\frac{s\epsilon^2}{8}} \tag{20}$$

Running testing on k test points, because if one test point fails, the testing result is "FAIL", the overall probability of failing is $1 - P_2^k$, and it is also the probability of type II error. To control it not to exceed α_2:

$$1 - P_2^k \leq \alpha_2 \tag{21}$$

Then,

$$k \leq \frac{\ln(1-\alpha_2)}{\ln P_2} \leq \frac{\ln(1-\alpha_2)}{\ln(1-2e^{-\frac{s\epsilon^2}{8}})} \tag{22}$$

Figure out s:

$$s \geq \frac{8}{\epsilon^2} \ln \frac{2}{1-(1-\alpha_2)^{\frac{1}{k}}} \tag{23}$$

B.2. s of unitarity checking

In Algorithm 4, we estimate variable $r = 1 - \frac{2s_1}{s}$. If the target program $\mathcal{P}$ is unitary, for any input (ρ_1, ρ_2) selected by the algorithm, $tr(\mathcal{P}(\rho_1), \mathcal{P}(\rho_2)) = 0$. So $E(r) = 0$ and $E(s_1) = \frac{s}{2}$. According to the *Chernoff's Bound* (Chernoff, 1952),

$$Pr(|s_1 - E(s_1)| \geq \epsilon E(s_1)) \leq 2^{-\frac{s\epsilon^2}{2}} \tag{24}$$

That is

$$Pr\left(|s_1 - \frac{s}{2}| \geq \frac{s\epsilon}{2}\right) \leq 2^{-\frac{s\epsilon^2}{2}}$$

$$Pr(|r| \geq \epsilon) \leq 2^{-\frac{s\epsilon^2}{2}} \tag{25}$$

$Pr(|r| \geq \epsilon)$ is the probability of failing in one test point. Note if the target program is unitary, $Pr(|r| \geq \epsilon)$ is the same for all test points. Let P_2 be the probability of passing in one test point, then

$$P_2 > 1 - 2^{-\frac{s\epsilon^2}{2}} \tag{26}$$

Similarly, limit the probability of failing:

$$1 - P_2^k \leq \alpha_2 \tag{27}$$

Then,

$$k \leq \frac{\ln(1-\alpha_2)}{\ln P_2} \leq \frac{\ln(1-\alpha_2)}{\ln(1-2^{-\frac{s\epsilon^2}{2}})} \tag{28}$$

Figure out s:

$$s \geq \frac{2}{\epsilon^2} \log_2 \frac{1}{1-(1-\alpha_2)^{\frac{1}{k}}} \tag{29}$$

B.3. Proof of formula (7)

The left less-than sign. Note that $\forall a \in (0,1)$ and $x \in \mathbb{R}$,

$$a^x \geq x \ln a + 1 \tag{30}$$

It deduces that

$$\frac{1}{1-a^x} \geq \frac{1}{-x \ln a} \tag{31}$$

Substitute x with $1/k$ and a with $1-\alpha_2$,

$$\frac{1}{1-(1-\alpha_2)^{\frac{1}{k}}} \geq \frac{k}{-\ln(1-\alpha_2)} \tag{32}$$

The right less-than sign. $k \geq 1$, so $1/k \in (0,1]$. Note that if $a \in (0,1)$ and $x \in [0,1]$,

$$a^x \leq (a-1)x + 1 \tag{33}$$

It deduces that

$$\frac{1}{1-a^x} \leq \frac{1}{(1-a)x} \tag{34}$$

Substitute x with $1/k$ and a with $1-\alpha_2$,

$$\frac{1}{1-(1-\alpha_2)^{\frac{1}{k}}} \leq \frac{k}{\alpha_2} \tag{35}$$

References

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

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

Barenco, A., Berthiaume, A., Deutsch, D., Ekert, A., Jozsa, R., Macchiavello, C., 1997. Stabilization of quantum computations by symmetrization. SIAM J. Comput. 26 (5), 1541–1557.

Barnett, S.M., Croke, S., 2008. Quantum state discrimination. arXiv preprint arXiv:0810.1970v1.

Beizer, B., 1995. Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc..

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

Buhrman, H., Cleve, R., Watrous, J., De Wolf, R., 2001. Quantum fingerprinting. Phys. Rev. Lett. 87 (16), 167902.

Burgholzer, L., Wille, R., 2020. Advanced equivalence checking for quantum circuits. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 40 (9), 1810–1824.

Cerezo, M., Poremba, A., Cincio, L., Coles, P.J., 2020. Variational quantum fidelity estimation. Quantum 4, 248.

Chen, K., Wang, Q., Long, P., Ying, M., 2023. Unitarity estimation for quantum channels. IEEE Trans. Inform. Theory 1.

Chernoff, H., 1952. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. Ann. Math. Stat. 23 (4), 493–507.

Chuang, I.L., Nielsen, M.A., 1997. Prescription for experimental determination of the dynamics of a quantum black box. J. Modern Opt. 44 (11–12), 2455–2467.

D'Ariano, G.M., Paris, M.G., Sacchi, M.F., 2003. Quantum tomography. Adv. Imaging Electron Phys. 128, 206–309.

Deutsch, D., 1985. Quantum theory, the church–turing principle and the universal quantum computer. Proc. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci. 400 (1818), 97–117.

Ekert, A.K., Alves, C.M., Oi, D.K., Horodecki, M., Horodecki, P., Kwek, L.C., 2002. Direct estimations of linear and nonlinear functionals of a quantum state. Phys. Rev. Lett. 88 (21), 217901.

Farhi, E., Goldstone, J., Gutmann, S., 2014. A quantum approximate optimization algorithm. arXiv preprint arXiv:1411.4028.

Flammia, S.T., Liu, Y.-K., 2011. Direct fidelity estimation from few Pauli measurements. Phys. Rev. Lett. 106, 230501.

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

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

Fortunato, D., Campos, J., Abreu, R., 2022c. QMutPy: a mutation testing tool for Quantum algorithms and applications in Qiskit. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 797–800.

García de la Barrera, A., García-Rodríguez de Guzmán, I., Polo, M., Piattini, M., 2021. Quantum software testing: State of the art. J. Softw.: Evol. Process e2419.

Garcia-Escartin, J.C., Chamorro-Posada, P., 2011. Equivalent quantum circuits. arXiv preprint arXiv:1110.2998.

Grover, L.K., 1996. A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing. pp. 212–219.

Guo, J., Lou, H., Yu, J., Li, R., Fang, W., Liu, J., Long, P., Ying, S., Ying, M., 2023. isQ: An integrated software stack for quantum programming. IEEE Trans. Quant. Eng. 1–18.

Harrow, A.W., Hassidim, A., Lloyd, S., 2009. Quantum algorithm for linear systems of equations. Phys. Rev. Lett. 103 (15), 150502.

Hoeffding, W., 1963. Probability inequalities for sums of bounded random variables. J. Amer. Statist. Assoc. 58, 13–30.
Honarvar, S., Mousavi, M., Nagarajan, R., 2020. Property-based testing of quantum programs in Q#. In: First International Workshop on Quantum Software Engineering (Q-SE 2020).
Hong, X., Feng, Y., Li, S., Ying, M., 2022. Equivalence checking of dynamic quantum circuits. In: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design. ICCAD '22, Association for Computing Machinery, New York, NY, USA.
Hong, X., Ying, M., Feng, Y., Zhou, X., Li, S., 2021. Approximate equivalence checking of noisy quantum circuits. In: 2021 58th ACM/IEEE Design Automation Conference. DAC, IEEE, pp. 637–642.
Huang, Y., Martonosi, M., 2019. Statistical assertions for validating patterns and finding bugs in quantum programs. In: Proceedings of the 46th International Symposium on Computer Architecture. pp. 541–553.
Janzing, D., Wocjan, P., Beth, T., 2005. "Non-identity-check" is QMA-complete. Int. J. Quantum Inf. 3 (03), 463–473.
Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y., 2020. Projection-based runtime assertions for testing and debugging quantum programs. Proc. ACM Program. Lang. 4 (OOPSLA), 1–29.
Long, P., Zhao, J., 2023. Testing multi-subroutine quantum programs: From unit testing to integration testing. arXiv preprint arXiv:2306.17407.
McArdle, S., Endo, S., Aspuru-Guzik, A., Benjamin, S., Yuan, X., 2018. Quantum computational chemistry. arXiv preprint arXiv:1808.10402.
Mendiluze, E., Ali, S., Arcaini, P., Yue, T., 2021. Muskit: A mutation analysis tool for quantum software testing. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1266–1270.
Mingsheng, Y., 2016. Foundations of Quantum Programming, first ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
Miranskyy, A., Zhang, L., 2019. On testing quantum programs. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). IEEE, pp. 57–60.
Miranskyy, A., Zhang, L., Doliskani, J., 2020. Is your quantum program bug-free? In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). IEEE Computer Society, pp. 29–32.
Miranskyy, A., Zhang, L., Doliskani, J., 2021. On testing and debugging quantum software. arXiv preprint arXiv:2103.09172.
Montanaro, A., de Wolf, R., 2013. A survey of quantum property testing. arXiv preprint arXiv:1310.2035.
Mosca, M., 2018. Cybersecurity in an era with quantum computers: Will we be ready? IEEE Secur. Privacy 16 (5), 38–41.
National Academies of Sciences, Engineering and Medicine, et al., 2019. Quantum Computing: Progress and Prospects. National Academies Press.
Nielsen, M.A., Chuang, I.L., 2010. Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press.
Pointing, J., Padon, O., Jia, Z., Ma, H., Hirth, A., Palsberg, J., Aiken, A., 2021. Quanto: Optimizing quantum circuits with automatic generation of circuit identities. arXiv preprint arXiv:2111.11387.
Shor, P.W., 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Rev. 41 (2), 303–332.
Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M., 2018. Q#: enabling scalable quantum computing and development with a high-level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. pp. 1–10.
Viamontes, G.F., Markov, I.L., Hayes, J.P., 2007. Checking equivalence of quantum circuits and states. In: 2007 IEEE/ACM International Conference on Computer-Aided Design. IEEE, pp. 69–74.
Vogel, K., Risken, H., 1989. Determination of quasiprobability distributions in terms of probability distributions for the rotated quadrature phase. Phys. Rev. A 40, 2847–2849.
Wang, X., Arcaini, P., Yue, T., Ali, S., 2021a. Application of combinatorial testing to quantum programs. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 179–188.
Wang, X., Arcaini, P., Yue, T., Ali, S., 2021b. Generating failing test suites for quantum programs with search. In: International Symposium on Search Based Software Engineering. Springer, pp. 9–25.
Wang, X., Arcaini, P., Yue, T., Ali, S., 2021c. Quito: a coverage-guided test generator for quantum programs. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1237–1241.
Wang, X., Arcaini, P., Yue, T., Ali, S., 2022a. QuSBT: Search-based testing of quantum programs. arXiv preprint arXiv:2204.08561.
Wang, J., Gao, M., Jiang, Y., Lou, J., Gao, Y., Zhang, D., Sun, J., 2018. QuanFuzz: Fuzz testing of quantum program. arXiv preprint arXiv:1810.10310.
Wang, Q., Li, R., Ying, M., 2022b. Equivalence checking of sequential quantum circuits. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (9), 3143–3156.
Wang, X., Yu, T., Arcaini, P., Yue, T., Ali, S., 2022c. Mutation-based test generation for quantum programs with multi-objective search. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 1345–1353.
Yamashita, S., Markov, I.L., 2010. Fast equivalence-checking for quantum circuits. In: 2010 IEEE/ACM International Symposium on Nanoscale Architectures. IEEE, pp. 23–28.
Yang, Z., Fan, J.Z., Proppe, A.H., de Arquer, F.P.G., Rossouw, D., Voznyy, O., Lan, X., Liu, M., Walters, G., Quintero-Bermudez, R., et al., 2017. Mixed-quantum-dot solar cells. Nat. Commun. 8 (1), 1–9.
Zhang, C., Feng, Y., Ying, M., 2006. Unambiguous discrimination of mixed quantum states. Phys. Lett. A 353 (4), 300–306.
Zhang, C., Wang, G., Ying, M., 2007. Discrimination between pure states and mixed states. Phys. Rev. A 75, 062306.
Zhao, J., 2020. Quantum software engineering: Landscapes and horizons. arXiv preprint arXiv:2007.07047.
Zhao, P., Zhao, J., Miao, Z., Lan, S., 2021. Bugs4Q: A benchmark of real bugs for quantum programs. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1373–1376.

Peixun Long received the B.S. degree from the School of Physics, Peking University, China, in 2016, and received the M.E. degree from the Institute of High Energy Physics, Chinese Academy of Sciences, China, in 2019. He is currently working toward the Ph.D. degree at the Institute of Software, Chinese Academy of Sciences, China. His current research interests include quantum computing and quantum software engineering.

Jianjun Zhao received the BS degree from Tsinghua University, China, in 1987, and the Ph.D. degree from Kyushu University, Japan, in 1997, both in computer science. He is a full professor with the School of Information Science and Electrical Engineering, Kyushu University. He was an assistant/associate professor with the Fukuoka Institute of Technology, Japan (1997–2005), and a full professor with Shanghai Jiao Tong University, China (2005–2016). He was a visiting scientist with MIT LCS from 2002–2003. His research interests include software engineering and programming languages for classical and non-classical computing.