Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs

Shaukat Ali
Simula Research Laboratory
Fornebu, Norway
shaukat@simula.no

Paolo Arcaini
National Institute of Informatics
Tokyo, Japan
arcaini@nii.ac.jp

Xinyi Wang
Nanjing University of Aeronautics and Astronautics
Nanjing, China
wangxinyi125@nuaa.edu.cn

Tao Yue
Nanjing University of Aeronautics and Astronautics, China
Simula Research Laboratory, Norway
taoyue@ieee.org

***Abstract*—Quantum programs implement quantum algorithms solving complex computational problems. Testing such programs is challenging due to the inherent characteristics of Quantum Computing (QC), such as the probabilistic nature and computations in superposition. However, automated and systematic testing is needed to ensure the correct behavior of quantum programs. To this end, we present an approach called *Quito* (QUantum InpuT Output coverage) consisting of three coverage criteria defined on the inputs and outputs of a quantum program, together with their test generation strategies. Moreover, we define two types of test oracles, together with a procedure to determine the passing and failing of test suites with statistical analyses. To evaluate the cost-effectiveness of the three coverage criteria, we conducted experiments with five quantum programs. We used mutation analysis to determine the coverage criteria' effectiveness and cost in terms of the number of test cases. Based on the results of mutation analysis, we also identified equivalent mutants for quantum programs.**

***Index Terms*—quantum programs, software testing, coverage criteria, mutation analysis**

I. Introduction

Quantum computing (QC) has the potential to solve profoundly complex computational problems of societal and industrial nature. Researchers and industry are increasingly recognizing such QC potential as evident with the increased number of QC platforms. Notable QC platforms include Microsoft Quantum computing platform[1], IBM Quantum Experience[2], and D-Wave Leap™ platform[3]. Several quantum programming languages also exist to enable programming quantum computers such as Microsoft's Q#, and IBM's Qiskit [1].

Building QC applications relies on implementing quantum algorithms as quantum programs in quantum programming languages. To ensure the correct implementation of a quantum program, automated and systematic testing is a natural choice. However, testing quantum programs is challenging for several reasons, such as the QC's probabilistic nature and difficulties in estimating a quantum program's state in superposition [2].

Researchers have identified testing challenges of quantum programs, and some testing methods have started to emerge as surveyed in [3]. The notable recent work of Honarvar et al. [4], named as QSharpCheck, is closely related to the work we present in this paper. QSharpCheck is a property-based testing approach for testing quantum programs written in Q#, which includes the definition of a property specification language and a testing method to generate and execute test cases.

Following the direction of the work by Honarvar et. al. [4], we define three coverage criteria on inputs and outputs of a quantum program. Defining such coverage criteria is one of the key contributions of this paper as compared to the work by Honarvar et. al. [4]. In our approach, each coverage criterion is also complemented with a test generation and execution algorithm to test a quantum program. We call our approach *Quito* (QUantum InpuT Output coverage). Moreover, Quito defines two types of test oracles and a procedure to determine the passing and failing of a test suite based on statistical analyses to deal with the quantum programs' probabilistic nature.

To assess the cost-effectiveness of the coverage criteria, we experimented with five quantum programs. We used mutation analysis for quantum programs to measure the effectiveness of a test suite corresponding to a coverage criterion. Given the differences in mutation analysis for quantum programs compared to classical programs, we also highlighted their differences followed by performing mutation analysis for the two types of test oracles. Finally, as the result of test execution, we also identified equivalent mutants for quantum programs.

The paper is organized as follows: We compare our work with related works in Section II. Section III presents the necessary background and a running example. Section IV defines test oracle types, coverage criteria, and test generation algorithms. Section V presents mutation analysis to assess

This work is supported by the Qu-Test project (Project #299827) funded by Research Council of Norway. Paolo Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST; Funding Reference number: 10.13039/501100009024 ERATO.

[1]https://www.microsoft.com/en-us/quantum/

[2]https://quantum-computing.ibm.com

[3]https://www.dwavesys.com/take-leap

the criteria' effectiveness. Section VI presents the design of experiments, whereas results and discussion are presented in Section VII. Finally, we conclude the paper in Section VIII.

II. Related Work

The potential of QC to solve complex problems and bring scientific and technological breakthroughs has been well recognized [5], [6]. Researchers have started to investigate the feasibility of applying QC for addressing challenges on software/system verification and validation. A research group from the University of Technology Sydney [7] is conducting research on modeling and verifying quantum programs in terms of their correctness, with the focus on developing model checkers for quantum Markov chains, etc. Differently from the above studies, we focus on testing quantum programs written in high-level quantum programming languages (e.g., Q#), rather than verifying them with formal methods.

Most relevantly, Wang et al. [8] proposed *QuanFuzz*, a search-based test input generator for quantum software. QuanFuzz first statically extracts quantum-sensitive information, such as quantum register measurement and sensitive branches, from a quantum program. Then, it relies on a grey-box fuzz testing model to generate inputs to change the state of quantum registers and maximize the coverage. QuanFuzz was evaluated with seven benchmark quantum programs and compared with a random generator. Results show that QuanFuzz is more effective in terms of triggering sensitive branches than the random generator and traditional branch coverage guided search-based test case generation techniques.

A comprehensive survey on quantum software engineering, with a particular focus on the whole life cycle of quantum software development [3], has been recently conducted. Results show that few researchers have already initiated the effort of identifying bug types (e.g., incorrect quantum initial values, incorrect operations, and transformations) [9], and defining and specifying assertions in quantum software as invariants [10], pre- and post-conditions [11], or constructing assertion operations (e.g., *AssertProbability* [4]).

QSharpCheck [4] was recently proposed by Honarvar et. al. for testing quantum software in Q#. As part of QSharpCheck, the authors proposed a test property specification language for Q# programs, and a property-based testing method for generating and executing test cases, and analyzing test results. A test property specification contains the information of number of test cases to generate (10 by default), predefined confidence level (0.99 by default), number of measurements (350 by default), and number of experiments (300 by default). The confidence level is needed for the statistical method embedded in QSharpCheck as the basis to perform statistical test on obtained test data. QSharpCheck is also equipped with five assertion types, such as *AssertProbability* and *AssertEntangled*, some of which are program specific such as *AssertTeleported* specific to quantum teleportation. The effectiveness of QSharpCheck was evaluated with two examples via mutation analysis and results show that QSharpCheck achieved 80% and 60% of mutation scores for the two examples.

```
qc.reset(2); //Reserve two qubits

// Creating two qubits
var qubit1 = qint.new(1, 'Qubit1');
var qubit2 = qint.new(1, 'Qubit2');

qc.write(0); //Initializing to 0

qc.label("Entangle Qubits");
qubit1.had(); // Put qubit1 in superposition
qubit2.cnot(qubit1); // Entangle the two qubits
qc.label();

qc.nop();
var result = qc.read(); // Read both qubits

qc.print(result); //Output qubits values
```

Listing 1: Running example – Javascript code

As compared to QuanFuzz and QSharpCheck, our key contributions are: 1) We initiate the definition of three testing coverage criteria, which are independent of specific quantum programming languages and applications; 2) For each defined testing coverage criterion, we propose an algorithm to generate a test suite for a quantum program, to reach the full coverage, during the test execution; and 3) We define two test oracles independent of any quantum programming language and application. Based on the test oracles, we assess test execution results automatically with the help of statistical tests such as the Wilcoxon test.

III. Background and Running Example

A quantum program works on quantum bits, or *qubits* in short, as opposed to bits in the context of classical computers. A classical bit takes either value 0 or 1. For example, all the possible states of two classical bits are 00, 10, 01 and 11. In contrast, a quantum state can be defined based on two elements; (1) a value, which is one of the permutations of quantum bits values, i.e., same as classical bits. For example, for two quantum bits, we have four possible values; (2) an *amplitude* (α)—a complex number. For example, two qubits can be represented in the bra-ket/Dirac notation [12] (a commonly used notation for specifying quantum states [13]) as:

$$\alpha_0 |00\rangle + \alpha_1 |10\rangle + \alpha_2 |01\rangle + \alpha_3 |11\rangle$$

Here, α_0, α_1, α_2, and α_3 are the amplitudes associated with each of the states. The square of the absolute value of amplitudes (e.g., $|\alpha_0|^2$) determines the probability of a quantum program being in this state, if the qubits are read. Also, the follow relationship holds:

$$|\alpha_0|^2 + |\alpha_1|^2 + |\alpha_2|^2 + |\alpha_3|^2 = 1$$

An example of quantum program with two qubits is shown in Listing 1. This program is about quantum entanglement, i.e., the program entangles *Qubit1* with *Qubit2*, implying that when their values are read, they will always be the same, i.e., either both 0's or both 1's. The quantum program is written in

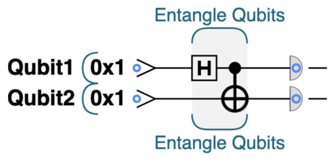
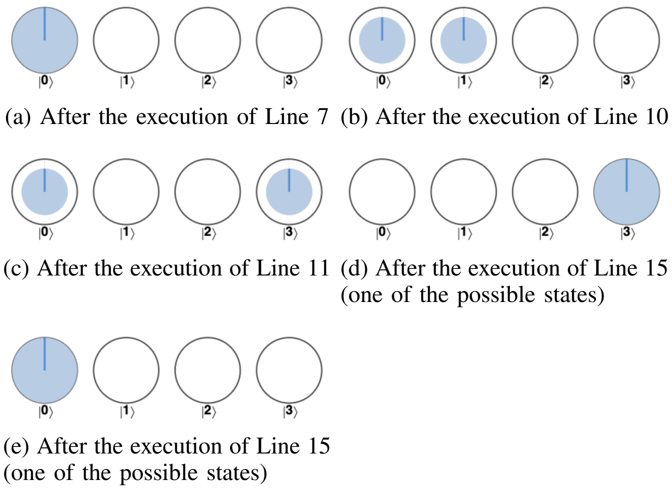


Fig. 1: Running example – Circuit diagram



(a) After the execution of Line 7 (b) After the execution of Line 10

(c) After the execution of Line 11 (d) After the execution of Line 15 (one of the possible states)

(e) After the execution of Line 15 (one of the possible states)

Fig. 2: Running example – Evolution of program states in the circle notation

the QCEngine's Javascript-based language and is an example from the book [13].

The equivalent circuit diagram of the program is shown in Figure 1, which is also taken from the book [13]. In Listing 1, Line 1 registers two qubits, Lines 4-5 create the two qubits (i.e., *Qubit1* and *Qubit2*), and Line 7 initializes the qubits to the 0 state. The state of the quantum program after the execution of Line 7 is shown in Figure 2a in the circle notation from [13]. Since there are two qubits, there are four possible values of qubits, i.e., 00, 10, 01, 11 (shown in decimal as $|0\rangle$, $|1\rangle$, $|2\rangle$, and $|3\rangle$ in Figure 2 in the bra-ket notation [13]). In the circle notation, the magnitude of a state is represented as a filled circle inside a circle indicating the probability of the program being in that particular state, if the qubits are read. At this stage, the program is in state 0, as shown in Figure 2a.

One of the key differences of quantum programs with classical programs is that at any given point of time, a quantum program can be in *superposition*. To put a quantum program in superposition, we apply the Hadamard operation [13] on quantum bits. Line 10 of the code in Listing 1 shows that we applied the Hadamard operation to qubit 1, which is also shown as an *H* gate on the circuit diagram (Figure 1). After the execution of Line 10, the program will be in a superposition state, and the state is represented in the circle notation in Figure 2b. In the figure, the 0 and 1 states have the filled circles of the same size. This means that there is an equal probability that the quantum program is in any of these two states. Another property of a quantum state is *phase*, which is shown as a line inside the circle. The phase ranges from the 0.0 to 360.0 degrees. For instance, the phase of the circle labeled with $|0\rangle$ in Figure 2b is 0 degrees. In this paper, we don't consider testing the phases of a quantum program. This will be part of our future work. Therefore, we don't discuss phases further in this paper. Interested readers may consult [13].

Line 11 in Listing 1 entangles the two qubits using the conditional not (*CNOT*) gate, shown as the gate on the qubits right after the *H* gate in the circuit diagram (Figure 1). After the execution of Line 11, the state of the program is shown in Figure 2c, indicating that the program is in the superposition of 0 and 3 with equal probability. In other words, if we read the qubits, for instance using the *read()* operation (Line 15), there is a 50% probability that we will observe 3 as output (Figure 2d) and a 50% probability that we will observe 0 as output (Figure 2e). Note that the *read()* and *write()* operations destroy superpositions. Thus, to fully utilize the power of quantum computing, most of the computation is performed when qubits are in superposition.

In this paper, we test a quantum program as a black-box, i.e, we set the initial state of the program by assigning values to registered qubits (e.g., Figure 2a). Also, we only check the state of the program at the end of its execution by reading values of qubits (e.g., Figure 2d and Figure 2e). Therefore, during a test execution, we don't interfere with any change of any state in superposition of the program (e.g., Figure 2b and Figure 2c), since doing so would destroy the superposition. However, we acknowledge that there might exist indirect ways to estimate superposition states (including magnitudes and phases) of a program without destroying them [13]. We will investigate these ways in the future work.

IV. Test Oracles and Coverage Criteria

In this section, we first provide necessary definitions (Section IV-A), based on which we then present the proposed test oracles (Section IV-B), coverage criteria (Section IV-C), and testing process (Section IV-D) of *Quito*.

A. Definitions

Definition 1 (Inputs and outputs)**.** Let $Q = \{q_1, \ldots, q_n\}$ be the set of qubits of the quantum program QP. All the qubits in Q are considered as *inputs*. Instead, only a subset of qubits O identify the *output*, i.e., $O \subseteq Q$. We define $D_I = \mathcal{B}^{|Q|}$ as the set of input values, and $D_O = \mathcal{B}^{|O|}$ as the set of output values.

In the following, we will report input and output values in their decimal representation.

Definition 2 (Quantum program)**.** A quantum program QP can then be defined as a function QP: $D_I \rightarrow 2^{D_O}$.

Note that a quantum program, given the same input value, can return different output values. Each possible output value is produced with a certain probability.

The program specification specifies preconditions on the allowed input values and the expected probabilities of occurrences of the output values, as described in Def. 3.

Definition 3 (Program specification)**.** Given a quantum program QP: $D_I \rightarrow 2^{D_O}$, we identify the expected behavior of the program in PS, i.e., the *program specification*. First of all, PS identifies a subset of input values $VD_I \subseteq D_I$ (i.e., *valid input values*) for which the program computation makes sense; therefore, the program should not be executed with invalid input values $D_I \setminus VD_I$ (it is a kind of precondition). Then, for each valid input assignment $i \in VD_I$, the program specification states the expected probabilities of occurrences of all the output values $o \in D_O$:

$$\texttt{PS}(i) = \{(o_1, p_1), \ldots, (o_{|D_O|}, p_{|D_O|})\}$$

where (o_h, p_h) indicates that value o_h can be produced with probability p_h by input value i (with $0 \leq p_h \leq 1$); it holds $\sum_{h=1}^{|D_O|} p_h = 1$. We further write $\texttt{PS}(\langle i, o_h \rangle) = p_h$ to specify the probability of occurrence of output value o_h for input value i.

The definition of program specification leads also to the definition of the valid output values that can be produced by the program.

Definition 4 (Valid output values)**.** The set of *valid output values* is the set of output values that, according to the program specification, can be produced at least by an input value. Formally:

$$VD_O = \{o \in D_O \mid (\exists i \in D_I : \texttt{PS}(\langle i, o \rangle) \neq 0)\}$$

Definition 5 (Test input, test outcome, and test suite)**.** A *test input* t is a valid assignment to the qubits, i.e., $t \in VD_I$. A *test outcome* is defined by the pair $\langle t, res \rangle$, where $res = \text{QP}(t)$ is the output value returned by the program for t in a given execution. A *test suite* TS is a set of test outcomes.[4]

Recall that QP(t) could return different values in different executions (see Def. 2). Therefore, we need test oracles and coverage criteria specific for quantum programs.

B. Test Oracles

The assessment of a test suite for quantum programs is different from a classical and deterministic program, and it must take into account the probabilistic nature of quantum programs. In a classical deterministic program, a test fails if a given assertion (during execution and/or on the output) is violated, while passes if all the assertions hold. In a quantum program, a test suite could have three outcomes:

- *definitely fail*. In this case, the outcome of the program clearly shows a failure (e.g., the program returns a value that is not a valid output value);
- *likely fail*, with a given confidence. In this case, multiple executions of the test seem to show that a desired property does not hold. For example, after executing the program *multiple* times with a valid input value i, which should produce a valid output o with a given probability p, we observe o with a probability p' that significantly deviates from p according to a statistical test. Such a statistical test can, to a certain extent, tell us the confidence that the real occurrence probability is really different from p; if we can reject the null hypothesis of the statistical test, we say that the test suite fails with a given confidence.
- *inconclusive*. In this case, multiple executions of the test do not allow to reject the null hypothesis of a statistical test. Therefore, we cannot claim anything about the failure of the test, and we say that it is inconclusive.

[4]Note that our definition of test suites is different from the classical definition, as we consider the obtained result as part of the test suite.

Following the previous considerations, we propose two test oracles (i.e., WOO and OPO) to assess test executions produced by the generation algorithms that will be described in Section IV-C. These two oracles target different types of faults: WOO checks whether a wrong output is produced, while OPO checks whether the outputs are produced with unexpected occurrence probabilities.

*1) Wrong Output Oracle (*WOO*):* The first oracle simply checks that the quantum program only produces valid output values. This oracle is applied by checking whether test outcome $res = \text{QP}(i)$ returned for test input i is invalid, i.e., the probability of occurrence according to the program specification (i.e., $\texttt{PS}(\langle i, res \rangle)$) is 0. In this case, a failure of a test is a *definitely fail*.

Example 1. Valid outputs for our running example are 0 and 3. If we observe another output, then it means that we observed a wrong output. Assume that we have the following test suite: $\{\langle 0, 0 \rangle, \langle 1, 2 \rangle\}$. We can see that the output corresponding to input 1 is 2, which is not expected, and therefore the test assessment will return $\texttt{fail}_{\texttt{WOO}}$.

*2) Output Probability Oracle (*OPO*):* This oracle checks whether the quantum program returns an expected output with its corresponding expected probability. Since the oracle needs to observe multiple executions of the same input to estimate the occurrence probability, the oracle assessment is not done on the single tests, but on a set of test suites:

$$\widetilde{TSs} = \left\{ \widetilde{TS}_1, \ldots, \widetilde{TS}_M \right\}$$

The output probability oracle is checked for each input-output pair $\langle i, o \rangle$ such that $\texttt{PS}(\langle i, o \rangle) \neq 0$. Given the test suites $\widetilde{TSs} = \{\widetilde{TS}_1, \ldots, \widetilde{TS}_M\}$, we compute the *input-output occurrence* as $\texttt{ioo}(\widetilde{TSs}, \langle i, o \rangle) = \{p_1, \ldots, p_M\}$, where each p_j identifies the percentage of times that the program produces the output o given input i in the test suite $\widetilde{TS}_j$, i.e.,

$$p_j = \frac{\{\langle x, y \rangle \in \widetilde{TS}_j \mid x = i \wedge y = o\}}{\{\langle x, y \rangle \in \widetilde{TS}_j \mid x = i\}}$$

The oracle assessment is then performed using a statistical test (e.g., the one-sample Wilcoxon signed rank test); the oracle assesses a failure if the distribution is significantly different from the expected probability, i.e.,

$$\texttt{fail}_{\texttt{OPO}}(\widetilde{TSs}, \langle i, o \rangle) = \begin{array}{l} \text{null hypothesis between} \\ \texttt{ioo}(\widetilde{TSs}, \langle i, o \rangle) \text{ and } \texttt{PS}(\langle i, o \rangle) \\ \text{is rejected} \end{array}$$

Algorithm 1 Test generation for input coverage

```
Require: the quantum program QP under test
Require: the program specification PS
1: $TSs \leftarrow \emptyset$                                  ▷ Test suites
2: for $h \in \{1, \ldots, K\}$ do
3:     $TS \leftarrow \emptyset$                               ▷ Test suite
4:     for $i \in VD_I$ do                        ▷ Iterate over input values
5:         $res \leftarrow \mathrm{QP}(i)$                     ▷ Program execution
6:         $TS \leftarrow TS \cup \{\langle i, res\rangle\}$
7:         if $\mathtt{PS}(\langle i, res\rangle) = 0$ then    ▷ The result is not expected
8:             $TSs \leftarrow TSs \cup \{TS\}$
9:             return $\langle TSs, \mathtt{fail}_{\mathtt{WOO}}\rangle$    ▷ A fault has been found
10:    $TSs \leftarrow TSs \cup \{TS\}$
11: return $\langle TSs, \mathtt{pass}_{\mathtt{WOO}}\rangle$
```

Instead, if the null hypothesis cannot be rejected, we say that the test is *inconclusive*, i.e., $\mathtt{incon}_{\mathtt{OPO}} = \neg\mathtt{fail}_{\mathtt{OPO}}$.

Example 2. For our running example, the program specification states that for each valid input value (i.e., 0 and 1), the probability of observing output value 0 is 50% and output value 3 is also 50%. Let's assume that we have 20 test suites. Then, for each input-output pair $\langle i, o\rangle$, we apply the statistical test: the input-output occurrence $\mathtt{ioo}(\widetilde{TSs}, \langle i, o\rangle)$ (composed of 20 values) is compared with $\mathtt{PS}(\langle i, o\rangle)$ (50% for all the pairs in this example). We assume the significance level of 0.01, i.e., if the computed p-value is less than or equal to 0.01, it means that the sample is significantly different than 50% and hence the test suite is failed with a confidence level of 99%. However, if the computed p-value is greater than 0.01, it means that there is no significant difference between the sample and 50%, and we cannot reject the null hypothesis.

C. Coverage Criteria

In this section, we present our coverage criteria.

1) Input Coverage (IC):

Definition 6 (Input Coverage)**.** The *input coverage* criterion requires that for each valid input value $i \in VD_I$, there exists a test $t = i$.

In this case, the criterion can be easily achieved by statically generating a test suite. The exact number of required tests is $|VD_I|$, i.e., one for each valid input. Differently from classical and deterministic programs, a single execution of a test suite is not sufficient to assess whether the test suite passes or fails for some types of oracles: this is the case of the oracle OPO (*Output Probability Oracle*) described in Section IV-B2 that checks the probability of occurrences of output values. Therefore, we generate K test suites $TSs = \{TS_1, \ldots, TS_K\}$ (being K a parameter of the input coverage criterion), all having the same test inputs, but with possibly different outputs. The complete test generation is shown in Alg. 1.

Each of the K test suites (Line 2) is generated as follows. The algorithm iterates over the valid input values (Line 4), executes the program with each input value (Line 5), and adds the test input and the test result (i.e., the *test outcome*) to the test suite (Line 6). If the returned output is not expected for the current input, the generation stops, returning the tests generated so far and signaling a failure (Lines 7-9); oracle WOO (*Wrong Output Oracle*) checks whether a wrong output has been returned (see Section IV-B1). Each generated test suite is collected in the set TSs (Line 10). If no failure occurs, at the end, the algorithm returns all the generated test suites, and a flag signaling that no failure occurred for oracle WOO related to wrong outputs (Line 11).

Algorithm 2 Test generation for output coverage

```
Require: the quantum program QP under test
Require: the program specification PS
Require: budget $\mathtt{Budget}_{\mathtt{OC}}$ on the generation of a single test suite
1: $TSs \leftarrow \emptyset$                                  ▷ Test suites
2: for $h \in \{1, \ldots, K\}$ do
3:     $TO\_COV \leftarrow VD_O$                  ▷ Set of output values to cover
4:     $TS \leftarrow \emptyset$                               ▷ Test suite
5:     while $TO\_COV \neq \emptyset \wedge \mathtt{Budget}_{\mathtt{OC}}$ not expired do
6:         for $i \in VD_I$ do                    ▷ Iterate over input values
7:             $res \leftarrow \mathrm{QP}(i)$                 ▷ Program execution
8:             $TS \leftarrow TS \cup \{\langle i, res\rangle\}$
9:             $TO\_COV \leftarrow TO\_COV \setminus \{res\}$
10:            if $\mathtt{PS}(\langle i, res\rangle) = 0$ then ▷ The result is not expected
11:                $TSs \leftarrow TSs \cup \{TS\}$
12:                return $\langle TSs, \mathtt{fail}_{\mathtt{WOO}}\rangle$  ▷ A fault has been found
13:    $TSs \leftarrow TSs \cup \{TS\}$
14: return $\langle TSs, \mathtt{pass}_{\mathtt{WOO}}\rangle$
```

Example 3. For our running example (see Section III), the set of valid input values are 0 and 1. Thus, one of the test suites in TSs generated with Alg. 1 could be $\{\langle 0, 0\rangle, \langle 1, 3\rangle\}$.

2) Output Coverage (OC):

Definition 7 (Output Coverage)**.** The *output coverage* criterion requires that each valid output value $o \in VD_O$ is observed at least once, i.e., there exists a test t whose result is $\mathrm{QP}(t) = o$.

Test suites for achieving the full output coverage cannot be generated statically. This is because, with a given input value, the program produces a specific output value o with an expected non-zero probability, so there is no guarantee to obtain o at a specific run (even on a correct program). Therefore, to generate tests for the full output coverage, we keep on generating and executing tests until all the expected valid output values in VD_O are observed at least once, or the budget allocated to the generation expires.

As for the input coverage, also for the output coverage we cannot rely on a single execution of tests to assess whether they pass or fail. Therefore, also in this case, we generate K test suites, and collect them in TSs. Differently from the input coverage, in this case, the different test suites can contain different test inputs. The whole test generation is described in Alg. 2. Since the generation for a test suite may not be able to achieve the criterion (if the program is faulty), or it could take too much time, the algorithm requires in input also a budget $\mathtt{Budget}_{\mathtt{OC}}$ (e.g., time or the maximum number of tests) for the generation of each test suite.

Each of the K test suites (Line 2) is generated as follows.

Algorithm 3 Test generation for input-output coverage

```
Require: the quantum program QP under test
Require: the program specification PS
Require: budget $\texttt{Budget}_{\texttt{IOC}}$ on the generation of a single test suite
 1: $TSs \leftarrow \emptyset$                                          ▷ Test suites
 2: $\texttt{Budget}^{\texttt{inp}}_{\texttt{IOC}} \leftarrow \texttt{Budget}_{\texttt{IOC}}/|VD_I|$
 3: for $h \in \{1,\dots,K\}$ do
 4:     $TS \leftarrow \emptyset$                                       ▷ Test suite
 5:     for $i \in VD_I$ do                                ▷ Iterate over input values
 6:         $TO\_COV \leftarrow \{o \mid \texttt{PS}(\langle i,o\rangle) \neq 0\}$      ▷ Possible outputs
 7:         while $TO\_COV \neq \emptyset \wedge \texttt{Budget}^{\texttt{inp}}_{\texttt{IOC}}$ not expired do
 8:             $res \leftarrow \text{QP}(i)$                        ▷ Program execution
 9:             $TS \leftarrow TS \cup \{\langle i, res\rangle\}$
10:             $TO\_COV \leftarrow TO\_COV \setminus \{res\}$
11:             if $\texttt{PS}(\langle i, res\rangle) = 0$ then ▷ The result is not expected
12:                 $TSs \leftarrow TSs \cup \{TS\}$
13:                 return $\langle TSs, \texttt{fail}_{\texttt{WOO}}\rangle$  ▷ A fault has been found
14:     $TSs \leftarrow TSs \cup \{TS\}$
15: return $\langle TSs, \texttt{pass}_{\texttt{WOO}}\rangle$
```

As long as a valid output value is not covered and the given budget $\texttt{Budget}_{\texttt{OC}}$ is not expired (Line 5), the algorithm keeps on iterating over the valid input values (Line 6) and, for each value i: it runs the program with i (Line 7), collects the test outcome in TS (Line 8), and marks the test result as covered (Line 9). If a wrong output is obtained, the generation stops, and returns the tests generated so far and signals a failure (Lines 10-12).

When all the valid output values are covered or the budget $\texttt{Budget}_{\texttt{OC}}$ expires, the test suite TS is added to the set of all test suites (Line 13), and the generation continues with the next test suite. In case of no failure, at the end, the set of all test suites is returned with a flag indicating that no failure for oracle $\texttt{WOO}$ occurred (Line 14).

Example 4. For our running example presented in Section III, the set of valid output values for the program are 0 and 3. Thus, to generate a test suite satisfying the output coverage, we need to create and execute test cases to observe these two valid output values. Thus, one potential test suite may be $\{\langle 0,0\rangle$, $\langle 1,0\rangle$, $\langle 0,0\rangle$, $\langle 1,3\rangle\}$.

3) Input-Output Coverage (IOC):

Definition 8 (Input-Output Coverage)**.** The *input-output coverage* criterion requires that, for each input-output pair $\langle i,o\rangle$ such that $\texttt{PS}(\langle i,o\rangle) \neq 0$, there exists a test $t = i$ whose result is $\text{QP}(t) = o$.

Same as for the output coverage, it is not possible to generate a test suite offline. We therefore propose a test generation algorithm as presented in Alg. 3. Also for this criterion, the test generation is performed K times, and there is a budget $\texttt{Budget}_{\texttt{IOC}}$ on the generation of each test suite. In order to avoid that the generation for an input value consumes the whole budget, the budget is divided equally among the different input values (Line 2). For each of the K test suites (Line 3), the generation is as follows. For each valid input value i (Line 5), the algorithm collects all the possible output values according to the specification (Line 6). Then, as long as some expected output of input i is not covered and the budget $\texttt{Budget}^{\texttt{inp}}_{\texttt{IOC}}$ is not expired (Line 7), the algorithm performs the following instructions: it runs the program with the input value i (Line 8), collects the test outcome in TS (Line 9), and removes the obtained output res from the list of the outputs yet-to-be covered (Line 10). If the returned output is not allowed, the generation stops returning the tests generated so far and signaling a failure (Lines 11-13).

When all the expected outputs of input i are covered or the budget expires, the loop terminates, and the algorithm continues the generation for the next input value. At the end of the generation for all the inputs, TS is saved in the set of test suites (Line 14).

If no failure occurs, at the end the algorithm returns all the generated test suites, and a flag signaling that no failure occurred for the oracle $\texttt{WOO}$ (Line 15).

Example 5. Our running example (Section III) has two valid input values, i.e., 0 and 1. For each valid input value, valid output values are 0 and 3. Thus, a test suite for input-output coverage will contain test cases that ensure observing both 0 and 3 output values with each valid input value. A potential test suite is: $\{\langle 0,0\rangle$, $\langle 0,0\rangle$, $\langle 0,3\rangle$, $\langle 1,3\rangle$, $\langle 1,0\rangle\}$.

D. Application of the testing process

We here explain how our testing framework must be used. Given quantum program QP having specification $\texttt{PS}$:

1) We generate a set of test suites using one of the three coverage criteria.
2) As output, we obtain a pair $\langle TSs, \texttt{res}_{\texttt{WOO}}\rangle$, where TSs is the set of generated test suites, and $\texttt{res}_{\texttt{WOO}}$ the assessment of oracle $\texttt{WOO}$, either $\texttt{pass}_{\texttt{WOO}}$ or $\texttt{fail}_{\texttt{WOO}}$. If the oracle assessment is $\texttt{fail}_{\texttt{WOO}}$, we obtain a failure and the evaluation terminates.
3) Otherwise, we check oracle $\texttt{OPO}$ using TSs and $\texttt{PS}$, as described in Section IV-B2. To perform the evaluation, we need to have *sufficiently large* test suites. Since the test suites in $TSs = \{TS_1, \dots, TS_K\}$ may be small, we merge some of them as follows. We identify a constant M such that $K \mod M = 0$; then, we merge groups of $R = \frac{K}{M}$ test suites at a time, i.e., $\widetilde{TSs} = \left\{\widetilde{TS}_1, \dots, \widetilde{TS}_M\right\}$, with $\widetilde{TS}_j = \bigcup_{h \in \{1+(j-1)\cdot R, \dots, j\cdot R\}} TS_h$ $(j = 1, \dots, M)$. So the statistical test is applied to $\widetilde{TSs}$. The outcome of the oracle assessment is either $\texttt{incon}_{\texttt{OPO}}$ or $\texttt{fail}_{\texttt{OPO}}$.

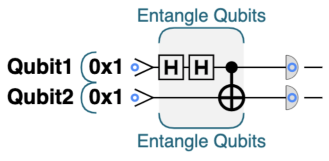
Therefore, the whole testing process can be described as:

$$\texttt{TP}_C(\text{QP}, \texttt{PS}) = \langle TSs, \texttt{res}\rangle$$

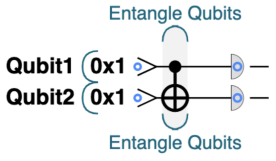
where $C \in \{\text{IC, OC, IOC}\}$ identifies the criterion, and $\texttt{res} \in \{\texttt{fail}_{\texttt{WOO}}, \texttt{incon}_{\texttt{OPO}}, \texttt{fail}_{\texttt{OPO}}\}$ is the assessment obtained by the evaluation of the oracles.

V. Assessment of the proposed coverage criteria through mutation analysis

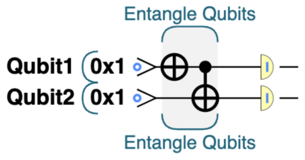
To assess the coverage criteria' effectiveness in terms of finding faults, we perform mutation analysis, which typically



(a) Add Gate (*AG*)



(b) Delete Gate (*DG*)



(c) Replace Gate (*RG*)

Fig. 3: Examples of mutations on the running example

generates a test suite with a particular test coverage criterion followed by executing the generated test suite on mutated programs to see if seeded faults can be found. Mutation analysis for evaluating the effectiveness of testing techniques for quantum programs is not entirely new, and it was also performed in a recent paper [4]. The process of mutation analysis is similar to the classical computing [14]; however, there are two key differences, because executing the same test suite more than once even on the original quantum program might produce different results due to QC's probabilistic nature. We highlight the two differences below: (1) For Input Coverage, test cases in a test suite for a mutant remain the same as for the original program. This is the same as in the classical mutation analysis. However, since executing the same test suite even on the original program will produce different results, we need to execute the same test suite for a mutant K number of times; (2) For Output Coverage and Input-Output Coverage, the test suite changes with each execution on a mutant. In other words, not each execution of the same test suite on even the original program may achieve the same coverage. Thus, for a particular mutant and for a specific coverage, we need to generate and execute test cases K number of times. This is different from the classical mutation analysis, where the test suite is fixed.

We define four categories of mutation operators: Add Gate (*AG*), Delete Gate (*DG*), Replace Gate (*RG*), and Replace Mathematical Operator (*RMO*). The first three types are specific to quantum programs, whereas the fourth type is similar to classical mutation operators, but adapted for quantum programs. There could be other types of mutation operators from classical computing, e.g., related to relational operators, which can also be adapted to quantum programs. However, such adaptation needs an investigation of its own.

The *AG* mutation operator introduces a gate on one of the qubits of the program. For example, as shown in Figure 3a, we added an *H* gate on *Qubit1* before the existing *H* gate of the program. The *DG* mutation operator deletes an existing gate from the program. For instance, as shown in Figure 3b, we deleted the *H* gate from the running example. The *RG* mutation operator replaces a gate with another one. For instance, in Figure 3c, we replaced the *H* gate with a *Not* gate. For the *RMO* category, we replaced a mathematical operator with another one. For instance, the addition is implemented as an *add()* operation. We, therefore, can replace this operation with, for instance, built-in *subtract()* operation.

Each mutation operator type is used to seed different faults in a program. Let $\mathit{MUTs} = \{m_1, \dots, m_n\}$ be a set of mutants of a program QP having specification $\mathtt{PS}$. Given a coverage criterion C, we test each mutant $m \in \mathit{MUTs}$, i.e., $\mathtt{TP}_C(m, \mathtt{PS}) = \langle \mathit{TSs}, \mathit{res} \rangle$ (see Section IV-D); the mutant is *killed* if *res* is $\mathtt{fail}_{\mathtt{WOO}}$ or $\mathtt{fail}_{\mathtt{OPO}}$, otherwise it is not killed (in this case, *res* is $\mathtt{incon}_{\mathtt{OPO}}$). Given all the test results $\mathit{MR} = \{\langle \mathit{TSs}_1, \mathit{res}_1 \rangle, \dots, \langle \mathit{TSs}_n, \mathit{res}_n \rangle\}$, we partition them in three subsets:

$$\mathit{MR}_{\mathtt{fail}_{\mathtt{WOO}}} = \{\langle \mathit{TSs}, \mathit{res} \rangle \mid \mathit{res} = \mathtt{fail}_{\mathtt{WOO}}\}$$
$$\mathit{MR}_{\mathtt{fail}_{\mathtt{OPO}}} = \{\langle \mathit{TSs}, \mathit{res} \rangle \mid \mathit{res} = \mathtt{fail}_{\mathtt{OPO}}\}$$
$$\mathit{MR}_{\mathtt{incon}_{\mathtt{OPO}}} = \{\langle \mathit{TSs}, \mathit{res} \rangle \mid \mathit{res} = \mathtt{incon}_{\mathtt{OPO}}\}$$

We can then define the mutation score (for each oracle) as:

$$ms_{\mathtt{WOO}} = \frac{|\mathit{MR}_{\mathtt{fail}_{\mathtt{WOO}}}|}{|\mathit{MUTs}| - \#\mathit{eqMuts}}$$
$$ms_{\mathtt{OPO}} = \frac{|\mathit{MR}_{\mathtt{fail}_{\mathtt{OPO}}}|}{|\mathit{MUTs}| - \#\mathit{eqMuts}}$$

where $\#\mathit{eqMuts}$ is the number of equivalent mutants. Some equivalent mutants may not be known before test execution; therefore, the *eqMuts* in the above equations refers to known equivalent mutants. Regarding quantum programs, there is no existing body of knowledge about equivalent mutants. Also, there is no approach available to determine equivalent mutants in quantum programs; therefore, we identified a few equivalent mutants in our experiment after test execution. Thus, we discuss our results, first without knowing about equivalent mutants (i.e., $\mathit{eqMuts} = 0$), and then discuss the results after text execution which resulted into identification of equivalent mutants. We further measure the percentage of total killed mutants as $ms_{\mathtt{TOTAL}} = ms_{\mathtt{WOO}} + ms_{\mathtt{OPO}}$.

We also defined the average size of the test suites killing the mutants with a given oracle as:

$$at_{\mathtt{WOO}} = \frac{\sum_{\langle \mathit{TSs}, \mathtt{fail}_{\mathtt{WOO}} \rangle \in \mathit{MR}_{\mathtt{fail}_{\mathtt{WOO}}}} \sum_{\mathit{TS} \in \mathit{TSs}} |\mathit{TS}|}{|\mathit{MR}_{\mathtt{fail}_{\mathtt{WOO}}}|}$$
$$at_{\mathtt{OPO}} = \frac{\sum_{\langle \mathit{TSs}, \mathtt{fail}_{\mathtt{OPO}} \rangle \in \mathit{MR}_{\mathtt{fail}_{\mathtt{OPO}}}} \sum_{\mathit{TS} \in \mathit{TSs}} |\mathit{TS}|}{|\mathit{MR}_{\mathtt{fail}_{\mathtt{OPO}}}|}$$

The total average size of test suites is calculated as follows:

$$at_{\mathtt{TOTAL}} = \frac{\sum_{\langle \mathit{TSs}, \mathit{res} \rangle \in \mathit{MR}} \sum_{\mathit{TS} \in \mathit{TSs}} |\mathit{TS}|}{|\mathit{MR}|}$$

VI. Experiment Design

In this section, we first present research questions (Section VI-A), then the subject systems used in the empirical study in Section VI-B, followed by experimental settings and evaluation metrics in Section VI-C. A replication package consisting of data, code, and R scripts is available online.[5]

[5]https://simula-complex.github.io/Quantum-Software-Engineering/ICST21.html

TABLE I: Selected Quantum Programs, their Characteristics, and Number of Mutants

Program	$\|Q\|$	$\|VD_I\|$	$\|VD_O\|$	Number of mutants			
				AG	*DG*	*RG*	*RMO*
Ent	2	2	2	7	2	1	0
Swap	3	4	1	15	4	3	0
RCR	2	4	4	13	4	3	0
Inc	3	8	8	12	0	0	1
Dec	3	8	8	12	0	0	1

A. Research Questions

We will answer the following two research questions (RQs).

- **RQ1** How do the three coverage criteria compare to each other in terms of overall cost and effectiveness of testing?
- **RQ2** How do the three coverage criteria compare to each other when studying their cost-effectiveness in terms of the two types of test oracles?

B. Subject Systems

We selected five quantum programs from [13], as summarized in Table I. The first program (*Ent*) is the same as the running example. Program 2 (i.e., *Swap*) aims to compare two qubits. Program 3 (*RCR*) studies the properties of entanglement related to randomness. In particular, it studies how reading one random qubit affects the probability of reading the second random qubit. Programs 4 and 5 (i.e., *Inc* and *Dec*) increment and decrement qubits. The interested readers can find details of these programs and code in [13].

Table I shows the number of qubits ($|Q|$), the number of valid input values ($|VD_I|$), and the number of valid output values ($|VD_O|$) for each program. We also show the number of mutants created with each of the four categories of the mutation operators. For instance, we have 10 mutants for *Ent*: seven for *AG*, two for *DG*, and one for *RG*.

C. Experimental Settings and Evaluation Metrics

Recall from Section IV-B that we need to set two parameters, i.e., K and M for experiments. We set $K = 2000$ and $M = 20$ (so, $R = 100$). There is no existing guide in the literature on how to select values of these parameters, therefore, further investigation, either empirical or theoretical, is needed to determine appropriate values for these parameters. Also, we needed to select a statistical test to assess passing or failing according to `OPO`. We needed a one sample test since we want to compare one sample with a specified probability value in test oracle. To this end, we selected one-sample Wilcoxon signed rank test with 99% confidence interval. This means that a sample is considered significantly different from the specified value (e.g., the expected probability of each valid output being 0.5 for our running example), if p-value< 0.01.

For each coverage criterion and each program, we report a total mutation score (i.e., $ms_{\texttt{TOTAL}}$) and mutation scores for each test oracle type (i.e., $ms_{\texttt{WOO}}$ and $ms_{\texttt{OPO}}$) as described in Section V. Also, we used various cost measures related to the number of test cases as described in Section V. We set $\texttt{Budget}_{\texttt{OC}}$ in Alg. 2 and $\texttt{Budget}_{\texttt{IOC}}$ in Alg. 3 both to 200, as we think it is acceptable, considering that we had limited time to execute all the generated test cases and 200 is relatively large. Different budgets may affect results, so further investigation is needed to know how to set a value for the budget.

TABLE II: Overall Mutation Scores and Average Number of Test Cases – RQ1

(a) Input Coverage

Program	`WOO`		`OPO`		`TOTAL`	
	$ms_{\texttt{WOO}}\%$	$at_{\texttt{WOO}}$	$ms_{\texttt{OPO}}\%$	$at_{\texttt{OPO}}$	$ms_{\texttt{TOTAL}}\%$	$at_{\texttt{TOTAL}}$
Ent	50%	1.8	30%	4000	80%	2001
Swap	–	–	72.8%	8000	72.8%	8000
RCR	–	–	80%	8000	80%	8000
Inc	69.2%	1.56	15.4%	16,000	84.6%	4924
Dec	69.2%	1	23.1%	16,000	92.3%	4924

(b) Output Coverage

Program	`WOO`		`OPO`		`TOTAL`	
	$ms_{\texttt{WOO}}\%$	$at_{\texttt{WOO}}$	$ms_{\texttt{OPO}}\%$	$at_{\texttt{OPO}}$	$ms_{\texttt{TOTAL}}\%$	$at_{\texttt{TOTAL}}$
Ent	50%	2.2	30%	4000	80%	2397
Swap	–	–	72.8%	31,415	72.8%	24,853
RCR	–	–	80%	15,212	80%	15,259
Inc	69.2%	1.1	23.1%	25,110	92.3%	8144
Dec	69.2%	1.2	15.4%	23,509	84.6%	8337

(c) Input-Output Coverage

Program	`WOO`		`OPO`		`TOTAL`	
	$ms_{\texttt{WOO}}\%$	$at_{\texttt{WOO}}$	$ms_{\texttt{OPO}}\%$	$at_{\texttt{OPO}}$	$ms_{\texttt{TOTAL}}\%$	$at_{\texttt{TOTAL}}$
Ent	50%	1.2	30%	4000	80%	122,396
Swap	–	–	72.8%	88,708	72.8%	68,873
RCR	–	–	95%	160,768	95%	160,990
Inc	69.2%	1.2	15.4%	111,402	84.6%	24,518
Dec	69.2%	1.3	15.4%	112,146	84.6%	24,690

VII. Experiment Results

We present results and analysis for RQ1 and RQ2 in Section VII-A and Section VII-B respectively. Discussions are presented in Section VII-C, limitations in Section VII-D, and threats to validity in Section VII-E.

A. Results and Analysis for RQ1

Table II summarizes results for each program for the three coverage criteria. Results related to RQ1 are shown in the last two columns of the tables under `TOTAL`. For *Ent*, we achieved the same 80% of mutation scores for input, output, and input-output coverage criteria with 2001, 2397, and 122,396 test cases respectively. For *Swap*, we achieved the same mutation score of 72.8% for input, output, and input-output coverage criteria with 8000, 24,853, and 68,873 test cases. These results suggest that there are no differences between the three coverage criteria for these two programs. Even the most expensive coverage criteria, i.e., the output and input-output coverage, couldn't increase the mutation scores when comparing with the input coverage, a less expensive coverage criterion.

For *RCR*, the input coverage and output coverage criteria achieved the mutation score of 80% with 8000 and 15,259 test cases respectively, whereas the input-output coverage criterion reached a mutation score of 95% with 160,990 test cases. This is an increase in 15% at the cost of extra 145,731 test cases as compared to the output coverage criterion, and 152,990 number of test cases with the input coverage criterion.

For *Inc*, the three coverage criteria achieved the mutation scores of 84.6%, 92.3%, and 84.6% at the cost of 4924, 8144, and 24,518 test cases respectively. Here, we see that the output coverage criterion achieved a higher mutation score than the input and input-output coverage criteria. For *Dec*, the input, output, and input-output coverage criteria reached the mutation scores of 92.3%, 84.6%, and 84.6% at the cost of 4924, 8337, and 24,690 test cases respectively. In this case, we can see that the input coverage criterion has the highest mutation score. For *Inc* and *Dec*, the input coverage and output coverage criteria have higher mutation scores as compared to the expensive input-output one. This may be due to the fact that the mutants are killed with lower confidence since less expensive criteria may have smaller sample sizes.

Based on the results, we can also conclude that even with a less expensive coverage criterion (i.e., input coverage), we managed to achieve higher mutation scores and even with expensive coverage criteria (i.e., output and input-output coverage criteria), we couldn't manage to increase mutation scores much except for *RCR*. There are two possible explanations. First, our quantum programs are small in size as it can be seen in Table I. Thus, we may observe differences in the effectiveness of the coverage criteria with complex programs. Second, the input coverage criterion is already expensive, especially when a fault cannot be caught with `WOO`, since we exercise each valid input once. As a result, for a simple program, the effectiveness of input coverage could be high.

B. Results and Analysis for RQ2

Now, we look at the results in terms of the two test oracle types, i.e., `WOO` and `OPO` for all the programs for the three coverage criteria to answer RQ2. Regarding the input coverage criterion (Table IIa), for *Ent*, *Inc*, and *Dec*, the mutation scores are higher for `WOO`, i.e., 50%, 69.2%, and 69.2% respectively as compared to 30%, 15.4%, and 23.1% respectively for `OPO`. Similar results can be observed for the output and input-output coverage criteria (Table IIb, Table IIc) for these three programs, where `WOO` had higher mutation scores than `OPO`. In general for `WOO`, we observed that the average number of test cases is quite low, i.e., minimum 1 and maximum 2.2 for these three programs. These results suggest that if the fault in a program results in a wrong output, it can possibly be caught with a lower number of test cases. Recall from Section IV-C that once a wrong output is observed, test case generation stops. This is the reason why the numbers of test cases for `WOO` are smaller. For *Swap* and *RCR*, mutation scores for `WOO` are shown as "–" since, for all the three coverage criteria, there were no mutants killed with wrong outputs. Regarding `OPO` for *Swap*, all the three coverage criteria reached the mutation score of 72.8%, implying no differences among the three coverage criteria in terms of mutation scores. However, in terms of cost, the input-output coverage criterion is much more expensive (i.e., 88,708 test cases) than the output coverage criterion (i.e., 31,415 test cases) and the input coverage criterion (i.e., 8000 test cases). For *RCR*, the input, output, and input-output coverage criteria reached the mutation scores of 80%, 80%, and 95% respectively implying that the input-output coverage criterion has a higher effectiveness than the other two criteria. However, the input-output coverage criterion has a higher cost, i.e., 145,556 more test cases than the output coverage criterion and 152,768 more test cases than the input coverage criterion.

In general, `OPO` is expensive as it can be seen in the $at_{\texttt{OPO}}$ columns of the three tables. Based on the results, we can conclude that with `WOO` the cost of finding a fault is low. However, if certain faults cannot be found with `WOO`, then the cost of finding faults with `OPO` could be quite higher. However, this cost may be reduced with a proper upper limit of test execution with `OPO`. For instance, we set $\texttt{Budget}_{\texttt{OC}}$ and $\texttt{Budget}_{\texttt{IOC}}$ to 200, i.e., the maximum test suite size. As a result, output and input-output coverage can have a maximum of 400,000 test cases. This is quite a large number of test cases. However, considering we wanted to have more confidence on results, we used the largest sample size as possible within the practical constraint of executing tests on quantum computer simulator hosted by QCEngine. Finding an optimal limit on the budget could potentially reduce the number of test cases; however, this requires further investigation.

C. Discussion

We further study the mutants that weren't killed by the test suites generated from any coverage criterion for each program. These mutants were candidates for being equivalent mutants. To check this, we employed a simple process. We checked a program's state with and without a mutation just before the final reading through the QC Engine's step by step execution facility [13]. In case of equivalent mutants, the magnitudes (see Section III) of the states that determine the probabilities of the output values remain the same, but the overall program states were altered, e.g., with different phases that didn't affect the probabilities of the output values. Below, we analyze each of the programs and provide examples.

For *Ent*, two mutants weren't killed by any of the three coverage criteria. These two mutants are about adding a *Not* gate before and after the *H* gate on Qubit1 of our running example. Now, let's assume that *Ent* is prepared in 0 state, i.e., the same as in Figure 2a. Without any mutant, the program state before reading should be as shown in Figure 2c. However, once we add a mutant, e.g., adding a *Not* gate before the *H* gate on Qubit1, the program state will be as shown in Figure 4. When comparing mutated program state in Figure 4 with program without mutation in Figure 2c, we can see that both $|0\rangle$ and $|3\rangle$ have the same magnitudes. This means that when read, both programs have equal probabilities of producing 0 or 3. However, the difference is on the phases of $|3\rangle$, i.e., 180 in the mutated program, whereas 0 in the original program. Based

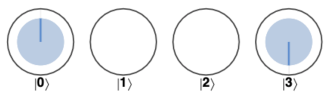


Fig. 4: Running example – Mutated program state

on this observation, we can conclude that with specialized test strategies (part of our future work), one should be able to identify differences in phases due to a fault resulting in a phase different than expected. In general, for *Ent*, the two equivalent mutants only caused differences in terms of phases and the magnitudes remained the same.

For *Swap*, six mutants weren't killed with test suites generated with any coverage criterion. All these were equivalent mutants. In case of *RCR*, all the mutants were killed by test suites generated with at least one coverage criterion except for one equivalent mutant. For *Inc*, all mutants were killed by at least one coverage criterion. For *Dec*, there was only one mutant that wasn't killed by all the three coverage criteria. This mutant was once again an equivalent mutant.

D. Limitations

Our work is preliminary, thus it has limitations that we will address in the future. First, we acknowledge that our coverage criteria will face scalability issues with the increased number of qubits. Thus, we can see the need for test optimization approaches in combination with the coverage criteria. Second, in our current coverage criteria, we didn't deal with qubits' phases (Section III) since we treated the programs as black-box. We can see the need for more advanced coverage criteria that can also ensure the phase coverage of qubits. Third, in this experiment, we only performed mutation analysis with one qubit gates and seeded one mutant at a time. Also, we didn't study qubit gates that manipulate phases of qubits. Thus, we see several improvements in mutation analysis in the future: (1) Studying the effectiveness of coverage criteria in terms of finding faults due to multiple qubits mutation operators; (2) Higher-order mutants are also possible, i.e., applying more than one mutation operator simultaneously and studying effectiveness of the coverage criteria.

E. Threats to Validity

Based on the guidelines reported in [15], we discuss threats to validity. An *internal validity threat* is related to the sampling strategies of the output coverage and input-output coverage criteria. A sampling strategy may affect the number of test cases to observe a desired output with a specified probability. However, which sampling strategy to use is a research question of its own and requires a separate experiment.

One *conclusion validity threat* is about the effect of the probabilistic nature of QC in drawing conclusions. Thus, we generated and executed test suites 2000 times (i.e., K). We also selected $M = 20$. This means that we had a sample size of 100 (i.e., K/M) to calculate percentages of observed outputs corresponding to each valid input. The selection of K and M may affect results and thus we need more empirical evaluations to select values of these parameters.

A *construct validity threat* is that the measures we used for drawing our conclusions may not be sufficient. At first, we computed the percentage of times that an output was observed corresponding to a valid input. This is important, since we need to compare the observed results with the specified probabilities of outputs in program specifications. Hence, we believe that this metric is adequate. Also, to draw more stable conclusions, we assessed the statistical significance of the results with the Wilcoxon test. More specifically, we compared the percentages of observed outputs corresponding to each valid input with the probabilities specified in the program specifications.

An *external validity threat* is that the results may not be generalizable since we used only five quantum programs. We are aware that such a selection is inherently partial, and we need more quantum programs with different characteristics to generalize the results. The lack of real-world quantum programs is a well-known challenge.

VIII. Conclusion and Future Work

To test quantum programs, we proposed three coverage criteria based on inputs and outputs of a quantum program, together with two types of test oracles. We also provided a procedure to determine passing and failing of test suites with statistical analyses. These contributions all together form our approach *Quito* (QUantum InpuT Output coverage). The coverage criteria were evaluated with five quantum programs using mutation analysis. We also presented a set of mutation operators for quantum programs and provided definitions of mutation scores for the two types of test oracles. Based on the test execution results and analyses, we identified a set of equivalent mutants for quantum programs.

In the future, we will improve the coverage criteria of *Quito* in several dimensions, such as developing test minimization methods to reduce the number of test cases across the three coverage criteria, introducing boundary value analysis and equivalence partitioning, and including test execution time in our experiments. Also, we would like to investigate white-box test coverage criteria for quantum programs. Finally, we would like to mention that we worked with qubits with no hardware errors. Thus, our results are valid for testing quantum programs executed on quantum computer simulators without simulating hardware errors. Developing testing techniques for pure qubits is essential since such techniques ensure that quantum programs' faults are due to faulty logic and not due to hardware errors. Naturally, we need to make these testing techniques hardware error-aware in the future.

References

[1] S. Garhwal, M. Ghorani, and A. Ahmad, "Quantum programming language: A systematic review of research topic and top cited languages," *Archives of Computational Methods in Engineering*, pp. 1–22, 2019.

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

[3] J. Zhao, "Quantum software engineering: Landscapes and horizons," *CoRR*, vol. abs/2007.07047, 2020. [Online]. Available: https://arxiv.org/abs/2007.07047

[4] S. Honarvar, M. R. Mousavi, and R. Nagarajan, "Property-based testing of quantum programs in Q#," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 430–435.

[5] D-Wave, "Applications – helping solve some of the most difficult challenges," 2021. [Online]. Available: https://www.dwavesys.com/applications

[6] IBM, "A primer on quantum computing," 2021. [Online]. Available: https://www.research.ibm.com/quantum-computing/

[7] UTS, "Quantum programming and verification," 2020. [Online]. Available: https://www.uts.edu.au/research-and-teaching/our-research/centre-quantum-software-and-information/qsi-research/qsi-research-programs/quantum-programming-and-verification

[8] J. Wang, M. Gao, Y. Jiang, J. Lou, Y. Gao, D. Zhang, and J. Sun, "QuanFuzz: Fuzz testing of quantum program," *CoRR*, vol. abs/1810.10310, 2018. [Online]. Available: http://arxiv.org/abs/1810.10310

[9] Y. Huang and M. Martonosi, "QDB: From Quantum Algorithms Towards Correct Quantum Programs," in *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*, ser. OpenAccess Series in Informatics (OASIcs), T. Barik, J. Sunshine, and S. Chasins, Eds., vol. 67. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 4:1–4:14. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10196

[10] M. Ying, S. Ying, and X. Wu, "Invariants of quantum programs: characterisations and generation," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 818–832, 2017.

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

[12] P. A. M. Dirac, "A new notation for quantum mechanics," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, no. 3, pp. 416–418, 1939.

[13] M. Gimeno-Segovia, N. Harrigan, and E. Johnston, *Programming Quantum Computers: Essential Algorithms and Code Samples*. O'Reilly Media, Incorporated, 2019. [Online]. Available: https://books.google.no/books?id=LZY1vgEACAAJ

[14] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 275–378.

[15] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.