Application of Combinatorial Testing to Quantum Programs

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

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

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

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

***Abstract*—The capability of Quantum Computing (QC) in solving complex problems has been increasingly recognized. However, similar to classical computing, to fully exploit QC's potential, it is important to ensure the correctness of quantum programs. Doing so via software testing is, however, very challenging because of QC's inherent properties: superposition and entanglement. Towards the direction of ensuring the correctness of quantum programs, we propose an approach called *QuCAT* (QUantum CombinAtorial Testing) for systematic and automated testing of quantum programs by benefiting from combinatorial testing, which has been proven to be cost-effective in testing classical programs. *QuCAT* supports two combinatorial test suite generation scenarios, i.e., generating combinatorial test suites of a given strength, and incrementally generating and executing combinatorial test suites of increasing strength until a fault is found. The approach employs two types of test oracles to assess test results. We performed an empirical study with 18 faulty versions of quantum programs to evaluate *QuCAT* with strengths of two, three, and four in the two test generation scenarios. We compare the cost-effectiveness of combinatorial testing of various strengths and random testing (taken as baseline approach). Results show that combinatorial testing always performs better than random testing with the same cost and finds faults more quickly (in terms of required number of test cases). In addition, in most cases, combinatorial testing with a higher strength outperforms the lower strength in terms of effectiveness.**

***Index Terms*—quantum programs, quantum software testing, combinatorial testing**

I. Introduction

Quantum programming enables the development of Quantum Computing (QC) applications. These days, several programming languages are available for quantum programming, e.g., IBM's Qiskit, Google's Cirq, and Microsoft's Q#. Although there exists programming support, there is still a lack of systematic and automated techniques for cost-effective testing of quantum programs [1], [2] such that correct and reliable QC applications can be developed. The lack of such testing techniques is due to the fact that quantum programs possess characteristics that are different from classical programs, such as being probabilistic, performing computations in superposition, and supporting entanglement. These characteristics make testing quantum programs very challenging.

Due to the increasing awareness of the importance of quantum software testing, recently researchers have started proposing some specific testing techniques. Examples include the definition of testing coverage criteria [3], leveraging property-based testing [4], handling the assertions at runtime [5], and applying search method to generate test suites for quantum programs [6].

In a quantum program, a fault could be related to a faulty gate which is directly (or indirectly) connected to the input qubits; a failure could be triggered by particular values of these qubits. However, no testing approach for quantum programs exists that directly targets these input values combinations.

Combinatorial testing (CT) is a testing technique, aiming to deal with various combinations of input variables. For instance, a commonly applied CT technique is pair-wise testing, i.e., testing all pairs of input variable values. Motivated by the successful application of CT in different domains [7], [8], in this paper, we present a testing approach, *QuCAT*, for automated and systematic testing of quantum programs with CT. The approach supports two combinatorial *usage scenarios* for test generation. For the first usage scenario (`UsageScenario1`), it supports generating combinatorial test suites of a given *strength* (e.g., pair-wise, 3-wise, etc.). For the second usage scenario (`UsageScenario2`), it implements an incremental algorithm for generating and executing combinatorial test suites of increasing strength, which stops once a fault is found, or a maximum strength is reached. To assess the passing and failing of test suites, we employed two types of test oracles defined for testing quantum programs [3].

To assess the cost and effectiveness of *QuCAT*, we performed an empirical evaluation with 18 faulty versions of quantum programs. We assessed the combinatorial approach with the strengths of two, three, and four for the first usage

This work is supported by the National Natural Science Foundation of China under Grant No. 61872182 and Qu-Test (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.

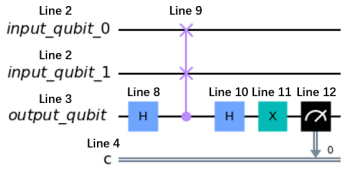

Fig. 1. Swap Test – Circuit diagram

scenario, and with maximum strength four for the second usage scenario. For both scenarios, we used random testing as the comparison baseline. Results showed that in most cases, CT with high strength, along with high cost, can outperform CT with low strength. In addition, CT is significantly better than random testing in terms of effectiveness with the same cost, and CT can find faults more quickly.

Paper structure. Sect. II presents the background and illustrates it with an example, followed by a comparison with related works in Sect. III. We describe the proposed approach in Sect. IV. The experimental design and results are presented in Sect. V and Sect. VI, respectively. In Sect. VII we discuss some threats that may affect the validity of our evaluation and how we tried to mitigate them. Finally, we present conclusion and future works in Sect. VIII.

II. Background and Running Example

Classical computers work on bits, represented by 0 and 1. During a computation, each bit can only be either 0 or 1 at every point in time. Quantum programs work on *qubits*, which is an extension of bits according to quantum mechanics. However, each qubit, before getting measured, is in *superposition*, which means that it can be both 0 and 1. During the computation, the probabilities of being 0 or 1 change.

For each qubit q, we can define its *quantum state* as follows:

$$|q\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$$

The Dirac notation $|q\rangle$ refers to the quantum state of q and α_0 and α_1 are the *amplitudes*, which are complex numbers defining two elements [9]. The first one is the *magnitude*: when we measure the qubit, the probability of being in the $|0\rangle$ state is $|\alpha_0|^2$ while the probability of being in the $|1\rangle$ state is $|\alpha_1|^2$, which satisfy $|\alpha_0|^2 + |\alpha_1|^2 = 1$. The second one is the *relative phase*, i.e., the angle of the complex number in polar form.

Table I shows some common quantum gates and their description. According to the table, a HAD gate puts a qubit into superposition. A control qubit implements a kind of condition for the quantum circuit. A phase gate changes the relative phase of one qubit. A measure gate is for reading values of qubits as the outcomes of a quantum program.

Fig. 1 shows the circuit of a three-qubit program (*Swap Test* taken from [10]). The program uses the *swap test* to compare the similarity of two input qubits: *input_qubit_0* and *input_qubit_1*. If the two qubits are initialized with the same states, the probability of getting value 1 as outcome after measuring qubit *output_qubit* is 100%. However, the probability will decrease when the initialized values of these two input qubits get increasingly different: e.g., if *input_qubit_0* is initialized with $|1\rangle$ and *input_qubit_1* is initialized with $|0\rangle$, the probability of getting outcome value 1 will be 50%.

TABLE I
Definitions of some gates of quantum circuits

Name	Description
Hadamard (HAD)	It places a qubit into a superposition, such that the qubit has an equal chance of being in state $\|0\rangle$ or $\|1\rangle$.
NOT	The NOT gate negates a qubit, i.e., turning from state $\|0\rangle$ into state $\|1\rangle$, or turning from state $\|1\rangle$ to state $\|0\rangle$.
Controlled-NOT (CNOT)	A two-qubit gate consisting of one control qubit and one target qubit. If the control qubit is in state $\|1\rangle$, the target qubit will get flipped.
Controlled-Controlled-NOT (CCNOT)	It is a three-qubit version of CNOT which has two control qubits. If both of these control qubits are in state $\|1\rangle$, the target qubit will get flipped.
SWAP	It is a two-qubit gate, which swaps two qubits.
Controlled-SWAP (CSWAP)	It is a three-qubit gate, which swaps two target qubits under the condition that the control qubit is in state $\|1\rangle$.
Phase (P)	It rotates the relative phase of a qubit.
Controlled-Phase (CP)	It is a two-qubit gate. The target qubit will change its relative phase, under the condition that the control qubit is in state $\|1\rangle$.
Measure	This gate can store a measurement into a classical bit, which can be either 0 or 1.

```
1  # Initialization statements
2  input_qubit = QuantumRegister(2, 'input_qubit')
3  output_qubit = QuantumRegister(1, 'output_qubit')
4  c = ClassicalRegister(1, 'c')
5  qc = QuantumCircuit(input_qubit, output_qubit, c)
6
7  # Implementation statements
8  qc.h(output_qubit)
9  qc.cswap(output_qubit, input_qubit[0], input_qubit[1])
10 qc.h(output_qubit)
11 qc.x(output_qubit)
12 qc.measure(output_qubit, c)
```

Listing 1. Swap Test – Qiskit Code

Listing 1 shows the code of this program in the Qiskit framework [11] in Python. We have marked each operation in the circuit shown in Fig. 1 with the corresponding line number in the code. Lines 2 and 3 create two quantum registers, which initialize two input qubits (*input_qubit[0]* and *input_qubit[1]* of array *input_qubit*) with state $|0\rangle$, and one output qubit (*output_qubit*) with state $|0\rangle$. Line 4 creates a classical register (c) which will store the result after measuring *output_qubit*. Line 5 creates the quantum circuit of *swap test*. Line 8 applies a HAD gate on *output_qubit*, thereby putting this qubit into superposition. Line 9 applies a CSWAP gate, which is a conditional operation with one control qubit (*output_qubit*) and two target qubits (*input_qubit[0]* and *input_qubit[1]*). If the control qubit is in state $|1\rangle$, the two target qubits will be exchanged. Instead, if the control qubit is in superposition, there will be a certain probability for the two target qubits to get exchanged, depending on the state of the control qubit. Line 10 applies the second HAD gate and due to the reversibility of gates in quantum computing, *output_qubit* returns to the original state. Line 11 applies a NOT gate to *output_qubit*, which flips a qubit from state α_0 to α_1 or from α_1 to α_0. Line 12 measures

output_qubit and stores the result into the classical bit *c*. Since in this example the initialized values of the two input qubits are both 0, we will get the result 1 at the end.

III. Related Work

We here discuss literature on quantum software testing (Sect. III-A) and CT in classical computing (Sect. III-B).

A. Quantum Software Testing

The *Quito* method [3], along with its accompanying tool [12], were defined based on inputs and outputs of quantum programs; the approach has two types of testing oracles and three coverage criteria. Moreover, *Quito* judges failing and passing of test suites with one-sample Wilcoxon signed ranked test. To assess the effectiveness of its three coverage criteria, an empirical study with five quantum programs was conducted, in addition to mutation analyses with four types of mutation operators.

Mendiluze et al. [13] proposed *Muskit*, a quantum mutation analysis tool. It defines mutation operators on gates of quantum programs, generates faulty versions of quantum programs automatically to execute, and produces results for test analyses.

Liu et al. [14] proposed quantum circuits for runtime assertions with the help of *ancilla qubits* (additional qubits for obtaining information of qubits of interest without affecting them), to overcome the challenge that qubits can not be copied and measured directly during testing.

Li et al. [5] proposed *Proq*, a runtime assertion scheme for testing and debugging quantum programs based on projections (i.e., closed subspaces of the state space). Only a small number of projective measurements are sufficient for checking the satisfaction of a projection by a quantum state. They conducted experiments to compare *Proq* with the approaches proposed in [15] and [14]. Results show that their assertions perform better in terms of high expressive power, flexible assertion location, few execution times, and low implementation overhead.

QuSBT [6] is a search-based method for generating test suites that can maximize the number of failing test cases within a given testing budget. The paper presents the problem encoding, a classification of failures, statistical tests for assessing test results, and, most importantly, the fitness function for search. The authors applied GA, compared with random search, to solve the test optimization problem, and evaluated *QuSBT* through mutation analysis with 30 carefully designed faulty programs of six quantum programs. Results show that *QuSBT* achieved a significant improvement over random search in more than 80% of the faulty programs.

B. Combinatorial Testing

Combinatorial testing (CT) aims to find faults due to interactions among inputs of a program, such as 2-ways and 3-ways interactions [7], [8], [16]. The concept behind CT was initially developed in the 1980s [16]. Since then, it has been extensively applied in testing various software systems. For example, a popular repository [17] lists 783 publications relevant to CT as of January 2021. Nie and Leung [7] made an extensive survey up to 2011, while a more recent survey by Tzoref-Brill [8] covers works up to 2018; Wu et al. [16], instead, made a survey specific to constrained CT.

A line of research in CT consists in the development of test generation algorithms producing test suites of a given strength, such as greedy algorithms [18], [19], [20] and search algorithms [21], [22].

CT has been successfully applied in different domains. For example, Tao et al. [23] apply CT for validating autonomous driving systems; they build an ontology for describing the environment in which the autonomous driving system is operating, and apply CT over the ontology to generate tests for the system. Ma et al. [24] investigate the use of CT for Deep Neural Networks. They first propose different coverage criteria requiring that different t-way combinations of Deep Neural Network neurons are activated. Then, they also propose a test generation approach for generating tests achieving these criteria.

IV. Combinatorial Testing for Quantum Programs

We here first provide some definitions about quantum programs in Sect. IV-A. We then introduce, in Sect. IV-B, the notions of test and test assessment for quantum programs. Finally, we introduce our proposed approach to use combinatorial testing (CT) for quantum programs in Sect. IV-C.

A. Definitions for quantum programs

Definition 1 (Inputs, outputs, and quantum program)**.** Let Q be the set of qubits of a quantum program QP. A subset of qubits $I \subseteq Q$ identifies the *input*, while a subset $O \subseteq Q$ identifies the *output*.[1] $D_I = \mathcal{B}^{|I|}$ are the *input values*, and $D_O = \mathcal{B}^{|O|}$ are the *output values*. A quantum program QP can be described as a function $\mathrm{QP}\colon D_I \to 2^{D_O}$.

The definition shows that, given an input value, a quantum program can return different output values. The program specification, if available, specifies the *expected* probability distribution followed by the output values.

Definition 2 (Program specification)**.** Given a quantum program $\mathrm{QP}\colon D_I \to 2^{D_O}$, we identify with $\mathtt{PS}$ the *program specification*, i.e., the expected behavior of the program. For a given input assignment $i \in D_I$, the program specification states the expected probabilities of occurrence of all the output values $o \in D_O$, i.e.,:

$$\mathtt{PS}(i) = [p_0, \ldots, p_{|D_O|-1}]$$

where p_h is the expected probability (with $0 \leq p_h \leq 1$) that, given the input value i, the value h is returned as output. It holds $\sum_{h=0}^{|D_O|-1} p_h = 1$. We introduce the following notation to identify the probabilities of outputs that can actually occur:

$$\begin{aligned} \mathtt{PS}_{NZ}(i) &= [p \in \mathtt{PS}(i) \mid p \neq 0] \\ &= [p_{j_1}, \ldots, p_{j_k}] \qquad \text{with } j_1, \ldots, j_k \in D_O \end{aligned}$$

[1] I and O do not need to be disjoint. Moreover, some qubits could be neither inputs nor outputs, i.e., $I \cup O \subseteq Q$.

B. Testing of quantum programs

Before introducing CT for quantum programs, we provide general definitions regarding testing of quantum programs.

Definition 3 (Test input)**.** A *test input* is a pair $\langle i, n \rangle$, where i is an assignment to qubits (i.e., $i \in D_I$), and n indicates how many times QP must be run with i.

A test needs to be executed multiple times to get an *estimate* of the probability distribution followed by the output values. The number of repetitions must be sufficient to be representative enough. To this aim, we here follow the approach proposed in [6]. The idea of the approach is that different inputs require different numbers of repetitions to get representative results; namely, the number of required repetitions is proportional to the number of possible output values, as specified by the program specification. Given an input i, the number of required repetitions is defined as follows:

$$\texttt{numReps}(i) = |\texttt{PS}_{NZ}(i)| \times 100$$

Definition 4 (Test execution and test result)**.** Given a test input $\langle i, n \rangle$ for a quantum program QP, the *test execution* consists in running QP n times with input i. We identify with $res = [\text{QP}(i), \ldots, \text{QP}(i)] = [o_1, \ldots, o_n]$ the *test result*, where o_j is the output value of the jth execution of the program.

1) Test assessment: The notion of *failure* in quantum programs is very specific, as it takes into account the stochastic nature of quantum computing. Following [3], [6], we identify two types of failures: *unexpected output failure* and *wrong output distribution failure*. Test assessment of a test result res is done by checking whether these two failures occurred, as explained in the following.

a) Unexpected Output Failure (uof): Such failure occurs when the output o returned by the program for a given input i, is not allowed by the program specification PS, i.e., $\texttt{PS}(i, o) = 0$. To check *uof*, it is enough to check if some produced output is unexpected, i.e.,

$$\texttt{fail}_{uof} := (\exists o_j \in res : \texttt{PS}(i, o_j) = 0)$$

In case a failure of this type occurred, the test assessment is stopped, as we can already assess that the test is not passing. Otherwise, the checking for the other type of failure is done as follows.

b) Wrong Output Distribution Failure (wodf): Such failure occurs when the output values returned by multiple executions for a given input i follow a probability distribution *significantly different* from the one specified by the program specification. We assess this with a *goodness of fit* test using the Pearson's chi-square test [25]: the test compares the observed frequencies of the values of the categorical variable with the expected distribution. For our purposes, given a test input $\langle i, n \rangle$ and its test result $res = [o_1, \ldots, o_n]$, we apply the chi-square test as follows:

- we retrieve, from the program specification, the expected probabilities of the outputs that can occur given the input i, i.e., $\texttt{PS}_{NZ}(i) = [p_{j_1}, \ldots, p_{j_k}]$, with $j_1, \ldots, j_k \in D_O$ (see Def. 2);
- we build the *one-dimensional contingency table* $[c_{j_1}, \ldots, c_{j_k}]$ of the chi-square test, by counting the number of occurrences of each possible output $j_1, \ldots, j_k$ in the test result, i.e., $c_{j_h} = |\{o \in res \mid o = j_h\}|$. Note that it is guaranteed that each returned output is considered in one of the counts $c_{j_1}, \ldots, c_{j_k}$; indeed, *wodf* is checked only if *uof* did not reveal any failure, meaning that the program returned outputs only from $j_1, \ldots, j_k$;
- we apply the chi-square test over the contingency table $[c_{j_1}, \ldots, c_{j_k}]$ and the expected occurrence probabilities $[p_{j_1}, \ldots, p_{j_k}]$.

If the p-value is less than a given significance level α ($\alpha = 0.01$ in our experiments), we can reject the null hypothesis that there is no statistical significant difference. We record the assessment with the following predicate:

$$\texttt{fail}_{wodf} := (\text{p-value} < \alpha)$$

In the end, the test is considered *failed* if one of the two failures has been observed:

$$\texttt{fail} := \texttt{fail}_{uof} \vee \texttt{fail}_{wodf}$$

C. Combinatorial testing for quantum programs

CT assumes that faults are triggered by particular combinations, of a given *strength* (2, 3, etc.), of input parameters of the program. In a quantum program, faults may be related to the use of wrong gates in the quantum circuit, or to a wrong connection between gates. The faulty gate is directly or indirectly connected with some of the input qubits. Therefore, our research hypothesis is that some particular values combinations of some of the input qubits can trigger the fault.

Therefore, CT seems to be a viable solution for the testing of quantum programs. We here introduce the notion of CT tailored for quantum programs.

Value schema [7] is a key concept in CT: it specifies a combination of input values that must be observed in a test.

Definition 5 (Value schema)**.** Given a quantum program QP with input qubits I, a *k-value schema* ($k > 0$ and $k \leq |I|$) is a combination of values $(\ldots, v_{i_1}, \ldots, v_{i_k}, \ldots)$ for k input qubits. We identify with '$-$' the values of qubits that are not fixed by the schema.

An example of 2-value schema for a quantum program with 4 qubits could be $(-, 1, -, 0)$.

Definition 6 (Combinatorial Test Suite)**.** Given a quantum program QP with input qubits I, a *combinatorial test suite* T of *strength* k guarantees that each k-value schema is contained in the input values of at least a test $t \in T$.

The required strength k defines the type of testing: *pairwise testing* (when $k = 2$), *3-wise testing* (when $k = 3$), etc.

Different generation algorithms have been proposed to generate combinatorial test suites, such as AETG [19], IPO [20], IPOG [26], search-based approaches [22], etc. Also different

Algorithm 1 Incremental Combinatorial Test Generation

Require: the quantum program QP under test
Require: the set of input qubits I
Require: the program specification PS
Require: maximum required strength K
1: $T \leftarrow \emptyset$ ▷ to collect executed tests
2: **for** $k = 2, \ldots, K$ **do**
3: $T_k \leftarrow$ GENCOMBTESTSUITE(I, k)
4: **for** $i \in T_k$ **do**
5: $n \leftarrow$ numReps(i) ▷ required number of repetitions
6: $res \leftarrow [\text{QP}(i), \ldots, \text{QP}(i)]$ ▷ test executed n times
7: $T \leftarrow T \cup \{\langle i, n\rangle\}$
8: $\texttt{fail}_{uof} \leftarrow$ EVAL_*uof*(res, PS, i)
9: $\texttt{fail}_{wodf} \leftarrow$ *false*
10: **if** $\texttt{fail}_{uof} =$ *false* **then**
11: $\texttt{fail}_{wodf} \leftarrow$ EVAL_*wodf*(res, PS, i)
12: **if** $\texttt{fail}_{uof} \vee \texttt{fail}_{wodf}$ **then**
13: **return** $\langle T, failure\rangle$
14: **return** $\langle T, pass\rangle$

tools have been developed, such as ACTS [27], CASA [28], CITLAB [29], CTwedge [30], PICT [31], etc.

In this paper, we evaluate the effectiveness of using CT for quantum programs. We consider two different *usage scenarios* of CT for quantum programs.

In the first usage scenario (UsageScenario1), we assume that the tester specifies a strength k, and generates a test suite T_k for that strength. They can then assess the passing and failing of tests, as explained in Sect. IV-B1.

In the second usage scenario (UsageScenario2), instead, we propose an incremental test generation approach that keeps on generating test suites of incremental strength, till a failure is detected, or a maximum k is reached. The approach is shown in Alg. 1. At Line 1, the approach initializes a set to collect the tests that are actually executed. Then, for each possible strength k till a maximum value K (Line 2):

- it generates a test suite T_k of strength k (Line 3);
- for each test i in T_k:
  - it runs the program with the test i (Line 6) (see Def. 4), and collects the test in the set of executed tests (Line 7);
  - it checks whether a failure of type *uof* occurred (Line 8);
  - if no *uof* occurred (Line 10), it checks whether a failure of type *wodf* occurred (Line 11);
  - if at least one of the two failures occurred (Line 12), it returns the set of executed tests with the information that a failure was found (Line 13);

At the end, if no failure has been found, the executed tests are returned with the information that all tests passed (Line 14).

V. Experiment Design

We here present our experimental design. First, in Sect. V-A, we provide the set of research questions that we will answer. In Sect. V-B, we present the list of benchmark programs used in the experiments. We show the experimental settings in Sect. V-C and discuss the evaluation metrics and employed statistical tests in Sect. V-D. Code to reproduce the experiments, benchmarks, and experimental results are available online [32].

TABLE II
BENCHMARK PROGRAMS*

Program	$\lvert I\rvert$	# gates	depth	Injected faults (position: right after the inputs)		
AS	6	25	22	$\texttt{AS}_1$: CCNOT	$\texttt{AS}_2$: CCCNOT	$\texttt{AS}_3$: CCCCNOT
BV	7	21	3	$\texttt{BV}_1$: CCNOT	$\texttt{BV}_2$: CCCNOT	$\texttt{BV}_3$: CCCCNOT
CE	11	25	26	$\texttt{CE}_1$: CCHAD	$\texttt{CE}_2$: CCCHAD	$\texttt{CE}_3$: CCCCHAD
IQ	10	60	56	$\texttt{IQ}_1$: CCHAD	$\texttt{IQ}_2$: CCCHAD	$\texttt{IQ}_3$: CCCCHAD
QR	9	15	12	$\texttt{QR}_1$: CCNOT	$\texttt{QR}_2$: CCCNOT	$\texttt{QR}_3$: CCCCNOT
SM	5	40	5	$\texttt{SM}_1$: CCNOT	$\texttt{SM}_2$: CCCNOT	$\texttt{SM}_3$: CCCCNOT

$|I|$ and # gates are the number of input qubits and gates in the program (e.g., H, X), and *circuit depth* is the length of the longest sequence of quantum gates. One unit of the length is determined by the output of a gate given as the input to another gate. We use the Qiskit command depth.

A. Research Questions

We will answer the following research questions (RQs):

- **RQ1** How do applications of CT with different strengths compare to each other in terms of cost and effectiveness?
  This RQ helps us study whether various strengths have an effect on the effectiveness (i.e., finding faults), and cost in terms of size of the generated test suite.
- **RQ2** How is the effectiveness of CT with different strengths as compared to random testing?
  This RQ studies whether CT can outperform a simple baseline; thereby, warrant the need for the systematic CT. We do not study the cost explicitly here, since we let random testing have the same test suite size as CT; thus, the cost is the same for both approaches.
- **RQ3** How quickly can CT find a fault as compared to random testing?
  This RQ assesses the cost of applying CT in terms of finding a fault. Given that executing test cases on quantum programs can be computationally very expensive, an approach that can find a fault with a lower number of executed test cases is preferred.

Note that RQ1 and RQ2 are related to UsageScenario1, while RQ3 to UsageScenario2 (see Sect. IV-C).

B. Benchmark Programs and Faulty Programs

For our experimentation, we chose the following six quantum programs to assess the cost and effectiveness of CT for quantum programs. Our selection criterion is to pick quantum programs with different characteristics (e.g., the number of gates, the depth of quantum program). We summarize various characteristics of the selected quantum programs in Table II, and we provide their brief descriptions below.

- **Add Squared (AS):** It performs a mathematical addition of a squared quantum integer b and another quantum integer a (i.e., $a+b*b$), both of which are in superposition. This program demonstrates how this mathematical operation is performed according to the superposition principle.
- **Conditional Execution (CE):** It performs a conditional addition on one quantum integer b, which is in superposition. The condition is based on another quantum integer a, which is also in superposition.

- **Bernstein-Vazirani (BV) and Simon (SM) algorithms:** These two quantum programs implement two cryptography programs. Consult [33], [34] for more details;
- **invQFT (IQ):** it implements inverse quantum Fourier transform. With a given frequency as input, it produces a signal that has periodically varying magnitude [10];
- **QRAM (QR):** This program manages an increment with QRAM, which can dynamically store and read values in the memory address with the help of superposition, differently from a conventional RAM [10].

Note that we extended the implementation of these programs provided in [10] to handle a higher number of qubits. In addition, these programs have also been used in [6].

To assess whether tests generated with CT can find faults in quantum programs, we needed a set of faulty versions of the programs. To this end, we created three faulty versions of each benchmark program described in Table II by introducing one fault at a time in the program. These faulty programs will be referred to as by the name of their original program with fault number in subscript, e.g., three faulty versions of AS are referred to as $\texttt{AS}_1$, $\texttt{AS}_2$, and $\texttt{AS}_3$, respectively. Thus, in total, we have created 18 faulty programs. Fault number 3 is the most difficult one followed by fault 2 and then fault 1.

C. Experiment Settings

We wrote quantum programs in Python in the Qiskit framework 0.27.0 [11]. We executed the quantum programs on the ideal quantum computer simulator shipped with the framework. By "ideal", we mean that hardware errors are not simulated during program execution. To generate test suites of varying strengths, we use the PICT tool [31]. For our experiments, we chose strengths 2, 3, and 4.

To answer RQ1 and RQ2 (corresponding to UsageScenario1 in Sect. IV-C), for each strength k and each benchmark program, we generated and executed 500 k-wise test suites. Moreover, to compare the performance of CT with a random approach, we randomly generated and executed 500 test suites of the same size of the combinatorial test suites.

To answer RQ3 (corresponding to UsageScenario2 in Sect. IV-C), we have executed, for each benchmark program, the incremental combinatorial test generation reported in Alg. 1 using, as maximum strength, $K = 4$. We have also executed a *random* version of the approach in which tests are generated randomly. Both the combinatorial and the random approach have been executed 500 times.

D. Evaluation Metrics and Statistical Tests

Table III describes a set of metrics that we used to answer the RQs. For each RQ, we describe metric types (i.e., cost, or effectiveness), the metrics themselves, and statistics used.

1) Metrics and Statistics for RQ1: To answer RQ1 in terms of *cost*, we calculate the size of the generated test suite corresponding to each strength (i.e., $k = 2, 3, 4$), i.e., $|T_k|$. Regarding the *effectiveness*, for each faulty program and each strength k, we calculate *success rate* (sr), i.e., the number of times a test suite found a fault out of the total number of runs.

TABLE III
EVALUATION METRICS AND STATISTICAL TESTS

RQ	metric type	metrics	statistical tests
RQ1	effectiveness	sr_k^c	Mann-Whitney U test, $\hat{A}_{12}$
	cost	$\lvert T_k \rvert$	
RQ2	effectiveness	$sr_k^c, sr_k^r, fd_k^c, fd_k^r$	Fisher's Exact test, Odds Ratio (OR)
	cost	$\lvert T_k \rvert$	
RQ3	cost	$\lvert T^c \rvert$, $\lvert T^r \rvert$	Mann-Whitney U test, $\hat{A}_{12}$

This metric was chosen based on the guide [35]. We will refer to the success rate corresponding to each strength k as sr_k^c, where c represents the combinatorial testing approach.

To assess the statistical significance of the two metrics for CT of various strengths, we chose the Mann–Whitney U test as the statistical test, as suggested by the guide [35]. Furthermore, as also suggested by [35], we also calculated the effect size with the Vargha and Delaney's $\hat{A}_{12}$ statistics. When comparing the application of CT with two strengths (e.g., k_1 and k_2) for a given metric (i.e., cost or effectiveness), if p-value is less than 0.05, then there is a significant difference between k_1 and k_2. If $\hat{A}_{12}$ is 0.5, then it means that there is no difference between the two strengths. An $\hat{A}_{12}$ value higher than 0.5 suggests that the values of the metric for k_1 are highly likely to be higher than those of k_2, and the higher the value of $\hat{A}_{12}$, the higher the likelihood. An $\hat{A}_{12}$ value less than 0.5 suggests otherwise.

2) Metrics and Statistics for RQ2: For RQ2, the cost for a given strength k is calculated in the same way as in RQ1. Note that, for the random approach, the size of the test suite corresponding to a strength k is the same as the size of the test suite generated by CT of strength k. In terms of effectiveness, for a given strength k, in addition to sr_k^c, we also calculate the success rate of random for the same k, i.e., sr_k^r. In addition, for CT and random, we calculate the overall percentage of fault detection (fd), which is the average success rate across all the 18 faulty programs. We will refer to the percentage of fault detection corresponding to each strength k and its corresponding random as fd_k^c and fd_k^r, respectively.

To assess the statistical significance of success rates of CT of various strengths as compared to the random approach, we employed the Fisher's exact test to calculate p-values, and used odds ratio (OR) as the effect size measure, based on the guide [35]. A p-value less than 0.05 means that the difference between the success rates of CT and random is statistically significant. A value of OR equals to 1.0 means no difference between the success rates, whereas a value greater than 1.0 means that CT is highly likely to be better than random. The higher the value of OR, the higher the likelihood of CT to be better than random. A value less than 1.0 means vice versa.

3) Metrics and Statistics for RQ3: For RQ3 (UsageScenario2), we calculate the *cost*, i.e., the number of test cases needed to find a fault with CT (i.e., $|T^c|$) and random (i.e., $|T^r|$). Moreover, to determine the statistical significance of the results, we compare CT with random using the Mann-Whitney U test and $\hat{A}_{12}$ statistic in a similar way as described in RQ1.

TABLE IV
USAGESCENARIO1 (RQ1 AND RQ2) – EXPERIMENTAL RESULTS – COST $|T_k|$, AND EFFECTIVENESS sr_k^c AND sr_k^r FOR COMBINATORIAL AND RANDOM

	$k=2$			$k=3$			$k=4$		
	$\|T_2\|$	sr_2^c	sr_2^r	$\|T_3\|$	sr_3^c	sr_3^r	$\|T_4\|$	sr_4^c	sr_4^r
AS_1	6.3	100.0%	82.6%	13.3	100.0%	97.6%	26.2	100.0%	100.0%
AS_2	6.3	72.4%	59.4%	13.3	100.0%	83.2%	26.2	100.0%	96.0%
AS_3	6.3	43.0%	37.4%	13.3	77.4%	65.4%	26.2	100.0%	85.8%
BV_1	7.0	100.0%	88.6%	14.6	100.0%	98.0%	30.2	100.0%	100.0%
BV_2	7.0	73.8%	64.8%	14.6	100.0%	85.4%	30.2	100.0%	99.0%
BV_3	7.0	41.0%	39.0%	14.6	87.8%	61.6%	30.2	100.0%	87.8%
CE_1	8.5	100.0%	91.6%	19.3	100.0%	100.0%	45.2	100.0%	100.0%
CE_2	8.5	82.4%	69.8%	19.3	100.0%	94.6%	45.2	100.0%	99.8%
CE_3	8.5	54.6%	47.0%	19.3	76.0%	74.2%	45.2	100.0%	96.6%
IQ_1	8.0	100.0%	93.0%	17.2	100.0%	99.6%	38.5	100.0%	100.0%
IQ_2	8.0	80.0%	67.0%	17.2	100.0%	91.2%	38.5	100.0%	99.6%
IQ_3	8.0	50.4%	43.2%	17.2	85.6%	76.0%	38.5	100.0%	96.0%
QR_1	8.3	100.0%	91.2%	18.3	100.0%	99.8%	42.0	100.0%	100.0%
QR_2	8.3	76.8%	68.2%	18.3	100.0%	91.2%	42.0	100.0%	100.0%
QR_3	8.3	48.8%	49.4%	18.3	85.4%	73.4%	42.0	100.0%	93.8%
SM_1	6.0	100.0%	80.2%	12.0	100.0%	96.8%	18.2	100.0%	98.6%
SM_2	6.0	71.4%	55.6%	12.0	100.0%	81.0%	18.2	100.0%	93.2%
SM_3	6.0	43.6%	37.0%	12.0	74.2%	58.0%	18.2	100.0%	71.4%
fd_k		13.4/18	11.7/18		16.9/18	15.3/18		18.0/18	17.2/18

TABLE V
USAGESCENARIO1 (RQ1) – COMPARISON OF EFFECTIVENESS sr_k^c AND COST $|T_k|$ BETWEEN CT APPLICATIONS WITH SIGNIFICANTLY STRENGTHS

	2 vs 3				2 vs 4				3 vs 4			
	sr_k^c		$\|T_k\|$		sr_k^c		$\|T_k\|$		sr_k^c		$\|T_k\|$	
	p-value	OR	p-value	$\hat{A}_{12}$	p-value	OR	p-value	$\hat{A}_{12}$	p-value	OR	p-value	$\hat{A}_{12}$
AS_1	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00
AS_2	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	1.00	-	<0.01	0.00
AS_3	<0.01	0.22	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00
BV_1	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00
BV_2	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	1.00	-	<0.01	0.00
BV_3	<0.01	0.22	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00
CE_1	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00
CE_2	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	1.00	-	<0.01	0.00
CE_3	<0.01	0.17	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00
IQ_1	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00
IQ_2	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	1.00	-	<0.01	0.00
IQ_3	<0.01	0.17	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00
QR_1	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00
QR_2	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	1.00	-	<0.01	0.00
QR_3	<0.01	0.16	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00
SM_1	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00	1.00	-	<0.01	0.00
SM_2	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	1.00	-	<0.01	0.00
SM_3	<0.01	0.27	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00	<0.01	0.00

For k_1 vs k_2, values of sr_k^c highlighted in green show the cases in which $sr_{k_2}^c$ is significantly better than $sr_{k_1}^c$. In the rest of the cases, there is no significant difference between the two approaches in terms of sr_k^c. Note that $|T_{k_1}|$ is always significantly better than $|T_{k_2}|$; thus, we do not highlight it.

VI. RESULTS

In this section, we answer the three research questions (RQs) reported in Sect. V-A.

A. RQ1: Comparing CT with Various Strengths

Experimental results for `UsageScenario1` (RQ1 and RQ2) are reported in Table IV. For each benchmark program and each strength k, we report the average size $|T_k|$ of the generated test suites (across the 500 repetitions) as *cost* measure, and the success rates sr_k^c and sr_k^r (i.e., percentage of test suites triggering a failure) of the combinatorial and random approaches as *effectiveness* measure. In the table, we also report the overall fault detection fd across all the programs, both for the combinatorial and the random approaches.

To answer RQ1, starting from the results of Table IV, we have compared the effectiveness and cost of different applications of CT of various strengths. For each pair of strengths k_1 and k_2, we have compared their success rates $sr_{k_1}^c$ and $sr_{k_2}^c$ (i.e., effectiveness) with the Fisher's exact test, and their test suite sizes $|T_{k_1}|$ and $|T_{k_2}|$ (i.e., cost) with the Mann-Whitney U test and with the $\hat{A}_{12}$ statistic, for each of the 18 benchmark programs. Table V reports the results, in terms of p-value and Odds Ratio (OR) for the effectiveness, and p-value and $\hat{A}_{12}$ for the cost. Values highlighted in green in the table indicate the cases in which $sr_{k_2}^c$ is significantly better than $sr_{k_1}^c$. Note that $|T_{k_1}|$ is always significantly better than $|T_{k_2}|$ and so we do not highlight it in the table.

As expected, we observe that CT with a lower strength has significantly less cost (in terms of size of the generated test suite) than CT with a higher strength. This can also be observed from Table IV where $|T_{k_i}| < |T_{k_j}|$ for $k_i < k_j$. This is well known in CT research, and lower bounds on the size of a generated test suite can also be theoretically assessed [7][2] and also upper bounds can be estimated [36]. The question is whether a higher cost for a higher strength can be justified in terms of effectiveness (i.e., success rate sr_k^c).

As shown in Table V, when comparing sr_k^c between applications of CT with strength $k = 2$ and $k = 3$, 12 programs show that 3-wise CT is significantly better than pairwise testing, and there is no difference in the rest of 6 programs (all with fault number 1) whose sr_k^c, for both CT applications, is 100% (see Table IV). Also, in columns 2 vs 4 and 3 vs 4, the comparison results indicate that, except for programs in which both versions of CT always trigger the failures, CT with higher strength always outperforms that with lower strength.

According to success rate sr_k^c in Table IV, the six faulty programs with fault number 1 (the easiest fault to detect, e.g., AS_1) can always be caught by all 500 test suites, even with the pairwise CT. At the same time, the six programs with fault number 2 (e.g., AS_2) are always detected by applying CT with strength three and four. The rest of the programs (e.g., QR_3), which are the most difficult ones, are always detected only by applying CT with strength four.

Based on the analyses above, we can conclude that applying CT with a higher strength can indeed lead to the increase in effectiveness. But for some programs with faults easy to detect, CT with a lower strength (e.g., pairwise) can still achieve good results (i.e., $sr_k^c = 100\%$).

B. RQ2: Comparing Combinatorial Testing with Random

In this RQ, we are interested in assessing the effectiveness of CT with various strengths, when compared to the random

[2]For quantum programs, we need at least 2^k tests for strength k. However, more tests are needed with the increasing number of input qubits.

TABLE VI

USAGESCENARIO1 (RQ2) – COMPARISON OF THE EFFECTIVENESS sr_k^c BETWEEN CT AND RANDOM TESTING – P-VALUE AND ODDS RATIO (OR) OF THE FISHER'S EXACT TEST

	$k=2$		$k=3$		$k=4$	
	p-value	OR	p-value	OR	p-value	OR
AS$_1$	<0.01	INF	<0.01	INF	1.00	-
AS$_2$	<0.01	1.79	<0.01	INF	<0.01	INF
AS$_3$	0.08	1.26	<0.01	1.81	<0.01	INF
BV$_1$	<0.01	INF	<0.01	INF	1.00	-
BV$_2$	<0.01	1.53	<0.01	INF	0.06	INF
BV$_3$	0.56	1.09	<0.01	1.97	<0.01	INF
CE$_1$	<0.01	INF	1.00	-	1.00	-
CE$_2$	<0.01	2.03	<0.01	INF	1.00	INF
CE$_3$	0.02	1.36	<0.01	2.50	<0.01	INF
IQ$_1$	<0.01	INF	0.50	INF	1.00	-
IQ$_2$	<0.01	1.97	<0.01	INF	0.50	INF
IQ$_3$	0.03	1.34	<0.01	1.88	<0.01	INF
QR$_1$	<0.01	INF	1.00	INF	1.00	-
QR$_2$	<0.01	1.54	<0.01	INF	1.00	-
QR$_3$	0.90	0.98	<0.01	2.12	<0.01	INF
SM$_1$	<0.01	INF	<0.01	INF	0.02	INF
SM$_2$	<0.01	1.99	<0.01	INF	<0.01	INF
SM$_3$	0.04	1.32	<0.01	2.08	<0.01	INF

Values of sr_k^c highlighted in purple indicate the cases in which CT is significantly better than random testing. In the rest of the cases, there is no significant difference between CT and random testing.

TABLE VII

USAGESCENARIO2 (RQ3) – AVERAGE NUMBER OF EXECUTED TESTS $|T^c|$, $|T^r|$ – COMPARISON RESULTS BETWEEN CT AND RANDOM TESTING – P-VALUE OF MANN-WHITNEY U TEST AND $\hat{A}_{12}$ STATISTIC

	AS$_1$	AS$_2$	AS$_3$	BV$_1$	BV$_2$	BV$_3$
average $\lvert T^c\rvert$	2.9	5.7	11.0	2.9	6.0	12.5
average $\lvert T^r\rvert$	3.8	7.5	14.5	3.9	7.5	16.0
p-value	<0.01	<0.01	<0.01	<0.01	0.04	0.21
$\hat{A}_{12}$	0.45	0.45	0.44	0.44	0.46	0.48

	CE$_1$	CE$_2$	CE$_3$	IQ$_1$	IQ$_2$	IQ$_3$
average $\lvert T^c\rvert$	2.9	5.9	10.8	3.0	5.8	11.7
average $\lvert T^r\rvert$	4.0	7.3	13.2	3.8	7.3	12.7
p-value	<0.01	0.01	0.01	0.04	<0.01	0.68
$\hat{A}_{12}$	0.42	0.46	0.45	0.46	0.45	0.49

	QR$_1$	QR$_2$	QR$_3$	SM$_1$	SM$_2$	SM$_3$
average $\lvert T^c\rvert$	2.9	6.2	11.5	2.7	5.5	10.3
average $\lvert T^r\rvert$	3.9	7.5	14.3	3.7	6.9	14.3
p-value	<0.01	0.04	<0.01	<0.01	0.02	<0.01
$\hat{A}_{12}$	0.45	0.46	0.44	0.45	0.46	0.42

Values of sr_k^c highlighted in purple indicate the cases in which CT is significantly better than random testing. In the rest of the cases, there is no significant difference between CT and random testing.

approach (see Sect. V-C).

From Table IV, we observe that, with increasing strengths, the effectiveness of CT increases, to the point of always triggering the failures with strength 4. We can also observe that, as expected, the effectiveness of the random approach increases. However, when we consider the average percentage of the fault detection of all the 18 programs in the last row of Table IV, we can clearly see that the value of fd_k^c of CT is larger than that of fd_k^r for each strength k.

To draw more definitive conclusions, we compare the results reported in Table IV with the Fisher's exact test. Table VI reports, for each strength k and each faulty program, the results of the comparison between effectiveness values sr_k^c and sr_k^r of CT and random. The table reports the p-values of the Fisher's exact test and odds ratio (OR) values. The cases in which CT is significantly better are highlighted in purple. Recall that, for the same strength k, CT and random approaches have the same cost, i.e., the same test suite size $|T_k|$.

From Table IV, we observe that, both CT with strength 2 and random can not always trigger a failure; however, from Table VI, we observe that CT with strength 2 is almost always significantly better in terms of sr_k^c (for 15 out of 18 programs), showing that it produces more useful tests than random.

When comparing random with CT with strength 3, from Table VI we observe that again, for 15 out of 18 programs, CT performed better than random. From Table VI, we also observe that CT with strength 4 is still significantly better than random for half of the programs.

These results show that CT is more effective than random in terms of generating test suites that are more likely to expose failures of quantum programs.

C. RQ3: Combinatorial Testing's Quickness in Finding Faults

In this section, we evaluate how quickly CT can find faults. So, we consider `UsageScenario2`, as described in Alg. 1 and Sect. V-C; namely, CT generates and executes test suites by incrementally increasing its strength until a failure is observed. To have a fair comparison, we also implemented a random approach, which keeps on generating test inputs randomly and executing them until a failure is detected. We use the number of executed tests as the evaluation metric. For each faulty program, both CT and random approaches have been repeated 500 times. Table VII reports, for each faulty program, values of the average number of executed tests $|T^c|$ and $|T^r|$ of 500 runs, and results of the Mann-Whitney U test and $\hat{A}_{12}$ statistic. Cases in which CT significantly outperforms random testing are highlighted in purple.

From Table VII, we can observe that, for each program, the average number of test cases needed to find a fault with CT is consistently smaller than that of random, and the results of CT are significantly better than random in 16 of the 18 faulty programs. For all the 12 faulty programs of `AS`, `CE`, `QR` and `SM`, CT works better than random. Among the six faulty programs of `BV` and `IQ`, for four of them, CT significantly outperformed random. Thus, we can conclude that CT can find faults more quickly than random.

D. Overall Discussion

As discussed in the results of the RQs, as expected, CT with a higher strength performed better than CT with a lower strength in terms of fault detection (RQ1); the CT approaches

with the three different strengths performed significantly better than random using the same number of tests (RQ2); and less test cases are needed to find a fault with CT than with random (RQ3). In the rest of this section, we would like to discuss some of our additional observations, which we consider interesting and important for future further investigation.

Methodology-wise, our approach, in the context of `Usage-Scenario2`, can be optimized by reusing test cases generated by CT with a lower strength. For instance, when applying CT with strength 3, the approach can avoid to try to cover triples that have been coincidentally already covered by the tests generated by CT with strength 2. Doing so will save the time cost required for both the generation and execution.

In our empirical study, we employed 18 faulty programs of 6 quantum programs. These quantum programs are of various complexity in terms of the number of input qubits, the number of gates, and the circuit depth (see Table II). Although our current design of the experiment does not allow us to study and draw a conclusion on whether the complexity of a quantum program has any influence on the effectiveness of the CT approaches, we can still observe from Table IV that the CT approaches can outperform random testing for quantum programs of various complexity. In addition, still looking at Table IV, we do not observe noticeable differences in terms of the effectiveness of CT across the quantum programs with various complexity. For instance, the pairwise CT achieved 100%, 82.4% and 54.6% effectiveness for $\texttt{CE}_1$, $\texttt{CE}_2$, and $\texttt{CE}_3$, respectively, which are comparable with that for the three `SM` programs; notice that `SM` is the simplest program and `CE` is the most complex one, in terms of the number of input qubits.

As we discussed in Sect. V-C, in our experiments, we intentionally controlled the difficulty of faults being introduced into the six quantum programs. For instance, $\texttt{AS}_3$ is more difficult to be detected than $\texttt{AS}_2$, which is more difficult to be detected than $\texttt{AS}_1$. Therefore, when looking at Table IV, we can observe that the faulty programs with more difficult injected faults (e.g., $\texttt{AS}_3$) obtained less effectiveness than those with easier faults seeded (e.g., $\texttt{AS}_2$), unless when CT reaches 100% fault detection for the quantum programs of all levels of difficulty (e.g., with strength 4). This is easy to understand, because, in quantum programs, to detect a difficult faulty gate naturally requires test cases with particular combinations involving a higher number of qubits.

VII. Threats to Validity

Similar to any other experiment, our experiments can also be affected by some threats to their validity.

In terms of external threats to validity, we conducted our experiments with only six quantum programs and 18 (faulty) benchmark programs. Therefore, the results of our experiments and conclusions drawn based on these results can only be generalized to quantum programs with similar characteristics. Similar to many other experiments conducted in the software engineering domain, more experiments with diverse characteristics of quantum programs are definitely required to further generalize the conclusions we obtained.

To assess the passing and failing of tests with *wodf*, we employed the Pearson's Chi-square test. It could be that there are other better means of doing that, which forms a threat to the construct validity of our experiment. However, the Chi-square test has been used for this purpose in existing related quantum software testing literature [15], [4].

Another threat to the construct validity is about the selection of faults that were introduced in the quantum programs to create the 18 faulty benchmark programs. We chose a set of arbitrary faults to be seeded in quantum programs, which could potentially affect our results. However, currently, there does not exist any bug repository for quantum programs that we could use to seed realistic faults in quantum programs.

Empirical evidence for classical programs [37] suggests that combinatorial test suites of strength 4 can find more than 90% of the faults for most of the programs; therefore, in our experiment, we selected 4 as the maximum strength of CT. However, we understand that dedicated empirical studies are needed, supported with more complex quantum programs with more difficult faults, to understand whether test suites of higher strengths may be needed in some cases, and to provide solid evidence and guideline on how to select a CT strength.

In order to obtain coverage of a given strength t, generation algorithms as those employed by PICT can generate different test suites, depending on the initial seed. To account for this randomness in the generation, we repeated, for each faulty program, our experiments 500 times to ensure that the results were not obtained by chance; this, to a certain extent, reduces threats to the conclusion validity. Moreover, in order to draw significant conclusions, we compared the results of CT with different strengths and those of the random approach, with suitable statistical tests as suggested by an established guide on conducting experiments with randomized algorithms [35].

VIII. Conclusion and Future Work

In this paper, we proposed *QuCAT* for the systematic and automated testing of quantum programs with combinatorial testing (CT). The approach supports two test generation scenarios. The first one generates a combinatorial test suite of a given strength, whereas the second one incrementally generates and executes combinatorial test suites of increasing strengths till a fault is detected, or the maximum allowed strength is reached. In order to assess the cost and effectiveness of *QuCAT*, we performed an empirical study using six quantum programs with 18 faulty versions in total. We compared test results of CT with strength 2, 3, and 4. Also, we compared CT with random testing. Results showed that in terms of effectiveness (i.e., ability to detect faults), CT performed significantly better than random testing with the same cost, and CT with higher strength (and so higher cost) outperformed that with lower strength (and so lower cost). Moreover, experiments showed that CT can find faults more quickly than random testing.

As future work, we will develop a fault localization approach for quantum programs by analyzing test results. Also,

we will conduct experiments on more complex quantum programs with more qubits and gates, and higher circuit depth. In addition, we will increase the number of benchmark programs. Besides, we will test quantum programs on a real quantum computer (e.g., provided by IBM) to assess the effectiveness of our approach in the presence of hardware errors, which we have not taken into consideration in this paper.

REFERENCES

[1] A. Miranskyy and L. Zhang, "On testing quantum programs," in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '19. IEEE Press, 2019, pp. 57–60.

[2] J. Zhao, "Quantum software engineering: Landscapes and horizons," *CoRR*, vol. abs/2007.07047, 2020.

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

[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] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Projection-based runtime assertions for testing and debugging quantum programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[6] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Generating failing test suites for quantum programs with search," in *Search-Based Software Engineering*, U.-M. O'Reilly and X. Devroey, Eds. Cham: Springer International Publishing, 2021, pp. 9–25.

[7] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, Feb. 2011.

[8] R. Tzoref-Brill, "Chapter two - advances in combinatorial testing," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 79–134.

[9] 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.

[10] M. Gimeno-Segovia, N. Harrigan, and E. Johnston, *Programming Quantum Computers: Essential Algorithms and Code Samples*. O'Reilly Media, Incorporated, 2019.

[11] R. Wille, R. Van Meter, and Y. Naveh, "IBM's Qiskit tool chain: Working with and developing for real quantum computers," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1234–1240.

[12] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Quito: a coverage-guided test generator for quantum programs," in *The 36th IEEE/ACM International Conference on Automated Software Engineering, Tool Demonstration*. IEEE/ACM, 2021.

[13] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue, "Muskit: A mutation analysis tool for quantum software testing," in *The 36th IEEE/ACM International Conference on Automated Software Engineering, Tool Demonstration*. IEEE/ACM, 2021.

[14] J. Liu, G. T. Byrd, and H. Zhou, "Quantum circuits for dynamic runtime assertions in quantum computation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1017–1030.

[15] 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), vol. 67. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 4:1–4:14.

[16] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, "A survey of constrained combinatorial testing," *CoRR*, vol. abs/1908.02480, 2019.

[17] "Combinatorial testing repository," https://gist.nju.edu.cn/ct_repository/, 2021.

[18] R. C. Bryce and C. J. Colbourn, "A density-based greedy algorithm for higher strength covering arrays," *Software Testing, Verification and Reliability*, vol. 19, no. 1, pp. 37–53, 2009.

[19] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, Jul. 1997.

[20] Y. Lei and K. Tai, "In-Parameter-Order: a test generation strategy for pairwise testing," in *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, 1998, pp. 254–261.

[21] S. Ghazi and M. Ahmed, "Pair-wise test coverage using genetic algorithms," in *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, vol. 2, 2003, pp. 1420–1424 Vol.2.

[22] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 540–550.

[23] J. Tao, Y. Li, F. Wotawa, H. Felbinger, and M. Nica, "On the industrial application of combinatorial testing for autonomous driving functions," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019, pp. 234–240.

[24] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "DeepCT: Tomographic combinatorial testing for deep learning systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 614–618.

[25] A. Agresti, *An introduction to categorical data analysis*, 3rd ed. Wiley-Blackwell, 2019.

[26] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy for T-Way Software Testing," in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, 2007, pp. 549–556.

[27] ACTS, "Automated combinatorial testing for software," https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software/downloadable-tools, 2021.

[28] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *1st International Symposium on Search Based Software Engineering*, 2009, pp. 13–22.

[29] A. Gargantini and P. Vavassori, "CITLAB: A laboratory for combinatorial interaction testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 559–568.

[30] A. Gargantini and M. Radavelli, "Migrating combinatorial interaction test modeling and generation to the web," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018, pp. 308–317.

[31] PICT, "Pairwise independent combinatorial testing," https://github.com/microsoft/pict, 2021.

[32] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Qucat: Application of combinatorial testing to quantum programs – supplementary material," https://github.com/Simula-COMPLEX/qucat-paper, 2021.

[33] E. Bernstein and U. Vazirani, "Quantum complexity theory," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 11–20.

[34] D. R. Simon, "On the power of quantum computation," *SIAM J. Comput.*, vol. 26, no. 5, p. 1474–1483, Oct. 1997.

[35] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1–10.

[36] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, no. 1, pp. 143–152, 2004, optimal Discrete Structures and Algorithms.

[37] R. Kuhn, R. N. Kacker, Y. Lei, and D. Simos, "Input space coverage matters," *Computer*, vol. 53, no. 1, pp. 37–44, 2020.