Development of a Tool for Finding Equivalent Mutants in Quantum Program: A Perspective to Measure the Quality of Quantum Software

Ajay Kumar  
Thapar Institute of Engineering and Technology

Prisha Kwatra  
Thapar Institute of Engineering and Technology

Sunita Garhwal  
Thapar Institute of Engineering and Technology

Research Article

Keywords: Quantum Mutation Testing, Equivalent Mutant, Quantum Software, Test Cases, Qiskit, Quality of Quantum Software

Posted Date: May 30th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-2250025/v2

License: ☑️ ☑️ This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.
Development of a Tool for Finding Equivalent Mutants in Quantum Program: A Perspective to Measure the Quality of Quantum Software

Ajay Kumar, Prisha Kwatra and Sunita Garhwal

Computer Science and Engineering Department, Thapar Institute of Engineering and Technology, Patiala, India

Abstract

Software testing is an important activity to ensure the software’s quality. Mutation testing is a fault-based technique for checking the effectiveness of test cases for the verification and validation of software. Testing quantum programs is challenging due to the involvement of quantum mechanics concepts (such as superposition and entanglement) and the probabilistic nature of the outcome.

In this paper, we will propose the modified mutation testing approach for quantum programs. The proposed approach will help to reduce the cost, time, and effort of performing the mutation testing. Moreover, we also propose a method for determining the equivalent mutant and developing a tool to check whether the mutated and original quantum program behave semantically equivalent or differently.

Keywords: Quantum Mutation Testing, Equivalent Mutant, Quantum Software, Test Cases, Qiskit, Quality of Quantum Software.

1. Introduction

Quantum Computing can potentially solve industry-relevant complex computational problems more efficiently than classical computing. Shor [1] developed one such practical algorithm for integer factorization. Grover [2] developed a quantum search algorithm for finding the unique element from an unstructured list of N elements with complexity $O(\sqrt{N})$. Research in the quantum field has significantly progressed in the last three decades after the development of Shor’s and Grover’s algorithms. With the rapid growth of...
research in quantum computing, several quantum computers and simulators
are accessible and available on the cloud. Several large-scale industry appli-
cations have been developed, and there is a concern related to the quality of
quantum software similar to classical software.
In classical software testing, Mutation testing is a fault-based technique to
determine the effectiveness of test cases by introducing artificial faults in the
software. The introduced faults are called mutants. Test cases are adequate
if test cases can kill all the non-equivalent mutants. The cost of mutation
testing increased due to the vast number of mutants and the determination
of equivalent mutants.
There is a need to develop a tool to identify the equivalent mutants by
checking the semantic equivalence of the original and mutated program. Ev-
every quantum software runs several shots to observe the outcomes. Mutation
testing compares the original and mutated program’s measured outcomes
with each test case. If the original and mutated program outcomes differ for
at least one test case, then the mutant is called a killed mutant; otherwise,
it is either a live or equivalent mutant. Thereafter, the equivalence mutant
procedure is used to find whether the mutant is equivalent. We need to
run each mutated version with several shots in quantum programming. To
reduce the cost, time, and effort, we find the circuit matrix equivalence of
the original and mutated program by running on the simulator only once.
Thereafter, the remaining mutants are classified into live or killed by running
the mutated versions and original program on an actual quantum computer
on the cloud. Inspired by the application of mutation testing to ensure the
quality of classical software, we propose and modify the concept of mutation
testing for quantum software in this paper.

1.1. Prior work:
Mutation testing is based on the principle of the Coupling effect and the
Competent Programmer Hypothesis. DeMillo et al. [3] pointed out that the
program written by a competent programmer differ from the correct program
with a small number of syntactic fault. Offutt [4] pointed out the concept
of the mutation coupling effect (test cases that detect all simple mutants
can detect all complex mutants formed by the coupling of simple mutants).
Lipton [5] introduced the concept of mutation testing in 1971. Mutation
testing has been applied to several programming languages such as C [6, 7],
Python[8], Java[9, 10, 11, 12], FORTRAN[13, 14], Haskell[15], SQL[16, 17, 18]
for ensuring the effectiveness of test cases. Singla et al. [19] used the concept
of the same mutation operator under similar conditions occurring at different locations in a program to reduce the cost of mutation testing.

Miransky et al. [20] recently introduced the concept of testing and debugging quantum software based on use cases for quantum software. Ali et al. [21] proposed the Quantum Input Output coverage (Qito) for three coverage criteria defined by the input and output of the quantum program. They analyzed the coverage criterion using mutation testing. Moguel et al. [22] pointed out the need and motivation for quantum software engineering. Weder et al. [23] introduced a ten-phases software life cycle for gate-based quantum software. Zhao et al. [24] identified the bug patterns in Qiskit and discussed how bug patterns could be eliminated. Zhao [25] proposed metrics for measuring the size and structure of quantum software.

Campos and Souto [26] QBugs, an experimental infrastructure evaluation of new research and reproducibility of published research in quantum software engineering. Ali and Yue [27] introduced the Quantum Software modeling language for understanding and modeling quantum programs independent of the quantum platform. Mendiluze et al. [28] proposed a Mutation testing tool (Muskit) for performing mutation testing of Quantum Programs. They pointed out Muskit is not able to detect the equivalent mutants. Wang et al. [29] proposed a QDiff for performing differential testing of Qiskit quantum programs. They generated a semantically equivalent version by applying equivalent gate transformation and mutation. Further, they called a sequence of gates semantically equivalent to another sequence by executing and comparing their measurement. Fortunato et al. [30] carried out a case study on 24 quantum programs. They considered the concept of the syntactically equivalent gate.

Motivated by these concepts, the contributions of this paper are threefold:

- Modified approach of Mutation testing has been introduced for the quantum program.
- An algorithm has been designed for finding equivalent mutants for the Quantum program.
- Mutation testing tool has been designed for finding equivalent mutants.

The reader can refer to various comprehensive surveys in the area of quantum computing, specifically, quantum walk[31], quantum programming language [32], quantum software testing [33], non-classical automata [34],
[35] and quantum cryptography [36], for in-depth understanding the basics of quantum computing.

**Organization of the paper:** The rest of the paper is organized as follows. We will introduce a few basic terms and notations in Preliminaries Section 2 related to mutation testing and quantum computing. Section 3 introduces the concept of a modified mutation testing approach for quantum software. Section 4 consists of the procedure for finding equivalent mutants, and finally, section 5 consists of the conclusion and future scope.

2. **Preliminaries**

This section describes the basic concepts and terminology in quantum computing and mutation testing.

### 2.1. Qubit

In classical computing, the basic unit of information is a bit. Similarly, the basic unit of quantum information in quantum computing is a quantum bit (Qubit). A qubit is a linear superposition of two orthonormal basis states. Mathematically, we can write a quantum state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $\alpha$ and $\beta$ are complex number satisfying the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. Ket 0 and Ket 1 are represented by $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ respectively.

### 2.2. Quantum Gates

Quantum gates are used to manipulate the qubits and are known as the basic building block of quantum circuits. The number of inputs and outputs must be the same for quantum gates to satisfy the reversibility condition. Table 1 describes the single qubit gates (Pauli-X, Pauli-Y, Pauli-Z, Hadamard, S, and T gates).

Regarding the Bloch sphere, Pauli-X, Pauli-Y, and Pauli-Z gates rotate by $\pi$ around the X, Y, and Z-axis, respectively. Hadamard gate plays a crucial role in creating the quantum states in superposition. Consider a qubit is $|1\rangle$, on applying the Hadamard gate, the qubit is in $1/\sqrt{2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = 1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle$.

Table 2 describes the two-qubit CNOT, Controlled-Z, and Swap gates.
Table 1: Quantum Single Qubit Gates and their corresponding matrices

<table>
<thead>
<tr>
<th>Gate</th>
<th>Matrix Representation</th>
<th>Figure</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
</table>
| Pauli-X  | \[
| 0 & 1 \\
| 1 & 0 
|\] | X       | $|0\rangle$            | $|1\rangle$             |
|          |                       |        | $|1\rangle$            | $\alpha|0\rangle + \beta|1\rangle$ |
|          |                       |        |                        |                         |
| Pauli-Y  | \[
| 0 & -i \\
| i & 0 
|\] | Y       | $|0\rangle$            | $i|1\rangle$            |
|          |                       |        | $|1\rangle$            | $-i|0\rangle$           |
| Pauli-Z  | \[
| 1 & 0 \\
| 0 & -1 
|\] | Z       | $|0\rangle$            | $|0\rangle$             |
|          |                       |        | $|1\rangle$            | $-|1\rangle$            |
| Hadamard | $1/\sqrt{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ | H     | $|0\rangle$            | $\frac{|0\rangle + |1\rangle}{\sqrt{2}}$ |
|          |                       |        | $|1\rangle$            | $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ |
| S        | \[
| 1 & 0 \\
| 0 & i 
|\] | S       | $|0\rangle$            | $|0\rangle$             |
|          |                       |        | $|1\rangle$            | $i|1\rangle$            |
| T        | \[
| 1 & 0 \\
| 0 & e^{i\pi/4} 
|\] | T       | $|0\rangle$            | $|0\rangle$             |
|          |                       |        | $|1\rangle$            | $(\frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}})|1\rangle$ |

2.3. Mutation Testing

Mutation testing is a fault-based testing technique that modifies a portion of program code deliberately for induced errors in the original program. Test cases from the test suite are executed one by one to ensure that our test suite reveals the induced error or not. Given the original program $P$, a slight change in the original program has been introduced, and the resulting program is known as a mutated program $P'$. For example, the left side of Fig. 1 is a Qiskit program for Phase kickback. By changing the statement 6, circuit.h(qreg.q[0]) to circuit.x(qreg.q[0]) in the original program, a mutant is obtained, resulting in a mutated program shown on the right side of Fig.
<table>
<thead>
<tr>
<th>Gate</th>
<th>Matrix Representation</th>
<th>Figure</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
</table>
| CNOT          | \[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\] | ![CNOT Diagram] | ![CNOT Input Output] | | |
| Controlled-Z  | \[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 \\
\end{bmatrix}
\] | ![Controlled-Z Diagram] | ![Controlled-Z Input Output] | | |
| Swap Gate     | \[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\] | ![Swap Gate Diagram] | ![Swap Gate Input Output] | | |

1. Matrix equivalent of the original program is \[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{bmatrix}
\] and mutated program is \[
\begin{bmatrix}
0.707 & 0.707 & 0 & 0 \\
-0.707 & 0.707 & 0 & 0 \\
0 & 0 & -0.707 & 0.707 \\
0 & 0 & 0.707 & 0.707 \\
\end{bmatrix}
\], causing the outcome of the original and mutated program to differ.

Mutants can be classified as live, killed, or equivalent mutants. Any test
case cannot kill an equivalent Mutant. The original program and mutated program (generated by induction of an equivalent mutant in the original program) will be semantically equivalent and can not be differentiated by any test case. If the outcome of the mutated and original program differ by any one of the test cases from the test suite, then the mutant is called a killed mutant. The remaining mutants are classified into a live or equivalent mutants. The original and mutated programs are semantically different for live mutants, but our test cases are not strong enough to differentiate between the original and mutated programs. Fig. 2 represents the quantum circuit (the entanglement of three qubits) and its mutated version. Various researchers [21, 28, 29, 30] proposed the concepts of mutants (Gate Insertion/deletion, Gate Replacement, Gate Swap) in the Qiskit programming.

Mutation score helps in determining the quality of test suite T. The mutation score is computed by

\[
\text{Mutation Score (MS)} = \frac{K}{T - NE}, \quad \text{where}
\]

- \(K\) : # of Killed Mutants
- \(T\) : # of Mutants
3. Mutation Testing for Quantum Software

In this section, we will explain the concepts of mutation testing for quantum software. DeMillo and Offutt [37] specified the three conditions (Reachability, Necessity, and Sufficiency) to be satisfied for test case t must be satisfied to differentiate original program P and Mutated program P’. Consider the statement $S_i$ of the original program P where mutant needs to be introduced. The followings are the three conditions for the mutation testing of quantum software:

- **Reachability:** On executing the quantum mutated program P’ with test case t, the control must reach the mutated statement ($S'_i$) in P’.

- **Necessity:** The matrix representation of the gate operations of $S'_i$ in P’ must be different from the statement $S_i$ in P.

- **Sufficiency:** Matrix representation of $S'_i$ in mutated program P’ must be propagated to the end of the execution such that the outcome of original program P and mutated program P’ must be different, i.e., $O(P(t)) \neq O(P'(t))$. 

$NE : \# \text{ of Equivalent Mutants}$
Fig. 3 represents the mutation testing process for quantum software. The followings are the steps used in the mutation testing of quantum software:

- **Identification of Mutant Operators**: In this step, we need to list mutants for a particular quantum programming language. We are considering the Qiskit programming language for performing mutation testing of quantum software.

- **Mutant Generation**: Mutant generation is a critical step of mutation
testing. A mutated program version $P'$ is created from the original program $P$ by adding a mutant from the list of mutants.

- **Identification of Equivalent Mutant:** Every quantum software runs several times (several shots) to observe the outcomes. Running a Mutated program on the actual Noisy Intermediate-Scale Quantum (NISQ) hardware on the cloud incurs cost, time, and effort.

  In classical software testing, a mutated program is executed against all test cases from the test suite. If the mutated and original program result is the same for all test cases, then the mutant is either a live or equivalent mutant.

  We will first check the equivalent mutant in quantum software to save time, cost, and effort. To find the equivalent mutant, we run the mutated and original software on the simulator and find the matrix equivalence of the mutated and original software. For a details process of finding equivalent mutants, refer to Section 4.

- **Execution of Mutated and Original Software on NISQ Hardware:** Even running the same original software two times may result in a minor difference in the outcomes as quantum NISQ hardware is prone to error. Original and Mutated software executed on NISQ hardware for the same number of shots. We need to analyze the outcome of original and mutated software for the same number of shots to declare whether the mutant is live or killed. There is a chance that the outcome of the original and mutated program for several shots will not match exactly due to the probabilistic nature of the results. We need to analyze the outcome, and a permissible difference between the original and mutated program should be allowed.

  The mutation score will be computed automatically based on the result of mutated quantum software and the expected outcomes of quantum software.

4. Algorithm for Finding Equivalent Mutants

Circuit C describes the specification for the original quantum software. Consider Circuit C consists of Unitary operations on the quantum system at each time click described by Unitary operations $U_1, U_2, ..., U_n$. Consider the
initial quantum state $|\psi_0\rangle$ represents any valid possible input. Matrices representation of Circuit C is $U_n U_{n-1} ... U_2 U_1 |\psi_0\rangle$. Fig. 4 represents the unitary operation for gate-based quantum software.

![Diagram of unitary operation on gate-based quantum software](image)

Figure 4: Unitary operation on Gate-based Quantum Software

Circuit C’ describes the specification of the Mutated quantum program. Circuit C’ consists of Unitary operations on quantum system at each time click described by modified unitary operations $U'_1, U'_2, ..., U'_n$. Matrices representation of Circuit C’ is $U'_n U'_{n-1} ... U'_2 U'_1 |\psi_0\rangle$. Algorithm 1 represents the procedure to find an equivalent mutant.

**Algorithm 1** Procedure to Find Equivalent Mutant (Circuit C, Circuit C’)

**Step 1:** Matrices representation of Circuit C representing the specification of original quantum software is $M_1 = U_n U_{n-1} ... U_2 U_1 |\psi_0\rangle$.

**Step 2:** Matrices representation of Circuit C’ representing the specification of mutated quantum software is $M_2 = U'_n U'_{n-1} ... U'_2 U'_1 |\psi_0\rangle$.

**Step 3:**

if $(M_1 == M_2)$ then
  Declare Equivalent Mutant;
else
  Declare Non-equivalent Mutant;
end

We developed a Qiskit tool for the equivalent mutants in quantum software. Fig. 5 and Fig. 6 represent snapshots of equivalent and non-equivalent mutants.
Figure 5: Snapshot of Equivalent testing tool a) Original Circuit b) Mutated Circuit c) Example showing that mutant is not an equivalent mutant

[26]:
```python
if np.array_equal(originalunitary, mutatedunitary):
    print("Equivalent Mutant")
else: print("Not an Equivalent Mutant")
```

Not an Equivalent Mutant
5. Concluding Remarks

In this paper, we proposed the modified mutation testing approach for quantum software and developed a tool for identifying the equivalent mutant. We believe that the finding of equivalent mutants and the modification in the mutation testing process will be helpful for researchers and quantum software developers to ensure the quality of quantum software. The proposed approach will help in the reduction of cost, time and efforts for performing the mutation testing. In future, we would like to develop a tool for performing the proposed mutation testing approach.

CONFLICT OF INTEREST

The authors declare no competing financial interest.
DATA AVAILABILITY STATEMENT

Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

References


