Quantum symbolic execution

Jiang Nan · Wang Zichen · Wang Jian

Received: 28 August 2022 / Accepted: 4 October 2023 / Published online: 20 October 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract
With advances in quantum computing, researchers can now write and run many quantum programs. However, there is still a lack of effective methods for debugging quantum programs. In this paper, quantum symbolic execution (QSE) is proposed to generate test cases, which helps to find bugs in quantum programs. The main idea of quantum symbolic execution is to find the suitable test cases from all possible ones (i.e., test case space). It is different from the way of classical symbolic execution, which gets test cases by calculating instead of searching. QSE utilizes quantum superposition and parallelism to store the test case space with only a few qubits. According to the conditional statements in the debugged program, the test case space is continuously divided into subsets, subsubsets and so on. Elements in the same subset are suitable test cases that can test the corresponding branch in the code to be tested. QSE not only provides a possible way to debug quantum programs, but also avoids the difficult problem of solving constraints in classical symbolic execution.

Keywords Quantum symbolic execution · Test cases · Quantum program testing · Quantum program · Quantum computing

1 Introduction
Quantum computing has attracted much attention, because quantum superposition, entanglement and other properties can greatly improve the efficiency of computing [1, 2]. In recent years, with the development of quantum computer hardware [3, 4] and quantum software, quantum programming [5–8] has also been greatly developed.

✉ Wang Jian
wangjian@bjtu.edu.cn

1 Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China
2 School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China
3 Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing 100044, China
Researchers can write and run many quantum algorithms that have been proposed before but cannot be implemented due to limitations, such as Grover’s algorithm [9], quantum principal component analysis algorithm [10], and quantum phase estimation [11]. In the process of writing quantum programs, some errors will inevitably occur [12–14]. For example, Zhao [15] defined a few bugs that focus on misuses of features of the quantum programming language—Qiskit [6]. Huang [16] also recorded some bugs in the Scaffold compiler [17]. For quantum programs, we still need to take corresponding measures to find these errors and fix them. Due to the characteristics of quantum computing, we cannot debug programs as in the classical environment. This difficulty in debugging quantum programs hinders the development of quantum computing. An effective quantum program debugging scheme is needed.

Researchers have proposed some methods for debugging quantum programs, including quantum unit tests [18] and quantum assertions [16, 19–21]. Unit tests are used to determine whether a specific function is correct under a specific condition. The role of the assertion is that when the program executes to the assertion, the corresponding assertion should be true, and if the assertion is not true, the program should terminate execution. These methods have corresponding quantum versions. However, these methods are not very good to meet the needs. Currently, assertions in the quantum environment include statistical assertions [16] based on classical observations, dynamic runtime assertions [19] that use auxiliary qubits to obtain information indirectly, a projection-based runtime assertion [20], and dynamic assertion [21] that extend dynamic runtime assertions [19]. These assertions have two main shortcomings. Firstly, they are mostly used when an error has occurred during the running of the program or when the programmer suspects that there is an error somewhere in the program. Just like people do not directly set breakpoints on the entire program, but often set breakpoints only when the output is not as expected. Secondly, the use of assertions relies on the prediction of results. They need to compare the actual output with the expected result to judge whether the program is error. This is not simple for quantum programs. Microsoft’s Q# [18] provides a method for unit testing of quantum programs, which tests a unit of a quantum program individually to verify whether it meets expectations, and internally still uses assertions to achieve this goal. There is another method Quito (quantum input output coverage) [22]. The biggest contribution of [22] is to define three coverage criteria for the input and output of quantum program debugging. But the biggest flaw of this method is that it still uses statistical analysis to determine test pass and fail, which certainly does not reduce the complexity of quantum program debugging. Therefore, they cannot meet the programmer’s needs for quantum program debugging very well.

Only unit tests and assertion cannot meet the needs of program debugging. In classical program debugging field, symbolic execution is another important debug method and it has appeared much earlier [23]. With the development of constraint solving technology, symbolic execution has become an effective technology for generating high-coverage test cases [24] and been widely used in different areas such as software testing, analysis and verification [25–27].

This paper proposes a quantum symbolic execution (QSE) method, which focuses on generating high-coverage test cases for quantum programs. QSE uses quantum superposition and parallel characteristics to store the test case space with only a few
qubits. According to the conditional statements in the debugged program, the test case space is continuously divided into subsets. Elements in the same subset are suitable test cases that can test the corresponding branch in the code to be tested. QSE not only provides a possible way to debug quantum programs, but also avoids the difficult problem of solving constraints in classical symbolic execution.

2 Related works

In this section, we briefly introduce the classical symbolic execution and some existing quantum modules that will be used in QSE.

2.1 Classical symbolic execution (CSE)

Programs often have conditional statements, and each branch represents an execution path to the program. In software testing, symbolic execution is a way to generate test cases that cover each execution path. Symbolic execution works by two steps:

1. Creating execution paths, and
2. Using a constraint solver to calculate the answers to the execution paths, i.e., generating test cases.

To formally accomplish this task, symbolic execution maintains two states globally: a symbolic state $\sigma$, which maps variables to symbolic expressions, and symbolic path constraints ($PC$s), which are quantifier-free first-order logical formulas over symbolic expressions. At the beginning of a symbolic execution, $\sigma$ is initialized to an empty map and $PC$ is initialized to $true$. Both $\sigma$ and $PC$ are populated during the course of symbolic execution. The update rule of $\sigma$ is:

- At every read statement $\text{var} = \text{sym\_input}()$ that receives program input, symbolic execution adds the mapping $\text{var} \mapsto s$ to $\sigma$, where $s$ is a fresh symbolic value.
- At every assignment $\text{v} = e$, symbolic execution updates $\sigma$ by mapping $\text{v}$ to $\sigma(e)$, where $\sigma(e)$ is the mapping of the symbolic state $\sigma$ to the expression $e$.

The update rule of $PC$ is:

- At every conditional statement $\text{if } (e) \text{ S1 else S2, PC is updated to } PC_1 = PC \land \sigma(e)$ ("then" branch) and $PC_2 = PC \land \neg\sigma(e)$ ("else" branch).

For example, the symbolic execution of the code in Fig. 1 starts with an empty symbolic state $\sigma$ and a symbolic path constraint $true$. After Line 03, $\sigma = \{x \mapsto x_0, y \mapsto y_0\}$; after Line 05, a path constraint $(x_0 + y_0 < 4) \land (x_0 > y_0)$ is created; and after Line 09, a path constraint $(x_0 + y_0 \geq 4) \land (y_0 > 1)$ is created. Finally, there are 4 path constraints: $PC_{11}$, $PC_{12}$, $PC_{21}$, and $PC_{22}$. Each path constraint is solved with a constraint solver to obtain test cases. $\{x = 2, y = 1\}$, $\{x = 1, y = 2\}$, $\{x = 3, y = 2\}$, and $\{x = 4, y = 1\}$ are the possible outputs of the constraint solver for $PC_{11}$, $PC_{12}$, $PC_{21}$, and $PC_{22}$ respectively, i.e., they are suitable test cases.
Fig. 1 An example to illustrate symbolic execution

\begin{verbatim}
01. int main(){
  x=sym_input();
  y=sym_input();
  if(x>y<4{
    if(x>y){
      return 0;
    }
  }
  else{
    if(y>1{
      return 0;
    }
    return 0;
  }
}
\end{verbatim}

\[ \sigma_0, PC: true \]
\[ \sigma_1[x \rightarrow x_0, y \rightarrow y_0, PC: true \]
\[ \sigma_2[x \rightarrow x_0, y \rightarrow y_0, PC: (x_0 + y_0 < 4) \]
\[ \sigma_3[x \rightarrow x_0, y \rightarrow y_0, PC: (x_0 + y_0 < 4) \land (x_0 > y_0) \]
\[ \sigma_4[x \rightarrow x_0, y \rightarrow y_0, PC: (x_0 + y_0 < 4) \land (x_0 <= y_0) \]
\[ \sigma_5[x \rightarrow x_0, y \rightarrow y_0, PC: (x_0 + y_0 >= 4) \land (y_0 > 1) \]
\[ \sigma_6[x \rightarrow x_0, y \rightarrow y_0, PC: (x_0 + y_0 >= 4) \land (y_0 <= 1) \]

Fig. 2 The execution tree for the example in Fig. 1

All the execution paths of a program can be represented using a tree, called the execution tree. For example, Fig. 2 gives the execution tree of the code in Fig. 1. The 4 branches correspond to the 4 path constrains.

2.2 Related quantum modules

Suppose \( a \) and \( b \) are two \( n \)-qubit binary numbers, quantum adder [28] “A” implements the addition of two qubits:

\[ A(|ab\rangle|0\rangle^{\otimes n+1}) = |ab\rangle|a + b\rangle. \]

The quantum module is shown in Fig. 3a.
Quantum multiplier \([29]\) “\(M\)” implements multiplication of two qubits:

\[
M(|ab\rangle|0\rangle^{\otimes 2n}) = |ab\rangle|a \times b\rangle.
\]

The quantum module is shown in Fig. 3b.

The quantum comparator \([30]\) “\(C\)” is used to compare two binary numbers. \(c_1\) and \(c_2\) are two 1-qubit outputs to record the comparison:

\[
C(|ab\rangle|00\rangle) = |ab\rangle|c_1 c_2\rangle.
\]

When \(a > b\), \(|c_1 c_2\rangle = |10\rangle\); when \(a < b\), \(|c_1 c_2\rangle = |01\rangle\); and when \(a = b\), \(|c_1 c_2\rangle = |00\rangle\). The module is shown in Fig. 3c.

3 Quantum symbolic execution

In this section, we first give the workflow of quantum symbolic execution. Then, we explain how to prepare the initial test case space and use relational operators and logical operators to delineate subspaces. Then, we give the overall framework of QSE and finally give an example to illustrate.

3.1 Main idea

In Sect. 2.1, we briefly describe the process of symbolic execution in the classical environment. Generally speaking, it first traverses the program to collect the path
Fig. 4 The contrast between classical symbolic execution and quantum symbolic execution

Quantum symbolic execution is completely different, which works by two steps:

1) generating a test case space that includes all possible test cases, and
2) according to the conditional statements in the code to be tested, partitioning the test case space into subspaces, and each subspace contains all the test cases that fit into a path constraint.

Figure 4 contrasts classical symbolic execution and quantum symbolic execution. QSE uses two quantum registers: $|s\rangle$ and $|c\rangle$, where

$$|q\rangle = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |s_i\rangle \otimes |c_i\rangle$$

$|s\rangle = |s^{n-1}s^{n-2}\cdots s^0\rangle$ consists of $n$ qubits and $s_i$ is a value used to represent a test case. $|c\rangle = |c^{m-1}c^{m-2}\cdots c^0\rangle$ consists of $m$ qubits and is the flag to subspace. $|s\rangle$ and $|c\rangle$ entangle together to realize the partition of $|s\rangle$: $s_i$ with the same $c_i$ belongs to the same subset, i.e., test cases for the same branch. $|s\rangle$ and $|c\rangle$ are collectively referred to as $|q\rangle$.

The flag $|c\rangle$ plays an important role in QSE, and it is gradually modified as the conditional statements in the code to be tested. Different conditions correspond to different ways to modify $|c\rangle$. Therefore, it is necessary to know how many types of conditions there are when programming. According to [31–33], the conditions mainly include relational operation in Table 1 and logical operation in Table 2.

The effects of relational and logical operations on $|c\rangle$ will be described in detail in Sects. 3.3 and 3.4, respectively.
Table 1  Relational operation

| Relational operators | Meaning       |
|----------------------|---------------|
| <                   | less than     |
| <=                  | less than or equal to |
| >                   | greater than  |
| >=                  | greater than or equal to |
| ==                  | equal to      |
| !=                  | not equal to  |

Table 2 Logical Operation

| Logical operators | Meaning |
|-------------------|---------|
| &&                | AND     |
| ||                | OR      |
| !                 | NOT     |

3.2 Preparation of the test case space

Prepare $m + n$ qubits and set all of them to $|0\rangle$. The initial state of $|q\rangle$ is

$$|q\rangle_0 = |0\rangle^\otimes n \otimes |0\rangle^\otimes m$$

(2)

i.e., $|s\rangle_0 = |0\rangle^\otimes n$ and $|c\rangle_0 = |0\rangle^\otimes m$.

$n H$ quantum gates and $m I$ quantum gates are used to transform the initial state $|q\rangle_0$ to state $|q\rangle_1$, where

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

The quantum preparation of the test case space can be expressed as $U_1$:

$$U_1 = H^\otimes n \otimes I^\otimes m$$

(3)
Fig. 5 The preparation of the test case space

\[
\begin{align*}
|q\rangle_1 &= U_1(|q\rangle_0) \\
&= H^\otimes n (|s\rangle_0) \otimes I^\otimes m (|c\rangle_0) \\
&= (H|0\rangle)^\otimes n \otimes (I|0\rangle)^\otimes m \\
&= \frac{1}{\sqrt{2}} ((0) + |1\rangle) \otimes \frac{1}{\sqrt{2}} ((0) + |1\rangle) \otimes \cdots \otimes \frac{1}{\sqrt{2}} ((0) + |1\rangle) \otimes |0\rangle^m \\
&= \frac{1}{\sqrt{2}^n} \left( |0\cdots00\rangle + |0\cdots01\rangle + \cdots + |1\cdots11\rangle \right) \otimes |0\rangle^m \\
&= \frac{1}{\sqrt{2}^n} \left( |0\rangle + |1\rangle + \cdots + |2^n - 1\rangle \right) \otimes |0\rangle^m \\
&= \frac{1}{\sqrt{2}^n} \sum_{i=0}^{2^n-1} |i\rangle \otimes |0\rangle^m \\
&= |s\rangle \otimes |0\rangle^m
\end{align*}
\]

where \(|s\rangle = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |i\rangle\). The quantum circuit is shown in Fig. 5.

Equation (4) shows that the test case space \(|s\rangle\) stores all integers from 0 to \(2^n - 1\), which are all the possible test cases. If the code to be tested contains \(l\) (\(l > 1\)) variables \(x_1, x_2, \ldots, x_l\), \(|s\rangle\) is still able to store all possible test cases. Divide the \(n\) qubits of \(|s\rangle\) into \(l\) parts and each part stores all the possible values of a variable. The \(i\)th part contains \(n_i\) qubits \(|s_{x_i}^{n_i-1}s_{x_i}^{n_i-2}\cdots s_{x_i}^0\rangle\), where \(n = \sum_{i=1}^{l} n_i\). For example, the code in Fig. 1 has two variables: \(x\) and \(y\). They contain 3 and 2 qubits, respectively. Hence,

\[
|s\rangle = |s_x s_y\rangle = \frac{1}{\sqrt{2}^3} \sum_{i=0}^{7} |i\rangle \otimes \frac{1}{\sqrt{2}^2} \sum_{i=0}^{3} |i\rangle = \frac{1}{\sqrt{2}^5} \sum_{i=0}^{31} |i\rangle
\]

3.3 Relational operator

Relational operators compare two numbers. Therefore, QSE uses the quantum comparator to divide the test case space. Section 2.2 shows that the quantum comparator has two output qubits: \(|c_1 c_2\rangle\). Suppose they correspond to some two adjacent qubits in \(|c\rangle = |c^{m-1}c^{m-2}\cdots c^0\rangle\) and mark them as \(|c^i c^i-1\rangle\). Combining Table 1, we can get the relationship between the relational operators and the state of the output qubits.
as shown in Table 3. In this table, “∗” indicates that there is no requirement for the state of that qubit.

Sometimes, instead of directly comparing two variables, the code to be tested compares the values of two expressions. Suppose the two expressions are $e_1$ and $e_2$, and their outputs are $|\varphi_1\rangle$ and $|\varphi_2\rangle$, respectively. A quantum comparator is used to compare $|\varphi_1\rangle$ and $|\varphi_2\rangle$. $|c^i\rangle$ and $|c^{i-1}\rangle$ record the results of the comparison, i.e., they are the flags to segment the test case space. The segmentation of the test case space by a relational operator is expressed as $U_r$:

$$U_r = C \otimes e_1 \otimes e_2$$ (5)

$U_r$ can segment the test case space by modifying the state of $|c^i c^{i-1}\rangle$.

$$U_r(|s\rangle|0\rangle^{\otimes k}|0\rangle^{\otimes t}|00\rangle)$$

$$= C(e_1(|s\rangle|0\rangle^{\otimes k})e_2(|s\rangle|0\rangle^{\otimes t})|00\rangle)$$

$$= C(|\varphi_1\rangle|\varphi_2\rangle|00\rangle)$$

$$= |s\rangle \otimes C(|\varphi_1\rangle|\varphi_2\rangle|c^i c^{i-1}\rangle)$$

In $|s\rangle \otimes |c^i c^{i-1}\rangle$, due to the entanglement between $|s\rangle$ and $|c^i c^{i-1}\rangle$, different states of $|c^i c^{i-1}\rangle$ correspond to different subspaces of $|s\rangle$. The circuit is shown in Fig. 6.

In the following, we use $|c^i c^{i-1}\rangle_e$ to indicate that $|c^i c^{i-1}\rangle$ is in the output state of $e$, and $|c^i c^{i-1}\rangle_\tau$ to indicate that $|c^i c^{i-1}\rangle$ is not in the output state of $e$, where $e = (e_1 \circ e_2)$ and $\circ \in \{<, \leq, >, \geq, =, \neq\}$. For example, if $e = (e_1 < e_2)$, $|c^i c^{i-1}\rangle_e = |01\rangle$, and $|c^i c^{i-1}\rangle_\tau = |10\rangle$ or $|11\rangle$ or other non-$|01\rangle$ states.

### 3.4 Logical operators

#### 3.4.1 T module

Usually, the inputs to a logical operator are the outputs of rational operator(s). A rational operator has two outputs $|c^i c^{i-1}\rangle$. Hence, Module $T$ is defined firstly to facilitate later descriptions.
Fig. 6 The segmentation of the test case space by relational operations

Fig. 7 Six cases of Module T

\( T \) is a control module that acts on two qubits \(|c^i c^{i-1}\rangle\). According to Table 3, \(|c^i c^{i-1}\rangle\) have six states. Therefore, there are also 6 cases of \( T = \{ T_<, T_\leq, T_>, T_\geq, T_=, T_\neq \} \). Their circuits are shown in Fig. 7.

For example, in Fig. 7a, because it is \( T_\lt \), the state of \(|c^i c^{i-1}\rangle\) is \(|01\rangle\). Hence, we place a 0-control on qubit \(|c^i\rangle\) and a 1-control on \(|c^{i-1}\rangle\). Thus, these two control qubits represent that the result of the previous relational operation is “less than”.

3.4.2 Logical operators

There are three logical operators. We will give their quantum circuits one by one.

(1) **AND**

Suppose there is an expression \( e_1 \& \& e_2 \), where \( e_1 \) and \( e_2 \) are two rational operations. The logical AND in QSE is shown in Fig. 8a, where \( T_{e_1}, T_{e_2} \in T, |c_1^i c_1^{i-1}\rangle \) are the flags of \( e_1 \), and \( |c_2^i c_2^{i-1}\rangle \) are the flags of \( e_2 \). The output of logical AND is \(|c_A\rangle\): if and
only if both \( e_1 \) and \( e_2 \) are satisfied, \( |c_A\rangle \) becomes \( |1\rangle \); otherwise, it remains unchanged in \( |0\rangle \) state. That is to say, \( |c_A\rangle \) becomes a flag of logical AND.

Define

\[
U_{Ar} = T_{e_1} - T_{e_2} - \text{NOT}
\]

Then,

\[
U_{Ar}(|c_1^i\rangle\langle c_1^{i-1}| \otimes |c_2^j\rangle\langle c_2^{j-1}| |0\rangle) = T_{e_1} - T_{e_2} - \text{NOT}(|c_1^i\rangle\langle c_1^{i-1}| \otimes |c_2^j\rangle\langle c_2^{j-1}| |0\rangle) = |c_1^i\rangle\langle c_1^{i-1}| e_1 \otimes |c_2^j\rangle\langle c_2^{j-1}| e_2 \otimes |1\rangle + |c_1^i\rangle\langle c_1^{i-1}| e_1 \otimes |c_2^j\rangle\langle c_2^{j-1}| e_2 \otimes |0\rangle + |c_1^i\rangle\langle c_1^{i-1}| e_1 \otimes |c_2^j\rangle\langle c_2^{j-1}| e_2 \otimes |0\rangle + |c_1^i\rangle\langle c_1^{i-1}| e_1 \otimes |c_2^j\rangle\langle c_2^{j-1}| e_2 \otimes |0\rangle
\]

If \( e_1 \) and \( e_2 \) are two logical operations, it is only necessary to replace \( |c_1^i\rangle\langle c_1^{i-1}| \) with \( |c_1^i\rangle\langle c_1^{i-1}| \), \( |c_2^j\rangle\langle c_2^{j-1}| \) with \( |c_2^j\rangle\langle c_2^{j-1}| \), and \( T_{e_1} \) and \( T_{e_2} \) with 1-control, as shown in Fig. 8b, where \( |c_1\rangle \) and \( |c_2\rangle \) are the outputs of \( e_1 \) and \( e_2 \), respectively. Now

\[
U_{Al} = \text{CC-NOT}
\]

and

\[
U_{Al}(|c_1\rangle \otimes |c_2\rangle \otimes |0\rangle) = \text{CC-NOT}(|c_1\rangle \otimes |c_2\rangle \otimes |0\rangle) = |1\rangle \otimes |1\rangle \otimes |1\rangle + |0\rangle \otimes |1\rangle \otimes |0\rangle + |1\rangle \otimes |0\rangle \otimes |0\rangle + |0\rangle \otimes |0\rangle \otimes |0\rangle
\]

(2) OR

For logical OR, there is an expression \( e_1|\rangle e_2 \). Figure 9a shows the logical OR in QSE if \( e_1 \) and \( e_2 \) are two rational operations. The output of logical OR is \( |c_O\rangle \): as long as one of \( e_1 \) and \( e_2 \) is satisfied, \( |c_O\rangle \) becomes \( |1\rangle \); otherwise, it remains unchanged in \( |0\rangle \) state. That is to say, \( |c_O\rangle \) becomes a flag of logical OR.
Define

\[ U_{Or} = T_{e_1} - NOT \otimes T_{e_2} - NOT \otimes T_{e_1} - NOT \]  \hspace{1cm} (11)

Then,

\[
U_{Or} (|c_1^{i-1} c_2^{i-1} \rangle \otimes |c_2^{i-1} c_2^{i-1} \rangle \otimes |0\rangle) \\
= T_{e_1} - T_{e_2} - NOT \otimes T_{e_2} - NOT (T_{e_1} - NOT (|c_1^{i-1} c_1^{i-1} \rangle \otimes |c_2^{i-1} c_2^{i-1} \rangle \otimes |0\rangle)) \\
= T_{e_1} - T_{e_2} - NOT \otimes T_{e_2} - NOT (|c_1^{i-1} c_1^{i-1} \rangle e_1 \otimes |c_2^{i-1} c_2^{i-1} \rangle \otimes |1\rangle) \\
+ |c_1^{i-1} c_1^{i-1} \rangle e_1 \otimes |c_2^{i-1} c_2^{i-1} \rangle \otimes |0\rangle) \\
= T_{e_1} - T_{e_2} - NOT (|c_1^{i-1} c_1^{i-1} \rangle e_1 \otimes |c_2^{i-1} c_2^{i-1} \rangle e_2 \otimes |0\rangle) \\
+ |c_1^{i-1} c_1^{i-1} \rangle e_1 \otimes |c_2^{i-1} c_2^{i-1} \rangle e_2 \otimes |1\rangle) \\
+ |c_1^{i-1} c_1^{i-1} \rangle e_2 \otimes |c_2^{i-1} c_2^{i-1} \rangle e_2 \otimes |1\rangle) \\
+ |c_1^{i-1} c_1^{i-1} \rangle e_2 \otimes |c_2^{i-1} c_2^{i-1} \rangle e_2 \otimes |0\rangle) \\
+ |c_1^{i-1} c_1^{i-1} \rangle e_2 \otimes |c_2^{i-1} c_2^{i-1} \rangle e_2 \otimes |1\rangle) \\
+ |c_1^{i-1} c_1^{i-1} \rangle e_2 \otimes |c_2^{i-1} c_2^{i-1} \rangle e_2 \otimes |0\rangle) \\
+ |c_1^{i-1} c_1^{i-1} \rangle e_2 \otimes |c_2^{i-1} c_2^{i-1} \rangle e_2 \otimes |0\rangle) \\
+ |c_1^{i-1} c_1^{i-1} \rangle e_2 \otimes |c_2^{i-1} c_2^{i-1} \rangle e_2 \otimes |0\rangle) \\
(12)

If \(e_1\) and \(e_2\) are two logical operations, the quantum circuit is shown in Fig. 9b and represented as \(U_{Ol}\). The migration principle is the same as in Fig. 8 and will not be repeated.

(3) NOT

NOT does not need to be implemented with any quantum circuits. For \(!e\), not matter \(e\) is a rational operation or a logical operation, \(e\) divides \(|s\rangle\) into two subsets: one satisfies \(e\) and the other does not. \(!e\) just reverses the satisfiability and does not affect the division of the two subsets. Therefore, there is no need for quantum circuits to change the division of the subsets or to divide the subsets further.
3.5 Divide the test case space

Programs often have complex $e$ or the branch statements are nested. Therefore, multiple quantum operations are needed to be connected to continuously divide the test case space.

Define

$$U_2 = U^\otimes k \quad (13)$$

where $U \in \{ U_r, U_{Ar}, U_{Al}, U_{Or}, U_{Ol} \}$ and $k$ is a positive integer. Act $U_2$ on $|q\rangle_1$:

$$|q\rangle_2 = U_2(|q\rangle_1) = U^\otimes k (|s\rangle \otimes |0\rangle^m) = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |s_i\rangle \otimes |c_i\rangle \quad (14)$$

According to the definitions of $U_r, U_{Ar}, U_{Al}, U_{Or}, U_{Ol}$ in Sects 3.3 and 3.4, the qubits $|0\rangle^m$ in $|q\rangle_1$ are gradually modified based on the relational and the logical operators in the program to be tested. Eventually, through the entanglement of $|s\rangle$ and $|c\rangle$, the test case space is divided into multiple subsets. The values belonging to the same subset are test cases that can cover the same branch.

3.6 Complexity analysis

QSE derives test cases by dividing the space. The number of subspace divisions equals the number of conditional statements. Suppose a program has $n$ conditional statements, then QSE needs $n$ subspace divisions. The division of space is achieved through quantum comparators and multi-controlled-NOT gates, which are relatively simple and basic operations. Hence, the complexity of QSE is $O(n)$.

CSE first finds out all the path constraints and then uses a constraint solver to solve a equation for each path constraint. Suppose each conditional statement has 2 branches, there are $n+1$ path constraints for a program with $n$ conditional statements. Moreover, as we know, the complexity of solving a linear equation with $m$ variables is $O(m^3)$. Hence, the complexity of CSE is $O(nm^3)$.

Obviously, the complexity of QSE is less than that of CSE.

Note that complexity is related to granularity. In general, the algorithm complexity in classical computation is coarse-grained logical complexity. It does not care how many physical gates are used at each step of the physical implementation. As an example, a simple program is shown in Fig. 10. Because this program has only one loop, its complexity is $O(s)$. However, if we consider the issue from the point of view of physical elements, the complexity of multiplication is not low. If both $a$ and $b$ are integers and each stored in $t$ bits, then the complexity of the multiplication $c = a \times b$ is $O(t^2)$. Nonetheless, the complexity of this program is generally considered to be $O(s)$, instead of $O(st^2)$. 
To fairly compare the complexity of QSE and CSE, the complexities given above are both coarse-grained logical complexity. The complexity of CSE, $O(nm^3)$, does not care how many physical gates are used. By the same token, we treat the quantum comparator as a module and no longer consider the internal structure of it. Hence, the complexity of QSE is $O(n)$.

4 Experiments

4.1 An example

4.1.1 The division of the test space

The program shown in Fig. 1 is used as an example to further illustrate how QSE works. There are 3 branch statements in the program. Coupled with the process of preparing the test case space, the quantum circuit consists of 4 parts as shown in Fig. 11.

(1) Prepare the test case space

3 and 2 qubits are used to represent variables $x$ and $y$, respectively. Hence, 5 $H$ quantum gates transform the initial state $|0\rangle^5$ to state $|s_x\rangle \otimes |s_y\rangle$, i.e.,

$$H^\otimes 5 (|0\rangle^5) = (H |0\rangle^3 \otimes (H |0\rangle)^2$$

$$= \frac{1}{\sqrt{2}} \sum_{i=0}^{7} |i\rangle \otimes \frac{1}{\sqrt{2}} \sum_{i=0}^{3} |i\rangle = |s_x\rangle \otimes |s_y\rangle$$
That is to say, \(|s_x\rangle\) stores 0 ~ 7 and \(|s_y\rangle\) stores 0 ~ 3. This is the test case space.

(2) \(x + y < 4\)?

The outermost branch statement is to determine whether \(x + y\) is less than 4. The quantum adder “A” is used to get the sum of \(x\) and \(y\). We add a \(|0\rangle\) qubit as the highest bit of \(|s_y\rangle\) to make \(|s_y\rangle\) and \(|s_x\rangle\) both have 3 qubits. The quantum comparator “C” is used to compare \(|s_x + s_y\rangle\) and \(|4\rangle\), and the output is \(|c^1 c^0\rangle\). If \(x + y < 4\), \(|c^1 c^0\rangle = |01\rangle\); otherwise, \(|c^1 c^0\rangle = |\ast 0\rangle\). The whole process can be described with the following equation.

\[
(C \otimes A)(|s_x\rangle |s_y\rangle \otimes |0\rangle|4\rangle|0\rangle|0\rangle) = C(A(|s_x\rangle |s_y\rangle|0\rangle) \otimes |4\rangle|0\rangle|0\rangle)
\]

\[
= C((|0\rangle |0\rangle + |0\rangle |1\rangle |1\rangle + |0\rangle |2\rangle |2\rangle + |0\rangle |3\rangle |3\rangle)
+ |1\rangle |0\rangle |1\rangle + |1\rangle |1\rangle |2\rangle + |1\rangle |2\rangle |3\rangle + |1\rangle |3\rangle |4\rangle
+ |2\rangle |0\rangle |2\rangle + |2\rangle |1\rangle |3\rangle + |2\rangle |2\rangle |4\rangle + |2\rangle |3\rangle |5\rangle
+ |3\rangle |0\rangle |3\rangle + |3\rangle |1\rangle |4\rangle + |3\rangle |2\rangle |5\rangle + |3\rangle |3\rangle |6\rangle
+ |4\rangle |0\rangle |4\rangle + |4\rangle |1\rangle |5\rangle + |4\rangle |2\rangle |6\rangle + |4\rangle |3\rangle |7\rangle
+ |5\rangle |0\rangle |5\rangle + |5\rangle |1\rangle |6\rangle + |5\rangle |2\rangle |7\rangle + |5\rangle |3\rangle |8\rangle
+ |6\rangle |0\rangle |6\rangle + |6\rangle |1\rangle |7\rangle + |6\rangle |2\rangle |8\rangle + |6\rangle |3\rangle |9\rangle
+ |7\rangle |0\rangle |7\rangle + |7\rangle |1\rangle |8\rangle + |7\rangle |2\rangle |9\rangle + |7\rangle |3\rangle |10\rangle) \otimes |4\rangle|0\rangle|0\rangle)
\]

(3) \(x > y\)?
If \( x + y < 4 \), it needs to be further judged whether \( x \) is greater than \( y \). Hence, a \( T_{<} \)-C module acts on the subspace \( |s_x \rangle |s_y \rangle \otimes |c^3c^2c^1c^0 \rangle \).

\[
T_{<} \cdot C(\langle 0|0001|0\rangle + \langle 0|0101|0\rangle + \langle 0|1001|0\rangle + \langle 0|1101|0\rangle) = |0000\rangle
\]

If and only if \( |c^1c^0 \rangle = |01 \rangle \), \( |s_x \rangle \) and \( |s_y \rangle \) need to be compared, i.e., \( |c^3c^2 \rangle \) is changed according to \( |s_x \rangle \) and \( |s_y \rangle \): if \( |s_x \rangle > |s_y \rangle \), \( |c^3c^2 \rangle = |10 \rangle \); otherwise, \( |c^3c^2 \rangle = |00 \rangle \). As long as \( |c^1c^0 \rangle \neq |01 \rangle \), \( |c^3c^2 \rangle \) remains unchanged at state \( |00 \rangle \).

(4) \( y > 1 \)?

If \( x + y \geq 4 \), it needs to be further judged whether \( y \) is greater than \( 1 \). Hence, a \( T_{\geq} \)-C module acts on the subspace \( |s_y \rangle |1 \rangle \otimes |c^3c^2c^1c^0 \rangle \).

\[
T_{\geq} \cdot C(\langle 0|10001|1\rangle + \langle 1|0001|1\rangle + \langle 2|10001|1\rangle + \langle 3|10001|1\rangle)
\]

Springer
If and only if $|c^0⟩ = |0⟩$, $|s_y⟩$ and $|1⟩$ need to be compared, i.e., $|c^3c^2⟩$ is changed according to $|s_y⟩$ and $|1⟩$: if $|s_y⟩ > |1⟩$, $|c^3c^2⟩ = |10⟩$; otherwise, $|c^3c^2⟩ = |0∗⟩$. As long as $|c^0⟩ \neq |0⟩$, $|c^3c^2⟩$ remains unchanged.

Finally, the state of the subspace $|s_x⟩|s_y⟩ \otimes |c^3c^2c^1c^0⟩$ is

\[
|0⟩|0⟩|0001⟩ + |0⟩|1⟩|0101⟩ + |0⟩|2⟩|0101⟩ + |0⟩|3⟩|0101⟩ \\
+ |1⟩|0⟩|1001⟩ + |1⟩|1⟩|0001⟩ + |1⟩|2⟩|0101⟩ + |1⟩|3⟩|1000⟩ \\
+ |2⟩|0⟩|1001⟩ + |2⟩|1⟩|1001⟩ + |2⟩|2⟩|1000⟩ + |2⟩|3⟩|1010⟩ \\
+ |3⟩|0⟩|1001⟩ + |3⟩|1⟩|0000⟩ + |3⟩|2⟩|1010⟩ + |3⟩|3⟩|1010⟩ \\
+ |4⟩|0⟩|1000⟩ + |4⟩|1⟩|0100⟩ + |4⟩|2⟩|1010⟩ + |4⟩|3⟩|1010⟩ \\
+ |5⟩|0⟩|0110⟩ + |5⟩|1⟩|1010⟩ + |5⟩|2⟩|1010⟩ + |5⟩|3⟩|1010⟩ \\
+ |6⟩|0⟩|0110⟩ + |6⟩|1⟩|1010⟩ + |6⟩|2⟩|1010⟩ + |6⟩|3⟩|1010⟩ \\
+ |7⟩|0⟩|1100⟩ + |7⟩|1⟩|0100⟩ + |7⟩|2⟩|1010⟩ + |7⟩|3⟩|1010⟩
\]

There are 4 cases of the state $|c^3c^2c^1c^0⟩$:

- $|001⟩$: $|c^1c^0⟩ = |01⟩$ indicates $x + y < 4$ and $|c^3c^2⟩ = |10⟩$ indicates $x > y$. Hence, $|001⟩$ indicates $x + y < 4$ && $x > y$, which corresponds to $PC_{11}$ in classical symbolic execution.
- $|0∗0⟩$: $|c^1c^0⟩ = |01⟩$ indicates $x + y < 4$ and $|c^3c^2⟩ = |0∗⟩$ indicates $x ≤ y$. Hence, $|0∗0⟩$ indicates $x + y < 4$ && $x ≤ y$, which corresponds to $PC_{12}$ in classical symbolic execution.
- $|10∗⟩$: $|c^1c^0⟩ = |∗0⟩$ indicates $x + y ≥ 4$ and $|c^3c^2⟩ = |10⟩$ indicates $y > 1$. Hence, $|10∗⟩$ indicates $x + y ≥ 4$ && $y > 1$, which corresponds to $PC_{21}$ in classical symbolic execution.
- $|0∗∗⟩$: $|c^1c^0⟩ = |∗0⟩$ indicates $x + y ≥ 4$ and $|c^3c^2⟩ = |0∗⟩$ indicates $y ≤ 1$. Hence, $|0∗∗⟩$ indicates $x + y ≥ 4$ && $y ≤ 1$, which corresponds to $PC_{22}$ in classical symbolic execution.

These 4 states of $|c^3c^2c^1c^0⟩$ divide $|s_x⟩|s_y⟩$ into 4 subsets. As shown in Eq. 15,

- Subset $\{ |1⟩|0⟩, |2⟩|0⟩, |3⟩|0⟩, |2⟩|1⟩ \}$ contains all the test cases that can test the branch $x + y < 4$ && $x > y$.
- Subset $\{ |0⟩|0⟩, |0⟩|1⟩, |0⟩|2⟩, |0⟩|3⟩, |1⟩|1⟩, |1⟩|2⟩ \}$ contains all the test cases that can test the branch $x + y < 4$ && $x ≤ y$.
- Subset $\{ |2⟩|2⟩, |3⟩|2⟩, |4⟩|2⟩, |5⟩|2⟩, |6⟩|2⟩, |7⟩|2⟩, |1⟩|3⟩, |2⟩|3⟩, |3⟩|3⟩, |4⟩|3⟩, |5⟩|3⟩, |6⟩|3⟩, |7⟩|3⟩ \}$ contains all the test cases that can test the branch $x + y ≥ 4$ && $y > 1$.
- Subset $\{ |4⟩|0⟩, |5⟩|0⟩, |6⟩|0⟩, |7⟩|0⟩, |3⟩|1⟩, |4⟩|1⟩, |5⟩|1⟩, |6⟩|1⟩, |7⟩|1⟩ \}$ contains all the test cases that can test the branch $x + y ≥ 4$ && $y ≤ 1$.
4.1.2 Running on the `ibmq_qasm_simulator`

We use the `ibmq_qasm_simulator` quantum computer on the IBM Quantum platform to perform the example. The circuit is shown in Fig. 12. This experiment uses 28 qubits, with $q_0$ as the lowest bit and $q_{27}$ as the highest bit:

- $q_2q_1q_0$ represent $|s_x\rangle$;
- $q_5q_4q_3$ represent $|s_y\rangle$;
- $q_9q_8q_7q_6$ represent $|s_x + s_y\rangle$;
- $q_{12}q_{11}q_{10}$ are the auxiliary qubits of the quantum adder “$A$”;
- $q_{16}q_{15}q_{14}q_{13}$ are used to represent constant $|4\rangle$ and $q_{17}$ is used to represent constant $|1\rangle$;
- $q_{23}q_{22}q_{21}q_{20}q_{19}q_{18}$ are the auxiliary qubits of the quantum comparator “$C$”;
- $q_{24}q_{25}$ are the flags $|c^3c^2\rangle$ and $q_{26}q_{27}$ are the flags $|c^1c^0\rangle$.

The three purple bars in the figure are three quantum comparators. At the end of the circuit, $q_0q_1q_2q_3q_4q_5$ and $q_{24}q_{25}q_{26}q_{27}$ are measured, and they have 32 results as shown in Fig. 13. The abscissa displays all the results, and the default state of qubits that are not measured is 0. The ordinate represents the probability of each state in a total of 8192 measurements.

The 32 results can be divided into four test case spaces. Figure 14a gives the measurement results whose $|c_3c_2c_1c_0\rangle = |1001\rangle$, i.e., $x+y < 4$ 
$\& \& x > y$. Figure 14b gives the measurement results whose $|c_3c_2c_1c_0\rangle = |0*01\rangle$, i.e., $x+y < 4$ 
$\& \& x \leq y$. Figure 14c gives the measurement results whose $|c_3c_2c_1c_0\rangle = |10*0\rangle$, i.e., $x+y \geq$
4.1.3 The number of test cases

CSE uses a constraint solver to generate the test cases. Each time, the constraint solver receives only one PC and outputs one test case. As shown in Fig. 2, this example has 4 PCs. Hence, the constraint solver is used 4 times, and only one test case under a PC is obtained each time. QSE is more efficient than CSE. QSE gets all the test cases in the space by dividing the space.

Table 4 compares the test cases and the number of them produced by CSE and QSE. CSE uses z3 constraint solver [34, 35]. For each branch, z3 is used one time and only 1 test case is obtained. The test case for QSE can be derived from Fig. 14. For each branch, 4, 6, 13 and 9 test cases are obtained, respectively. For example, the first measurement in Fig. 14a is \(100000...1001\), therefore, \(x = |S_x\rangle = |001\rangle = |1\rangle\), \(y = |S_y\rangle = |000\rangle = |0\rangle\).
4.2 Experiment data

4.2.1 Complexity

Firstly, 16 real programs are used to evaluate the performance of QSE. They come from 6 references, as shown in Table 5. The “Line of code” column lists the number of source code lines in the program, excluding comments and empty lines. Figure 15 shows two of them.
We compare the complexity and the time consumption of CSE and QSE. The comparison results are shown in Table 6. The main factor that affects the complexity of CSE is the number of path constraints. (Here, we see a path constraint as a one-step operation and do not consider the number of variables $m$ as shown in Sect. 3.6, because different path constraints may have different variables.) The main factor that affects the complexity of QSE is the number of subspace divisions. Table 6 shows that the complexity of QSE is less than that of CSE. We also compare the actual time consumption of CSE and QSE. The tool to realize CSE is z3. In most cases, the time consumption of QSE is also smaller than that of CSE.

Figure 16 gives the execution tree of the two programs shown in Fig. 15, where diamonds indicate subspace divisions and circles indicate path constraints.

- Program “dart” consists of two levels of conditional statements. Although the inner conditional statement has only an “if” branch, there is also an implicit “else”
Table 6  The comparison of complexity and time consumption of CSE and QSE

|       | CSE Number of path constraints | Time/s | QSE Number of subspace divisions | Time/s |
|-------|--------------------------------|--------|----------------------------------|--------|
| dart  | 4                              | 0.48   | 3                                | 0.45   |
| power | 11                             | 1.32   | 7                                | 1.05   |
| stat  | 3                              | 0.36   | 2                                | 0.3    |
| tcas  | 5                              | 0.6    | 4                                | 0.6    |
| early | 2                              | 0.24   | 1                                | 0.15   |
| basic00181 | 3                      | 0.36   | 2                                | 0.3    |
| snp3-ok | 1                           | 0.12   | 1                                | 0.15   |
| CWE789| 6                              | 0.72   | 3                                | 0.45   |
| trimV | 8                              | 0.96   | 4                                | 0.6    |
| getSq | 4                              | 0.48   | 2                                | 0.3    |
| editor| 7                              | 0.84   | 3                                | 0.45   |
| mooc  | 6                              | 0.72   | 6                                | 0.9    |
| rofier| 3                              | 0.36   | 2                                | 0.3    |
| arr   | 20                             | 2.4    | 16                               | 2.4    |
| fun   | 4                              | 0.48   | 3                                | 0.45   |
| euler | 22                             | 2.64   | 11                               | 1.65   |

Fig. 16  The execution tree of the two programs

branch: do nothing when the condition is not met. Hence, Program “dart” has 4 path constraints and 3 subspace divisions.

• Program “power” consists of three levels of conditional statements. The middle conditional statement has two “if-else” branches, which means that the space needs to be divided twice. Plus the implied “else” of the outer conditional statement, the root node has three branches. The same is true of the four “if” in the inner conditional statement, each of which divides the space once. Hence, Program “power” has 11 path constraints and 7 subspace divisions.

Then, 266 programs that have conditional statements are selected from the dataset in Ref. [37] to carry out the time-consuming comparative experiment of CSE and QSE. The results are shown in Fig 17. The horizontal axis is the program number, the
Fig. 17 Comparison of time consumption between QSE and CSE

vertical axis is the time, blue is CSE, and orange is QSE. For 97% of the programs, QSE is faster than CSE, with an average improvement of 0.113s.

4.2.2 Branch coverage rate

We also show the effect of test case space on program branch coverage. The factor that affects the size of the test case space is the number of qubits used to represent the quantum variables. In the example given in Sect. 4.1.2, the variables $|S_x\rangle$ use 3 qubits and $|S_y\rangle$ uses 2 qubits. In fact, using more or less qubits can affect the performance of QSE.

- Too few qubits to represent variables may make it impossible for QSE to cover all branches. For example, if only one qubit is used to represent a quantum variable, the test case space has at most two test cases: $|0\rangle$ and $|1\rangle$. If the program has four branches, two test cases cannot cover all the branches. This is not due to the path constraints of the program, but because the test case space is too small.

- Isn’t the more qubits used, the better? No. Too many qubits will increase the difficulty of QSE and lead to the waste of quantum resources.

Therefore, the smallest number of qubits that can cover all branches is the best choice. Figure 18 shows the relationship between the number of qubits used by variables in the three programs in Table 5 and the program branch coverage rate. For Program “dart”, if all the variables $x$ and $y$ use total 1 to 3 qubits, the branch coverage rate is only 50%; if 4 qubits are used, the branch coverage rate is increased to 75%; if 5 or more qubits are used, the branch coverage rate is 100%. Hence, the best numbers of qubits for Program “dart” is 5. For the other two problems, the best numbers of qubits are 2 and 4 respectively.

5 Discussion

In the previous description, QSE’s function was described as generating test cases to help find bugs in quantum programs. In fact, QSE can test not only quantum programs, but also classical programs.

In the QSE application scenario, there are two programs: a program that to be tested (called the tested program) and a QSE. The tested program is quantum or classical,
and the QSE is quantum. For example, all the tested programs in Sect. 4 are all classical. QSE partitions the test case space into subspaces according to the conditional statements in the tested program, regardless of the form of the tested program. Whether the tested program is quantum or classical, as long as it is a program, it can express conditional statements. Then, QSE can divide the test case space according to these conditional statements.

If the tested program is classical, test cases can be obtained by measurement; if the tested program is quantum, test cases are measured or maintained in quantum states depending on the needs of the application.

Figure 19 gives a quantum tested program where $U |0\rangle^{\otimes n} = |a\rangle$ and $V |0\rangle^{\otimes n} = |b\rangle$. Hence, Fig. 19 realizes a function $f$:

$$f(x) = \begin{cases} 
a, & \text{if } x = 0 \\
b, & \text{if } x = 1 
\end{cases}$$

(16)

The tested program has two branches: $x = 0$ and $x = 1$. The corresponding QSE is shown in Fig. 20. Firstly, one $H$ quantum gate is used to prepare the test space $|x\rangle$, which include two values: 0 and 1.

$$H(|0\rangle) = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) = |x\rangle$$
Then, a quantum comparator is used to compare $|x\rangle$ and $|0\rangle$. If $x == 0$, $|c^1c^0\rangle = |00\rangle$; else $|c^1c^0\rangle = |10\rangle$.

$$C\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|00\rangle\right) = \frac{1}{2}C((|00\rangle + |10\rangle)|00\rangle) = C\left(\frac{1}{\sqrt{2}}(|00\rangle|00\rangle + |10\rangle|10\rangle\right)$$

In this example, $|c^1\rangle$ is the flag to divide the test space. Since this example is simple, it can be simplified to the right figure in Fig. 20. Eventually, QSE gets

$$|x\rangle|c^1\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

The $x$ corresponding to $c^1 = 0$ covers one branch of the function $f$, and the $x$ corresponding to $c^1 = 1$ covers the other branch of the function $f$.

6 Conclusion

This paper proposes a quantum symbolic execution for the first time to generate high-coverage test cases. It is completely different from not only classical symbolic executions, but also quantum debugging schemes. QSE divides the test case space into subsets according to the conditional statements in the debugged program, and a subset contains all test cases that can test the same program branch. QSE not only provides a possible way to debug quantum programs, but also avoids the difficult problem of solving constraints in classical symbolic execution, which obviously reduces the difficulty and improves the efficiency of the work.

Funding  This work is supported by the National Natural Science Foundation of China under Grants No. 61502016, Beijing Key Laboratory of Security and Privacy in Intelligent Transportation under Grants No. 202209300499.

Data Availability  All data generated or analyzed during this study are included in this article.

Declarations

Conflict of interest  The authors of this article declare that there are no potential conflicts of interest effecting the production of this work. In addition, the authors attest that this is an original article and has not been published or under peer-review elsewhere.
References

1. Nielsen, M.A., Chuang, I.: Quantum Computation and Quantum Information. American Association of Physics Teachers (2002)
2. Jiang, N., Liang, X., Wang, M.: Programmable quantum processor implemented with superconducting circuit. Commun. Theor. Phys. 73(5), 055102 (2021)
3. Zhong, H., Wang, H., Deng, Y., Chen, M., Peng, L., Luo, Y., Qin, J., Wu, D., Ding, X., Hu, Y.: Quantum computational advantage using photons. Science 370(6523), 1460–1463 (2020)
4. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J.C., Barends, R., Biswas, R., Boixo, S., Brandao, F.G., Buell, D.A.: Quantum supremacy using a programmable superconducting processor. Nature 574(7779), 505–510 (2019)
5. Broughton, M., Verdon, G., McCourt, T., Martinez, A.J., Yoo, J.H., Isakov, S.V., Massey, P., Halavati, R., Niu, M.Y., Zlokapa, A., et al.: Tensorflow quantum: a software framework for quantum machine learning. arXiv preprint arXiv:2003.02989 (2020)
6. Cross, A.: The ibm q experience and qiskit open-source quantum computing software. In: APS March Meeting Abstracts, vol. 2018, 58–003 (2018)
7. Paolini, L., Piccolo, M., Zorzi, M.: Qpcf: higher-order languages and quantum circuits. J. Autom. Reason. 63(4), 941–966 (2019)
8. Selinger, P.: Towards a quantum programming language. Math. Struct. Comput. Sci. 14(4), 527–586 (2004)
9. Adedoyin, A., Ambrosiano, J., Anisimov, P., Bärtschi, A., Casper, W., Chennupati, G., Coffrin, C., Djidjev, H., Gunter, D., Karr, S., et al.: Quantum algorithm implementations for beginners. arXiv preprint arXiv:1804.03719 (2018)
10. He, C., Li, J., Liu, W.: An exact quantum principal component analysis algorithm based on quantum singular value threshold. arXiv preprint arXiv:2010.00831 (2020)
11. O’Brien, T.E., Tarasinski, B., Terhal, B.M.: Quantum phase estimation of multiple eigenvalues for small-scale (noisy) experiments. New J. Phys. 21(2), 023022 (2019)
12. Paltenghi, M., Pradel, M.: Bugs in quantum computing platforms: an empirical study. arXiv preprint arXiv:2102.14560 (2021)
13. Wang, J., Gao, M., Jiang, Y., Lou, J., Gao, Y., Zhang, D., Sun, J.: Quanfuzz: fuzz testing of quantum program. arXiv preprint arXiv:1810.10310 (2018)
14. Miranskyy, A., Zhang, L., Doliskani, J.: Is your quantum program bug-free? arXiv preprint arXiv:2001.10870 (2020)
15. Zhao, P., Zhao, J., Ma, L.: Identifying bug patterns in quantum programs. arXiv preprint arXiv:2103.09069 (2021)
16. Huang, Y., Martonosi, M.: Statistical assertions for validating patterns and finding bugs in quantum programs. In: Proceedings of the 46th International Symposium on Computer Architecture, 541–553 (2019)
17. JavadiAbhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F.T., Martonosi, M.: Scaffcc: scalable compilation and analysis of quantum programs. Parallel Comput. 45, 2–17 (2015)
18. Bright, P.: Microsoft’s q# quantum programming language out now in preview. Ars Technica, 11 (2017)
19. Liu, J., Byrd, G.T., Zhou, H.: Quantum circuits for dynamic runtime assertions in quantum computation. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 1017–1030 (2020)
20. Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y.: Proq: Projection-based runtime assertions for debugging on a quantum computer. arXiv preprint arXiv:1911.12855 (2019)
21. Liu, J., Zhou, H.: Systematic approaches for precise and approximate quantum state runtime assertion. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 179–193 (2021). IEEE
22. Ali, S., Arcaini, P., Wang, X., Yue, T.: 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), 13–23 (2021). IEEE
23. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
24. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 82–90 (2013)
25. Zhao, W., Zhang, W., Wang, J., Wang, H., Wu, C.: Smart contract vulnerability detection scheme based on symbol execution. J. Comput. Appl. 40(4), 947–953
26. Yang, C., Guo, Y., Hu, H., Liu, W., Huo, S., Wang, Y.: Cache-based side-channel vulnerability detection based on symbolic execution. Acta Electron. Sin. 47(6), 1194 (2019)
27. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: Cached: Identifying cache-based timing channels in production software. In: 26th [USENIX] Security Symposium ([USENIX] Security 17), pp. 235–252 (2017)
28. Chang, L., Zhu, Y., Jiang, H.: Design of quantum full adder. Acta Electron. Sin. 47(9), 1863 (2019)
29. Yuan, S., Wang, Y., Wang, Y., Huang, F.: Quantum multiplier and its implementation method. J. Chongqing Univ. Posts Telecommun. (Natural Science Edition) (2019)
30. Wang, D., Liu, Z., Zhu, W., Li, S.: Design of quantum comparator based on extended general toffoli gates with multiple targets. Comput. Sci. 39(9), 302–306 (2012)
31. Prata, S.: C Primer Plus (2014)
32. Bruce, E.: Thinking in Java (Fourth Edition) (2006)
33. Eric, M.: Python Crash Course: A Hands-On, Project-Based Introduction to Programming (First Edition) (2015)
34. Jovanović, D., Moura, L.D.: Solving non-linear arithmetic (2012)
35. Mues, M., Howar, F.: Jdart: Portfolio solving, breadth-first search and smt-lib strings (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27, 448–452 (2021). Springer
36. Dinges, P., Agha, G.: Solving complex path conditions through heuristic search on induced polytopes (2014)
37. LeClair, A., McMillan, C.: Recommendations for datasets for source code summarization. arXiv preprint arXiv:1904.02660 (2019)
38. Minizinc (2018) https://www.minizinc.org/
39. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: a deep learning-based system for vulnerability detection (2018)
40. Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: Jdart: a dynamic symbolic analysis framework. In: Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 22, 442–459 (2016). Springer
41. Scientific Computation (2018) https://github.com/elizabethzhenliu/scientificcomputation

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.