METAMORPHIC TESTING IN QUANTUM COMPUTING

SINHUÉ GARCÍA GIL

FORMAL METHODS IN COMPUTER SCIENCE AND ENGINEERING
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
COMPLUTENSE UNIVERSITY OF MADRID

MASTER DEGREE THESIS

Year 2023-2024

Thesis directors:

Luis Fernando Llana Díaz

José Ignacio Requeno Jarabo

Madrid, 5^{th} September 2024

Thesis grade: 10/10 (MH)

Abstract

Quantum computing has been on the rise in recent years, evidenced by a surge in publications on quantum software engineering and testing. Progress in quantum hardware has also been notable, with the introduction of impressive systems like Condor boasting 1121 qubits, and IBM Quantum System Two, which employs three 133-qubit Heron processors. As this technology edges closer to practical application, ensuring the efficacy of our software becomes imperative. Mutation testing, a well-established technique in classical computing, emerges as a valuable approach in this context.

In this work, we aim to introduce metamorphic testing for quantum programs. To achieve this goal, we will develop QCRMut, a mutation tool specifically designed for quantum programs, leveraging the inherent structure of Quantum Circuits. Additionally, we will establish an automated framework for mutation testing. This framework will help us define our metamorphic rules and facilitate a systematic testing approach for quantum programs. By automating the testing process, we can efficiently identify potential faults and improve the reliability of quantum software.

Keywords: Quantum computing, mutation testing, metamorphic testing.

Contents

Chapter 1

Introduction

The following document and work will guide us through the journey from the basic principles of quantum mechanics and how they directly define quantum computing, to one of the possibilities we have for testing these programs. Before continuing with the methodology, let us see where it all began.

Quantum mechanics began to develop in the 1920s, but it was not until the 1980s that the possibility of applying this theory to computing started to be considered [1]. In 1980, Paul Benioff presented the quantum Turing machine [2], which used quantum theory to describe a simplified computer. In 1984, this theory was applied to cryptographic protocols by Charles Bennett and Gilles Brassard [3].

Since then, new algorithms have emerged, mainly to solve the oracle problem. Among them are the algorithms of Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, and Simon. Here we can already see the improvement in efficiency compared to the classical computer, as Richard Feynman conjectured in 1982 [4]. It could be said that what sparked the interest of the rest of the community in the potential importance of quantum computing happened thanks to Peter Shor in 1994 [5], with his algorithms that could break RSA and Diffie-Hellman encryption keys, which are still used today. Since then, investment and interest in the study of this field have been growing, and in 1998 the milestone of building the first two-qubit computer was achieved, making this possibility feasible.

IBM presented in 2023 IBM Condor, a 1,121 superconducting qubit quantum processor, currently the largest system in terms of the number of qubits. They also presented Heron, a high-performing quantum processor with 133 qubits, followed by IBM Quantum System Two, a modular-architecture quantum computing platform used to realise parallel circuit executions for quantum-centric supercomputing. This year, they presented their first stable version of the Qiskit SDK, Qiskit 1, with several changes from their previous versions.

With all this evolution occurring regarding the number of qubits and the improvements being made to achieve greater reliability, the door is beginning to open to the real use of quantum computing. However, how are we going to verify their correctness? One of the possibilities we propose in this work is how the union between quantum computing and metamorphic testing can help us answer this question. Metamorphic testing is one of the methods used for error detection since its introduction in 1998, and it is based on the study of the necessary properties that can be derived from the algorithms. We are going to use mutation testing to show the quality of the metamorphic rules proposed.

1.1 Methodology

The methodology followed in this work can be distinguished into three phases, which will be reflected in the following chapters of this document. The primary source used as the foundation has been the book Quantum Computation and Quantum Information by Michael A. Nielsen and Isaac L. Chuang [6], supported for algorithm checks by the book Quantum Computing for Computer Scientists by Noson S. Yanofsky and Mirco A. Mannucci [1], and IBM Quantum official documentation[1]. Next, we will briefly introduce each of these phases:

- **Background and Testing in Quantum**: We will start with a brief introduction to metamorphic testing and mutation testing, along with some of their applications in the classical realm. Then, we will introduce quantum computing and provide a state of the art overview of testing in quantum computing. For this state of the art review, we have primarily used publications found on Google Scholar. We began with three key papers related to our topic: *MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform* by Paltenghi, M. et al. [7], *Mutation Testing of Quantum Programs: A Case Study With Qiskit* by Fortunato, D. et al. [8], and *Metamorphic Testing of Oracle Quantum Programs* by Abreu et al. [9]. We then reviewed and expanded upon papers cited by these works or those that cite them.

- **Tool Development**: After a thorough review of the state of the art, we focused on identifying the tools needed to demonstrate the utility of our metamorphic rules. We decided to use mutation testing for this purpose. The question then arises: Can we use any of the tools we reviewed? This chapter will address that question and explain how we adapt and develop the necessary tools.

[1] `https://docs.quantum.ibm.com/`

- **Metamorphic Testing**: Finally, once we have all resources needed, we will analyse the metamorphic rules defined in previous work. We will review these rules and explain how we adapt them to our quantum circuits. This will be followed by automated testing and our analysis of the results. Will the metamorphic rules provide us with a powerful tool for testing in quantum computing?

1.2 Goals

Once the methodology and materials have been analysed, we will follow the same structure to analyse the objectives. It is important to highlight that the main objective of this work is to study the application of metamorphic testing in quantum computing. However, to reach this goal, we will accomplish a series of objectives and acquire secondary skills in each phase of the work.

- **Background and Testing in Quantum**: We will assume basic knowledge of quantum physics and quantum computing, while introducing some fundamentals to provide a brief background for our work. Our main objective is to gain a comprehensive understanding of the current state of quantum computing testing, acquire a broader perspective, and identify existing gaps. This groundwork should help us understand the research path and its insights, offering us methodologies, best practices, and potentially new ideas to apply to quantum software engineering.

- **Tool Development**: This phase should enable us to create or adapt new tools to meet our expectations and needs. It will allow us to explore Python and Qiskit in depth, likely involving the examination of various libraries and their usage. As we develop a new tool, we will compare it to existing tools, highlighting its unique features and how it better addresses our requirements.

- **Metamorphic Testing**: We will combine metamorphic testing with mutation testing for quantum programs. This approach will enable us to evaluate the effectiveness of our metamorphic rules. We will also assess the limitations of our approach, considering that we are working within the simulation domain of quantum computing and contending with the inherent exponential computational complexity of quantum simulations.

Chapter 2

Background

2.1 Metamorphic testing

Informally, we understand the term metamorphic rule, hereafter referred to as MR, as one that can be logically derived from a definition or specification. Let us start with an example to illustrate this concept, focusing on the sine function, $f(x) = \sin(x)$.

The definition we learn first when studying trigonometry is that the sine function represents the ratio of the opposite side of the angle to the hypotenuse in a right triangle. With this image in mind, it is easy to see that $\sin(x) = \sin(x + 2\pi) = \sin(\pi - x)$. These are two MRs of the sine function. However, we could obtain different MRs from other definitions.

Now, let us consider what we formally define as an MR and how we proceed with metamorphic testing, hereafter referred to as MT. This MT background review is based on *Metamorphic Testing: A Review of Challenges and Opportunities* by Chen et al. [10]:

- **Metamorphic Relationship, MR**: Let $f : X \to Y$ be a function. We consider $\mathcal{R} \subseteq X^n \times Y^n$ to be a **metamorphic rule** if it is a relationship between a sequence of inputs $\langle x_1, x_2, \ldots, x_n \rangle$ with $n > 1$ and their corresponding outputs $\langle f(x_1), f(x_2), \ldots, f(x_n) \rangle$. In other words, it is a necessary property of f.

- **Source/Follow-up Input**: Let $\mathcal{R}$ be a metamorphic relationship and let $\langle x_1, x_2, \ldots, x_k \rangle$ be the original sequence with their respective results. We denote $\langle x_1, x_2, \ldots, x_k \rangle$ as the **source input**, which is defined or characterised data, i.e., already known. In turn, we can generate $\langle x_{k+1}, x_{k+2}, \ldots, x_n \rangle$, which are constructed based on the original input and even on its output. We call this sequence the **follow-up input**.

- **Metamorphic Input Group**: We call the sequence defined by the source and follow-up inputs the **metamorphic input group**, i.e., $\langle x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_n \rangle$.

- **Metamorphic Testing (MT)**: Let f be the target function, P an implementation of f, and $\mathcal{R}$ a metamorphic rule of f.

  To perform **metamorphic testing** on P, we will follow these steps:

  - Define $\mathcal{R}'$ by replacing f with P in $\mathcal{R}$.
  - Given the source input, generate their outputs according to P, construct from these the follow-up input $\langle x_{k+1}, ..., x_n \rangle$, and obtain $\langle P(x_{k+1}), ..., P(x_n) \rangle$.
  - Study the results obtained with respect to $\mathcal{R}'$. If $\mathcal{R}'$ is not satisfied, then we can assert that P is not correct.

The strategy presented above for MT will be followed in the implementation of the MRs obtained for our algorithms. Let us briefly review some advantages and challenges associated with the approach we are taking to verify the correctness, or rather identify the incorrectness, of an algorithm.

Advantages:

- **Simplicity**. The concept and interpretation of an MR are quite simple compared to other concepts used within the field of testing. Studies have shown that even individuals with little experience can use these techniques relatively effectively in a short amount of time [10, 11, 12].
- **Ease of implementation**. Building on the previous advantage of simplicity in definition, the implementation of this test essentially involves following the steps outlined in the definition of MT.

Challenges:

- **Effective generation of metamorphic input groups**. It is still under investigation what guarantees we have regarding the effectiveness of the chosen metamorphic input group in demonstrating the correctness of an algorithm, particularly in how we obtain our source input, since follow-up input is generated based on it. Our goal is to maximise the identification of errors or defects in P.
- **Structure of MT**. Due to the wide variety of MRs, there is still no consensus on a defined and formal structure that provides security in our tests and encompasses all the possibilities within MRs. However, it is important to note that MT has been

useful in identifying various errors in systems extensively studied with other testing methods, such as the GCC[1] and LLVM[2] C compilers, where more than 100 errors were found [10, 13, 14, 15].

The challenges of MT can serve as a solid foundation for future work in this field and highlight the possibilities it can offer. To conclude this section on MT, I would like to provide a couple of examples demonstrating the wide applicability of this testing method.

- **Chess Engines**: The work presented by Méndez et al. [16], titled *Metamorphic Testing of Chess Engines*, applies metamorphic testing to the chess engine Stockfish[3]. The authors defined four different MRs they applied to Stockfish, leading them to discover problems in the engine. The number of problems found increases as the engine's depth increases.

- **OpenStreetMap,OSM**[4]: MT has also been applied to OSM. The first significant work was *Metamorphic Testing of OpenStreetMap* by Almendros-Jiménez et al. [17] in 2021, where the authors presented nine MRs to check for errors in the map. Some examples include checks for isolated ways or overlapping between buildings and ways. They analysed different cities, finding several issues with all their MRs. Their subsequent research in 2023 [18] aimed at checking the quality of tags. Additionally, Méndez et al. [19] combined MT and machine learning to enhance OSM.

2.2 Mutation testing

Mutation testing is a technique in which small changes are introduced into the code to simulate potential errors. The test suite is then executed to determine if it can detect these mutants. Over the years, extensive research has been conducted in this field, and a comprehensive collection of advancements is provided by Papadakis et al. in their 2019 study, *Mutation Testing Advances: An Analysis and Survey* [20], which is one of the most recent and authoritative surveys on mutation testing, defining all the essential concepts.

[1] `https://gcc.gnu.org/`
[2] `https://llvm.org/`
[3] `https://stockfishchess.org/`
[4] `https://www.openstreetmap.org/`

To further elaborate on some of these concepts related to mutation testing, I will complement them with notes from Professors Núñez M. and Merayo M. at Complutense University of Madrid, created for the course on *Formal Approaches to Testing*.

- **Mutant**: A mutant is a version of the original program where modifications have been inserted. These modifications can be syntactically correct or incorrect.
- **Mutation Operator**: A mutation operator is the rule by which we syntactically change the original algorithm to produce the mutant. This change can be applied to an operator or a variable. Offutt et al., in their work *An Experimental Determination of Sufficient Mutant Operators* [21], propose a five-operator set as a minimum standard for mutation testing. This set is composed of the following mutation operators: absolute value insertion, arithmetic operator replacement, logical connector replacement, relational operator replacement, and unary operator insertion.
- **Killing Mutants**: We consider a mutant to be killed if the output under a test produced by the mutant is different from the output of the original algorithm under the same test.
- **Equivalent Mutants**: No test can kill an equivalent mutant as it behaves like the original algorithm.
- **Mutation Score**: It is defined as the ratio of killed mutants to the total number of mutants. Alternatively, we can consider it as the ratio between killed mutants and non-equivalent mutants, which will always be higher or equal to the previously defined ratio.
- **Mutation testing process steps**:
  - **Generate Mutants**: Create the desired mutants. If possible, eliminate equivalent mutants before proceeding to the next step.
  - **Generate Test Cases**: Develop a comprehensive set of test cases to evaluate the mutants.
  - **Execution and Analysis**: Execute the mutants and analyse their results. If desired or possible, increase the number of test cases and re-run the remaining non-killed mutants to improve the results.

To effectively asses mutation testing we need to use well defined rules based on syntactic descriptions to make systematic changes to the syntax or to objects developed from the syntax. Let us observe some advantages and challenges that mutation testing provide us.

Advantages

- **Represents programmer faults**: The use of mutation operators tends to represent possible faults introduced during the implementation of the algorithm, allowing us to test for potential real faults.
- **Test suite quality**: Mutation testing provides information about the quality of our test suites. This helps us produce an effective set of tests to run on our algorithm, as these tests will have already been proven to detect faults in the program.

Challenges

- **Equivalent Mutants**: As explained previously, we aim to reduce the number of equivalent mutants in our set of mutants, as they waste resources. Recall that an equivalent mutant cannot be killed since its behaviour is identical to the original algorithm. However, detecting equivalent mutants is an undecidable problem [22]. One approach to address this challenge is the use of compiler optimisation techniques, as proposed by Offutt et al. in their work *Using Compiler Optimization Techniques to Detect Equivalent Mutants* [23].
- **Number of Mutants**: The number of mutants produced for a program can be very large, making reduction necessary. Strategies aim to select a representative subset of mutants, addressing the issue of large mutant sets. This leads to an interesting topic explored by Gopinath et al. in [24, 25], where they examine the limits and challenges of such reduction techniques.

2.3 Quantum computing

Finally, we will overview and provide a brief review of quantum computing. The foundation of quantum computing originates from quantum physics; let us briefly recall these postulates [1, 6] and observe how quantum computing directly derives from the postulates of quantum physics. We can categorise these four postulates into two distinct groups:

Kinematic or representation postulates:

- **First postulate**: The state of an isolated system at a given moment t is represented by $|\varphi(t)\rangle$ in a complex Hilbert space, $\mathcal{H}$.
- **Second postulate**: The space representing a quantum composite system is the tensor product of the spaces corresponding to each component system. Therefore, if we have n components, the space factors as $\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2 \otimes \cdots \otimes \mathcal{H}_n$.

Dynamic postulates:

- **Third postulate, deterministic evolution**: The evolution of a quantum system's state is determined by the Schrödinger equation, $\imath\hbar\frac{d|\varphi\rangle}{dt} = H|\varphi\rangle$, where H is the system's Hamiltonian. In an isolated system, the evolution can be described as a unitary transformation from the initial state.
- **Fourth postulate, system observation, probabilistic evolution**: To observe a quantum system and gain information from it, we need to measure the system.
  - Each measurement $\mathcal{A}$ is described by a Hermitian operator A acting on H. We say that this operator is an observable if its eigenvectors form a basis of $\mathcal{H}$. The result of measuring a quantity $\mathcal{A}$ must be one of the eigenvalues corresponding to the observable A. It can be identify as a positive operator-valued measure.
  - The probability of obtaining an eigenvalue λ with eigenvector v if the quantum state is $|\varphi(t)\rangle$ is:

$$Prob(\lambda) = \frac{|| \, P_{|v\rangle}|\varphi(t)\rangle \, ||^2}{|| \, |\varphi(t)\rangle \, ||^2} = \frac{|\, \langle v|\, \varphi(t)\rangle \, |^2}{|| \, |\varphi(t)\rangle \, ||^2} \tag{2.1}$$

  - If, after performing a measurement $\mathcal{A}$ on the state $|\varphi(t)\rangle$, the result is λ, then the state of the system collapses to the normalised projection of $|\varphi(t)\rangle$ onto the subspace of eigenvectors associated with λ, $P_{|\lambda\rangle}|\varphi(t)\rangle$.

We should now focus on quantum computing and how these postulates provide us with the basic components needed to proceed with quantum programming. Yanofsky and Mannucci [1] state that quantum programming is based on creating a quantum circuit or algorithm, typically represented geometrically, in which operations are performed with unitary

operators on different qubits, as well as measurements. Let us introduce some basic concepts of quantum computing, which can be related to the previously presented postulates.

- **Qubit**: A qubit, or quantum bit, is the basic unit of quantum information. It is defined as the simplest quantum system, represented by a two-dimensional complex Hilbert space, $\mathcal{H}$. This definition derives from the first postulate of quantum physics. The state of a qubit is represented by a ket $|\varphi(t)\rangle \in \mathcal{H}$, where $||\varphi(t)\rangle| = 1$.

- **Computational Basis of a Qubit**: The computational basis is defined by the kets $|0\rangle = \begin{bmatrix}1 & 0\end{bmatrix}^t$ and $|1\rangle = \begin{bmatrix}0 & 1\end{bmatrix}^t$, which form an orthonormal basis for $\mathcal{H}$.

- **Superposition**: A qubit's state is in superposition if it is a linear combination of the basis states. Specifically, if we consider the computational basis, $|\varphi\rangle = a\,|0\rangle + b\,|1\rangle = \begin{bmatrix}a & b\end{bmatrix}^t$, where $a, b \in \mathbb{C}$ and $|a|^2 + |b|^2 = 1$.

- **Amplitudes**: The coefficients of the superposition representation are known as amplitudes. For the state $|\varphi\rangle$ in the computational basis, the amplitudes are a and b.

- **Entanglement**: A system of multiple qubits is said to be entangled if its state cannot be represented as a tensor product of the individual qubit states. The most common example is one of the Bell states, which is represented by the equation 2.2.

$$|\varphi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{2.2}$$

If we consider the second postulate and how we generate composite quantum systems, we should note, which will be important for this work, that the basis of the quantum system grows exponentially with the number of qubits. Let us introduce some definitions on how we can change, evolve or observe the quantum system, along with some important results.

- **Unitary Gate**: A unitary gate is an operation on the quantum state of a system, defined by a unitary operator corresponding to a specific Hamiltonian that dictates the deterministic evolution of the quantum system. Examples include the Pauli gates and the CNOT gate.

- **No-Cloning Theorem** [6]: If $\mathcal{H}$ is a complex Hilbert space and $|\psi\rangle \in \mathcal{H}$, then there does not exist a unitary operator U over $\mathcal{H} \otimes \mathcal{H}$ such that $U|\varphi\rangle|\psi\rangle = |\varphi\rangle|\varphi\rangle$ for all $|\varphi\rangle \in \mathcal{H}$.

- **Universal Gate Set**: A universal gate set is a collection of gates such that any unitary operation can be approximated by a sequence of these gates. Although it is theoretically impossible to provide a finite basis for an infinite set, the Solovay-Kitaev theorem guarantees that any unitary operation can be approximated to a given precision efficiently using a universal gate set.

- **Measurement Gate**: Defined by the quantum measurement postulate, it allows us to extract information from the quantum system and store it in classical bits. However, the act of measurement modifies the quantum state.

Now that we have all the necessary building blocks for our quantum circuits, let us define them and explain how we will execute these algorithms.

Quantum Circuit: A quantum circuit is a sequence of quantum gates applied to a quantum system on an initial state.

Execution: We can execute our quantum circuits in two different ways: simulation or on a real quantum system. As previously explained, when we observe or measure a quantum circuit, we obtain a single eigenvalue. To derive the distribution that represents our quantum state at the measurement point, we repeat the same quantum circuit multiple times. The number of repetitions can be defined by the user, and throughout this work, we will observe that the number of repetitions is crucial, as it can significantly impact the accuracy of the results. Let us briefly define the two execution methods:

- Simulation: A quantum circuit can be simulated using classical computers. The quantum circuit consists of a sequence of quantum gates, each represented by a matrix. Simulation provides an exact solution by operating with these matrices. It is important to note that the size of these matrices grows exponentially with the number of qubits in the simulated circuit. Specifically, if a circuit has n qubits, the matrices will have a size of $2^n \times 2^n$.
- Real Quantum Systems: We can execute our quantum circuit on real quantum systems, in our case, we will use IBM Quantum systems. Each system has its own qubit structure and universal gate set, meaning the operations that a real quantum system can perform are restricted to those implemented in its gate set. Consequently, our quantum circuit will be transformed into an equivalent circuit within the system's gate set before execution. These systems are susceptible to operational errors and noise.

Noise and Gate Errors: Operations in real quantum systems are prone to errors that deviate from the expected theoretical values. However, errors are not the only challenge, noise can also affect the state of our system. Noise refers to any interference from the environment on our quantum system.

To execute our quantum programs and simulations, we will rely on Qiskit, an open-source development package created by IBM for the creation and manipulation of quantum programs, as well as for performing simulations[5]. This can be done either through simulation or by connecting our programs to IBM's quantum computers, which will provide more realistic results and allow us to observe the noise present in these computers today. Now, let us introduce a quick example to bring all of this together.

We are going to execute a simple circuit that represents the Bernstein-Vazirani, BV, algorithm, which will be one of the algorithms tested in this work. The algorithm solves an oracle problem that encapsulates a function $f(x) = x \cdot s$, where $\cdot$ denotes the dot product and s is the secret string we aim to obtain. The algorithm successfully retrieves this string. We will explore the algorithm in greater detail later in this work. The Figure 2.1 represents the algorithm with an oracle that implements this function for the string $s = 1010$.

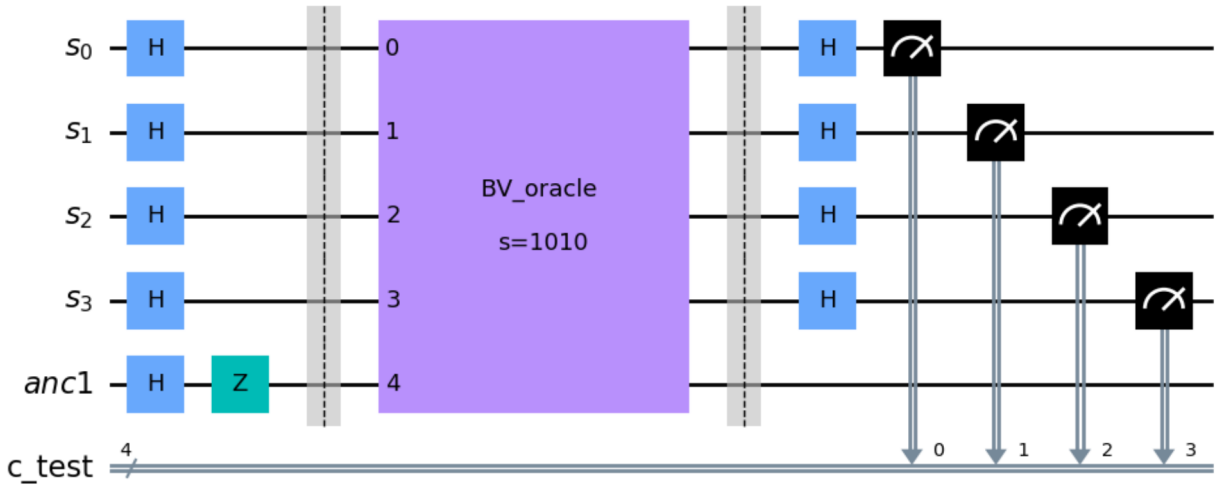
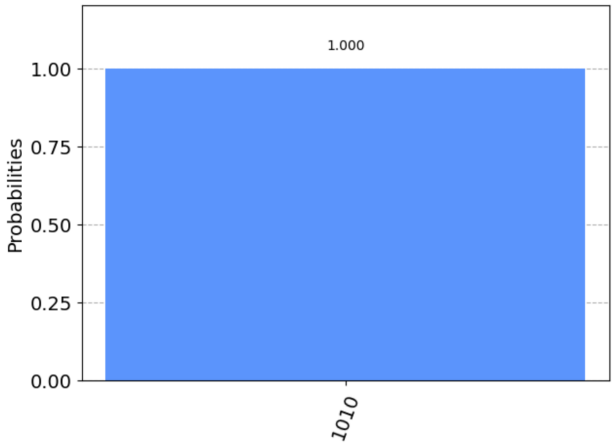


Figure 2.1: BV circuit

Let us execute it in both ways to see the differences. In Figure 2.2, we can see the results from the simulator execution compared to those obtained on the *ibm_kyiv* quantum system. In our simulation, we obtain the only possible result: **s**. However, on the real quantum system, we observe some undesired outcomes. It is important to note that not

[5] https://qiskit.org/

all quantum algorithms have only one possible outcome; we used this algorithm due to its simplicity in terms of results.



(a) Simulation

Results	Count	Probability
1010	1000	0.97656
1000	3	0.00293
0010	20	0.01953
0000	1	0.00098

(b) IBM_Kyiv

Figure 2.2: Results: Simulation vs IBM_Kyiv.

However, let us examine the process for *ibm_kyiv*. First, we can observe its structure on the error map[6] in Figure 2.4; the darker the colour, the lower the error it represents. Qiskit will transform our circuit and adapt it to the quantum system's structure, optimising the qubits assigned for better results. Our circuit, presented in Figure 2.1, will be transpiled and sent to *ibm_kyiv*, as shown in Figure 2.3. It might seem confusing that I only show three qubits instead of five; the other two qubits do not have any gates applied to them. This is due to the nature of the oracle, which does not apply any gate to the qubits represented by 0 on s, in inverse order from s_3 to s_0. The inverse order is inherent to Qiskit, as it organises the computational basis in this way. Since the oracle does not affect qubits s_0 and s_2, these two qubits will only have two Hadamard gates applied to them, which the transpilation process ignored since $HH = I$. This reduces the number of gates applied to the quantum system, helping to manage and control errors.

We can observe that $anc1$ is mapped to qubit 75, s_3 to qubit 76 and s_1 to qubit 90. These may seem like random numbers, but the architecture of the system is crucial, if two qubits do not have physical connectivity, two-qubit gates can not be applied. In Figure 2.4, we can see how this circuit was executed on the lower left half of the system. On the other hand, although not visible in Figure 2.3, s_0 and s_2 are assigned to qubits 6 and 42 and are measured before any gate operations begin.

[6] `https://quantum.ibm.com/services/resources?system=ibm_kyiv`

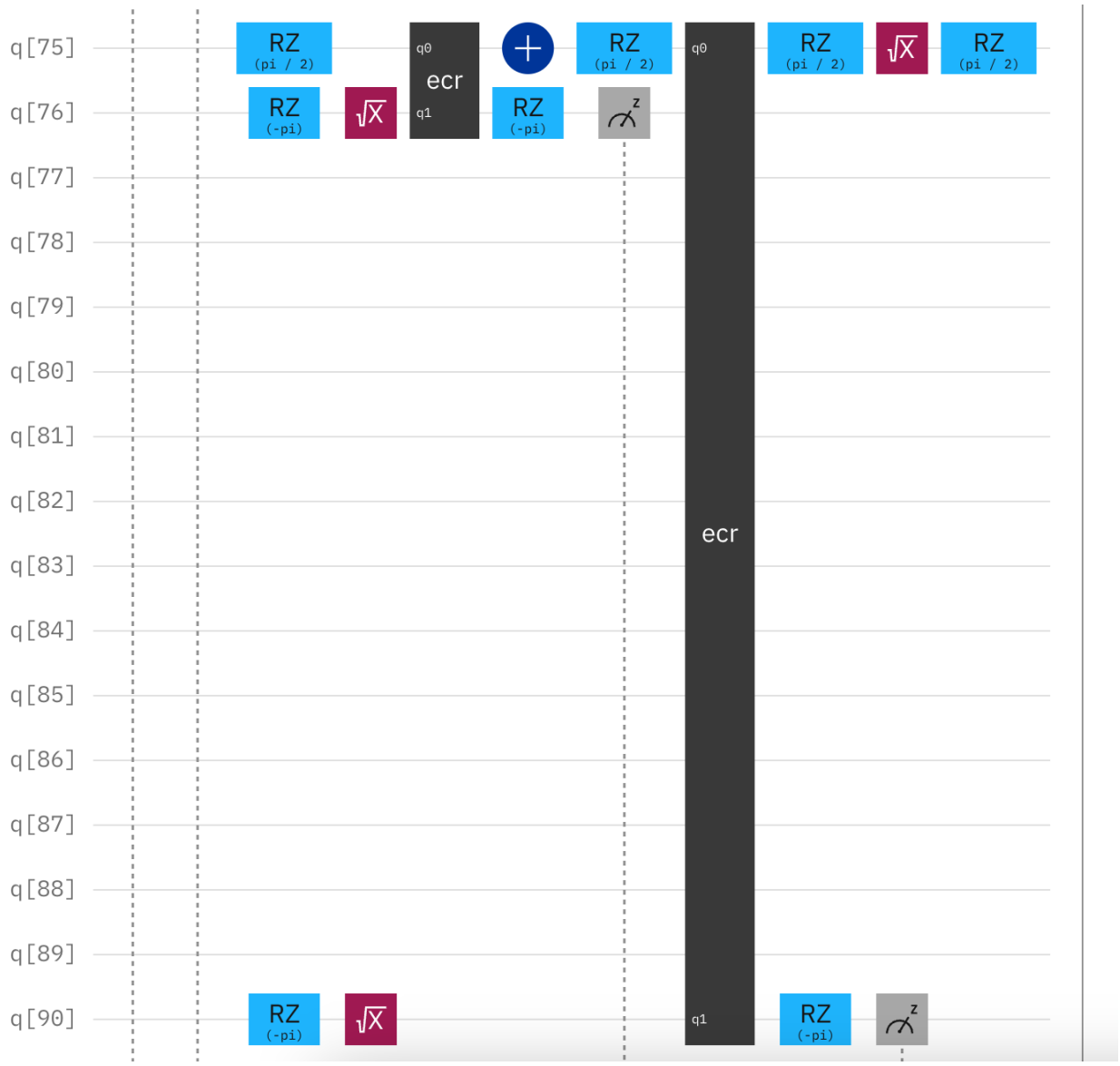


Figure 2.3: Snap of BV circuit adapted to ibm_kyiv.

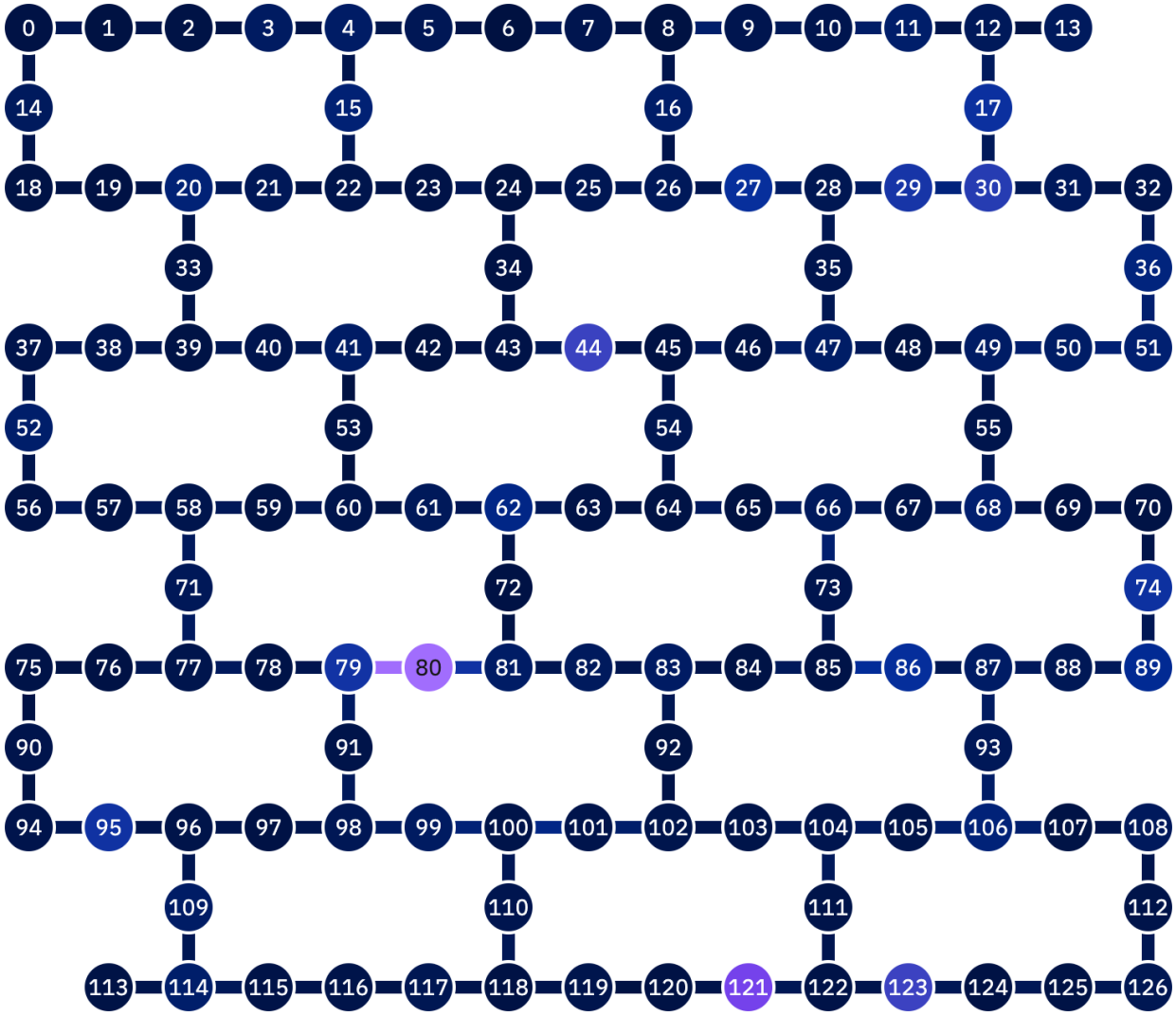


Figure 2.4: ibm_kyiv error map[7].

[7] https://quantum.ibm.com/services/resources?system=ibm_kyiv

Chapter 3

Testing in Quantum

After providing a brief introduction to quantum computing, let us allocate some time to delve into the latest developments in the field of quantum testing. This section will be divided into two different perspectives. Firstly, we will explore the testing of Quantum Platforms, with a specific focus on Qiskit and how it can evolve for different platforms. Subsequently, we will continue with various approaches and techniques for testing quantum programs, hereafter referred to as QP.

3.1 Quantum computing platforms

As mentioned earlier, we will begin by examining the testing of quantum platforms. We will introduce these approaches in chronological order and the authors will even assess how they compare to each other, observing improvements over time.

3.1.1 QDiff

QDiff, differential testing of quantum software stacks (QSS), is the approached presented by Wang et al. in their 2021 article titled *QDiff: differential testing of quantum software stacks* [26], they introduce it as a novel differential testing technique for QSS.

Their motivation behind this work arises from the increasing interest in quantum computing and the development of QSS as a high-level languages for quantum programming. These languages aim to shield users from the mathematical and physical complexities inherent in quantum computing. Another motivation is to offer a potential solution for identifying failures within QSS, which aids in resolving the confusion and incoherence that often arises when a user reports bugs. When reviewing the collection of issues reported for each QSS, these problems frequently reveal inconsistencies in identifying the potential source of the

bug or determining whether the issue originates from the probabilistic or noisy nature of quantum indeterminacy.

The authors idea is to be able to perform testing in the three most widely used QSS [27] at that time: Qiskit, Cirq and Pyquil. Now, let us explore the workflow (Figure 3.1) and key innovations in QDiff, which will include their propose solutions for technical challenges that complicate testing within QSS.

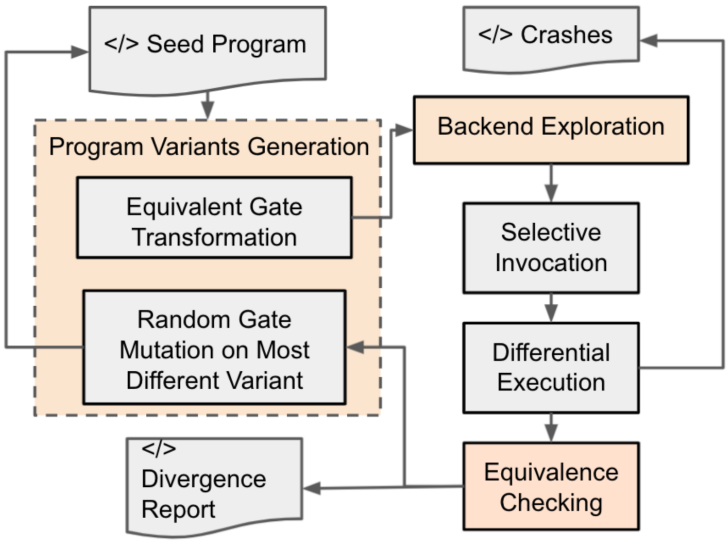


Figure 3.1: QDiff overview [26]

- QP generator: In order to fulfil our needs while testing QSS, we need to generate semantically equivalent programs. The authors proposed a set of equivalent gate transformation rules designed to preserve program semantics. These seven rules have been derived from the analysis of each gate and its behaviour when deployed sequentially.

  Followed by mutation modifications, QDiff utilises this process to diversify the pool of input programs. The initial seed consists of 6 QPs, and through mutation, it will be expanded to 730 variants. Quantum mutant operators used in QDiff:

  - M1: Gate insertion/deletion.
  - M2: Gate change.
  - M3: Gate swap.
  - M4: Qubit change.

Rule ID	Original Construct	Equivalent Construct
G1	SWAP(q_1,q_2)	CNOT(q_1,q_2) CNOT(q_2,q_1) CNOT(q_1,q_2)
G2	H(q_1)H(q_1)	Merged to Identity Matrix
G3	X(q_1)	H(q_1)S(q_1)S(q_1)H(q_1)
G4	Z(q_1)	S(q_1)S(q_1)
G5	CZ(q_1,q_2)	H(q_2)CNOT(q_1.q_2)H(q_2)
G6	CZ(q_1,q_2)CZ(q_1,q_2)	Merged to Identity Matrix
G7	CCNOT(q_1,q_2,q_3)	6 CNOT gates with 9 one-qubit gates

Figure 3.2: Semantic preserving rules [26]

- Filtering QP based on whether they are worth running. This determination relies on the execution system and the specific QP. A circuit is deemed suitable for execution on quantum hardware or a noisy simulator if it adheres to two key thresholds:

  – The maximum number of gates, determined by average execution gate and T1 time for the system, where T1 represents the decoherence time for a qubit.

  – The maximum number of 2-qubit gates, determined by the error rate tolerance willing to be added by the user the final measurements

  A concrete example of these boundaries is provided in *IV. QDIFF APPROACH - B. Quantum Simulation and Hardware Execution* [26].

- Comparing outcomes derived from two equivalent quantum programs. The authors employ two similar techniques discussed in *Equivalence Checking via Distribution Comparison* [26] and prior works [28, 29, 30]. Let us outline both methods.

  Firstly, it is imperative to establish a threshold t representing the maximum acceptable distance and a significance level p. Afterwards, the minimum number of executions required for statistically guaranteed comparison is calculated. Following the executions, empirical distribution functions (EDFs) are computed.

  – Kolmogorov-Smirnov (K-S) distance: This metric characterise the K-S distance as the most substantial difference between the two EDFs. If this difference is less than t, the QPs are deemed similar.

  – Cross entropy: The comparison involves evaluating the total entropy of both EDFs, equally constrained by the threshold t.

- Reporting divergences. Once a divergence has been found, the system will identify the differences. Then it identifies between: backend, frontend or API gate implementation.

We should now conclude by highlighting the key contributions of these experiments:

- QDiff successfully evaluated more than one QSS.
- 14799 QPs were generated, organised into 730 sets of semantically equivalent circuits.
- Among the 730 sets, 33 differing outcomes were identified, all which will undergo manual checking.
  - Out of these, 4 discrepancies resulted in simulator crashes, while the remaining divergences exceeded expected noise levels on IBM hardware.
  - Six distinct sources of instability were identified, comprising 4 software crash bugs in Pyquil and Cirq, along with 2 root causes potentially explaining 25 out of 29 cases of divergence beyond anticipated noise on IBM hardware, attributed to unreliable connections between two qubits.

3.1.2 QSS review

Matteo Paltenghi presented his paper titled *Cross-Platform Testing of Quantum Computing Platforms* in 2022, as documented in [31]. The primary objective of this paper is to delineate the challenges faced by quantum computing platform testing in 2022, offering hypotheses on potential solutions. We could consider this paper as the seed for Paltenghi's subsequent research with MorphQ, the finding of which will be presented upon shortly. Let us explore the challenges and hypotheses outlined by the author.

- **Challenge** 1: Quantum program generation. Platform testing requires a large amount of programs. However, there is only a few real quantum program examples available nowadays [32].
- **Challenge** 2: Multi-platform Program Translator. Quantum programs are often written in new programming languages or expressed using APIs on top of existing languages. The absence of standardised intermediate representations presents a challenge for cross-platform testing. Attempts have been made with *OpenQASM* [33, 34].

- **Challenge** 3: Multivariate Binary Distribution Comparison. The probabilistic nature of quantum measurements introduces complexity to testing, as outputs are represented by distributions. Research has been conducted on this challenge in quantum debugging [35, 36, 37], along with preliminary work using K-S or cross-entropy tests [26].

- **Hypothesis** 1: Cross-platform testing can be a promising direction to find bugs in QC platforms. The authors suggest the existence of bugs in traditional buggy components, such as compilers and optimisers, building on the work presented by Wang et al. [26].

- **Hypothesis** 2: Testing with automatically generated realistic quantum programs could boost the effectiveness of current differential testing on QC platforms.

- **Hypothesis** 3: Machine learning methods could be successful in generating realistic quantum programs. This hypothesis comes from classical computing and how genetic algorithms have been used for program synthesis [38] and reinforcement learning to generate equivalent quantum programs [39].

- **Hypothesis** 4: The quantum programs can be translated to run on multiple platforms.

- **Hypothesis** 5: A statistical test specific for multivariate binary distribution should perform better than existing unspecialised methods.

After outlining these brief ideas about the challenges and potential solutions in quantum platform testing, let us proceed to introduce the subsequent work by the same author.

3.1.3 MorphQ

Matteo Paltenghi and Michael Pradel presented in May 2023 their latest work on testing QQS, introducing metamorphic testing as a novel approach for quantum platforms. Their innovative method involves the design of metamorphic rules for Qiskit.3 Their article, titled: *MorphQ: Metamorphic testing of the qiskit quantum computing platform* [7], further develops this concept, presenting a fresh perspective on generating quantum programs using a defined grammar. This advancement brings us closer to the idea of automating the testing process, eliminating the need for a library of quantum programs as generation seed, which was the traditional approach up to this point.

MorphQ will focus on Qiskit, as quantum platform, employing metamorphic testing to address specific challenges presented in quantum computing. The oracle problem will

be avoided as it will be resolved by the use of MR, eliminating the need for a detailed specification of expected input behaviour. Let us provide a general overview of MorphQ as presented by the authors, then we will focus on the key innovations within each of its distinct components.

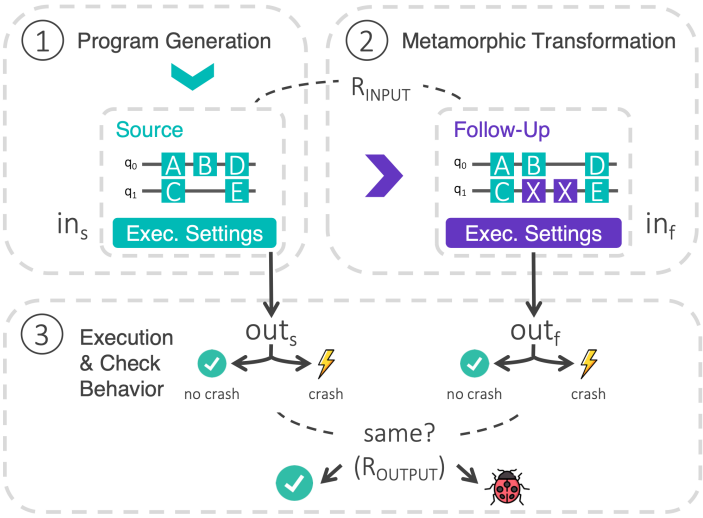


Figure 3.3: MorphQ overview [7]

As previously mentioned, one of the authors' significant contributions lies in their approach to program generation. Since the focus is on testing the quantum platform, the primary objective of this proposal is to create syntactically correct quantum programs, thus preventing execution crashes. These this automating generated programs will serve as test suit for the QSS. To begin, Paltenghi and Pradel introduced the grammar that will guide the program generation process. This grammar follows the typical recursive structure found in computing. A subset of it is illustrated in Figure 3.4, as provided by the authors in their paper. You can access all the related information about the grammar and the rest of the article on their GitHub[1] page.

The purpose of defining this grammar is to ensure that invalid quantum programs are avoided. A random approach would not be ideal, as it would likely generate a high number of invalid programs. Additionally, the authors will impose a constraint on the number of gates per program, limiting it to 30. Higher amount of gates can increase the execution time due to the complexity of quantum operations. Their commitment towards keeping execution times within reasonable limits, lies in the decision of the authors to only use simulators, where the size of the matrix for an operator grows exponentially with the number of qubits.

[1] `https://github.com/sola-st/MorphQ-Quantum-Qiskit-Testing-ICSE-23`

⟨*GATE_OPS*⟩ ::= ⟨*INSTR*⟩⟨*EOL*⟩⟨*GATE_OPS*⟩ | ⟨*EOL*⟩
⟨*INSTR*⟩ ::= ⟨*INSTR_1Q*⟩ | ⟨*INSTR_2Q*⟩ | ... | ⟨*INSTR_5Q*⟩
⟨*INSTR_1Q*⟩ ::= `qc.append(`⟨*GATE_1Q*⟩`,`
`qregs=[qr[`⟨*INT*⟩`]])`
⟨*INSTR_2Q*⟩ ::= `qc.append(`⟨*GATE_2Q*⟩`,`
`qregs=[qr[`⟨*INT*⟩`],qr[`⟨*INT*⟩`])`
⟨*GATE_1Q*⟩ ::= ⟨*HGate*⟩ | ⟨*RZGate*⟩ | ...
⟨*HGate*⟩ ::= `HGate()`
⟨*RZGate*⟩ ::= `RZGate(`⟨*FLOAT*⟩`)`
⟨*GATE_2Q*⟩ ::= ⟨*CXGate*⟩ | ⟨*CRZGate*⟩ | ...
⟨*CXGate*⟩ ::= `CXGate()`
⟨*CRZGate*⟩ ::= `CRZGate(`⟨*FLOAT*⟩`)`
⟨*EOL*⟩ ::= `\n`

Figure 3.4: Subset of the QP generation grammar [7]

Once we have been able to generate the source input for the testing desired, the authors defined the metamorphic rules that Qiskit should fulfil classifying them in 3 categories: Circuit transformation which modify the circuit, representation transformations which change the intermediate representation of QP and execution transformations, which affect the execution environment, we could observe this metamorphic rules in Figure 3.5.

Category	Name	Precondition
Circuit transformation	Change of qubit order	-
	Inject null-effect operation	-
	Add quantum register	Coupling map not fixed
	Inject parameters	-
	Partitioned execution	Non-interacting subsets of qubits
Representation transformation	Roundtrip conversion via QASM	-
Execution transformation	Change of coupling map	No added register
	Change of gate set	-
	Change of optimiz. level	-
	Change of backend	-

Figure 3.5: Qiskit MR used by QMorph [7]

The expected relationship between source and follow-up outputs is equivalence, with the only exception of non-semantic-preserving transformations, specifically, the change of qubit order and partitioned execution which will need a posterior treatment before comparing behaviours. This is why we will continue applying metamorphic transformations unless one

of the exceptions mentioned above is introduced. This transformations are executed through python AST or/and sectioning source code with python matching technique, adding needed elements and reconstructing the code.

In the final step of MorphQ, which involves execution and behaviour checks, the primary assessment is crash check. Identifying a crash in the follow-up program marks a critical failure. As we are all aware, when executing quantum programs without accounting for quantum noises, the result can be non-deterministic linked to the final state amplitudes, reflecting the well-known probabilistic nature of quantum measurement. Consequently, we would need to execute the programs a specific number of shots to compare two different distributions and decide if the output is equivalent or we have possibly found a failure. To determine this required number of executions, the authors follow the same technique as the one used in QDiff, employing the **L1 norm** [28]. The distribution are compared using Kolmogorov-Smirnov test (p-value $< 5\%$).

Another noteworthy aspect is the management of warning messages related to crashes. The authors implemented a semi-automatic clustering approach for these warnings, aiming to abstract from program-specific references. Afterwards, a random selection process is applied to each cluster, and the chosen programs undergo manual inspection. During these inspections, transformations are reversed step by step until the one responsible for the fault is reached. Then, they use **delta debugging** until they identify the minimal sequence of operations to trigger the crash.

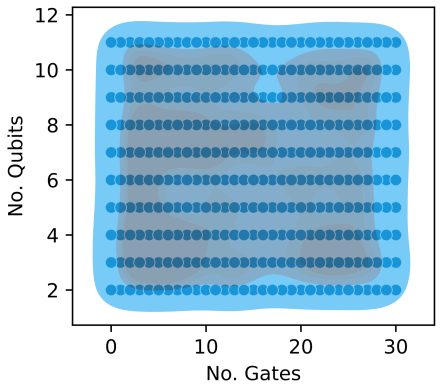


Figure 3.6: QP diversity [7].

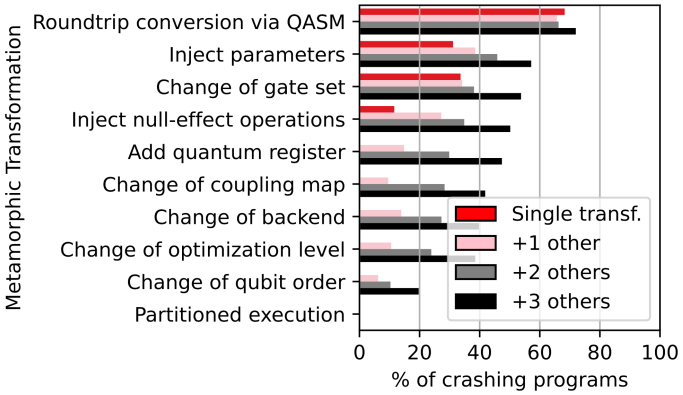


Figure 3.7: MR analysis in follow-up QP [7].

Let us summarise the key results and contributions of MorphQ, keeping in mind that the experiments were constrained by a 48-hour execution time-frame. The authors also adapted QDiff to the same execution time-frame to be able to compare results.

- QP generator:
  - Automatically generated 8360 quantum programs .
  - Source-QP executed without crashing.
  - Wide range of programs produced (Figure 3.6).
  - Higher code coverage vs QDiff: 8.1% vs 6.1%
- 13 bugs discovered in the latest version of Qiskit.
- Follow-up QP behaviour analysis:
  - Crashed in 23.2% of the cases,with only 56 programs showing a distribution difference.
  - Demonstrated wider diversity compared to QDiff, as measured by unique API calls.
  - *Roundtrip conversion via QASM* and *Inject null-effect operations* are the most effective MR, although a combination of several MR will produce better results, exposing 8 out of 13 bugs (Figure 3.6).
  - Non-crashing follow-up QP: Upon evaluating the 56 programs, it was determined that the differences in distribution were due to randomness, which is plausible given the test significance level.
- False positive: MorphQ may generate false positive warnings because the MR do not always hold in practice, even when is theoretically sound. This can occur when applying the *Change of gate set*. In Qiskit, A* algorithm is used to find equivalent gates, although exploring all possibilities is impractical. Therefore, we might encounter the warning *Unable to map source basis to target basis*, which could be recognised as a limitation of the platform.

3.2 Quantum computing programs

After considering some of the latest results regarding testing of quantum platforms and the diverse challenges they have encountered and overcame, we will shift our focus towards testing our QP implementations. This presents a different challenge where we have an algorithm or specification and an implementation that should represent that algorithm. We will explore the recent developments and analyse different techniques used for testing an implementation (QP) of a given algorithm.

We provided some information in the Methodology section 1.1 on how we selected these papers. Additionally, we identified several surveys on quantum testing and quantum software engineering that led us to additional papers, such as *Quantum software testing: State of the art* by García de la Barrera et al. [40] and *The Quantum Frontier of Software Engineering: A Systematic Mapping Study* by De Stefano et al. [41]. However, these surveys offer only brief summaries of each paper, typically no more than a small paragraph.

3.2.1 QSharpCheck

The initial approach we will discuss was presented by S. Honarvar et al. in 2020 with the title: *Property-based Testing of Quantum Programs in Q#* [42]. The authors position this approach as the first step towards structured testing techniques for quantum programs, offering a property-based framework tailored for *Q#*. Their work was influenced by Quantum Hoare Logic [43] and assertion language [36]. The authors introduce a syntax for specifying properties of *Q#* programs along with a tool named QSharpCheck, designed for test case generation, text execution, and analysis of outcomes.

```
1 Transform_Property;
2 (10, 99, 500, 300);
3
4 {q : Qubit (36,72)(0,360)};
5 TransformState(q);
6 [AssertTransformed(q,(108,144)(0,360))];
```

Figure 3.8: Property example, state transformation [42].

Let us delve into the syntax introduced for property specification. This involves associating a test property with a name and parameters, followed by allocation and setup, function call and finally assertion. This structured syntax is illustrated in Figure 3.10, and an example is provided in Figure 3.8.

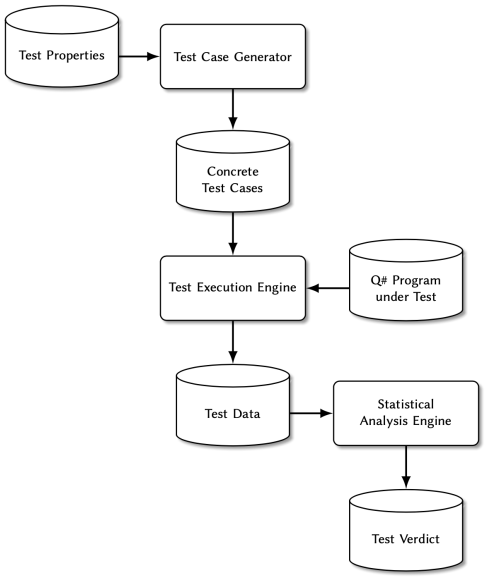


Figure 3.9: QSharpCheck architecture [42].

With the established syntax, we can now introduce QSharpCheck, illustrated in Figure 3.9. We could observe some output examples in Figure 3.11:

- Test case generator. QSharpCheck functions as a test case generator, producing the specified number of test cases randomly within the range specified in the parameters.
- Test execution Engine. This component repeats the execution as many times as required by the specification.
- Statistical Analysis Engine. QSharpCheck incorporates a statistical analysis engine that performs hypothesis testing, considering the last assertion in the test as the null hypothesis. It assumes the program follows a binomial distribution, and in cases where the user does not provide a confidence level, it defaults to 0.99.

The authors have designed QSharpCheck to accommodate various types of assertions as methods. Let us explore the types of assertions that can be used within this tool.

- AssertProbability. This assertion observes if a given qubit can reach a specified measurement with a designated probability.
- AssertEntangled. Given two qubits as input, this assertion tests whether or not they are entangled.

- AssertEqual. As the name suggests, this assertion checks whether two qubits are equal. The null hypothesis is defined based on the equality of the qubits.
- AssertTeleported. Specifically designed for testing teleportation, this method takes the sent and received qubits as arguments. It analyses the probability found on the sent qubit and executes the *AssertProbability* on the received qubit.
- AssertTransformed, This assertion is employed for unitary transformations.

```
//Preamble
Property name;
(numberOfTestCases, confidenceLevel,
     numberOfMeasurements, trials);

//Precondition and initialisation
{q_0...q_n : Qubit (θ interval)(φ interval)};
{b_0...b_n : Bool};

//Invoking system under test
operation_name (arguments) : return type;

//Postcondition
//Any combination of the following assertions
[AssertProbability(q_0,0,proposed propability)];
[AssertEntangled(q_0,q_1)];
[AssertEqual(q_0,q_1)];
[AssertTeleported(q_0)];
[AssertTransformed(q_0,(θ interval)(φ interval))];
```

Figure 3.10: Property syntax structure [42].

To asses the capabilities of this framework, the authors introduce mutation testing to two programs, teleportation and superdense coding. The results yielded a mutation score of 80% for teleportation and 60% for superdense coding, with 20 mutant programs examined in each case.

The authors conclude the article by summarising the completed work, emphasising that this marks the initial phase in property-based testing of QPs. The primary contribution of the article lies in the integration of a property language based on pre- and post-condition type properties and QSharpCheck.

```
Testing "Transform_Property"
Number of test cases:                  10
Confidence level:                      99%
Number of measurements:                500
Number of experiments:                 300
AssertTransformed was true in all test cases. Passed
      10 tests.
```

(a) QSharpCheck positive outcome.

```
Testing "Transform_Property"
Number of test cases:                  10
Confidence level:                      99%
Number of measurements:                500
Number of experiments:                 300
After 1 test, AssertTransformed was falsified when:
θ = 38 and ϕ = 5
```

(b) QSharpCheck negative test.

Figure 3.11: Two possible QSharpCheck outcomes [42].

3.2.2 QCEC

Bulgholzer et al. introduced the QCEC tool in their 2021 paper titled *QCEC: A JKQ tool for quantum circuit equivalence checking* [44]. They describe QCEC as a quantum circuit equivalence checker developed in C++ as part of the JKQ toolset, with bindings provided for Python users.

Overview of methods provided by QCEC.

Method	Strategy	Description	Remarks
Standard DD		Construct and compare the decision diagram for both circuits	Potentially constructs two large decision diagrams
$G \rightarrow \mathbb{I} \leftarrow G'$	Naive	Alternate between applications of G and G'	The most simple strategy
	Proportional	Proportionally apply gates according to the gate count ratio of G and G'	The most reliable general-purpose strategy
	Lookahead	Always apply the gate yielding the smaller decision diagram	High gain, high risk strategy
	Compilation Flow	A dedicated scheme for verifying results of the IBM Qiskit Compilation Flow explicitly exploiting certain knowledge about the compilation process	Best performance for specific application scenario
Simulation	Classical	Computational basis states	Very fast, but cannot detect certain rotations
	Local Quantum	Each qubit value is independently chosen from any of the six basis states ($\|0\rangle, \|1\rangle, \|+\rangle, \|-\rangle, \|L\rangle, \|R\rangle$)	Fast and reliable
	Global Quantum	Random stabilizer states	Slow, but very reliable

Figure 3.12: QCEC methods [44].

The core idea behind this method is that quantum programs are reversible when measurements have not occurred. Thus, one can compute the inverse of the quantum circuit and check if the product of the original circuit and its inverse yields the identity matrix. The methods implemented in QCEC, which are summarised in Figure 3.12, were previously detailed in their earlier works [45, 46, 47, 48, 49]. The authors conclude that due to the generality of the tool, it can be tailored for specific scenarios and even integrated into other programming languages.

3.2.3 Quito

The subsequent approach, presented by S. Ali et all in 2021 under the title: *Assessing the effectiveness of input and output coverage criteria for testing quantum programs*[50], introduces the Quito approach (QUantum InpuT Output coverage). The authors define three coverage criteria on the inputs and outputs of QPs along with their test generation strategies. This paper has been expanded to include new experiments, as detailed in [51].

In their paper, the authors formally establish definitions for QP, input, output, program specification, valid output values, and test input, output, and suite. As well, the definition of input, output and input-output criteria follows the usual such that every input/output/<input,output>, respectively, is covered by a test suite. Let us delve in their definition for outcome of a test suite:

- Definitely fail: The outcome returns a non-valid output.
- Likely fail: The tested probability significantly deviates from the expected probability based on the Wilcoxon test with a 99% confidence interval.
- Inconclusive: No failure has been detected.

To proceed, the authors propose the application of two different oracles in sequence:

- Wrong Output Oracle, WOO: Checks if QP only produces valid output values.
- Output Probability Oracle, OPO: Verifies if the QP returns an expected output with its associated probability. Providing one of the outcomes explained previously.

Let us introduce the test suit generators for testing.To initiate the process, the user will define a parameter K representing the desired number of test suites. Additionally, a budget for output coverage need to be allocates as a constraint for generating a single test suite, because there is a chance of not finding the correct input for a particular output. The test suite creation algorithms will directly assess the WOO, halting if a non-valid output is produced and saving that test suite as a WOO failure.

- Input coverage. For each value up to K, each possible input will be executed and saved if it produces a valid output.

- Output coverage. For each value up to K, while there is elements in the valid outputs set and our budget for a single test suite generation has not been exhausted:
  - Execute one element from the valid input set.
  - Add ⟨input, output⟩ to the test suite.
  - Remove the output from the valid outputs set.
- Input-Output coverage. For each value up to K and for each valid input, we define the valid outputs set from the specification and proceed similarly to output coverage with this set and budget.

Therefore, we can now outline the Quito workflow:

- Test suites generation.
- Check for WOO outcomes, already evaluated during test suite creation. The outcome will be *definitely fail* if fails WOO.
- For test suites that pass WOO, they will be assessed by OPO. Since larger test suites may be required for statistical significance, it may be necessary to merge some test suites to meet this criterion. The outcome will be either *likely fail* or *inconclusive*.

The authors intend to assess these coverage criteria using mutation analysis, employing the following categories of mutation operators: add gate (AG), delete gate (DG), replace gate (RG), and replace mathematical operator (RMO), these operator behave similarly to classical mutation operators. The effectiveness of their test suites will be evaluated on mutated programs. Since results may vary across executions, each test suite will be executed K times for each criteria.

A mutant will be considered killed if at least one of the proposed oracles fails. The mutation score is defined by subtracting the number of equivalent mutants from the total number of mutants. As the number of equivalent mutants is initially unknown, the authors start with 0 and analyse it after testing. The equivalent mutants are then studied manually, step by step, comparing states from the original process to a mutated one using the QCEngine execution facility.

Let us summarise the key results and contributions of Quito:

- Three coverage criteria: Quito introduces three coverage criteria that are independent of a specific language, each equipped with its own algorithm for creating test suites to achieve full coverage.
- Consistent Mutation Scores: The mutation scores tend to remain within the same range regardless of the coverage criteria. Although there are some variations, achieving higher mutation scores comes with a significant computational cost. For instance, RCR reaches an 80% mutation score with 8000 test cases in input coverage and over 15000 in output coverage. However, it attains a 95% mutation score on input-output coverage but requires 160k test cases.
- Equivalent Mutants: All non-killed mutants were identified as equivalent mutants, and some exhibited a phase difference.

3.2.4 QuSBT

Wang et al. presented their work *Generating Failing Test Suites for Quantum Programs With Search* [52] and a posterior shorter version [53]. The authors focus on test case generation, aiming to use a genetic algorithm to automate this process. These papers build on and support the definitions presented in their earlier work, which is mentioned or defined in Section 3.2.3 of this document.

One of the additions to the existing definitions is the introduction of a defined number of required executions for a certain input to achieve a reasonable level of confidence in its output. Depending on the possible output values, a different number of execution repetitions may be needed to approximate the correct distribution. The number of repetitions for an input i is defined by:

$$numRepetitions(i) = |PS_{NZ}(i)| \times 100$$

Where $PS_{NZ}(i)$ represents the set of non-zero probabilities for output values for an input i. Let us now examine the authors' proposal for generating the test suites.

Firstly, the number of tests must be selected for each test suite. The authors note that proposing the same number of tests for different algorithms may not be optimal. Instead,

they suggest that the user should specify a percentage β, where the number of tests is that percentage of all inputs. Let us now see the proposal for fitness using a genetic algorithm:

- **Search variables**: $\bar{x} = [x_1, ..., x_M]$, where x_i are integer variables, each representing an input chosen from the possible inputs.
- **Assignment**: $\bar{v} = [v_1, ..., v_M]$, which represents the assignment found by QuSBT for the M tests, aiming to maximise the number of tests that fail the program.
- **Fitness computation**:
  - Identify the required number of repetitions.
  - Execute the correct number of repetitions for each selected input.
  - Assess the results and identify the failed test.
- **Fitness function**: The fitness function is defined as the number of programs that fail during the assessment.

The authors designed an experiment that involved six algorithms with varying numbers of gates and depths. They manually created five faulty versions of each algorithm: two with faults introduced immediately after the inputs, two after reading the output, and one in the middle of the program. Detailed information about the programs and the changes made can be found in their work [52]. The genetic algorithm used in the study was obtained from the `jMetalPy 1.5.5` framework. For statistical assessment, they employed the Chi-square test with a significance level of $\alpha = 0.01$. The aim was to compare their genetic algorithm, GA, approach with random search, RS, in the input domain.

Let us now review the authors' RQs and their answers:

- **RQ1**: Does QuSBT (which is GA-based) perform better than Random Search (RS)? RQ1 assesses whether GA can identify test inputs that contribute to failures, as compared to RS.
  - The authors observe that, out of 30 executions, GA performed better in 26 cases. Thus, GA is significantly better than RS.
  - In the remaining 4 cases, no statistical difference was found between RS and GA. Two of these results were from the smallest program (in terms of qubit number), where the number of possible inputs is smaller.

- **RQ2**: How does QuSBT perform on the benchmark programs? RQ2 assesses the variability on the final results, and how fast the GA converges to better solutions.
  - QuSBT was almost always unable to find 50 failing tests for their programs, or even 26 failing tests for different algorithms. The authors hypothesise that this could be due to the limited number of failing tests or insufficient search time.
  - The authors propose as future work the need for a dedicated and large-scale empirical study to better understand this variability.

3.2.5 Testing QSharp with Microsoft development kit

Mykhailova et al. presented a paper titled Testing Quantum Programs Using Q# and Microsoft Quantum Development Kit [54], in which they explore the use of Q# and the open-source Quantum Development Kit (QDK) for implementing testing strategies. A key contribution of their work is the progression from unit testing to testing unitary transformations through small, illustrative examples. They utilise Katas [55], a tutorial framework with interactive feedback, and apply the Choi-Jamiolkowski isomorphism [56, 57] to simplify verification processes.

We have decided to include this paper in the state of the art, as it may be interesting to showcase the variety of tools available for use in quantum computing. However, it is primarily an introduction to the Microsoft Quantum Development Kit for Q# and how it can be utilised.

3.2.6 Muskit

The third approach was introduced by E. Mendiluze et al. with the title: *Muskit: A Mutation Analysis Tool for Quantum Software Testing* [58] in 2021. The authors present Muskit as the novel tool to autonomously implement mutation testing in Quantum Programs (QPs). While there were some previous works [50, 51] utilising mutation testing in quantum, all mutated programs were manually generated.

As evident from the Muskit architecture (Figure 3.13), Muskit comprises two main components, with the added Test Analyser. Let us delve into these components. The first, and a key result of this paper, is the introduction of a mutation generator. Let us explore the various mutation operators employed and the criteria the tool uses to implement them.

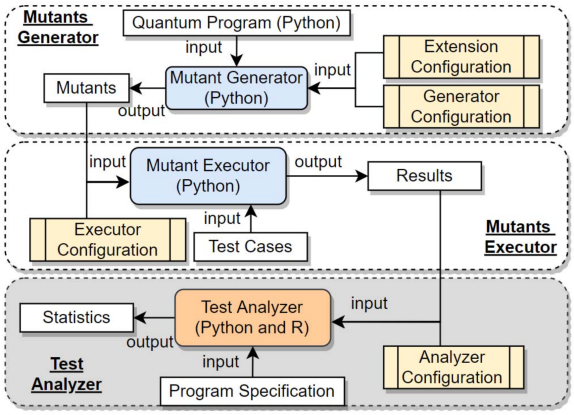


Figure 3.13: Muskit architecture [58]

Mutation operators supported for 19 gates:

- Add Gate (AG).
- Remove Gate (RemG).
- Replace Gate (RepG). Replacing an existing gate for one compatible, applied to the same number of qubits.

Mutation criteria, will reduce the number of mutants to be generated:

- All. All possible mutants are generated.
- Operator Selections. The user has the flexibility to choose which mutant operators they would like to use.
- Gate Selection. The user controls mutant generation based on the number of qubits. Muskit supports three categories: one qubit gates, two qubit gates and three or more qubit gates.
- Gate Number Selection. The user can control what gate will be mutated, as Muskit maps gates to numbers.
- Location Number. The user can locate where to apply AG.
- Phase Selection. The user can select a phase between 0 and 360 when mutants are applied to phase gates.

- Maximum Number. One can limit the number of mutants to be generated.

When focusing on the Mutants Executor component, it takes mutants and test cases as input, executing them as many times as specified in a file called *ExecutorConfiguration*. In cases where no specification is provided for the input test cases, it defaults to using the *Input coverage* as presented in 3.2.3 [50] by S. Ali et al. The tool follows a similar approach for the Test Analyser component, importing both oracles—Wrong Output Oracle (WOO) and Output Probability Oracle (OPO)—from 3.2.3 [50]. Once analysed, all results are reported.

	Number of mutants				**Mutation Score(%)**	
	#AG	***#RemG***	***#RepG***	**#Total**	WOO	OPO
IQFT	285	9	78	372	0	100
QRAM	304	9	58	371	87.27	6.66
BV	255	9	79	343	77.35	0
CE	342	12	104	458	99.26	0.74

Figure 3.14: Muskit results [58]

The authors conducted experiments using four different QPs to evaluate Muskit, yielding the results shown in Figure 3.14, to demonstrate the capability of automatically generating mutants in QP. Until now, all mutations had been performed manually. The paper identifies a limitation that Muskit currently faces, it is unable to detect equivalent mutants. The authors express concerns about the possibility that these quantum mutation operators may be related to each other in terms of the killability of their mutants. This leaves room for exploration in theoretical and experimental future works, presenting an area that could be further investigated.

3.2.7 QuCat

Wang et al. presented *Application of Combinatorial Testing to Quantum Programs* [59] in 2021, where they introduce QuCAT, an approach for systematic and automated testing of quantum programs. The core definitions follow those previously established by Ali et al., as noted in Section 3.2.3.

QuCAT generates combinatorial test suites using various generation algorithms, including AETG [60], IPO [61], and IPOG [62]. Their experiments aim to assess the effectiveness

of combinatorial testing for quantum programs in two scenarios: generating a test suite for a specified strength and incrementally increasing strength. To evaluate QuCAT, they propose an experiment involving six algorithms with injected faults. The proposed research questions, RQs, and their answers are detailed in their work.

- **RQ1** How do applications of combinatorial testing, CT, with different strengths compare in terms of cost and effectiveness?
  - CT with lower strength is associated with significantly lower costs due to smaller test suite sizes.
  - Analysis indicates that while CT with higher strength is generally more effective, lower-strength CT can still yield good results for simpler faults.
- **RQ2** How does the effectiveness of CT with different strengths compare to random testing?
  - On average, CT demonstrates higher success rates in fault detection compared to random testing across all tested strengths.
- **RQ3** How quickly can CT detect faults compared to random testing?
  - CT can identify faults more rapidly than random testing, as evidenced by results from Scenario 2, across all faulty programs.

The authors propose as future work the development of a fault localisation approach for quantum programs based on these results. They plan to continue testing this approach with more complex quantum programs.

3.2.8 Quantum bug fixes comprehensive study

Luo et al. presented the article titled *A Comprehensive Study of Bug Fixes in Quantum Programs* [63], in which they analyse bugs in quantum programs by collecting data from GitHub, Stack Overflow, and Stack Exchange related to Qiskit, Q#, Circ and ProjectQ. Their goal is to study current bugs and their fixes in quantum programming.

The authors gathered bug reports and fixes from these platforms, then filtered them based on three criteria: relevance to quantum programs, authorship by developers, and the availability of valid pre- and post-fix code. After collecting and processing this information,

they analysed the different types of bugs. Their approach involves categorising the bugs using two frameworks: the first, defined by Zhong and Su [64], focuses on fault complexity, and the second, based on Paltenghi and Pradel's work [65], classifies bug patterns. This categorisation enables them to address their research questions and present their findings.

Bug pattern			Count
API-related	API misuse		30
	Outdated API client		5
Incorrect application logic	Intermediate representation	Missing information	4
		Wrong information	6
	Refer to wrong program element	Wrong concept	2
		Wrong identifier	1
	Incorrect scheduling		1
	Wrong declaration		1
	Wrong way of implementation		7
	Qubit-related	Incorrect qubit order	4
		Incorrect qubit count	1
Math-related	Incorrect numerical computation	Incorrect gate	8
		Incorrect matrix computation	9
		Incorrect initial state	2
		Incorrect measurement	3
		Potential all-zeros matrix	1
	Incorrect randomness handling		3
Others	Misconfiguration		4
	Type problem		1
	Typo		2
	Overflow error		1

Figure 3.15: Bug patterns [63]

- **RQ1** Does this bug occur only in quantum programs?
  - Quantum-specific bugs constitute over 80% of the bugs identified, significantly more than classical bugs.
- **RQ2** Where are the bugs located in the programs?
  - All bugs have been fixed within their respective source files. This finding may be skewed, as many programs only consist of a single source file.
  - Over 70% of the bugs can be resolved by modifying just one line of code.

- **RQ3** How complex is it to fix bugs in quantum programs?
  - Many bugs are fixed with single-line changes, indicating that a significant proportion of quantum program bugs are relatively easy to fix.
  - The most complex bugs are often due to incorrect implementation methods. These issues arise when users are unsure of the correct implementation. However, this complexity tends to decrease with user experience. This type of error presents challenges for automated bug-fix approaches.
- **RQ4** Are there identifiable bug patterns in quantum programs?
  - Bug patterns are illustrated in Figure 3.15.
  - While classical bugs are more evenly distributed across categories, quantum bugs are predominantly concentrated in API-related and math-related issues.

3.2.9 Debugging QC

Metwalli et al. presented an article titled *A Tool For Debugging Quantum Circuits* [66], introducing a debugging tool developed on Qiskit. The main objective was to provide users with a novel tool to help them debug their quantum programs.

The authors outline the tool's functionality, which is divided into three parts:

- **QP Slicer**: This component breaks a quantum program into different subprograms.
- **Testing Individual Slices**: This involves testing each subprogram independently.
- **Gate Tracking**: This feature tracks the gates used in the quantum circuits.

The tool has been implemented as an extension of the `QuantumCircuit` class, eliminating the need to create an entirely new type. The authors introduced two new methods: `breakBarrier()`, which helps slice the program vertically, and `gateInfo()`, which provides necessary information about a particular gate. Additionally, the tool includes functions to initiate the debug state, produce slices, and obtain the positions and number of occurrences of a specific gate in a circuit.

Let us examine each part of the tool and understand how it functions.

- **QP Slicer**: The tool differentiates between two types of slicing: vertical and horizontal. The vertical slicer divides the QP into different slices, while the horizontal slicer reduces the number of qubits in each subprogram by excluding unused qubits. Slicing can be cumulative, where each new slice is added to the previous one.
- **Testing Each Slice**: The authors propose categorising each slice, with different testing techniques for each category. The current version categorises slices based on the circuit size, the types of gates involved, and their potential effects on state entanglement. There are three testing options: uniform superposition state, specific state, and symmetric states such as GHZ, W, and Dicke states.
- **Gate Tracking**: To address potential errors caused by incorrect gate implementation, the tool includes gate tracking. This feature helps debug issues by providing detailed information about the gates used in the quantum circuit.

The authors present an example demonstrating how the slicer works and how the different slices are categorised. They use the triangle graph problem solver algorithm, implemented with Grover's algorithm, as a case study. The evaluation was conducted by releasing a beta version of the tool to a group of quantum programmers with varying levels of expertise in quantum computing. The goal was to assess the tool's capabilities for users of all levels, including its installation and usage. Survey results indicate that the tool simplifies and accelerates the process of locating bugs.

In their future research, the authors aim to enhance the tool by incorporating higher capabilities. This includes developing a more accurate categorisation criteria, expanding gate tracking features, and creating an automated testing function for commonly used subroutines.

3.2.10 QPs testing with MR

Abreu et al. presented an article titled *Metamorphic Testing of Oracle Quantum Programs* [9] in May 2022. The authors aim to address some of the main challenges and key difficulties in quantum computing testing:

- The inability to use methods with step-by-step monitoring.
- The high-dimensional and difficult-to-interpret nature of quantum states, which limits testing for debugging misbehaving quantum programs.

- The lack of existing guidelines for quantum computing testing.

They propose metamorphic testing, MT, to primarily address the quantum measurement problem. Metamorphic relations, MR, will be specified in the quantum setting to avoid measuring the quantum state directly. The proposed approach is demonstrated by applying it to a simple modular adder implementation, using Grover's algorithm.

The main idea is to identify the MR that the implementation under test should satisfy and then express this in a straightforward quantum manner. The authors present $MR_0 : x +_m 0 = x$, requiring a comparison of the equality of two integers. To validate this, they use different adders, with their proposed implementation under test, to obtain a result for the equality. They will take the complement of one result and perform the addition; the result should be all bits set to 1.

Once the metamorphic relations, MR, have been studied and the approach proposed, the authors compare the classical and quantum approaches for MR verification. The classical approach raises two main concerns:

- Do we need to implement the MR as a quantum operation when performing classical tests on the MR?
- Is it possible to perform MR verification using quantum computing, thereby taking advantage of quantum parallelism?

The authors then focus on the quantum approach for MR verification, primarily based on Grover's algorithm. They explain how this approach can leverage Grover's algorithm, as the metamorphic rule can be treated as a search function for identifying failing tests: $f(t) = 1$ if the test t fails the metamorphic rule indicated by f. However, this approach faces two challenges:

- Grover's algorithm requires that the number of successful tests is lower than half of the cases. If this condition is not met, a Monte Carlo approach could be used, or the search space could be increased by 1 qubit.
- Grover's algorithm does not inherently stop when a solution is found. Therefore, the number of steps must be known beforehand. The authors propose a hybrid program to ensure enough iterations are performed until results are obtained with high probability.

The authors apply this approach directly to the previously defined MR and obtain the expected results. However, to demonstrate how Grover's algorithm behaves in the presence of bugs, they introduce mutants. The results show how the algorithm finds a solution, indicating that it has successfully identified the mutants.

In conclusion, the main contributions of this proposal are highlighted along with some challenges that remain as future work.

- Metamorphic testing concepts can be effectively applied to QPs.
- The entire process is executed within the quantum realm.
- The validity of the approach is demonstrated through the experiments presented.
- The approach shows a significant speed-up compared to classical methods for MT.
- Further research is needed on extracting adequate input test cases and applying the method to different and more complex QPs.

3.2.11 MutTG

Wang et al. presented the article with title: *Mutation-Based Test Generation for Quantum Programs with Multi-Objective Search* [67]. The authors adopted the same theoretical framework as applied in [50] and [51], which were previously discussed in Section 3.2.3. They define two new concepts of possible output failures directly connected to the names of the appropriate oracles.

The authors propose a search-based approach called MutTG, which aims to generate the minimal test suite that can kill as many mutants as possible, targeting all non-equivalent mutants. They define an individual in the search as a set of k elements, where there are k mutants. These elements can be derived from the testing domain or any other relevant sources. This set represents possible tests with a cardinality less than k.

MutTG applies an algorithm to each set of test cases, saving dictionaries to track whether a test killed a mutant or not. Once a mutant is killed, the algorithm proceeds to the next mutant without testing the remaining test cases for that particular mutant. They denote KM as the dictionary of pairs $\langle t, \{M_1, .., M_{n_t}\}\rangle$, where test t killed mutant $M_i, \forall i \in 1, ..., n_t$. Similarly, NKM represents the dictionary for cases where t does not kill M_j.

These dictionaries are global and will be used in following executions, starting from empty dictionaries.

The authors' goal is to assess, using these dictionaries, whether a mutant is highly likely to be an equivalent mutant. To achieve this, they create the following functions:

- $discFactor$(Mutant, NKM): The discount factor represents the ratio of the number of tests that have not killed the mutant to the total number of possible tests.
- $notKilledScore$(Mutant, search, KM, NKM): This function assigns a score to each mutant based on the following criteria:
  - 0, if the mutant has been killed during this search.
  - 1, if the mutant has not been killed in this search but has been killed previously.
  - 1 - $discFactor$, otherwise.
- f_{not_killed}(search): The sum of all $notKilledScore$ values for mutants in the search.

Experiment design:

- Five QP: BV, QRAM, invQFT, AddSquare, and ConditionalExecution.
- Three sets of mutants categorised by difficulty:
  - Easy: killed by at least 25% of the inputs in the domain.
  - Medium: killed by at least 1.26% and less than 25% of the inputs in the domain.
  - Hard: killed by at most 1.26% of the inputs in the domain.
- Statistical test used for distribution comparison: Pearson Chi-square test with $\alpha = 0.01$ as the significance level.
- Three search approaches: MutTG, MutTG without $discFactor$ and Random Search, specifically NSGA-II.
- Three metrics:
  - $Hypervolume$: A quality indicator obtained from [68]. Higher values indicate better quality.

  - $MNNKM$: Minimum number of not-killed non-equivalent mutants. Lower values are better.
  - TSS_{MNNKM}: Minimum size of the test suite needed to achieve $MNNKM$. Lower values are better.

- Statistical test within metrics: Mann–Whitney U test with significance level 0.05 and Vargha and Delaney's $\hat{A}_{12}$ statistics as the measure of effect size.

Research questions and results:

- **RQ1** What is the influence of the discount factor on the effectiveness of the search in MutTG?
  - MutTG is shown to be the most effective if there is any difference between the approaches.
  - The effectiveness improvement is evident in experiments with higher difficulty, which involve a greater number of equivalent mutants.
- **RQ2** How does the difficulty of the benchmarks affect the effectiveness of MutTG?
  - The authors' results show that difficulty significantly impacts test suite size. Even when values are low, there are cases where not all non-equivalent mutants are killed, even with more than 500 generations.
- **RQ3** How does the number of equivalent mutants in the targeted mutant set affect the effectiveness of MutTG?
  - It is more challenging to kill non-equivalent mutants when the number of equivalent mutants is lower, as expected due to the reduced number of mutants overall.

3.2.12 QMutPy

The initial approach we are presenting was authored by Fortunato, D. et all in 2022 in their article titled *Mutation testing of quantum programs: A case study with qiskit* [69]. In this work, they explore mutation testing on QP with the extension of the MutPy library by introducing new quantum mutation operators. This article can be divided into two distinct sections. The first section focuses on the introduction of QMutPy and the initial experiments. The second section delves deeper into the previous results to identify potential causes and solutions. The authors implement some of these solutions to determine if they are on the right path.

First, let us understand what QP are going to be tested and why did the authors choose this library. The oracle problem was one of the challenges discussed earlier, where given an input, we need the expected behaviour. In this article, the authors have decided to prioritise and initially focus on this new approach for QP testing. To accomplish this, they opted to use Qiskit-Aqua's repository, which has since been moved to Qiskit-Terra[2]. This repository provides the implementation of 24 QP along with their respective test suites. In these repository, you can discover a variety of programs, including pure classical, hybrid, and pure quantum programs.

Similarly, they carefully selected the tool they believed would be the most suitable for implementing their ideas about quantum mutation testing. Their criteria included support for Python programs, testing frameworks, mutation operators, and the capability to generate appropriate reports. For that reason, they choose MutPy, as it fulfils their specific needs. Now, let us see how they develop these new quantum mutation operators, expanding the library mentioned earlier and naming it QMutPy.

They have introduced in QMutPy 5 quantum mutation operators, QMO:

- Quantum gate replacement, QRG.
- Quantum gate deletion, QRD.
- Quantum gate insertion, QRI.
- Quantum measurement insertion, QMI.
- Quantum measurement deletion, QMD.

[2] `https://github.com/Qiskit/qiskit-aqua/#migration-guide`

These new operators are designed to capture the main errors that can occur during the implementation of an algorithm. The key concept around QMO is gate equivalence. We will consider two gates to be syntactically equivalent if and only if the number and type of the arguments are identical. The authors have identified 40 gates with at least one syntactical equivalent gate. As an example, gate h would have the following equivalent gates: x, y, z, i, id, s, sdg, sx, t and tdg. The authors illustrated all equivalence gates in Figure 3.16. There may be some gates which may not meet such criteria.

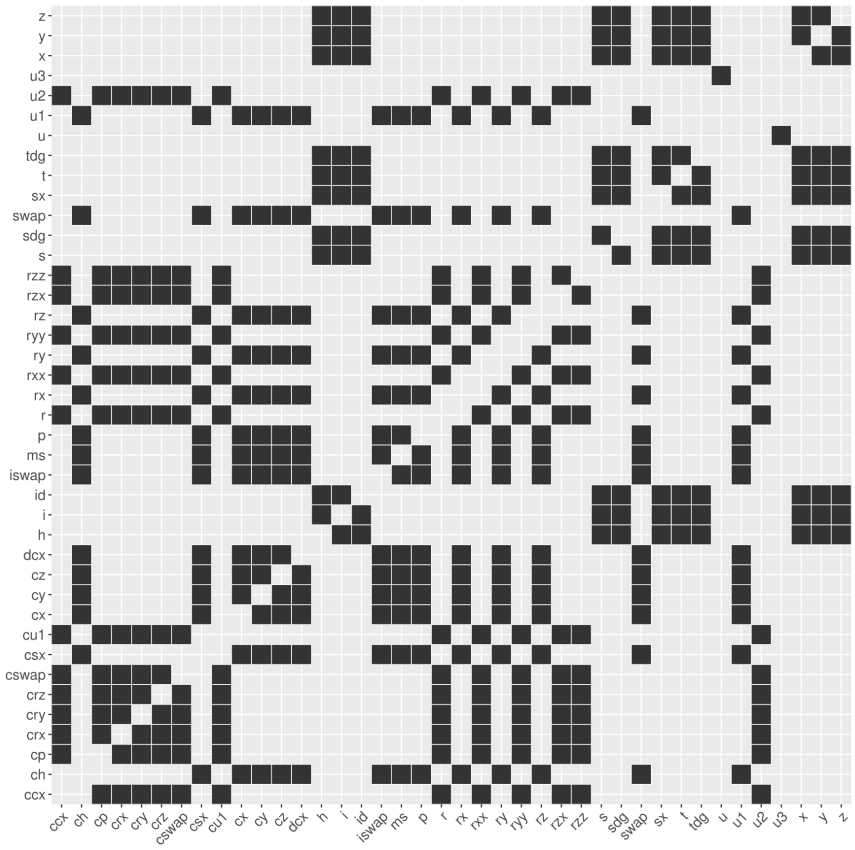


Figure 3.16: Gate equivalence matrix [69]

There is no need to provide an explanation for the mutant operators, as they are inherently self-explanatory. However, let us dig into a more comprehensive understanding of gate operators. As mentioned earlier, the fundamental concept that makes these operators effective is gate equivalence. Whenever we encounter a gate addition, `.append()`, and we intend to employ QRG or QRI, we will either replace the existing gate with a syntactically equivalent one or simply add a new gate that preserves the syntax of our program. The process will generate as many mutants as there are syntactically equivalent gates. All of

these modifications are implemented using Python's Abstract Syntax Tree (AST).

QMutPy workflow, analogous to MutPy:

- Loads QP source code and test suite.
- Executes test suite on the unmutated QP.
- Applies mutant operators, including QMO.
- Executes test suite on all mutated QP and provides a summary of results.

Because the works in quantum mutation [26, 58, 50] were in a preliminary state, the authors opted to compare MutPy and QMutPy by running experiments on the same set of quantum programs and test suites. The experiment's metrics will be assessed through two distinct approximations of the mutation score. The first one follows the classical interpretation [70], while the second allows for the potential achievement of 100% mutantion score. This will be achieved by excluding from the total mutants generated those that have not been executed.

The authors will perform when needed Kruskal-Wallist non-parametric test with $p = 0.01$ and Cohen's d effect-size measure on the results reported by QMutPy to evaluate the statistical significance.

Let us conclude by highlighting the key contributions and outcomes of these experiments:

- Expansion of the set of mutation operators for quantum programs, QMutPy.
- Introduction of gate equivalence definition based on the number and type of arguments.
- QMutPy generates quantum mutants for 11 out of 24 QPs. The remaining 13 QPs should exclude quantum gates and measurements. On average, 4 lines of code (LOCs) are mutated, resulting in 13 mutants and 64 mutants per QP. MutPy, on the other hand, generates mutants for all programs with an average of 64 LOCs mutated, 3 mutants, and 147 mutants per QP.
- QMutPy's performance in mutant creation is not statistically significantly lower than MutPy when we considering the potential number of classical vs quantum mutations per quantum program.

- Test suits may only focus in the quantum part of the program, as evidenced by the difference in mutation scores between classical and quantum mutations and the type of killing, error vs test assertion.
- QMutPy exhibits reduced efficiency in creating quantum mutants. One proposed solution involves defining new operations in Python AST for quantum gates to enhance creation time.

As mentioned earlier, the authors delved into potential enhancements for these results, particularly focusing on test suites. They concentrated on two criteria for improvement: coverage and test assertions. By selecting specific programs for each test, they identified why mutants survived and implemented new assertions or tests as necessary. Therefore, the augmentation of test suites and assertions resulted in an increased mutation score for the studied programs.

- The mutation score for *hhl*'s rose from 50% to 100% (with coverage increasing from 86.55% to 89.16%), and for *vqc*'s, it increased from 0% to 50% (coverage rising from 93.26% to 94.43%).
- In the case of Shor's algorithm, the original test suite achieved a 53.34% mutation score, whereas the augmented test suite reached 72.81%.

3.2.13 QuMu

García de la Barrera et al. present *Quantum Software Testing: Current Trends and Emerging Proposals* [71] in 2022 as one of the chapters in the book *Quantum Software Engineering* [72]. This chapter provides an overview of the state of the art in quantum computing testing. We will focus on the *Quantum Mutation Support Tool, QuMu* prototype introduced by the authors in this chapter.

They use Quirk[3], a quantum circuit simulator, where each circuit is represented by an ordered set of columns, with each column containing an ordered set of gates. This structure is utilised to track positions where mutations can occur. An abstract operation, `isApplicableTo`, is defined for this purpose. This operation identifies which gates can be passed as parameters, thereby creating a mapping of gates that could be mutated.

[3] `https://algassert.com/quirk`

The authors define several mutation operators, categorised into different groups, which are detailed in [71]:

- Initialisation: Change of initial value, first gate duplication, and additional gate duplication.
- Wrong gate: Introduction of incorrect gates.
- Missing gate: Removal of gates, including one-qubit gates, multi-qubit gates, and control gates.
- Bridging faults: Swapping controls and controlled qubits.
- Entanglement faults: Incorrect entanglement initialisation, entanglement corruption, forced unentanglement, and forced entanglement.

The process is divided between two engines:

- Mutant generation: Iterates over selected mutation operators using the mapping explained previously.
- Mutant execution: Uploads the mutated circuit to the Quirk website, runs the simulation, and then downloads and saves the results in the database.

They use distribution matrices to compare results and identify killed, alive, and injured mutants. Injured mutants are those that produce the same distribution matrix but leave the qubits with different phases, as illustrated in Figure 3.17. This prototype includes a mutants results analyser.

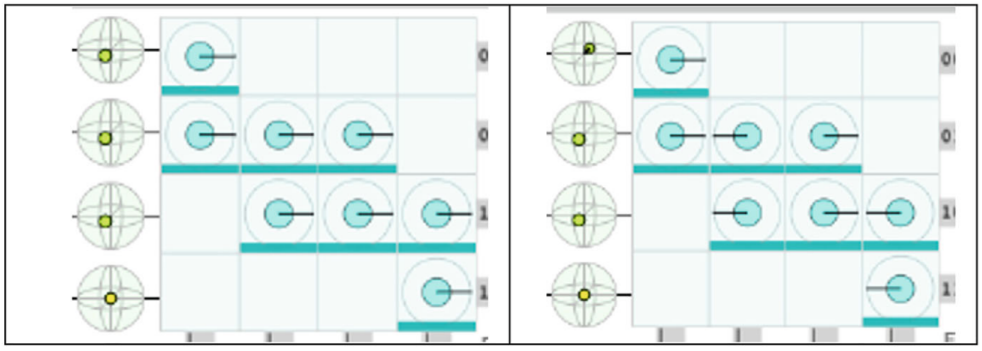


Figure 3.17: Injured mutant [71]

3.2.14 Shor's correctness with MR

Nuno Costa et al. presented in 2022 the paper with title: *Asserting the correctness of Shor implementations using metamorphic testing* [73]. In this work, they used metamorphic testing to validate the correctness of the well-known Shor's algorithm.

Firstly, the authors explore Shor's algorithm, which is crucial for metamorphic testing. Understanding the algorithm is important for defining appropriate MR, as these rules are based on the characteristics of the algorithm being tested. Here is their proposal for MR:

- In the domain of `number factoring` where p is a given prime number, q_1 is a given number, and $f(x)$ represents the set of factors of x, the following MR, named Add Prime, should hold: if q_2 is equal to $q_1 \times p$ then $f(q_2) \backslash f(q_1) \subseteq p$.
- In the domain of number factoring where q_1 and q_2 are given numbers and $f(x)$ represents the set of factors of x, the following MR, named Multiply Factors, should hold: if q_3 is equal to $q_1 \times q_2$ then $f(q_3) = f(q_1) \cup f(q_2)$.

The authors proceed to implement the metamorphic rule `Add Prime` and conduct experimental studies using mutants. They note that with the current number of qubits available in real quantum systems, it is not feasible to reproduce experiments for Shor's algorithm due to its high qubit requirements. Given the non-deterministic nature of Shor's algorithm, even a correct implementation could fail the test. To address this, the authors introduce a threshold of 20 iterations, considering the test a pass if at least one iteration meets the MR. They conclude that, despite these challenges, further research is needed in this approach. However, with the appropriate threshold, they observe that the MR effectively detects implementation failures.

3.2.15 Automatic test circuit generation

Garcia et al. presented preliminary work at the end of 2022 tittled *Automatic generation of test circuits for the verification of Quantum deterministic algorithms* [74]. The authors aim to introduce test cases to be executed on a quantum machine. Their approach involves embedding the quantum circuit under test, CuT, within a QP that checks whether the expected output matches the computed output, producing a verdict for the test case rather than evaluating the outputs directly.

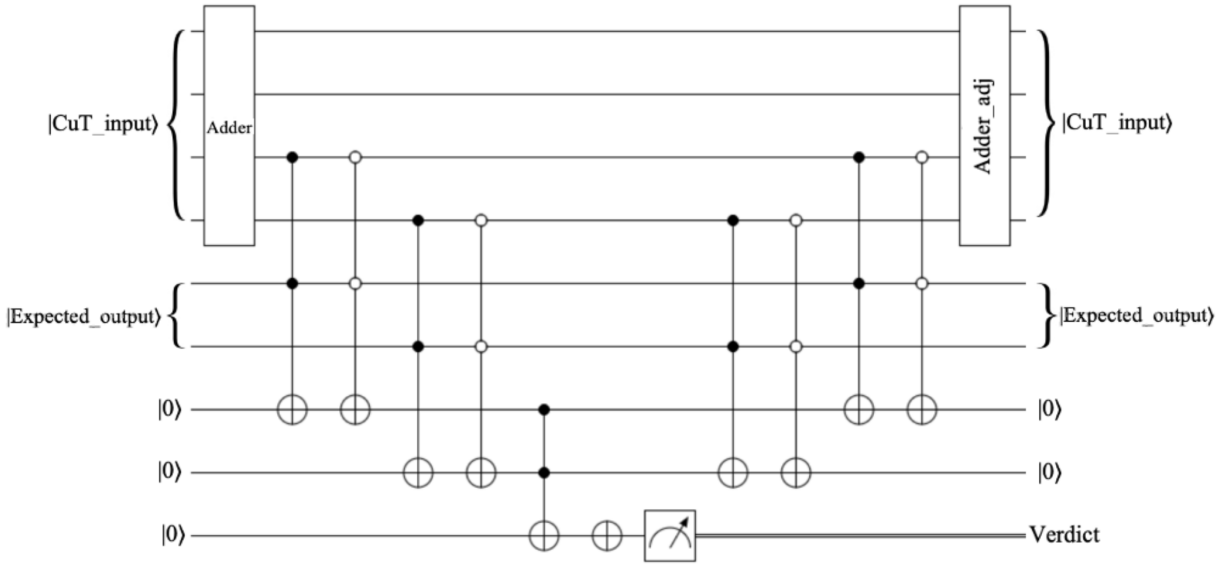


Figure 3.18: Adder QTC [74]

They implement this approach using a two-qubit adder algorithm, applying both a correct test suite and a modified one to observe how the program detects non-matching results, thereby generating a quantum test circuit, QTC.

In my opinion, a key point to highlight about this approach is the increased number of qubits used. As illustrated in Figure 3.18, the original adder required 4 qubits, while the entire system needed 9 qubits to verify the test suite. A similar approach was used previously by Abreu et al. in [9] to represent the output of a metamorphic rule.

There is a new version of this paper in which the authors delve further into this approach and present new case studies. The title is *Automatic Generation of Test Circuits for Deterministic Quantum Algorithms* [75], and it is currently in preprint status at the time of writing this section.

3.2.16 Coverage Criteria for Quantum Software Testing

Ajay Kumar presented the paper titled: *Formalisation of Structural Test Cases Coverage Criteria for Quantum Software Testing* [76], where he introduces the concept of cyclomatic complexity into the realm of quantum computing to address the complexity of quantum testing. The original concept of cyclomatic complexity was introduced by McCabe in 1976 [77]. The author defines **cyclomatic complexity** for quantum computing as the number of in-

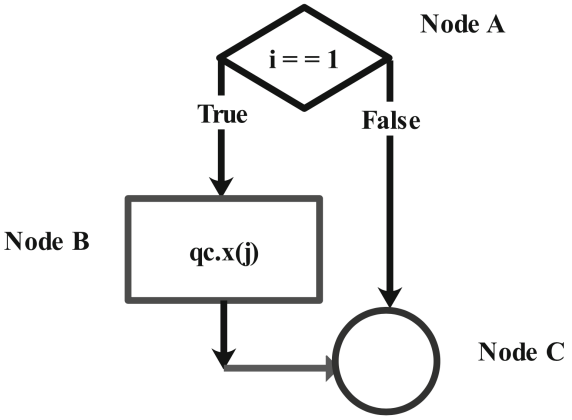
dependent paths and paths generated due to the superposition of each control qubit. This concept will guide the development of a structural coverage criterion for testing QPs.

McCabe [77] introduced the concept of cyclomatic complexity based on graph theory. Cyclomatic complexity is a quantitative metric used to determine the number of independent paths in a control flow graph. For a control flow graph with n nodes, e edges, and p connected components, the complexity is given by $e-n+2p$. He proved that the cyclomatic complexity of n different modules of a program is equal to the sum of the cyclomatic complexities of all individual modules.

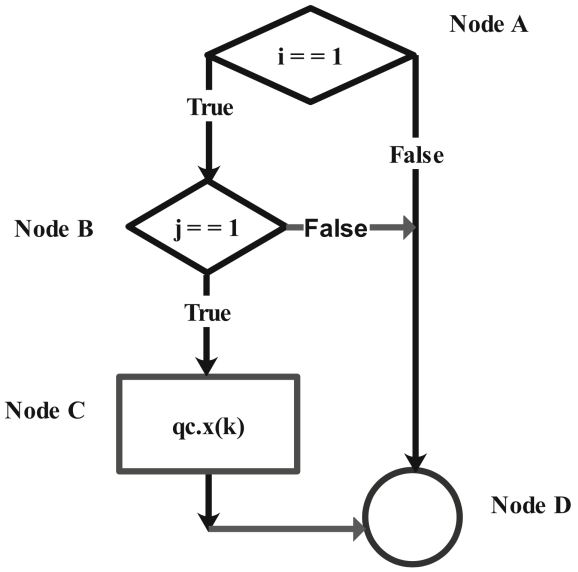
If we focus on quantum computing, we have three different possibilities for control qubits:

- If the control qubit is in the state $|1\rangle$ or $|0\rangle$, we are in the same situation as classical computing, with complexity defined by $e - n + 2p$.
- If the control qubit is in superposition, we need to consider test cases that cover that state. If there are c instances of control qubits in the QP, then the cyclomatic complexity will be $e - n + 2p + c$.

Let us now examine different control flow graphs for various gates, Figure 3.19.



(a) CNOT (i, j) control flow graph.



(b) Toffoli (i, j, k) control flow graph.

Figure 3.19: Two possible control flow graphs [76].

Let us see the possible paths in Figure 3.19b :

- Linear Independent Path 1: A, B, C, D.
- Linear Independent Path 2: A, D.
- Linear Independent Path 3: A, B, D.
- Superposition Path 1 (Control qubit $|i\rangle$ in superposition): A, $\alpha|\mathrm{B}\rangle\mathrm{CD} + \beta|\mathrm{B}\rangle$ where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$.
- Superposition Path 2 (Control qubit $|j\rangle$ in superposition): A, B, $\alpha|\mathrm{C}\rangle\mathrm{D} + \beta|\mathrm{C}\rangle$ where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$

The authors then present coverage criteria for 1, 2, or more qubits, considering the gates in use and making two assumptions: all gates are reachable and the QP is optimised (i.e., no consecutive Hadamard gates). They conclude that this criteria should be further explored, as it has only been introduced in relation to cyclomatic complexity and currently covers only quantum control.

3.2.17 Testing Multi-Subroutine Quantum Programs

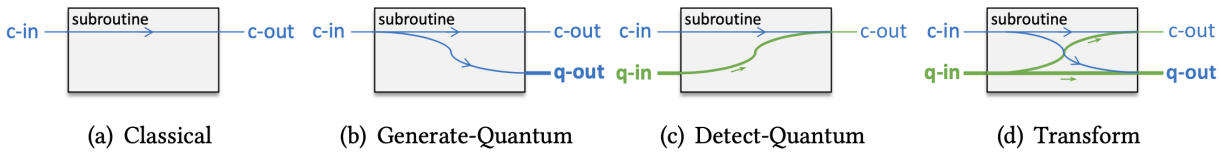
Peixun Long and Jianjun Zhao unveiled their paper, titled *Testing Multi-Subroutine Quantum Programs: From Unit Testing to Integration Testing* [78], in July 2023, building upon their earlier work presented in [79]. Their objective is to introduce novel testing techniques tailored to the unique challenges and characteristics associated with multi-subroutine quantum programs.

Until now, most research efforts have predominantly focused on testing small or fixed-scale quantum programs, without addressing the complexities inherent in practical quantum programs that involve multiple subroutines and a mix of classical and quantum inputs and outputs. The authors aim to bridge this gap in the existing literature by exploring the intricacies associated with such multifaceted quantum programs.

The authors focused their research on the following RQs:

- **RQ1**: What are the critical properties of multi-subroutine quantum programs, and how do they impact testing?

- **RQ2**: Are our unit testing design strategies effective for testing various quantum subroutines?
- **RQ3**: Is it necessary to cover both classical and superposition states as inputs to ensure adequate test coverage?
- **RQ4**: How effectively does the Superposition-Cover-All-Qubit (SCAQ) criterion reveal bugs in quantum programs?
- **RQ5**: Are our testing design strategies effective for testing multi-subroutine quantum programs?



(a) Classical (b) Generate-Quantum (c) Detect-Quantum (d) Transform

Figure 3.20: IO types [80]

To address these questions, the authors propose Q-UML diagrams to define unit testing and integration testing between subroutines.

- **A-RQ1**: Focused on approaching a quantum program (QP) from different perspectives: program structure, subroutines, and I/O types. The authors define four I/O types in subroutines: Classical, Generate-Quantum, Detect-Quantum, and Transform, as represented in Figure 3.20.
- **A-RQ2**: Based on the I/O types presented in RQ1, the authors propose 17 subroutines to cover all four types. They design a unit testing plan for each subroutine, which can be seen in [78].
- **A-RQ3/4**: The authors conduct experiments on the subroutines `CRk, Teleport, Empty, Reverse, MultiSWAP` and `QFT` using mutation testing. They use the following mutation operators: quantum gate mutation (GM), subroutine mutation (SM), classical mutation (CM), and measurement mutation (MM). CM and SM are new mutation types not used in previous research. To address the specific questions, they categorise inputs into three types: classical input (CI), random two-value superposition input (RTI), and complementary superposition input (CSI). The experiments reveal that

the trigger rate is higher for RTI than for CI, highlighting the importance of covering both classical and quantum inputs. CSI shows the highest trigger rate, providing an answer to RQ4, indicating that a broader range of superposition input states yields better results.

- **A-RQ5**: The idea is to validate this approach for QPs. To evaluate RQ5, the authors propose three case studies: Quantum Phase Estimation, Shor's Algorithm, and the Linear System Solver. The rationale for selecting these QPs is their variety of subroutines. For each case, the authors present the corresponding Q-UML diagram to facilitate the testing design. They obtained the following insights:
  - In top-down integration, quantum subroutines with an I/O type of classical can be replaced with equivalent classical subroutines.
  - If feasible, it is recommended to prioritise the use of transform-based methods for verifying the running output.
  - A recommended testing approach is to systematically scale the testing, starting from small-scale scenarios and progressively moving to larger ones.

Chapter 4

QCRMut

We have seen the different approaches different authors have taken to tackle some of challenges in quantum engineering, particularly in testing. The motivation for this work and the development of our tool is to establish metamorphic testing as an approach that can include most of the testing process in the quantum realm, aiming to delay or simplify the measurement and its inherent probabilistic nature. We will assess the quality of the proposed metamorphic rules. The approach we have chosen for this goal is mutation testing.

All tools, experiments, and results discussed in this document are accessible on our GitHub[1], where a `README.md` file is available as a guide.

4.1 Assessment of the adequacy of current tools

We have reviewed several papers that utilise mutation testing, on a smaller scale, where only a few mutants were created manually. However, we decided that automating this process would be necessary. This has already been done, with some peculiarities, by Muskit in Section 3.2.6 and QMutPy in Section 3.2.12. Initially, we considered using one of these tools with some adaptations. However, we concluded that this would not be feasible for various reasons, and therefore, we needed to create a new tool. In this chapter, we present the tool we developed, QCRMut (Quantum Circuit Random Mutation tool), which is tailored to meet the requirements for our final goal. However, why are we unable to use these tools?

We will begin with QMutPy by Fortunato et al. [8, 69]. This tool expands upon the already existing and widely used Python tool, MutPy. However, using such a tool would make it more difficult for us to modify and adapt it to our needs, including handling immutable positions, randomising generation, and working within the quantum realm in the mutated circuit prior to test execution. We will consider an important contribution from

[1] `https://github.com/sinugarc/TFM`

this paper for our research: equivalent gates. However, while we agree with the concept, we do not agree with the table they propose. Let me recall this table in Figure 4.1.

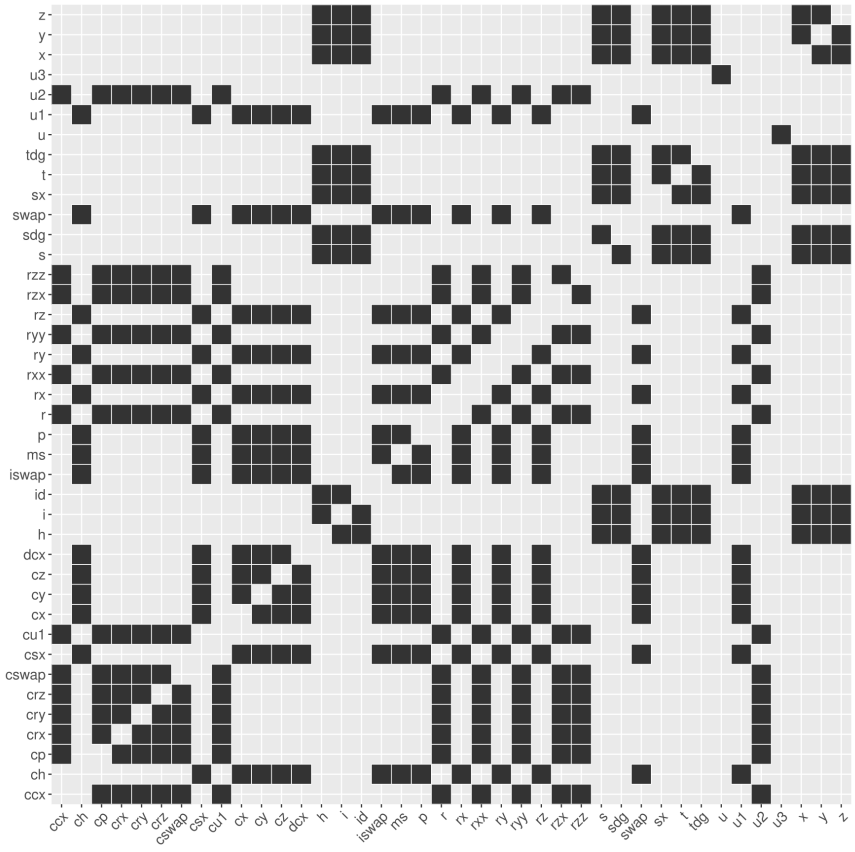


Figure 4.1: Gate equivalence matrix [69]

One of the examples is the proposed equivalence between the `ccx` gate and the `crx` gate. The `ccx` gate is applied to three qubits, whereas the `crx` gate is applied to only two qubits. Additionally, `ccx` has no parameters, while `crx` has one parameter. Due to the disagreement on the proposed equivalence classes and the difficulty of adapting MutPy to our needs, we have decided that this tool is not suitable for our research.

This brings us to one last option, Muskit by Mendiluze et al. [58]. This tool would allow us to perform mutations over the lines of code and save them as `.py` files. This is promising, as it means we could modify the mutated circuit and potentially introduce operations prior to measurement. While it is true that there are no immutable positions, there is the possibility of not mutating a position index. With this basic reduction of Muskit's applications and capabilities, it seemed like a potential option to adapt and make suitable for our research.

Therefore, we proceeded to understand the use and process of the tool.

First, we reviewed the code, which can be found on their GitHub[2] page. This revealed an unstructured, lengthy code, including nearly 10 instances of deeply nested `if` statements. This gave the code an illegible appearance, and what we can infer from this is that they operate directly over the source code. Despite our discomfort with the code, we decided that we might still be able to work with it.

We then decided to replicate the experiments, which had initially led us to question the suitability of this tool. Even after following their replication instructions, we were unable to reproduce any results. This led us to identify the first problem: the oracles required for execution were incomplete. For example, the quantum conditional execution, `CE`, oracle is incomplete; it is designed for the execution of three qubits, not the six as proposed. However, this was not the issue when trying to replicate the experiment, as it would have worked up until the distribution checks.

Afterwards, we decided to execute their mutants to see if we could replicate their results. Wanting to stay within the quantum realm as much as possible, we decided that we could save our quantum circuits as a serialised object using Python's `pickle` library and `gzip` as the compression format. This allowed us to preserve the structure of the quantum circuit in the file. Our first step was to read the mutants and save them using these libraries for subsequent analysis. This led to our first surprise: several mutants produced errors when saved as a `QuantumCircuit`, which is the class used by Qiskit to define a quantum circuit. These errors were due to faulty mutant generation, where the process was producing gates without the correct number of arguments or assigning gates to non-existent qubits. For example, a `ccx` gate, which requires three qubits, might be inserted with only one. The results of this experiment can be found in Table 4.6 in Section 4.3.1, as this will form part of a bigger experiment trying to replicate Muskit's results. However, we can find the errors in the folder `Errors` of our `muskitTest` section on our GitHub[3] page. We might have thought that the issue was limited to a specific set of gates, such as three-qubit gates. However, the fault cannot be confined to just those gates, as these gates can be inserted correctly at lower indices. This issue can be observed by comparing the mutants 151 and 189 in the QRAM, quantum random access memory, mutants on their GitHub.

In conclusion, after encountering all these inconveniences, we determined that Muskit was not suitable for our research. Having studied both possibilities, we decided to create

[2] `https://github.com/Simula-COMPLEX/muskit/tree/main/Muskit`
[3] `https://github.com/sinugarc/TFM/tree/main/Results/QCRMut/muskitTest`

a new tool. Given this opportunity, we believed that working in the quantum realm would be simpler and more effective than in the classical realm. Hence, we present the Quantum Circuit Random Mutation generator tool, QCRMut.

4.2 Tool development

In this section, we will explain the concepts guiding the development of our tool. Our primary aim is to utilise Qiskit's class structure to simplify the mutation process through high-level operations, eliminating the need to delve into the program source code. On a broader scale, due to the close relationship between Qiskit's class structure and QASM, a low-level language used by most quantum SDKs, this tool could be adapted not only to serve Qiskit but also as a foundation for other quantum SDKs. Let us introduce some concepts that will shape the structure of our mutants.

4.2.1 Insights and expectations

We aim to apply metamorphic testing to quantum algorithms. So far, it has been shown to be effective in quantum computing. We have seen how MorphQ [7] studied the Qiskit platform and how metamorphic testing can be applied to quantum programs, as demonstrated in [9], where a simple adder was analysed.

We would like to view each quantum algorithm as a whole entity, recognising that specific peculiarities may impact the mutation process. It is important to note that some algorithms do not use an initial input to set up the quantum system; instead, they incorporate the input as an oracle within the algorithm itself. For example, early quantum algorithms such as Deutsch, Deutsch-Jozsa, Simon [1], and Bernstein-Vazirani [81] are designed to solve oracle problems. These algorithms do not receive inputs at the start of the quantum circuit; rather, their input is an oracle situated within a specific part of the circuit.

Our tool has been developed to address the needs of metamorphic testing, which current quantum mutation tools do not yet meet. Since metamorphic testing relies on the relationship between inputs and outputs, preserving the position of the input is essential for accurate mutation of the algorithm. This focus on maintaining input positions is a key differentiator of our tool, ensuring the integrity and stability of the circuit for our purposes. Additionally, our tool can account for subroutines within the programs whose placement

must also be preserved. By doing so, we ensure that positions within the circuit, whether for inputs, quantum subroutines, or other components, are maintained, thereby creating a sense of immutability over these positions.

Another change from previous work is our approach to generating mutants. We will introduce a random, uniformly distributed method, across all levels. This decision stems from the consideration that future quantum algorithms under test will have many more gates than those currently used in research. An exhaustive process could result in an overwhelming number of mutants. It is true that this randomisation approach will need to be studied and compared with exhaustive methods.

We will use the structure of the `class QuantumCircuit` defined in Qiskit. We are able to obtain all our quantum circuit information with the attributes defined in the class. One of these attributes, `data`, allows us to observe the quantum circuit ordered list of `CircuitInstructions`. This provides us with the possibility of modifying the structure of the QC and even being able to change each of the instructions. This process will be fully explained in the following subsections.

To compare with previously created tools and align our research with our objectives, we propose the following research questions specifically for the study of this tool:

- **RQ1**: Does the tool produce correct mutated quantum circuits?
- **RQ2**: Does the tool hold the spaces required?
- **RQ3**: How efficient is this tool? (in terms of time and space)
- **RQ4**: How does this tool compare to already existing tools?
- **RQ5**: Does the generation seed used in the random approach affect the results?

Before we delve into the experiments and how we can answer these questions, let us explore the tool and structure, which will already provide an answer to **RQ1** and **RQ2**.

4.2.2 Definitions and basic implementation

After defining our tool and identifying its unique features, we have highlighted its ability to preserve a specific position in a quantum circuit without undergoing mutation. Let this be our first definition in the development of our tool.

To achieve this feature, we considered two different options. One option was to add an attribute to the `QuantumCircuit` class indicating the position for the input in the instruction list. However, this approach might require the attribute to be modified when mutated. Instead, we decided to implement this feature differently. We will create the `PlaceHolder` class, which is an abstraction of the `CircuitInstruction` class. This will allow us to introduce immutable elements into our circuits, and we can even name them as desired. It will help us identify them when we decide to instantiate them with the real input or subroutine. The chosen option will preserve the `QuantumCircuit` class, treating this position like any other `CircuitInstruction` since it will be a generic child of that class. This will provide us with the capability of visualisation when representing the circuit, which will assist us in metamorphic testing.

The definition can be found in Listing 4.1. Here, we can observe the differences between the classes `CircuitInstruction` and `Instruction`. The class `Instruction` will hold the name of the instruction, which, in basic terms, is the name of the gate and some basic data. However, the `CircuitInstruction` class will have the instruction defined as before and where it applies. As a clarification, `qubits` and `clbits` are attributes of `QuantumCircuit` as a list of their respective classes. Each qubit or classical bit have an specific name which depends on the way we created our quantum circuit. For example, one qubit could be seen as `Qubit(QuantumRegister(2, 'q'), 0)`, but the same qubit could be called `Qubit (QuantumRegister(2, 's'), 0)` depending on the creation process.

```
class Placeholder(CircuitInstruction):
    def __init__(self,  num_qubits: int, qubits: list[Qubit],
                 name: str = "Input", clbits: list[ClBit] = [],
                 num_clbit: int = 0, param: list[float] = [],
                 label: str = "  Input  "):
        self.name = name
        super().__init__(Instruction(name, num_qubits, num_clbit, param,
                             label), qubits = qubits, clbits = clbits)
```

Listing 4.1: Placeholder class definition

Another principle guiding our tool is to generate only mutated quantum circuits that are correct by construction. Let us first provide a couple of definitions that will help us achieve this goal. Then, we will need to work with our mutation operators and ensure that once they are applied, the produced circuit is correct.

Firstly, we are going to define a set of gates our tool will work with. We are going to use 38 gates, which we collect in the set `mutableGateSet`. These gates have been selected from a combination of the gates in QMutPy [8] and the gates in the Qiskit equivalence library. For example, `gms` is a valid gate for QMutPy but is no longer included in the Qiskit equivalence library; therefore, we decided to eliminate it. We need to ensure we understand the use of this set: the tool uses this collection to identify which gates may be mutated, but it does not affect the gates used to create new mutants. Modifying this set could create correctness problems or even raise errors when trying to access information that does not exist. We introduce `mutableGateSetIndex` as the set of indices that represent mutable positions.

Before we explain our mutant operators, let us introduce another concept: gate equivalence. The concept is similar to the one introduced by Fortunato et al. in [8], but with slight differences as we did not agree all of their proposed equivalences.

Gate equivalence: We define two gates as equivalent if they apply to the same number of qubits and if they have the same number of parameters.

This notion of equivalence allows us to exchange gates without having to change parameters, thereby maintaining the correctness of the circuit.

In our tool, this will be represented as a `dict` where we generate our class equivalence of our primary set of gates based on this definition. The key names given are representative with the definition and they will have structure `nqmp` as `str`, where `n` is the number of qubits and `m` is the number of parameters. The classes of equivalence can be seen in Table 4.1, and we may observe some differences compared to Figure 4.1 from Fortunato et al. in [8].

Key name	Gates
1q0p	{x, h, z, y, t, sx, sdg, s, tdg, id}
1q1p	{p, u1, r, rz, ry, rx}
1q2p	{u2}
1q3p	{u, u3}
2q0p	{swap, iswap, dcx, cz, cy, cx, csx, ch}
2q1p	{rzz, rzx, ryy, rxx, cu1, crz, cry, crx, cp}
3q0p	{cswap, ccx}

Table 4.1: Gate equivalence

Now that we have created the structure of `Placeholder`, defined our set of mutable gates, and established our equivalent classes, we can focus on our method of creating mutants.

First, let us examine our mutation operators, and then we will proceed with the process of mutant generation and the capabilities we have included.

4.2.3 Mutant operators

We are going to use a reduced number of mutant operators. There are more operators presented in previous works [8, 58]. However, our interest is the main quantum circuit without the consideration of measurement as we will want to have posterior treatment of the circuits. Therefore, we are going to ignore any mutation which includes measurement gates. Let us see the mutants operators and their characteristics:

Insertion operator: This operator allows the insertion of a gate from our gate set at any position in the circuit. It is used as a "joker" operator for the extreme case where the circuit does not have any mutable gates. As expected, this operator does not use the `mutableGateSetIndex`, as we could insert at any position.

Deletion operator: This operator removes the corresponding `CircuitInstruction` from the circuit, based on the list provided by `data`.

Gate name operator: This operator swaps the corresponding gate with a new equivalent but different gate, based on the class equivalence between gates.

Gate attributes operator: This operator modifies the gate attributes: qubit application and parameters of the gate. As it is able to obtain the class of the mutable gate, it will have enough information to produce the right attributes for such a gate.

All these operators represent common errors that programmers may encounter while coding. We will show how these mutations produce correct circuits by construction, thanks to the definitions introduced earlier. However, let us see some visual examples of these operators first in Table 4.2. The first column represents our original circuit, which QCRMut receives to be mutated.

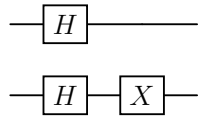
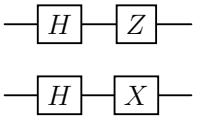
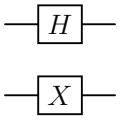
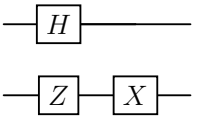
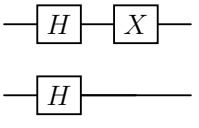
Op	Original QC	Insert	Delete	Name	Attribute
QC					

Table 4.2: Mutant operators example

Lets briefly explain what the operators did in Table 4.2. We can not ensure which index they using as the visualisation would allow different index for the same representation.

Insertion operator: A Z gate is inserted on the first qubit.

Deletion operator: A H gate is deleted on the bottom qubit.

Gate name operator: The bottom H gate has been transformed into a Z gate, with the attributes kept the same.

Gate attributes operator: The X gate of the quantum circuit has been modified, where we have changed the qubit application within the class `CircuitInstruction`.

4.2.4 Mutant generation

Once we have all the basic components for our tool, it is important to remember that we employ a uniformly random approach throughout the mutation process. Let us observe the main structure of our tool, as represented in Figure 4.2, which outlines the workflow of QCRMut for create a single mutant:

- The main function receives the quantum circuit and obtains the mutation index randomly for each mutant produced.
- An auxiliary function is responsible for generating a mutant. It randomly selects the mutation operator, and if there are no exceptions, it calls the appropriate operator function. If an exception occurs, it selects another operator at random.
- The operator function creates the new mutant.
- Once a mutant, not equal to the original quantum circuit, has been produced, it is returned and saved, if desired, to a text file using the libraries pickle and gzip.

Let us begin with the first step of our process. We will initiate the tool executing `mutant_generator` with arguments: the quantum circuit to be mutated, the number of mutants desired, and options for specifying the seed, saving the mutants, and choosing the path for saving them. This function will prepare the quantum circuit for mutation.

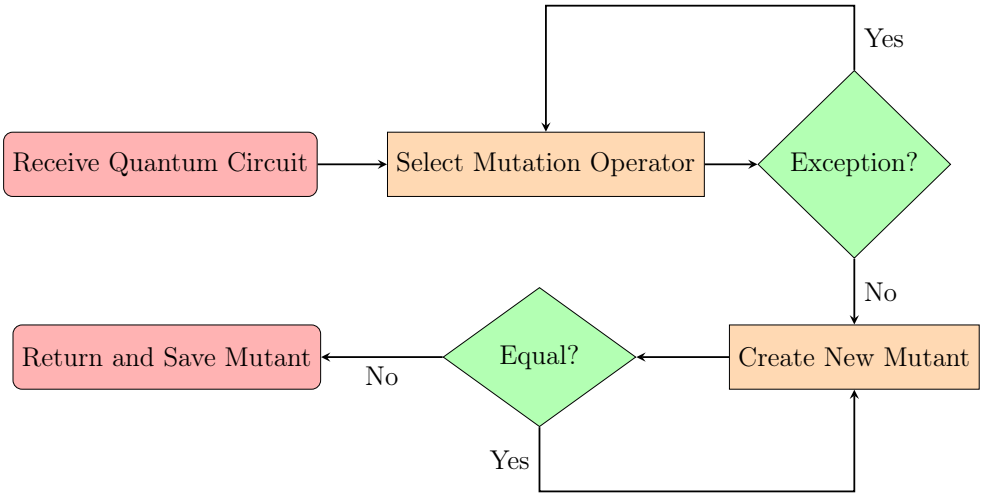


Figure 4.2: QCRMut workflow

Once we receive a quantum circuit for mutation, the first task is to identify the mutable gates. As mentioned before, the structure of the `QuantumCircuit` with the `data` attribute allows us to obtain a ordered `list` of `CircuitInstruction`. We will collect the indices of all mutable gates in their `data`, checking if the name of each internal `Instruction` is in our `mutableGateSet`. This will be represented with a variable called `mutableGateSetIndex`.

Then, the function checks if `mutableGateSetIndex` is empty as this is the first exception we might encounter. If this is the case, the only possible operator that can be applied is the **insertion operator**, which will be called for each required mutant. Otherwise, it will pass the generation to the auxiliary function with an index to be mutated obtained at random on every loop. Once the mutant is created, a serialised and compressed version can be saved, if desired. However, note that the main function will always return a generator, calling the appropriate function depending on the exception encountered.

Now, in the auxiliary function, we already know the index that we are going to mutate. Once the mutation operation is randomly selected, the function will handle various exceptions as needed and select a new mutation operator if necessary. For example, this could occur if the change name operator is selected and there is only one equivalent gate available, or if we attempt to change the attributes of a gate and are trying to mutate a single-qubit operator with no parameters in a single-qubit quantum circuit.

Afterwards, if needed, it will obtain its equivalence class name and call the appropriate function for each operator, except for the deletion operator, which is as simple as removing the element at the desired index. Each operator function has its own characteristics.

Let us discuss them separately and examine how we ensure uniform randomness.

Inserting operator: The tool identifies the number of qubits in the quantum circuit being mutated, which determines the gates available for insertion. Depending on the number of available qubits, it randomly selects a number, with weighted probabilities. The aim is to ensure that all gates have an equal chance of being inserted. These weights are obtained from the dictionary created with our 38 gates, which can referenced in Table 4.1. Similarly, it selects the number of parameters, which depends on the number of qubits obtained. For example, if we are inserting a 1-qubit gate, the possible parameters range from 0 to 3, whereas for a 3-qubit gate, the only option is 0 parameters. However, we will not allow the identity gate to be inserted or any rotation with a parameter of 0, as this would generate equivalent mutants.

Gate name operator: We have identified the gate equivalent class at the desired index, ensuring there is more than one element in that class. Then, it randomly selects a new name different from the original one. Finally, we change the `name` attribute in the `Instruction` class to the new name. Since we do not modify any other elements of the quantum circuit, this is the only change required for the mutant in a deep copy of the original circuit.

Gate attributes operator: It will obtain as many qubits as needed from the entire set of qubits in the quantum circuit using the `sample` function from the `random` library. Then, it will proceed in the same manner for the number of parameters needed, but in this case, it will use the `uniform` function. Once it has ensured that they are not the same as the original ones, it will proceed to change them in a copy of the original circuit. However, in this case, the change must be done at two different levels: qubits will be an attribute of the `CircuitInstruction` class, and parameters will be in the `Instruction` class.

When the main function has finished if we have selected to save the mutants, it will produce a report with relevant information such as the time needed for all mutant generation, the number of mutants, or the seed used.

Once we understand the entire generation process, we can then address **RQ1**: Does the tool produce correct mutated quantum circuits? Assuming the quantum circuit provided is correct, which it will be as it will only receive an element of the `QuantumCircuit` class, we can ensure that the mutated circuit will also be correct. Any changes made in the internal

structure of the `QuantumCircuit` obtain their information from the same circuit or from the equivalent gate dictionary. The most complex operation would be the insertion operator, but even this one follows the entire structure of creating a `CircuitInstruction` element. Finally, the last test is that we are using `pickle` to save our mutants into a text file. This library saves the circuit as an element of the `QuantumCircuit` class, so if it was not correct, it would produce an error.

We could also address **RQ2**: Does the tool preserve the necessary spaces? Understanding the structure of the tool, it only inserts new instructions or modifies the ones in the `mutableGateSetIndex`. Since no placeholder instruction can be included in the `mutableGate SetIndex`, we can conclude that such positions are kept immutable and in the corresponding space, which will always be in the same position in the original `qc.data` or at most one position further, as we could insert a gate in a previous gap.

4.2.5 Coverage and analysis

When it comes to testing, one of the key concepts we rely on is coverage analysis of the code. In classical computer engineering, it is common to measure the lines of code, LOC, covered during the mutation process. Previous works such as Muskit [58] and QMutPy [8] have adopted this principle in their approaches. Their methods are exhaustive, aiming to cover either all LOC [58] or all nodes of the Python syntax tree [8], utilising MutPy's structure.

However, in our case, as we are mutating the `QuantumCircuit` class, we will assess the coverage based on the number of gates rather than the LOC, which is common in quantum.

As our tool mutates the quantum circuit at random, we could have implemented an internal checker which saves all gates or positions mutated. However we will study our concept of full coverability in a different way. We would like to study how many mutants we need to generate to achieve full coverage on average. This study can be approached using The Coupon Collector's problem [82] as a reference.

We consider achieving full coverage when we mutate each position at least once, including both the positions of insertion and those within `mutableGateSetIndex`. This concept can be related to The Coupon Collector's problem. To simplify this approach, let us assume that the cardinality of `mutableGateSetIndex` is bounded by `len(qc.data)`, denoted as N. Therefore, we aim to collect $N+1$ *coupons* from the insertion operator and N *coupons* from `mutableGateSetIndex`. Each group of *coupons* has a different probability: the insertion group has a probability of $1/(N+1)$, while the rest are bounded by $1/N$. Our goal is to

ensure that we obtain all *coupons*, which, in our tool, would mean mutating each position.

We could use a formula to solve this problem with different probabilities, but for simplicity, we will take a different approach. We will focus on a single question: How many mutants do we need to generate using the insertion operator to obtain the $N+1$ *coupons*?

The solution to this question aligns with the equiprobable Coupon Collector's problem for n coupons. It is important to remember that this solution represents the expected number of executions needed. It is the expected value or mean of the distribution representing all possible values of obtaining a full collection:

$$\mathsf{exe}_{\mathsf{ins}}(n) = n\, H_n = n \sum_{i=1}^{n} \frac{1}{i} \tag{4.1}$$

Using Cauchy's Integral Test, also known as Maclaurin–Cauchy test, we can bound H_n:

$$ln(n) + \frac{1}{n} \leq H_n \leq ln(n) + 1 \tag{4.2}$$

If we consider the upper bound and $n = N + 1$ we obtain:

$$\mathsf{exe}_{\mathsf{ins}} = (N+1)\lceil ln(N+1) + 1 \rceil \tag{4.3}$$

Therefore, to achieve the desired coverage, we are going to use the insertion operator $(N+1)\lceil ln(N+1)+1 \rceil$ times, where N represents the number of instructions in the circuit. Given that we only utilise this operator a quarter of the time during the execution of our mutant generator, ensuring full coverage for the tool entails:

$$\mathsf{exe} = 4\,(N+1)\lceil ln(N+1) + 1 \rceil \tag{4.4}$$

However, we decided that full coverage had to include obtaining all desired coupons, including the ones for modifying or deleting existing gates. Let us now examine how this number of executions also covers the *missing* positions and the reason why we decided to simplify the problem. We will access these executions with a probability of $3/4$, thus:

$$3(N+1)\lceil ln(N+1) + 1 \rceil > N \lceil ln(N) + 1 \rceil \geq N\, H_N \tag{4.5}$$

Name	# Gates	# Mut Gates	# Min mutants
IQFT	24	24	500
CE	17	12	288
BV	9	9	160

Table 4.3: Quantum Algorithms

We could conclude, then, that with $4(N+1)\lceil ln(N+1)+1\rceil$ executions, we fulfil the desired coverage, as this serves as an upper bound for the solution of the problem involving other operators, which is $N\,H_N$. There is even a possibility of expanding our definition of coverage. With this number of executions, we could say that not only has every position been mutated at least once, but also that every position has been mutated at least once by every operator. However, we should recall that the solution to the Coupon Collector's problem provides the expected value and does not ensure it. Despite that, we are going to consider such a value as a good approximation. As we are considering this as a good approximation, we will use the experiments in the following section to strengthen this hypothesis.

4.3 Empirical experiments

We are going to conduct two main experiments. The first will give us insights into the efficiency in time and space of this tool, allowing us to compare it with QMutPy [8] and Muskit [58]. Then, we will proceed with a second experiment where we try to reproduce the same experiment as used by Muskit in [58]. This will enable us to compare our results with random approach and their results with an exhaustive one. To facilitate comparisons, we will use three quantum algorithms from [58], which can be seen in Table 4.3, where we present the name and number of gates. We have also included the calculation, using Equation 4.4, for the bound to reach our full coverability.

All experiments are fully reproducible. To ensure this, we used two different seeds: one to generate the same mutants and another to execute them, ensuring that the information we aim to achieve remains unaffected. To reproduce the results, download the folder `Replication` and follow the instructions in the `README.txt` file. The results can be found in the `Results/QCRMut` folder.

4.3.1 Efficiency experiment

We will proceed by generating 10^2, 10^3, 10^4, and 10^5 mutants for each algorithm in Table 4.3 and analyse their execution time and memory usage. Firstly, we will calculate the average execution time per mutant and then, per mutant and gate. We aim to determine whether the number of gates in each circuit directly affects the running time of our tool. Finally, we will perform simple calculations regarding the size of our saved files. As explained previously, to ensure reproducibility, we have used `seed 1` for the `random` library.

Before executing our tool, we obtain the code for each algorithm from [58] and adapt it to generate a `QuantumCircuit`, as our tool does not mutate over lines of code (LOC). Results for each algorithm and number of mutants are saved and organised in the folder `Experiment_I`, with a summary available in the folder `Experiment_I_Results`. All information in Tables 4.4 and 4.5 is obtained from these folders. However, due to the large number of files, in the case of 10^5 mutants, we have decided to retain only information related to time execution.

Name	# Mutants	Total (s)	Avg Mut[1] (ms)	Avg Gate[2] (μs)
IQFT	10^2	0.043	0.430	17.917
	10^3	0.435	0.435	18.125
	10^4	4.405	0.441	18.370
	10^5	44.751	0.448	18.660
CE	10^2	0.039	0.390	22.941
	10^3	0.376	0.376	22.118
	10^4	3.709	0.371	21.824
	10^5	38.158	0.382	22.471
BV	10^2	0.028	0.280	31.111
	10^3	0.282	0.282	31.333
	10^4	2.905	0.291	32.333
	10^5	29.375	0.294	32.667

[1] Average time per mutant
[2] Average time per mutant and gate

Table 4.4: Experiment I, Creation time

Let us dive into the results from Table 4.4, which will also be compared with the time efficiency experiments conducted by Fortunato et al. in [8, 69]. However, since those experiments were conducted on a different machine, we will only provide a brief insight into the efficiency of our tool.

If we focus on a single mutant, we observe that all algorithms have an average creation time lower than 0.5 ms, indicating that our tool is efficient. According to [69], it is considered adequate for each mutant to be produced in less than 0.1 seconds, a threshold we meet. Notably, QCRMut can generate over 10^5 mutants in less than one minute, even for the algorithm with the highest number of gates (24 gates). However, when considering the average creation time per mutant and gate, as shown in the last column of Table 4.4, we observe that the algorithm with the highest number of gates has the fastest creation time. This contradicts the expectation that creation time increases proportionally with the number of gates in the circuit.

This observation could be explored further in future work, although the efficiency of the tool surpasses the threshold, suggesting that the impact may not be significant. Nevertheless, it is possible to hypothesise that creation time is mainly linked to the type of gates, with certain types being easier to modify (e.g., `3q0p` compared to `2q1p`), and to a lesser extent, the number of qubits in the algorithm. This hypothesis stems from our commitment to ensuring that every gate is equiprobable when it comes to being inserted or modified.

Name	# Mutants	Total (MB)	Avg Mut (KB)
IQFT	10^2	0.134	1.374
	10^3	1.341	1.373
	10^4	13.418	1.374
CE	10^2	0.129	1.318
	10^3	1.291	1.322
	10^4	12.915	1.322
BV	10^2	0.117	1.195
	10^3	1.168	1.196
	10^4	11.687	1.197

Table 4.5: Experiment I, Memory

While Table 4.5 was obtained directly from the files mentioned before, we now need to compare it with the mutants generated by Mendiluze et al. in [58]. This initial comparison could be easily done by obtaining the mutant files from [58]. However, in subsequent experiments, we aimed to reproduce their results but were unable to do so. Therefore, as explained at the start of this chapter, we decided to replicate them more closely to align with our structure.

Our first step was to transition to PGZ files as `QuantumCircuit` objects from plain Python text files. We added a few lines of code at the beginning and end of each file to save them in our chosen format. There were some errors, previously mentioned, in this transformation, which can be found within the `muskitTest` folder. This is why, in Table 4.6, there is a discrepancy in the number of mutants between the original and the serialised version.

Name	# Mutants # PGZMutants[1]	Total (MB) Total (MB)	Avg Mut (KB) Avg Mut (KB)
IQFT	372 360	0.238 0.482	0.655 1.372
BV	343 333	0.084 0.392	0.250 1.205
CE	458 446	0.186 0.585	0.415 1.342

[1] Number of mutants saved as PGZ, serialised object.

Table 4.6: Muskit, Memory

Analysing Tables 4.5, we can observe that the number of gates in a `QuantumCircuit` does not significantly affect the size of the saved file, as indicated by column 3. However, when we refer to Table 4.6, we notice a slight difference compared to our results, which becomes more pronounced when the number of gates is reduced. This discrepancy arises from the structural difference in how a line of code (LOC) affects a Python text file compared to how a gate affects a `pickle` element. The latter has an initial greater weight due to the necessity of saving the class of the element.

However, following these trends, there will be a point where, for more complex algorithms, the memory needed to store the mutants is either equal to or even lower than that required for a serialised element. Saving our mutants in this manner has an additional benefit: we can be assured that all saved mutants are correct `QuantumCircuit` objects.

Memory experiment: Py - PGZ

We have proposed a hypothesis about the memory usage of our mutants compared to lines of code (LOC). Let us introduce a quick experiment to observe how they change and at what point the space used by the Python (`py`) file will possibly be higher than the PGZ file.

We will consider the IQFT algorithm as it has the highest number of gates, which we will first round up to 25 gates. Then, we will consider an inclusion of 5 Hadamard gates at a time, which is probably one of the shortest possible gate insertion texts. We will proceed until we have passed 100 gates. You can observe the results of this experiment in Table 4.7.

# Gates	Py (MB)	PGZ (KB)
24	0.676	1.354
25	0.688	1.354
30	0.743	1.363
35	0.798	1.368
40	0.853	1.370
45	0.907	1.375
50	0.962	1.374
55	1.017	1.377
60	1.071	1.377
65	1.126	1.376
70	1.181	1.387
75	1.235	1.397
80	1.290	1.393
85	1.345	1.396
90	1.399	1.395
95	1.454	1.396
100	1.509	1.400

Table 4.7: Memory experiment Py - PGZ

However, let us take a visual approach. Figure 4.3 represents the previous results in Table 4.7. We can observe how the size of `PyFiles` grows faster than `PGZFiles`, while the latter maintains the space usage within a reasonable range.

Let us recap and revisit **RQ3**: How efficient is this tool? We could conclude that QCRMut is an efficient mutant generator tool that meets or exceeds the standards of current tools in terms of creation time and memory usage, and it even has the potential to improve upon them. Additionally, it offers features such as ensuring the correctness of the mutants.

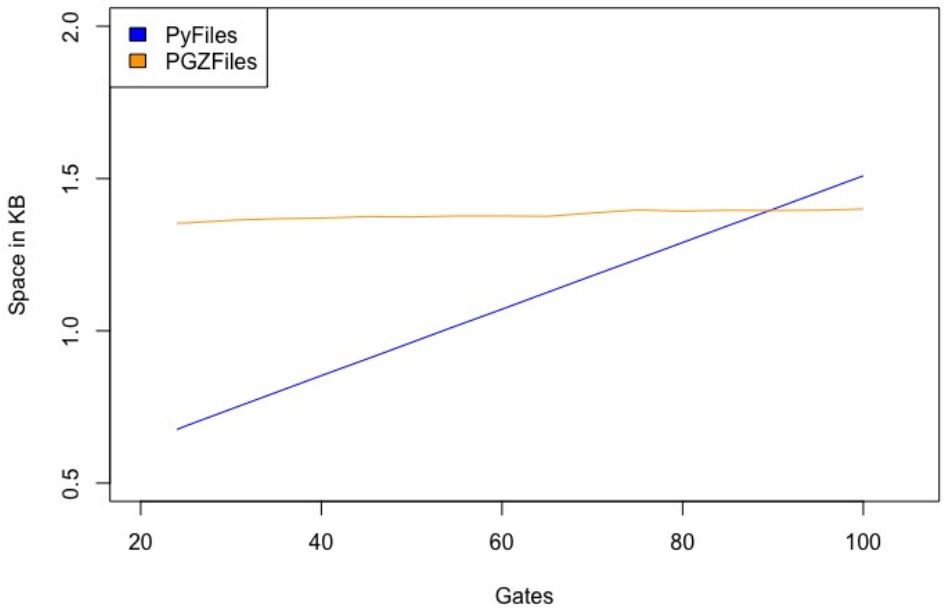


Figure 4.3: Memory experiment Py - PGZ results

4.3.2 Mutation experiment

So far, we have demonstrated that QCRMut produces correct mutants using an equiprobable random approach, and we have shown its efficiency. We have also introduced a formula that can provide a different perspective on coverage, moving away from the classical LOC percentage. Now, let us delve into the next experiment, where we will execute our mutants and compare the results with those provided by Mendiluze et al. in [58].

Our goal of comparing with Muskit [58] has led us to design this experiment to ensure maximum comparability. We will start by reading the PGZ file, then adding our different inputs and measurements to our mutants. Although the inputs can be generated at random, we will consider all possible inputs to mimic their experiments. Subsequently, we will proceed to execute them using `seed 42` in `AerSimulator`, with 100 and 1024 executions. The oracle analysis with the execution results will have the same setup as presented by Mendiluze et al. in their experiments [58], which has also been used in previous works such as [26] and [53]. For this analysis, we will set the `p-value` to 0.05 and utilise the `chiSquare` function from the `scipy.stats` library. To proceed, we are using the oracles from their experiments, although

we have increased them for completeness. These oracles will be read and transformed into a `dict` to facilitate the analysis process. The results for this experiment, reflected in Table 4.8, can be found in folders `Experiment_II` and `Experiment_II_Results`.

Name	# Mutants	# Rep	# Mutants killed	Mutation Score
IQFT	500	100	479	95.80%
		1024	157	31.40%
CE	288	100	288	100%
		1024	288	100%
BV	160	100	160	100%
		1024	160	100%

Table 4.8: Experiment II, Mutation Score

We could have relied on the results from [58]. However, due to some inconsistencies and our inability to reproduce the results, we opted to execute the files we had prepared from Experiment I using the same process to obtain the mutation scores. One reason we chose to use 1024 repetitions is to gather sufficient data to apply the `chiSquare` test on our output distribution, making the results statistically significant. Therefore, after observing the differences between 100 and 1024 executions for IQFT in Table 4.8, we decided to apply 1024 repetitions to Muskit's IQFT mutants as well. Results from Table 4.9 can be found within the `muskitTest/Results` folder.

Name	# Mutants	# Rep	# Mutants killed	Mutation Score
IQFT	360	100	359	99.72%
		1024	96	26.67%
CE	446	100	446	100%
BV	333	100	333	100%

Table 4.9: Muskit, Mutation Score

If we compare Tables 4.8 and 4.9, we can observe that the results are similar, although not completely identical. We will assume this discrepancy is likely due to the different gate sets we used to produce mutants. However, we can still conclude that our results, obtained with a random approach using the calculated number of mutants based on the number of gates, yield similar outcomes to the existing work done by Mendiluze et al. [58].

One notable difference is observed between the results obtained with 100 and 1024 repetitions in IQFT. The main reason for this difference is the statistical test we chose to use, `chiSquare`. This test requires a minimum amount of data for each entry to yield meaningful results. With only 100 repetitions, the test may not provide sufficient data for meaningful analysis, potentially leading to inconclusive or non-significant results. This issue is particularly pronounced with IQFT due to its specific characteristics or the distribution of results, which may require a larger sample size for accurate statistical analysis. Why is this only a problem with IQFT?

We can observe that BV or CE do not encounter the same issue, which stems from differences in the oracles and possible outcomes. For BV and CE, the potential outcomes for each input are at most 4, and they are equiprobable. Therefore, when we conduct 100 repetitions, there is sufficient data for each value to apply such a test effectively.

However, in the case of IQFT, every input can yield any of the possible 64 values. This variability means that if we only produce 100 repetitions, our analysis may not yield reliable results due to the wide range of potential outcomes.

This could align with the initial idea behind our tool development, which was to understand each quantum program on its own terms rather than through classical paradigms. Each algorithm possesses unique characteristics, one of which may be the representation of inputs. While we introduced inputs as an initial state to closely mimic the setup of [58], this approach does not necessarily reflect how we understand quantum programs.

Survivors study

If we observe Tablea 4.8 and Table 4.9, we can identify slight differences in the IQFT results with 100 repetitions between Muskit and QCRMut. While one mutant survived in Muskit's results, 21 mutants survived in QCRMut's results. Considering that Muskit conducts an exhaustive study of mutants, all mutants generated by QCRMut should be a subset of Muskit's mutants for the same algorithm. Let us summarise the survived mutations.

Since QCRMut does not record where the algorithm has been mutated, we need a different approach. Firstly, we must analyse the results to identify the surviving mutants, which can be done using the `grep` command to search for the result `[0,64]`, indicating that no input has killed the mutant. Once we have identified the surviving mutants, we can obtain the corresponding `QuantumCircuit` represented in each file. Then, we can compare them with the original circuit. It is important to note that some mutations may be deeper in

the data list as parameters, so how we define equality when comparing is crucial. Following these steps, we can arrive at our results as shown in Table 4.10.

Mutant	Mutant operator	Pos	Mutation
3	Gate name	22	cu1 -> crx
48	Delete	1	-
56	Delete	0	-
81	Insert	6	cu1
91	Gate name	9	h -> sx
126	Delete	5	
135	Gate attributes	17	(cu1) Qubits : (2,5) -> (0,2) (cu1) λ : 0.393 -> 0.426
144	Gate attributes	2	(swap) Qubits : (2,3) -> (0,4)
157	Delete	10	-
173	Insert	15	swap gate
192	Delete	0	-
222	Insert	22	rxx gate
264	Insert	15	cu1 gate
292	Gate name	5	cu1 -> rxx
312	Insert	9	h gate
391	Insert	19	y gate
440	Gate name	6	cu1 -> rzz
443	Insert	19	t gate
458	Delete	10	-
492	Insert	8	cu1 gate
495	Gate attributes	8	(cu1) Qubits : (0,5) -> (2,4) (cu1) λ : 0.098 -> -0.018

Table 4.10: QCRMut, surviving mutants for IQFT with 100 Rep

One might think that the differences in the gate sets used by these tools are affecting the results. However, we wanted to identify the mutants which survived in our tool but they were killed on theirs. On this try of identifying the mutants, we noticed that several files had big difference on their size with the main algorithm. They were reasonably smaller, a third of the size. Therefore, we decided to identify them, obtaining a big surprise and it is that the problems within the tool were a lot more complicated than we expected.

Let us show one example, although there are several more. In Muskit's IQFT mutants, we are going to consider the original algorithm and the mutation *78ReplaceGate.py* in Figure 4.4, where we ignore the imports and initiation of the quantum circuit:

```
11      qc.swap(q[0], q[5])
12      qc.swap(q[1], q[4])
13      qc.swap(q[2], q[3])
14      qc.h(q[0])
15      qc.cu1(pi/2**(1), q[0], q[1])
16      qc.cu1(pi/2**(2), q[0], q[2])
17      qc.cu1(pi/2**(3), q[0], q[3])
18      qc.cu1(pi/2**(4), q[0], q[4])
19      qc.cu1(pi/2**(5), q[0], q[5])
20      qc.h(q[1])
21      qc.cu1(pi/2**(1), q[1], q[2])
22      qc.cu1(pi/2**(2), q[1], q[3])
23      qc.cu1(pi/2**(3), q[1], q[4])
24      qc.cu1(pi/2**(4), q[1], q[5])
25      qc.h(q[2])
26      qc.cu1(pi/2**(1), q[2], q[3])
27      qc.cu1(pi/2**(2), q[2], q[4])
28      qc.cu1(pi/2**(3), q[2], q[5])
29      qc.h(q[3])
30      qc.cu1(pi/2**(1), q[3], q[4])
31      qc.cu1(pi/2**(2), q[3], q[5])
32      qc.h(q[4])
33      qc.cu1(pi/2**(1), q[4], q[5])
34      qc.h(q[5])
```

(a) IQFT original code

```
11      qc.swap(q[0], q[5])
12      qc.swap(q[1], q[4])
13      qc.swap(q[2], q[3])
14      qc.h(q[0])
15      qc.h(q[1])
16      qc.h(q[2])
17      qc.h(q[3])
18      qc.h(q[4])
19      qc.sx(q[5])
```

(b) IQFT mutated code

Figure 4.4: Differences in a single mutation in Muskit.

That mutation replaced all `cu1` gates for a single `sx` gate. There are some mutants that treat the blocks of `cu1` gates as one, however this one treats all of them as one even when in the original algorithm are introduced separately. Once we found this inconsistency, or in our opinion, concluded that this should not be considered a single mutation, we decided to abort any further comparison. This further strengthened our decision to create a new tool.

Finally, we can address our last aim, **RQ4**: How does this tool compare to existing tools? We can conclude that our tool produces similar results to previous works, even when implementing a random approach with a non-exhaustive mutant generation.

All these experiments and the ones in the following sections have been executed on Python 3.9.13 and qiskit 0.38.0 of a system with Linux version 6.1.0-17-amd64, Debian 12.2.0-14, 6 physical/12 virtual Intel(R) Xeon W-2235 CPU @ 3.80GHz processors with 128GB RAM.

4.4 QCRMut Summary

In this section, we introduced QCRMut as an alternative approach to creating mutants for quantum circuits, offering the flexibility to maintain a desired structure for algorithm robustness. Our method provides an efficient means of generating random mutants that are correct by construction, yielding results comparable to those produced by exhaustive mutant generators. Departing from the classical notion of mutating over lines of code, LOC, our approach operates within the quantum paradigm, focusing on the number of gates in the algorithm. This allows users the freedom to execute, measure, and analyse quantum circuits with ease.

Our proposed execution aligns with previous works by Mendiluze et al. [58], which require the use of an oracle. Alternatively, Fortunato et al. [8] employ an assertion-based approach, utilising a non-mutated version of the algorithm to derive results.

We have addressed most of the RQs proposed in Section 4.2.1. However, we have left the final question, **RQ5**: Does the generation seed used in the random approach affect the results? We chose to defer this question because, although it applies to the mutant generator, we would like to conduct the experiment on the mutants generated for our MRs. Therefore, while the question is introduced here, it will be answered in the following chapter.

Chapter 5

Metamorphic Testing

Finally, we present this chapter, in which we will develop the main topic of this work: applying metamorphic testing to quantum algorithms. To achieve this, we will introduce our algorithms and the metamorphic rules we will apply to them. We will also discuss how we adapt these rules and their implementation as quantum circuits. Then, we will assess the quality of our rules using mutation testing, which is the reason for creating the tool presented in the previous chapter, QCRMut (Chapter 4).

One of the goals of any testing procedure is the automation of the testing process. Therefore, we will begin by introducing an abstract structure designed to automatically generate the entire testing process. Additionally, we will provide a class definition that will allow us to consolidate all the information from a rule, making it ready for testing. Let me begin with this, then we will follow with each algorithm.

5.1 Mutation testing automatic tool

The definition of these abstract classes is derived from the general workflow of the testing process, which can be found in Figure 5.1. We will also develop the execution of a single mutant, as shown in Figure 5.2. This process will be repeated for each mutant.

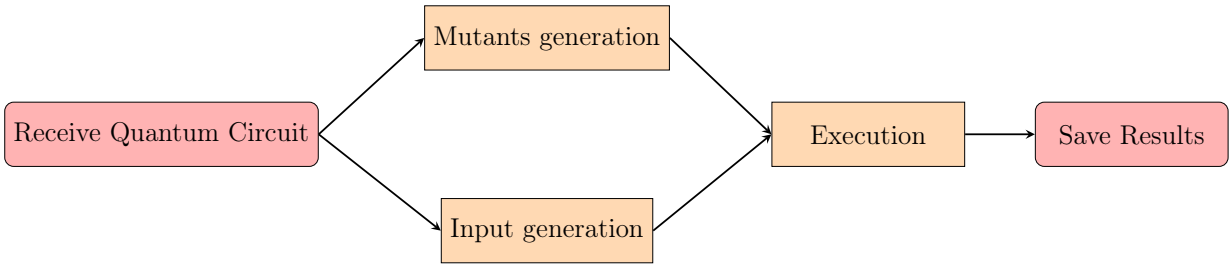


Figure 5.1: Simplified Testing Workflow

We can now define the first class, `QMutation_TestingStructure`. This class will collect the information needed to perform mutation testing in quantum computing: the original algorithm, mutant generator, input generator, test circuit generator, execution method, and analysis function. All these functions will be assigned as attributes and will be accessible as class properties. Once this class is instantiated with all its attributes, it will define a metamorphic rule for a chosen algorithm, as we will have provided all the necessary information to perform all the required steps.

Recalling the definition of a metamorphic rule, MR, by Chen et al. [10]: a MR is a relationship between a sequence of more than one input and their corresponding outputs. The input generator will provide the flexibility to create the appropriate input relationship for our MR, the test circuit generator will help establish the relationship between inputs and outputs, and the analysis function will enable us to determine the survival of the mutant based on the expectations of our MR.

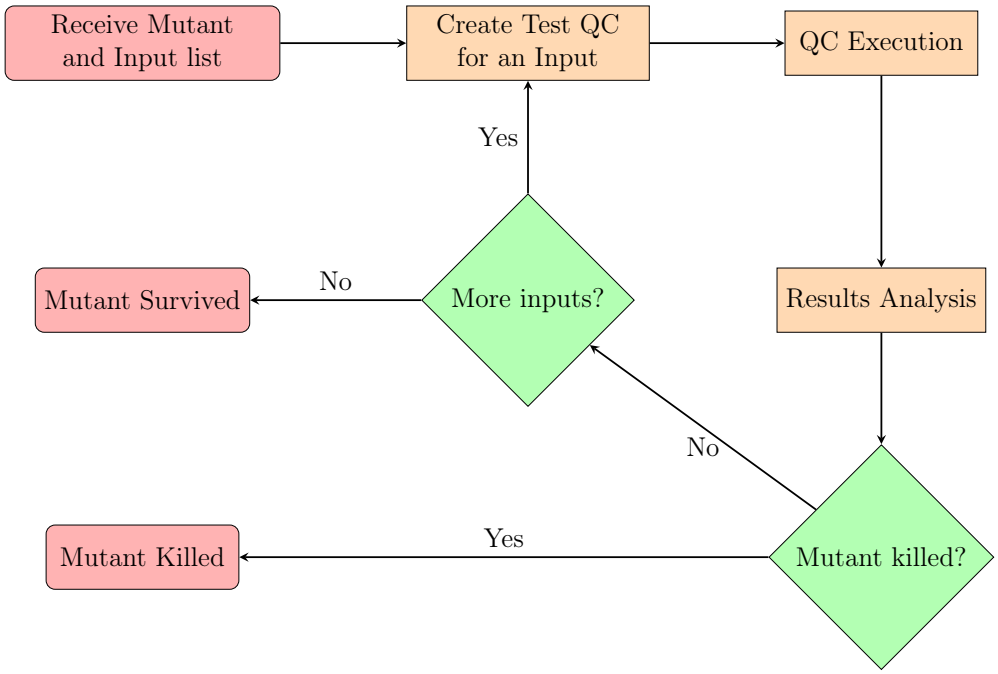


Figure 5.2: Execution Workflow for a Mutant

Let us now focus on the execution process. Although we could have combined this as a method within the previous class, we decided to create a new class and separate it from the initial one. Therefore, we distinguish between defining an MR as an instantiation of the class `QMutation_TestingStructure`, which only allows consultation of the attributes, and the new abstract class, `Q_Test`, which automates the testing process. In this new class, the

attributes will define various characteristics, such as the number of qubits or mutants, as well as any other arguments we wish to use as inputs for our functions.

`Q_Test` will receive as arguments the test structure, which is defined by the previous class, and the information needed for mutant generation, as we consider this the starting point of the testing process. We have also created a method called `execution`, which requires the following parameters: the results path, the number of processes to use for parallel execution, and the necessary arguments for input generation and the execution function.

This method will return an iterable of all non-killed quantum circuits and the mutation score of the test. The results will also be saved, including all argument information. This method will follow the structure outlined in Figure 5.2. It is important to note that if any user wishes to utilise this structure, it is their responsibility to ensure that the function types and arguments match. To assist users, we have chosen to type all functions defined in our code. According to the workflow defined by Figure 5.2, it is expected that connected functions have corresponding types. For example, the output type of *QC Execution* must match the input type for *Results Analysis*.

We are now going to introduce a couple of functions we created to support our classes, as these will be shared or indirectly used by all MRs. There are a few more functions that can be found in the file `shareFunctions.py`, but these two will be the most important ones:

Input generator: As the name indicates, we will use this function to generate random inputs, all different and within the correct scope, defined by the number of qubits. We can set a maximum number of inputs using a keyword argument, with a default initial value of 64. The function identifies the maximum number of inputs possible, 2^n where n is the number of qubits, and if we request more inputs than possible, it will return all possible inputs.

Execution: This function will execute our quantum programs according to Qiskit's requirements. We have included the option of not using simulation. However, in this work, we will always use the simulation option.

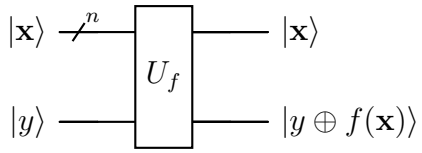
Let us now introduce the algorithms we are going to study and explain how we adapt them and their MR to the testing process defined in this section.

5.2 Deutsch-Jozsa

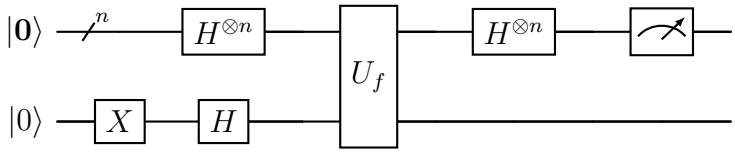
The Deutsch-Jozsa algorithm was proposed by David Deutsch and Richard Jozsa in their 1992 work titled *Rapid solutions of problems by quantum computation* [83]. This algorithm aims to solve the following problem: Let $f : \{0,1\}^n \to \{0,1\}$ be a function with $n \in \mathbb{N}, n > 1$ that has one of the following properties:

- We say f is **constant** if all the inputs map to either 0 or 1.
- We say f is **balanced** if exactly half of the inputs map to 0 and the other half map to 1.

If we consider f as an oracle, the Deutsch-Jozsa algorithm, DJ, will determine whether the function is balanced or constant. Let the oracle of f, U_f, be the define by its unitary matrix as follows:



Deutsch-Jozsa algorithm:



Let us examine the possible outcomes depending on the function's properties:

- **Constant**: The outcome can only be $|\mathbf{0}\rangle$.
- **Balanced**: The outcome can be anything except $|\mathbf{0}\rangle$.

It is important to note that if the function is neither balanced nor constant, the algorithm's output will not be interpretable. Therefore, this is a precondition for f.

5.2.1 Metamorphic rules

Now that we have considered the algorithm, let us recall the metamorphic rules that were obtained in my undergraduate dissertation. This work can be found in the following Github repository[1]. In particular, these MRs and all following MRs with their respective proofs can be found in Chapter 4. Let us consider P as the implementation under test:

MR1: Let $f : \{0,1\}^n \to \{0,1\}$ be a function defined as determined by the problem, and let $g : \{0,1\}^n \to \{0,1\}^n$ be an automorphism. Then, $P(f) = |\mathbf{0}\rangle$ if and only if $P(f \circ g) = |\mathbf{0}\rangle$.

MR2: Let $f : \{0,1\}^n \to \{0,1\}$ be a function defined as determined by the problem, and let $h : \{0,1\}^n \to \{0,1\}$ be the function such that $h(x) = 1 - f(x)$. Then, $P(f) = |\mathbf{0}\rangle$ if and only if $P(h) = |\mathbf{0}\rangle$.

The validity of these MRs can be understood directly from the definition of f, as this is a precondition of the problem.

5.2.2 Application of metamorphic rules

However, the question now is: How can we transform these MRs into code? In particular, how can we develop a testing structure using the `QMutation_TestingStructure` class? This will be our goal for this section. To achieve this, we will go step by step through each of the functions needed to instantiate our MRs. However, we will ignore the creation of mutants as QCRMut will be used for all of them.

Firstly, we need to create the original algorithm with our `PlaceHolder` in the input position, leaving us to the quantum circuit in Figure 5.3.

Secondly, we will need to create the inputs. This can be one of the tricky parts of any algorithm, as the input will depend on the requirements of the oracle generator. In this case, it is more complex than just a number within a certain scope. Our first consideration is how we are going to implement the oracle. To keep it as simple as possible, we considered the function proposed by Qiskit in their learning courses[2]. However, we adapted the function,

[1] `https://github.com/sinugarc/TFG`

[2] `https://learning.quantum.ibm.com/course/fundamentals-of-quantum-algorithms/quantum-query-algorithms`

as it was originally set up to obtain random numbers to decide whether it is balanced or constant, and which values would be balanced. Every random decision taken inside the function body is now considered an input. Therefore, our input for the oracle will be whether we flip the output qubit, whether the function is constant or balanced, and the set of states produced that will be mapped to 1 if it is balanced.

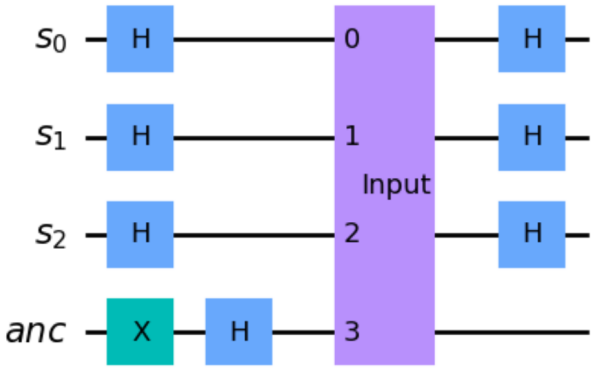


Figure 5.3: DJ Algorithm for 3 qubits

We can then create our base DJ input generator, which will return a list of tuples with three elements, representing the ones explained previously. We have decided that the constant inputs must always be included, as there are only two of them: if all map to 0 or if all map to 1, represented as `(0, 1, -)` or `(1, 1, -)`, respectively. These inputs will determine the f oracle. One example of a balanced input would be `(0, 0, (2, 3, 6, 7))` for an algorithm with three qubits.

For MR1, we need to consider the automorphism g as an input. We will implement this in our quantum programs by permuting the qubits before applying our oracle. To be precise, we would need to undo this permutation after the oracle. However, since we want to measure whether the output is $|\mathbf{0}\rangle$ or not, the order will not affect the result. To simplify the number of gates, the permutation will only be applied before the oracle. Therefore, for MR1, the input will be a pair of the first type and a list defining the permutation. To apply the permutation, we use a built-in Qiskit method. The input would have the form `((a, b, QubitTuple), Permutation)`, where $a, b \in \{0, 1\}$.

For MR2, we will provide only one input defining f, as the function h can be constructed directly from f. If we consider f to be defined by `(1, 0, QubitTuple)`, then h will be defined by `(0, 0, QubitTuple)`.

Once we have the mutants and the inputs, we need to create the test quantum circuits. Both MRs will return two quantum circuits: one with the circuit substituting the `PlaceHolder` by the f oracle, and the second one will differ. For MR1, we will include the permutation, representing g, before the oracle f. For MR2, we will only include in the input the h oracle.

Finally, we need to proceed with the analysis of both results. We need to adapt the idea that the first circuit under test is always $|\mathbf{0}\rangle$ if and only if the second circuit under test is always $|\mathbf{0}\rangle$. Let us consider all possibilities and build our analysis based on them:

- If the first circuit is balanced, it does not contain $|\mathbf{0}\rangle$ in the results. Therefore, the mutant will be killed if $|\mathbf{0}\rangle$ is present in the second circuit's results. This would mean that the first circuit is balanced, and the second one may be constant or neither.
- If the first circuit is constant, it has only one possible solution, which is $|\mathbf{0}\rangle$. Then, the mutant will be killed if there is more than one possible outcome, or if $|\mathbf{0}\rangle$ is not the only results of the second circuit.
- The only remaining possibility is that the first circuit has output more than one string, and $|\mathbf{0}\rangle$ is one of them. Therefore, this circuit is neither balanced nor constant, and the mutant will be killed.

We can then construct our DJ analysis function, directly following this separation of all possible cases in the first quantum circuit under test: balanced, constant, and neither.

Finally, we can construct our testing structures using the defined functions:

MR1: DJ Algorithm, QCRMut, DJ input generation for MR1, DJ MR1 circuit generator, Execution, and DJ Analysis.

MR2: DJ Algorithm, QCRMut, DJ input generation, DJ MR2 circuit generator, Execution, and DJ Analysis.

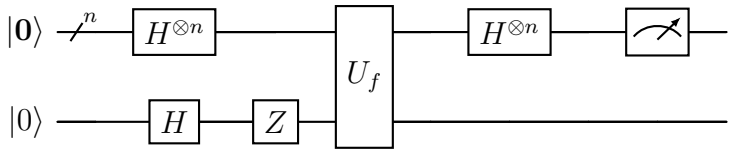
We can observe how the slight changes in the input generation and the creation of the circuit differentiate both DJ MR rules. These definitions can be found in the python file `DJ_Testing.py` within the corresponding algorithm folder.

5.3 Bernstein-Vazirani

The Bernstein-Vazirani, BV, algorithm was presented by Ethan Bernstein and Umesh Vazirani in their work Quantum Complexity Theory [81]. We can understand this algorithm as a generalisation of the Deutsch-Jozsa algorithm[3]. Let us introduce the problem that this algorithm solves.

Problem: Let $\mathbf{s} = s_0 s_1 \dots s_{n-1} \in 0,1^n$ with $n \in \mathbb{N}, : n > 1$, and let $f : 0,1^n \to 0,1$ be a function such that $f(\mathbf{x}) = \mathbf{s} \cdot \mathbf{x} \bmod 2 = s_0 x_0 \oplus s_1 x_1 \oplus \dots \oplus s_{n-1} x_{n-1}$. We would like to obtain the string $\mathbf{s}$. The oracle interpretation is identical to the one proposed for the DJ oracle.

The BV algorithm to solve this problem is similar to the one proposed in the DJ algorithm; however, it is usually represented slightly differently:



We can see that the difference lies in the initialisation of the ancilla qubit. This is due to the following equivalence between gates: $X = HZH$. Therefore, $XH = HZHH = HZ$. The only theoretically possible outcome is the string $\mathbf{s}$ encapsulated in the f oracle.

5.3.1 Metamorphic rules

Let us now introduce the three MRs for the BV algorithm. Assume that P is the implementation of the BV algorithm:

MR1: If $\mathbf{s} \in \{0,1\}^n$ with $n \in \mathbb{N}, : n > 1$, and let $\bar{\mathbf{s}}$ be its binary complement, then $P(\mathbf{s}) \oplus P(\bar{\mathbf{s}}) = |\mathbf{1}\rangle$.

MR2: If $\mathbf{s}, \mathbf{s}' \in \{0,1\}^n$ with $n \in \mathbb{N}, : n > 1$, then $P(\mathbf{s}) \oplus P(\mathbf{s}') = \mathbf{s} \oplus \mathbf{s}'$.

MR3: If $\mathbf{s}, \mathbf{s}' \in \{0,1\}^n$ with $n \in \mathbb{N}, : n > 1$, and we understand the composition as the successive application of the oracles, then $P(\mathbf{s} \circ \mathbf{s}') = P(\mathbf{s}) \oplus P(\mathbf{s}') [= P(\mathbf{s} \oplus \mathbf{s}')]$.

[3] `https://learn.qiskit.org/course/ch-algorithms/bernstein-vazirani-algorithm`

We can observe that MR1 and MR2 are similar. In particular, MR1 is a specific case of MR2. Why do we consider them as two different rules? If we refer to the definition of a metamorphic relation, MR, provided by the text [10], it specifies an analogous example to demonstrate the importance of this distinction. The example is as follows: Imagine there is an MR such that $f(k \times x) = k \times f(x)$. We could consider the MR where $k = -1$. This MR is weaker than the previous one, but it is much easier to verify.

In contrast to the MRs proposed for the Deutsch-Jozsa algorithm, which are derived directly from its definition, the proofs of these MRs are based on the mathematical development of the BV algorithm, especially MR3. These proofs can be found in Chapter 4 of my undergraduate dissertation, available in the following GitHub repository[4].

5.3.2 Application of metamorphic rules

Following the structure of this work, we should now see how we are going to adapt these rules to our framework. For the sake of simplicity, we will avoid any repetitions when the functions are analogous to the previous ones. For example, the BV original circuit would be analogous to the one mentioned earlier.

Let us continue with the inputs, as we need strings for this case. We utilise the `input_generator` function adjusting this list to generate random pairs or the list provided with its binary complement.

The corresponding circuit creation for each MR follows the development of the rules, with our main aim being to stay in the quantum realm as much as possible before measuring. Examples of each quantum circuit under test, for a three-qubit algorithm, are shown in Figures 5.4 and Figure 5.5.

Now, let us proceed with the analysis. MR1 is defined as obtaining the ket $|\mathbf{1}\rangle$. However, to simplify the application of all rules at once, we will focus on obtaining the ket $|\mathbf{0}\rangle$. In the circuit creation, we can observe how we add a gate that represents the relation between inputs. This gate yields the expected result. Thus, by combining the inputs with themselves, we obtain the desired result due to $\mathbf{z} \oplus \mathbf{z} = \mathbf{0}$. Therefore, the mutant will be killed if $|\mathbf{0}\rangle$ is not the only key value in the results.

[4] `https://github.com/sinugarc/TFG`

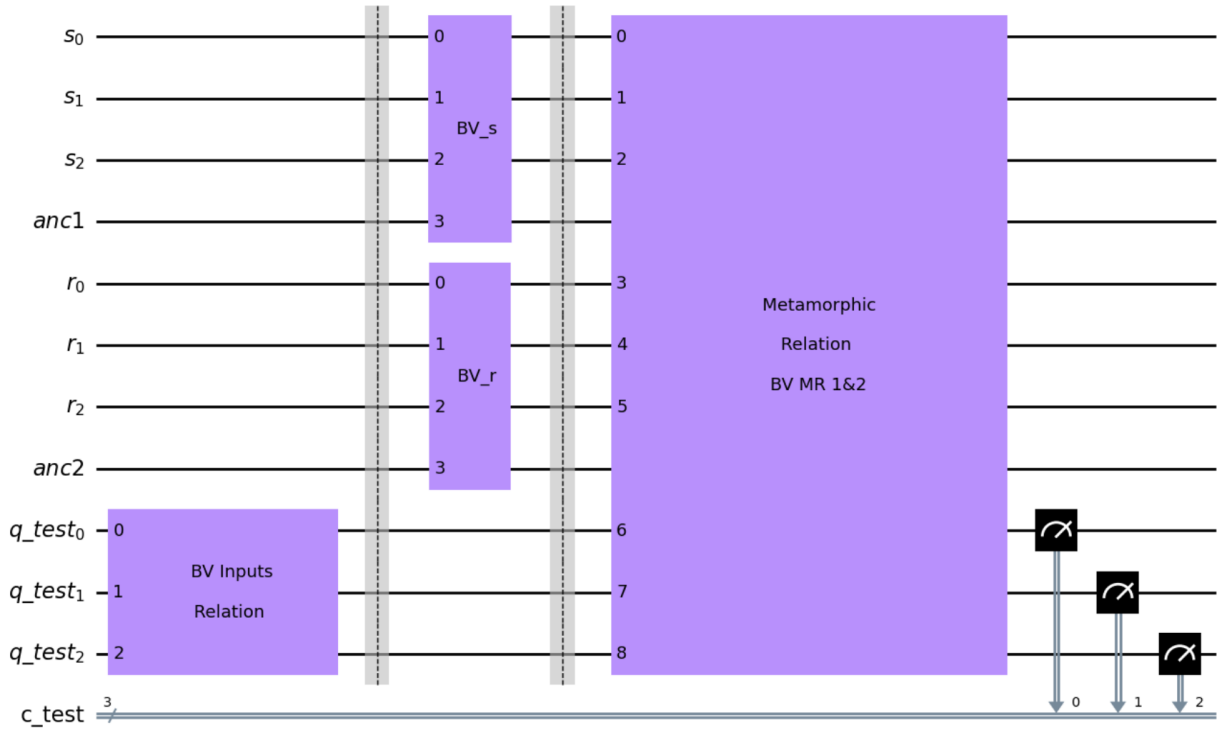


Figure 5.4: BV test QC for MR1 and MR2

Finally, we can reduce our MRs to the testing structure:

MR1: BV Algorithm, QCRMut, BV pair input generation, BV MR1-2 circuit generator, Execution, and BV Analysis.

MR2: BV Algorithm, QCRMut, BV xor input generation, BV MR1-2 circuit generator, Execution, and BV Analysis.

MR3: BV Algorithm, QCRMut, BV pairs input generation, BV MR3 circuit generator, Execution, and BV Analysis.

We can observe that the difference between MR1 and MR2 lies in the relationship between `s` and `r`. In the MR3 circuit shown in Figure 5.5, there is no input relationship, as this relationship is contained within the `q_test` oracle.

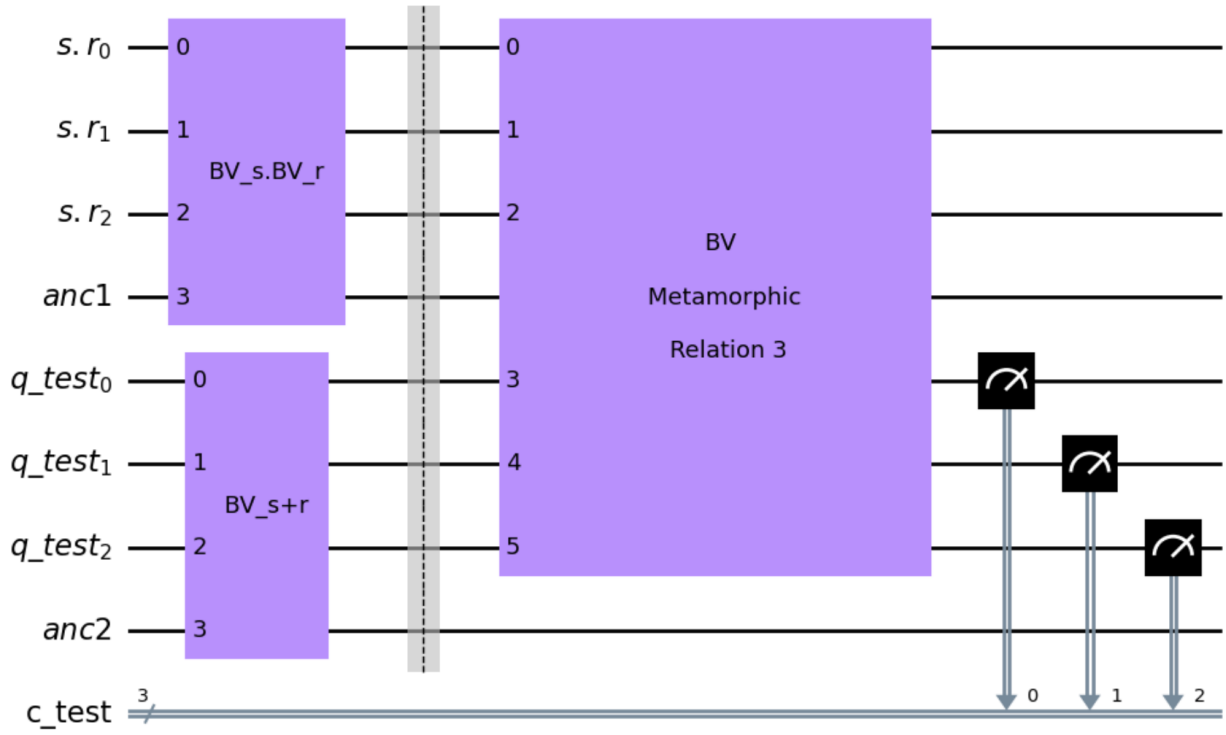


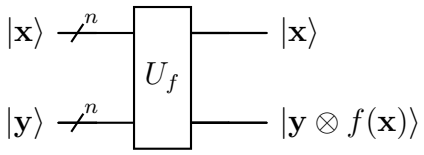
Figure 5.5: BV test QC for MR3

5.4 Simon

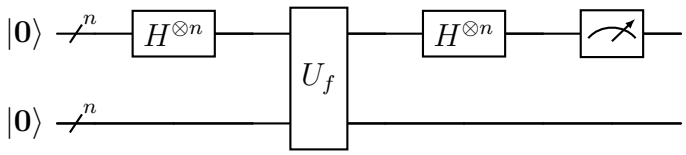
Let us introduce the last algorithm of this study, Simon's algorithm [1], the first algorithm that does not only use quantum computing; however, we will focus only on this part. The algorithm was presented by Daniel Simon in his 1997 work titled *On the Power of Quantum Computation* [84].

Problem: Let $f : 0,1^n \rightarrow 0,1^n$, with $n \in \mathbb{N}$, be a periodic function where the period is $\mathbf{c} = c_0 c_1 \ldots c_{n-1}$. We would like to determine the period of f, understanding that this function is provided as an oracle.

The oracle representing f behaves as follows:



Simon's quantum algorithm routine will provide us with the information needed to solve it. The algorithm proceeds as follows:



We note that this is only the first step of the solution. What are the outcomes provided by this algorithm? It provides the strings $\mathbf{z}$ such that $\langle \mathbf{z}, \mathbf{c} \rangle = 0$. To obtain the period, we need to solve the system of equations defined by all possible values of $\mathbf{z}$.

5.4.1 Metamorphic rules

We should now introduce the MRs we obtained from this algorithm, assuming P is the implementation of the algorithm:

MR1: If we consider $S_{\mathbf{c}}$ as the set of all $\mathbf{z}$ for a function f with period $\mathbf{c}$, then $(S_{\mathbf{c}}, \oplus)$ is a group.

MR2: If we consider $S_{\mathbf{c}}$ as the set of all $\mathbf{z}$ for a function f with period $\mathbf{c}$, $\bar{\mathbf{c}}$ as the inverted $\mathbf{c}$, and $\bar{S_{\mathbf{c}}}$ as the set of all inverted strings in $S_{\mathbf{c}}$, then $\bar{S_{\mathbf{c}}} = S_{\bar{\mathbf{c}}}$.

We are presenting both MRs as they were presented in my undergraduate thesis. However, we question the validity of MR1 as an MR. By definition, an MR is a relation between more than one input and their corresponding outputs, but MR1 involves only one input. We will examine whether this affects our results when verifying the validity of our MRs.

5.4.2 Application of metamorphic rules

Let us now observe how we apply these MRs to our quantum strategy. Firstly, the oracle for Simon's algorithm has been obtained from Qiskit's textbook [85]. We used to be able to import such an oracle from their libraries. However, due to ongoing changes aimed at stabilising the Qiskit version, some libraries have been affected.

This is one of the most straightforward adaptations, as it closely resembles the original circuit. The test circuit for the three-qubit algorithm is illustrated for MR1 in Figure 5.6 and for MR2 in Figure 5.7. These QC results will be compared to the single mutated circuit.

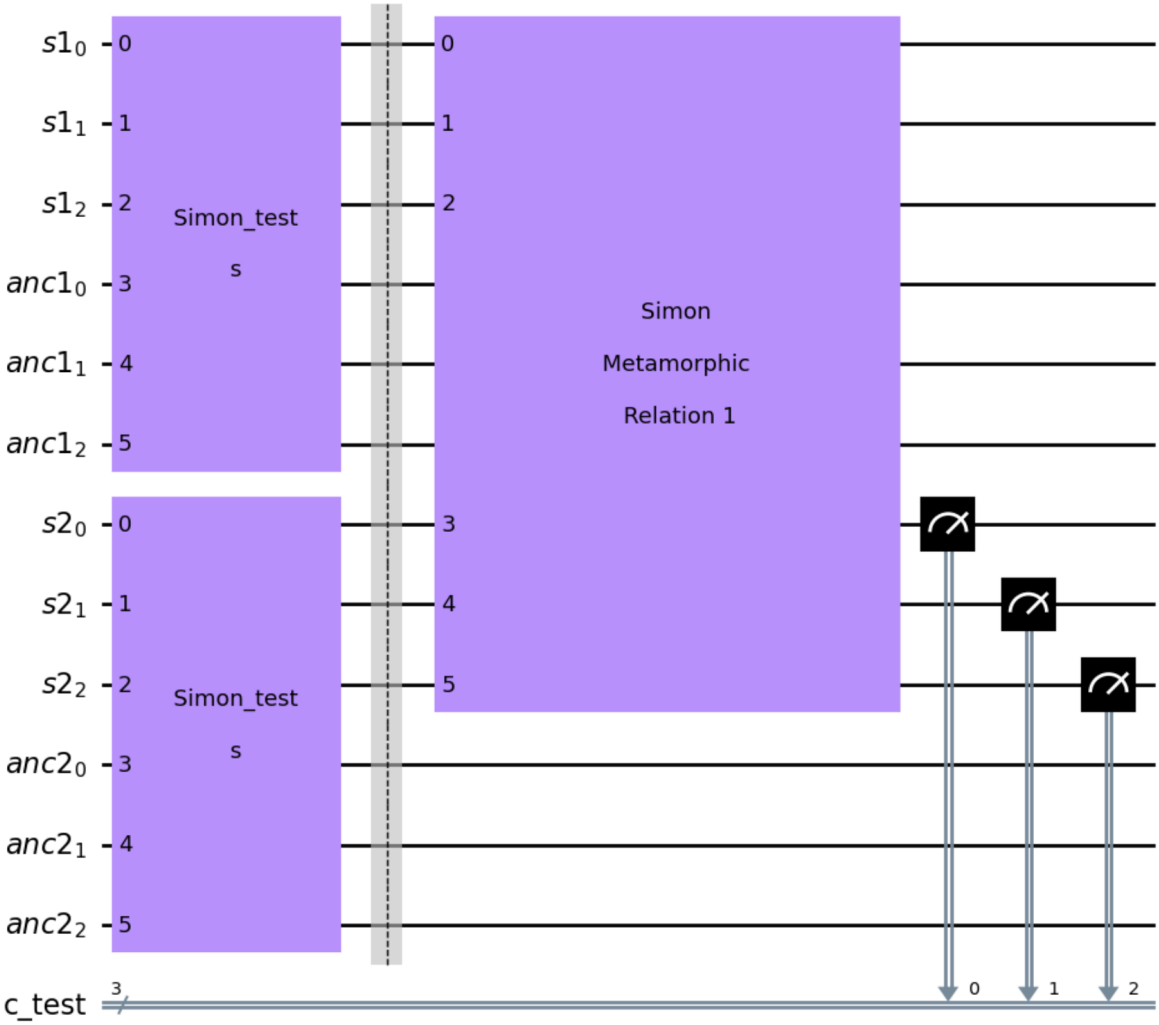
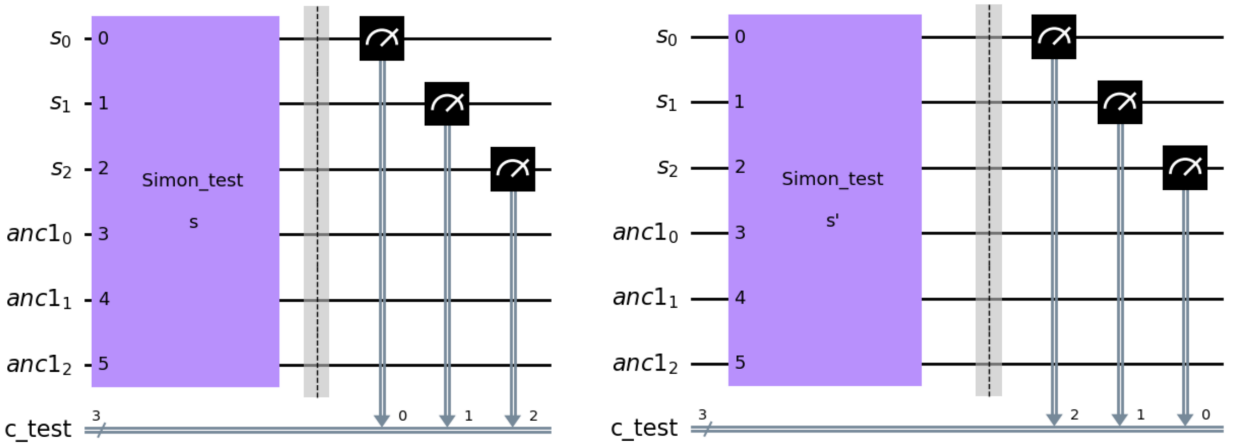


Figure 5.6: Simon test QC for MR1.

We could conclude with the analysis of the rules. Since we are working with a set of outcomes, we have a direct method to identify this set, thanks to Qiskit's results structure. The results are obtained as a dictionary where the keys are the strings, and the values are the number of repetitions. Therefore, our comparison as a set will be derived from the set of keys in the results dictionaries. To simplify the analysis function, we can observe in Figure 5.7b how the measurement is inverted as needed for Simon MR2.



(a) Simon's mutated circuit for s

(b) Simon's mutated circuit for s', inverse circuit.

Figure 5.7: Simon test QC for MR2.

Let us finish this section with the last MR definitions as a quantum testing structure:

MR1: Simon Algorithm, QCRMut, input generation, Simon MR1 circuit generator, Execution, and Simon Analysis.

MR2: Simon Algorithm, QCRMut, input generation, Simon MR1 circuit generator, Execution, and Simon Analysis.

We can observe that the only difference lies in the creation of the circuit, which was evident from the figures presented previously.

5.5 Metamorphic testing experiments and results

We are finally at the heart of our work, the point we have been striving toward, to demonstrate that metamorphic testing can be applied to quantum programs and that the MRs defined in the previous section are effective at finding errors in our mutants. Let us define the main experiment to obtain the mutation score for our MRs.

We will use the structure created for this purpose, incorporating some basic arguments to keep it as simple as possible. To reproduce these results, we have set the same seeds as in the QCRMut experiments. Specifically, we will use `random.seed(1)` for mutant creation and `random.seed(42)` for Qiskit's `AerSimulator` execution. The code for all experiments can

be found in the `Replication` folder, with replication instructions available in the `README.md` file, and all results are located in the `Results/MetTesting` folder.

Our testing structures will be set up for five tests with 3, 4, 5, 6, and 7 qubits, respectively. Instead of proposing a specific number of mutants, we will use the study done in Section 4.2.5 on coverability. Recall the number of mutants we will create for each circuit, depending on the algorithm and the number of qubits, n, as shown in Table 5.1.

Algorithm	# Gates	# Qubits	# Mutants
DJ and BV	$2 * n + 3$	3	160
		4	192
		5	224
		6	256
		7	288
Simon	$2 * n + 1$	3	128
		4	160
		5	192
		6	224
		7	256

Table 5.1: Number of mutants for each test.

One of the problems we may face with this simulation is the growth of qubits for our MRs. This problem was already outlined in previous works [9, 74]. Therefore, let us study how many qubits we are going to use for each MR, depending on the number of qubits in the initial algorithm, denoted as n. These results can be found in Table 5.2.

Algorithm	MR	Max # Qubits
DJ	1	$n + 1$
	2	$n + 1$
BV	1	$3n + 2$
	2	$3n + 2$
	3	$2n + 2$
Simon	1	$2n$
	2	$2n$

Table 5.2: Number of qubits needed at most for each MR.

The reason for this problem was already stated in Section 2.3, when we introduced simulation as one of the possible ways of executing quantum programs. These simulations calculate the theoretical result by operating with the matrices that represent each gate. The size of these matrices grows exponentially with the number of qubits; as we mentioned, the size is $2^n \times 2^n$, where n is the number of qubits.

Therefore, if we want to test the BV algorithm with 10 qubits, the number of qubits needed to execute such a test for MR1 and MR2 would be 32 qubits. The size of the representative matrix would be $2^{32} \times 2^{32}$, which may be optimised. However, this is not the only metric affecting the simulation; the number and type of gates, as well as the depth of the quantum circuit, also affect the simulators used.

Finally, we have all the necessary tools to proceed with the test. The results are stored in their respective folders, named according to the rule in the `Metamorphic_Test_I` folder. A serialised object representing the results is available, which will facilitate further processing if needed. The results are presented in Table 5.3.

Let us analyse the results:

- The quality of most of our MRs is excellent, being able to kill over 75% of mutants.
- Less than 10 inputs are needed to kill all killable mutants created. However, there are times when five or fewer inputs are sufficient.
- Simon's MR1: The results support our impression that this rule should not have been considered as such.

Let us expand on the problem with Simon's MR1. We already mentioned that if we use the definition of MR, this rule should not be considered valid. The reason for this is that the input is related to itself, rather than to a different one. As we proceed with a mutation, the mutated gate or position can be included, theoretically, in the input, recovering the correct quantum circuit but with a different oracle. The problem is that, since we are only dealing with one input, the oracle containing the mutation is the same for all testing circuits of that input. Therefore, any rule or logical derivation between them appears correct, as we are working with the correct circuit but with an unknown oracle.

Another question we could ask ourselves is whether the number of qubits affects the mutation score. We have produced figures where we can observe the evolution of the mutation score for each MR, except Simon's MR1, depending on the number of qubits in the algorithm. This will be our second metamorphic test experiment.

Algorithm	Rule	# Qubits	# Inputs used[1]	% Mutation Score
DJ	MR1	3	6	74.38
		4	6	75.52
		5	7	76.79
		6	4	81.25
		7	4	82.64
	MR2	3	6	74.38
		4	4	72.92
		5	5	75.89
		6	5	73.05
		7	7	79.86
BV	MR1	3	1	100
		4	1	100
		5	1	100
		6	1	100
		7	1	100
	MR2	3	2	83.12
		4	3	84.38
		5	2	84.38
		6	1	83.98
		7	2	88.19
	MR3	3	3	78.75
		4	3	78.12
		5	1	79.91
		6	2	78.91
		7	2	83.68
Simon	MR1	3	2	2.34
		4	1	1.25
		5	5	4.17
		6	5	2.68
		7	4	6.25
	MR2	3	4	74.22
		4	4	78.75
		5	4	79.17
		6	4	77.23
		7	10	80.47

[1] Number of inputs needed to kill mutants

Table 5.3: Mutation score for each MR.

Algorithm	Rule	# Qubits	64 Inputs		10 Inputs	
			% Mut Score	Exec time (s)	% Mut Score	Exec time (s)
DJ	MR1	8	81.25	499.64	81.25	219.37
		9	82.73	2098.17	82.73	690.51
	MR2	8	75.62	1371.48	73.75	235.03
		9	75.45	4152.69	71.59	833.39
BV	MR1	8	100	522.52	100	566.73
		9	100	7703.03	100	5479.63
	MR2	8	89.06	4642.51	89.06	1156.3
		9	87.73	60397.35	87.73	14524.99
	MR3	8	85.31	531.86	85.31	114.32
		9	84.32	906.51	84.32	195.87
Simon	MR2	8	100	13.24	100	13.21
		9	100	15.9	100	16.16

Table 5.4: Mutation score and execution time (Experiments II and III).

Once we removed Simon's MR1, we proceeded to execute a new test, this time with 8 and 9 qubits. The execution times in this case will be higher, as the complexity continues to increase exponentially. However, we have observed that, at most, we need 10 random inputs to kill as many mutants as possible. Let us set up a third experiment: we will continue with 8 and 9 qubits but limit the inputs to 10. Our aim with this approach is to see if we can achieve the same or a similar mutation score while reducing the execution time. The results of this experiment can be found in Table 5.4. Let us compare the last two experiments and examine how the execution time changes depending on the number of inputs. These results are due to the reduction of 54 executions per non-killed mutant, as we are limiting our inputs to 10.

- Most of the MRs maintain the same mutation score even with the reduction in the number of inputs. DJ MR2 is the only one that shows a decrease in the mutation score. However, this reduction does not significantly impact the results, as it only decreases by 2% and 4%, respectively.
- When the mutation score is 100%, there is no improvement in execution time. This is because the initial inputs are sufficient to kill all mutants without requiring additional executions. If we check the files, since the inputs are randomised, we can see that this is achieved even with different inputs.

- We observe a significant improvement in performance when using 10 inputs, provided the rules do not achieve a 100% mutation score. We can reduce the execution time to a quarter of the original time, as demonstrated in BV MR2, where we managed to obtain the same mutation score in 4 hours instead of the approximately 16 hours needed in Experiment II.

5.6 Mutants and seed experiments

Once we have analysed the effectiveness of our MRs, we should ask ourselves more questions. Let us recall **RQ5:** from Chapter 4: Does the generation seed used in the random approach affect the results? Let us also delve deeper into the analysis of our results and our proposals for *QCRMut*. Does the number of mutants selected for our coverage represent the correct mutation score?

To answer these questions, we propose a final set of experiments in this work. We will set up some of our MRs for an extensive test using different numbers of mutants and different seeds. We have selected DJ MR1 and MR2, BV MR2 and BV MR3, and Simon's MR2. The reason for selecting these MRs is their mutation score in the lower qubits; they are neither 100% nor near 0%. We would like to see the variation in the mutation score, which would be harder to observe with such extreme values. Let us call this set the *Seed Experiments*. All results for this final set of experiments are located in their folders, such as `Seed_Test_I`.

The first and main experiment will be as follows: we will test each MR for all possible combinations of $[10, 20, 30, 50, 70, 100, 150, 200, 300, 500]$ numbers of mutants generated and the seed variation from 0.5 to 10, evenly distributed across 20 different seeds. Since this means we are producing a high number of individual tests, we have decided to conduct these experiments only over 3, 4, and 5 qubits. However, even though all results are saved, we believe that a graphical representation would make it easier to perceive the results without having to delve into the details of the mutation scores. These figures can be found in the folder `Results/Figures` of the corresponding test. Figure 5.8 represents a sample of the tests conducted in Experiment I, where we have collated the figures for BV MR2.

We can observe that different seeds do not affect the results significantly, as they all stay within the same range of values when applied to a higher number of mutants. However, with a smaller number of mutants, the mutation score highly depends on the chosen seed, with differences sometimes exceeding 50%. This is due to the randomness implemented

in the mutant generation; the seed directly affects the mutants created. If the exhaustive mutation score is 70% with over 400 mutants, the chances of obtaining a sample of only 10 or 20 mutants can produce significantly different sets.

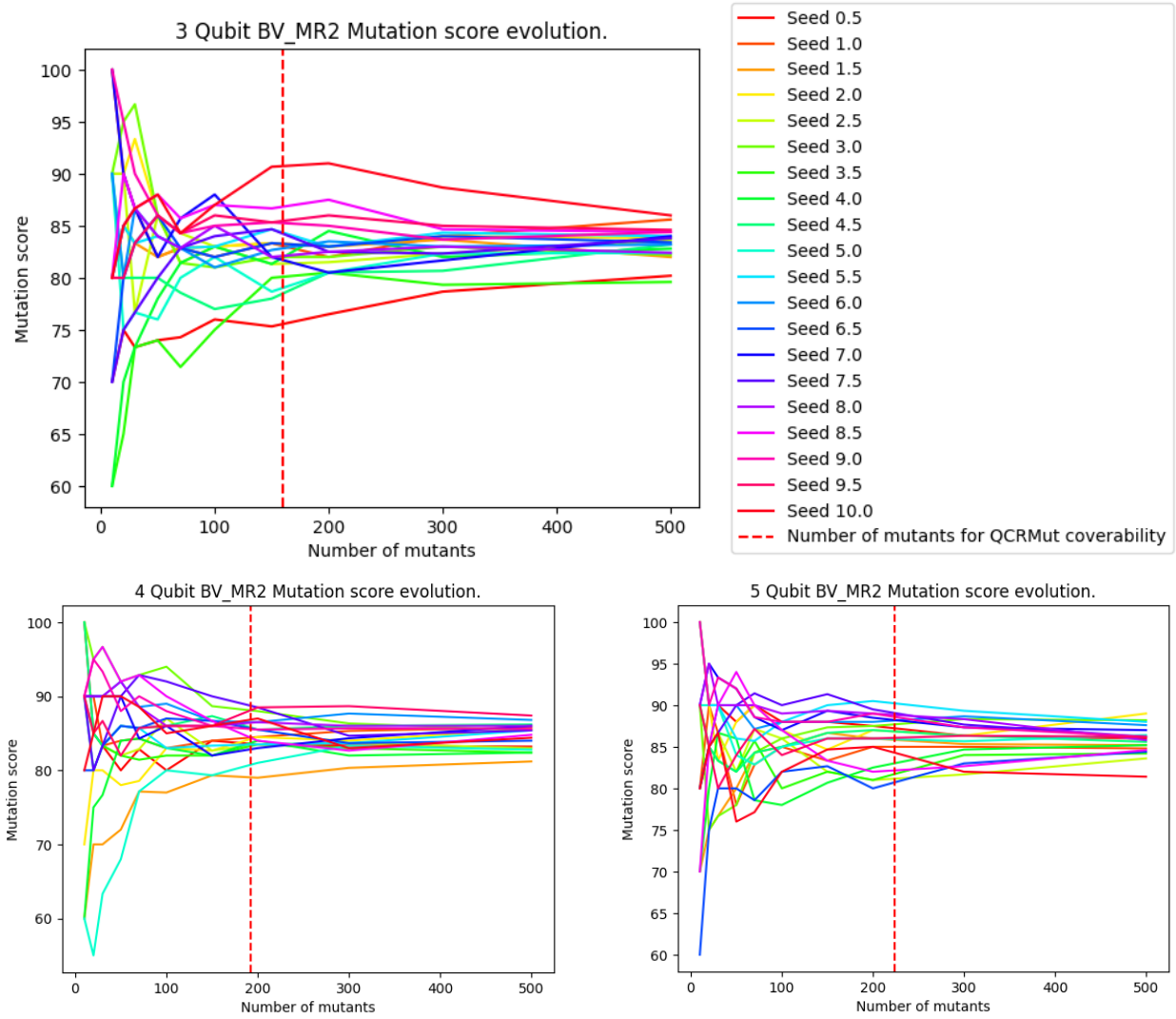


Figure 5.8: Seed experiment I results, BV MR2.

Another question arises: Does the chosen range and the evenly divided set of seeds affect the results? We propose two quick experiments using only BV MR2 as an example. In the first experiment, we will use a set of integer seeds in a greater range, from 50 to 1000, which we will call Experiment II. In the third experiment, we will use a set of seeds chosen uniformly in $[0, 1]$ using the `random` library. We have selected a specific seed to produce this set to ensure all our results are reproducible. The results of these experiments can be seen

in Figures 5.9 and 5.10, where it is clear that there is no significant difference in the chosen set of seeds, as they all behave similarly.

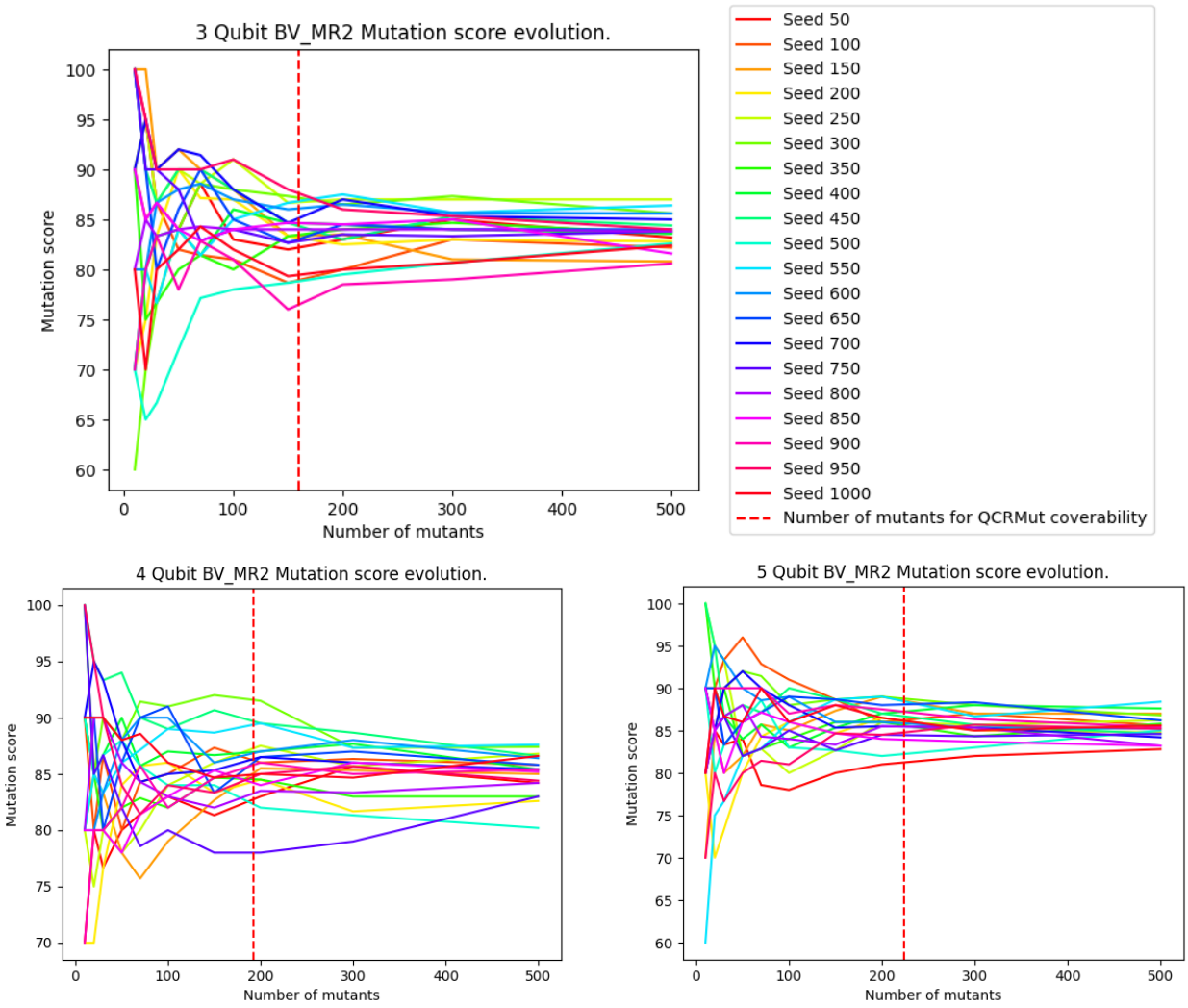


Figure 5.9: Seed experiment II results, BV MR2.

Let us conclude this chapter with one final reflection on our coverage definition and estimation. We have been able to observe from the figures how many mutants we would produce under our coverage definition. At times, as shown in the figures, it may appear that the choice of seed under our number of mutants can affect the results. However, we must recall that the number of mutants generated reflects the mean of the expected distribution for achieving full coverage. Therefore, as the number of qubits, correspondingly the gates, increases, the hope stabilises, which is why we observe a bigger deviation at our chosen point only in the lowest number of qubits.

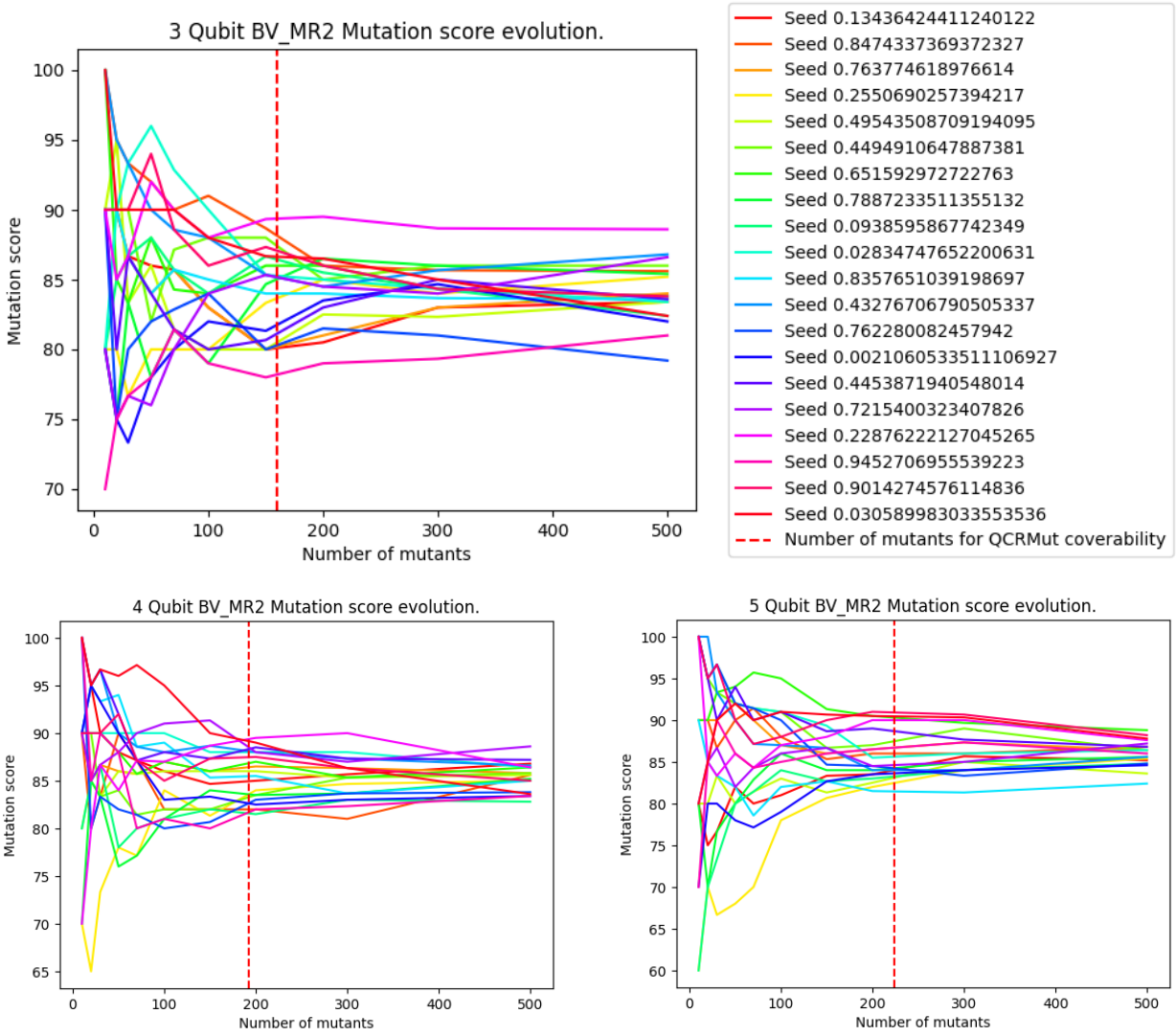


Figure 5.10: Seed experiment III results, BV MR2.

Chapter 6

Conclusion and future work

To culminate this work, we will provide a summary that encapsulates the efforts undertaken. We have explored the state of the art in quantum computing, developed a novel mutation tool, and created an automated testing tool. Furthermore, we conducted experiments to empirically validate the quality of our MRs. However, before we conclude, it is essential to discuss some potential threats to the validity of our research.

6.1 Threads to validity

Rather than dividing this section into two separate subsections, one focusing on the challenges or issues associated with QCRMut and the other on metamorphic testing, we have chosen to consolidate everything into a single section. By doing so, we aim to preserve the order in which these topics were presented throughout this work.

One main concern that could be raised regarding the validity of this tool, QCRMut, is the limited number of quantum algorithms studied in our experiments. If we had included more algorithms, we might have been able to better understand any potential differences in outcomes, particularly those stemming from randomness. Muskit [58] used only four algorithms, but we had to exclude one, QRAM, due to inconsistencies in the oracle and our inability to reproduce it.

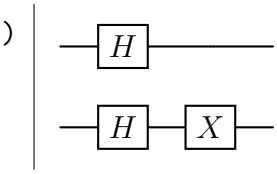
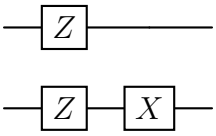
The second issue we encountered in this paper was the use of the `chiSquare` test and its influence on the number of executions needed. It may be beneficial to explore more sophisticated statistical methods that impose fewer preconditions.

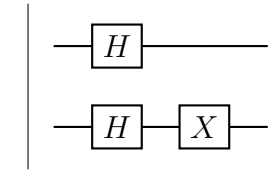
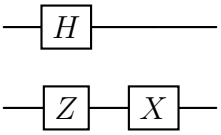
Additionally, our experiment comparing the space used by our mutants to the number of lines of code, LOC, may not have been as precise as desired. This is because including loops in our code can increase the number of gates without significantly affecting the LOC. However, our comparison was based on a linear algorithm script. A more comprehensive analysis

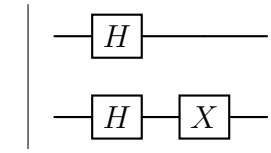
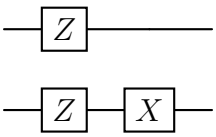
could include assessing the RAM memory used to serialise objects with such libraries, which we propose as future work.

To finalise with QCRMut, we have to be careful and understand the insights of Qiskit and Python, specifically how they create items and use classes. There are cases where the mutation may appear duplicated or even modify more than one gate at once. Ideally, this should not occur with our tool, as we are modifying only a single position in the list of all `QuantumInstruction`. However, a well-known behaviour in Python is the use of references to memory locations when using the assignment operator (=). Since Qiskit is implemented in Python, this behaviour has been derived into Qiskit. Therefore, when creating a new quantum circuit, the code will define the relation, in memory terms, between the different gates. We provide some examples in Table 6.1.

Original code	Original circuit	Mutated circuit

`qs = QuantumRegister(2,"s")`
`qc = QuantumCircuit(qs)`
`qc.h(qs)`
`qc.x(1)`
		

`qc = QuantumCircuit(2)`
`qc.h(0)`
`qc.h(1)`
`qc.x(1)`
		

`qc = QuantumCircuit(2)`
`qc.h(range(2))`
`qc.x(1)`
		

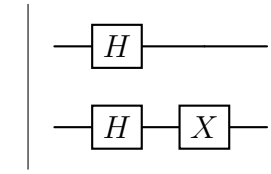
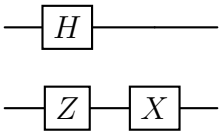
`qc = QuantumCircuit(2)`
`for i in range(2):`
`  qc.h(i)`
`qc.x(1)`
		

Table 6.1: Gate operator behaviour; Second Hadamard change for 'Z'.

We observe that if the set of gates is created simultaneously, either due to the range of qubits or through the `QuantumRegister` class, the name of the gate is stored in the same memory location. Consequently, when any of the gates is modified, the change affects all of them. This behaviour can be observed in the first and third circuits. The next question is

whether this affects our mutants. We propose that the answer is negative; we do not believe this behaviour impacts the mutants. Such a modification would represent a single fault in the code, which then propagates to different locations within the code. However, to avoid any issues related to this matter, we have ensured that all our code is implemented in a way that avoids reference issues between the gates.

We could raise similar concerns regarding our metamorphic testing experiments, particularly with the use of a small number of qubits. However, as explained throughout this work, this choice primarily affects execution times, especially when using simulations. The increase in execution times can be observed in Table 5.4 in Section 5.5, where we noted the differences between using 64 inputs and 10 inputs. The use of only three algorithms was due to time constraints in obtaining more MRs for different algorithms.

To conclude this section on threats to validity, we would like to revisit the study conducted with the seed experiments, where we aimed to determine whether the seed that initiates randomness in mutant generation could impact our results, particularly the mutation score and, consequently, the validation of the quality of our MRs. Although this is not the only seed we used, it may be considered the most significant one. We used `seed 42` for the execution of our quantum programs. It would be valuable to provide more insights into how these seeds might affect our results in future studies.

6.2 Conclusion

Firstly, in Chapter 3, we introduced an extensive state-of-the-art review of the preceding years, where we delved into the most interesting ideas and solutions proposed by different researchers around the globe. Aligning with our research, we observed how metamorphic testing has been effective, as demonstrated in the paper presented by Paltenghi et al. [7], which revealed several bugs in the Qiskit quantum platform. A brief introduction to metamorphic testing in quantum programs was first provided by Abreu et al. [9].

Simultaneously, there has been significant research related to mutation testing in the quantum realm. Automation of mutant generation was achieved through tools like Muskit by Mendiluze et al. [58] and QMutPy by Fortunato et al. [8, 69]. However, we began Chapter 4 with an introduction to the reasons why these two tools were not suitable for our research, providing insights into these issues and identifying several faults in Muskit, as well as unaligned concepts, such as equivalent gates in QMutPy, with our research objectives.

We presented QCRMut as an alternative approach to creating mutants for quantum circuits, offering the flexibility to maintain a desired structure for algorithm robustness. Our method provides an efficient means of generating random mutants that are correct by construction, yielding results comparable to those produced by exhaustive mutant generators. Departing from the classical notion of mutating lines of code, LOC, our approach operates within the quantum paradigm, focusing on the number of gates in the algorithm. This allows users the freedom to execute, measure, and analyse quantum circuits with ease.

It is important to consider the significance and application of the statistical tests used in our research, as well as in much of the literature. Often, there is a lack of validation for the models created, and the ability of these tests to confidently provide answers within our models is crucial.

Finally, in Chapter 5, we revisited the metamorphic rules for three quantum algorithms: Deutsch-Jozsa, Bernstein-Vazirani, and the quantum routine of Simon's algorithm. We provided an abstract structure for fully automating the testing process, with the aim of providing all necessary tools, combined with QCRMut, to verify the quality of our MRs.

We demonstrated the adaptation process of a mathematical object, such as an MR, to code, and how it fits as an instance of our abstract class to define itself as a rule ready for testing. Our experiments have confirmed the quality of our MRs, where all MRs produced over a 70% mutation score, with an impressive result showing that with such MRs, we would only need to provide at most 10 inputs to achieve a similar mutation score. This led to improvements in our execution times during simulations.

We addressed and resolved some of our concerns with experiments that demonstrated the robustness of our approach, such as the finding that the seed of our random mutation approach does not affect the results, or how executing an excessive number of mutants does not significantly alter the results of our estimation of mutants needed.

In conclusion, QCRMut promises to support future applications by bringing testing closer to the quantum paradigm, facilitating a more straightforward interpretation of measurements. We also presented Metamorphic Testing as a valid and interesting formal approach to testing in the quantum realm, reducing the number of inputs needed to detect any faults in our quantum algorithms and thereby decreasing the execution time of any desired tests. Throughout this study, we have successfully achieved all our objectives and provided insights into potential future work, which will be discussed in the next section.

6.3 Future work

All through this document, we have mentioned several possibilities for future work. Here, we will highlight some of the more relevant ones discussed in the last two chapters, Chapter 4 and Chapter 5, which are the primary focus of our current research. Our future work could proceed along several different lines:

QRCMut: Adapting QCRMut to a stable version of Qiskit is essential for the continuation of this research. This adaptation will provide a more solid foundation for utilising current and future versions of Qiskit.

QASM: As mentioned, the structures of Qiskit and QASM are quite similar. A potential future direction could involve adapting QCRMut to QASM, allowing it to be developed and used in any quantum SDK, thus not limiting its applicability to Qiskit alone.

Exhaustive mutant generation: To better evaluate the effectiveness and validity of our random approach in QCRMut, it would be valuable to adapt the tool to support exhaustive mutant generation. This adaptation would allow us to generate all possible mutants, enabling a more thorough comparison of mutation scores within the same set of generation parameters.

Simulator Seed Influence: Further tests are needed to assess whether the simulator seed can influence our results in mutation testing. Understanding this impact will help ensure the reliability and validity of our testing approach.

Increase of the number of quantum programs: Expanding the range of quantum programs tested would benefit both chapters of this work. A broader range of tested quantum programs could provide a deeper understanding of QCRMut and its applications. Additionally, applying metamorphic testing to larger and more complex algorithms would increase our confidence in supporting metamorphic testing as a promising approach for testing in the quantum realm.

Simulation vs real quantum systems: Another important future research direction involves transitioning from simulation to real quantum systems. We would have liked to include more insights and possibly some experimental tests using real quantum systems. However, this was not feasible due to the incompatibility between the version of Qiskit used in our research and the current version employed by IBM.

Bibliography

[1] Noson S Yanofsky and Mirco A Mannucci. *Quantum computing for computer scientists*. Cambridge University Press, 2008.

[2] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of statistical physics*, 22:563–591, 1980.

[3] Bennett, C. H. and G. Brassard, "Quantum Cryptography: Public-Key Distribution and Coin Tossing", Proceedings of the International Conference on Computers, Systems and Signal Processing, Bangalore, India (1984).

[4] Richard P Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467–488, 1982.

[5] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[6] Michael A Nielsen and Isaac Chuang. *Quantum computation and quantum information*. American Association of Physics Teachers, 2002.

[7] Matteo Paltenghi and Michael Pradel. Morphq: Metamorphic testing of the qiskit quantum computing platform. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2413–2424. IEEE, 2023.

[8] Daniel Fortunato, José Campos, and Rui Abreu. Qmutpy: A mutation testing tool for quantum algorithms and applications in qiskit. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 797–800, 2022.

[9] Rui Abreu, João Paulo Fernandes, Luis Llana, and Guilherme Tavares. Metamorphic testing of oracle quantum programs. In *Proceedings of the 3rd International Workshop on Quantum Software Engineering*, pages 16–23, 2022.

[10] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.

[11] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2014. How effectively does metamorphic testing alleviate the oracle problem? IEEE Transactions on Software Engineering 40, 1, 4–22.

[12] Pak-Lok Poon, Fei-Ching Kuo, Huai Liu, and Tsong Yueh Chen. 2014. How can non-technical end users effectively test their spreadsheets? Information Technology and People 27, 4, 440–462.

[13] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14). ACM, New York, NY, 216–226.

[14] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). ACM, New York, NY, 65–76.

[15] John Regehr. 2014. Finding compiler bugs by removing dead code.

[16] Manuel Méndez, Miguel Benito-Parejo, Alfredo Ibias, and Manuel Núñez. Metamorphic testing of chess engines. *Information and Software Technology*, 162:107263, 2023.

[17] Jesús M Almendros-Jiménez, Antonio Becerra-Terón, Mercedes G Merayo, and Manuel Núñez. Metamorphic testing of openstreetmap. *Information and Software Technology*, 138:106631, 2021.

[18] Jesús M Almendros-Jiménez, Antonio Becerra-Terón, Mercedes G Merayo, and Manuel Núnez. Using metamorphic testing to improve the quality of tags in openstreetmap. *IEEE Transactions on Software Engineering*, 49(2):549–563, 2022.

[19] Manuel Méndez, Antonio Becerra-Terón, Jesús M Almendros-Jiménez, Mercedes G Merayo, and Manuel Núñez. Combining metamorphic testing and machine learning to enhance openstreetmap. *IEEE Transactions on Reliability*, 2024.

[20] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in computers*, volume 112, pages 275–378. Elsevier, 2019.

[21] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[22] Timothy A Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta informatica*, 18(1):31–45, 1982.

[23] A Jefferson Offutt and W Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[24] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. On the limits of mutation reduction strategies. In *Proceedings of the 38th international conference on software engineering*, pages 511–522, 2016.

[25] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, 66(3):854–874, 2017.

[26] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. Qdiff: Differential testing of quantum software stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 692–704. IEEE, 2021.

[27] Ryan LaRose. Overview and comparison of gate level quantum software platforms. *Quantum*, 3:130, 2019.

[28] Siu-On Chan, Ilias Diakonikolas, Paul Valiant, and Gregory Valiant. Optimal algorithms for testing closeness of discrete distributions. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1193–1203. SIAM, 2014.

[29] Scott Aaronson and Lijie Chen. Complexity-theoretic foundations of quantum supremacy experiments. *arXiv preprint arXiv:1612.05903*, 2016.

[30] Andrew W Cross, Lev S Bishop, Sarah Sheldon, Paul D Nation, and Jay M Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3):032328, 2019.

[31] Matteo Paltenghi. Cross-platform testing of quantum computing platforms. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 269–271, 2022.

[32] José Campos and André Souto. Qbugs: A collection of reproducible bugs in quantum algorithms and a supporting infrastructure to enable controlled quantum software testing and debugging experiments. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pages 28–32. IEEE, 2021.

[33] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[34] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S Bishop, Steven Heidel, Colm A Ryan, Prasahnt Sivarajah, John Smolin, Jay M Gambetta, et al. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022.

[35] Yipeng Huang and Margaret Martonosi. Qdb: from quantum algorithms towards correct quantum programs. *arXiv preprint arXiv:1811.05447*, 2018.

[36] Yipeng Huang and Margaret Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 541–553, 2019.

[37] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[38] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4:87–112, 1994.

[39] Lorenzo Moro, Matteo GA Paris, Marcello Restelli, and Enrico Prati. Quantum compiling by deep reinforcement learning. *Communications Physics*, 4(1):178, 2021.

[40] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. Quantum software testing: State of the art. *Journal of Software: Evolution and Process*, 35(4):e2419, 2023.

[41] Manuel De Stefano, Fabiano Pecorelli, Dario Di Nucci, Fabio Palomba, and Andrea De Lucia. The quantum frontier of software engineering: A systematic mapping study. *Information and Software Technology*, page 107525, 2024.

[42] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. Property-based testing of quantum programs in q#. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 430–435, 2020.

[43] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):1–49, 2012.

[44] Lukas Burgholzer and Robert Wille. Qcec: A jkq tool for quantum circuit equivalence checking. *Software Impacts*, 7:100051, 2021.

[45] Lukas Burgholzer and Robert Wille. Improved dd-based equivalence checking of quantum circuits. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 127–132. IEEE, 2020.

[46] Lukas Burgholzer and Robert Wille. The power of simulation for equivalence checking in quantum computing. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[47] Philipp Niemann, Robert Wille, David Michael Miller, Mitchell A Thornton, and Rolf Drechsler. Qmdds: Efficient quantum function representation and manipulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):86–99, 2015.

[48] Lukas Burgholzer, Rudy Raymond, and Robert Wille. Verifying results of the ibm qiskit quantum circuit compilation flow. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 356–365. IEEE, 2020.

[49] Lukas Burgholzer, Richard Kueng, and Robert Wille. Random stimuli generation for the verification of quantum circuits. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 767–772, 2021.

[50] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao 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)*, pages 13–23. IEEE, 2021.

[51] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. Quito: a coverage-guided test generator for quantum programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1237–1241. IEEE, 2021.

[52] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. Generating failing test suites for quantum programs with search. In *International Symposium on Search Based Software Engineering*, pages 9–25. Springer, 2021.

[53] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. Qusbt: search-based testing of quantum programs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 173–177, 2022.

[54] Mariia Mykhailova and Mathias Soeken. Testing quantum programs using q# and microsoft quantum development kit. In *Q-SET@ QCE*, pages 81–88, 2021.

[55] Mariia Mykhailova. The quantum katas: Learning quantum computing using programming exercises. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1417–1417, 2020.

[56] Man-Duen Choi. Completely positive linear maps on complex matrices. *Linear algebra and its applications*, 10(3):285–290, 1975.

[57] Andrzej Jamiołkowski. Linear transformations which preserve trace and positive semidefiniteness of operators. *Reports on Mathematical Physics*, 3(4):275–278, 1972.

[58] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. Muskit: A mutation analysis tool for quantum software testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1266–1270. IEEE, 2021.

[59] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. Application of combinatorial testing to quantum programs. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 179–188. IEEE, 2021.

[60] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[61] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*, pages 254–261. IEEE, 1998.

[62] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James 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)*, pages 549–556. IEEE, 2007.

[63] Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. A comprehensive study of bug fixes in quantum programs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1239–1246. IEEE, 2022.

[64] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923. IEEE, 2015.

[65] Matteo Paltenghi and Michael Pradel. Bugs in quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

[66] Sara Ayman Metwalli and Rodney Van Meter. A tool for debugging quantum circuits. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 624–634. IEEE, 2022.

[67] Xinyi Wang, Tongxuan Yu, Paolo Arcaini, Tao Yue, and Shaukat Ali. Mutation-based test generation for quantum programs with multi-objective search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1345–1353, 2022.

[68] Ke Shang, Hisao Ishibuchi, Linjun He, and Lie Meng Pang. A survey on the hypervolume indicator in evolutionary multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 25(1):1–20, 2020.

[69] Daniel Fortunato, Jose Campos, and Rui Abreu. Mutation testing of quantum programs: A case study with qiskit. *IEEE Transactions on Quantum Engineering*, 3:1–17, 2022.

[70] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[71] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and José A Cruz-Lemus. Quantum software testing: Current trends and emerging proposals. In *Quantum Software Engineering*, pages 167–191. Springer, 2022.

[72] Manuel A Serrano, Ricardo Perez-Castillo, and Mario Piattini. *Quantum Software Engineering*. Springer Nature, 2022.

[73] Nuno Costa, João Paulo Fernandes, and Rui Abreu. Asserting the correctness of shor implementations using metamorphic testing. In *Proceedings of the 1st International Workshop on Quantum Programming for Software Engineering*, pages 32–36, 2022.

[74] Antonio García de la Barrera Amo, Manuel A Serrano, Ignacio García Rodríguez de Guzmán, Macario Polo, and Mario Piattini. Automatic generation of test circuits for the verification of quantum deterministic algorithms. In *Proceedings of the 1st International Workshop on Quantum Programming for Software Engineering*, pages 1–6, 2022.

[75] Antonio García de la Barrera Amo, Manuel A Serrano, Ignacio García-Rodríguez de Guzmán, Macario Polo Usaola, and Mario Piattini. Automatic generation of testing circuits for deterministic quantum algorithms. *Available at SSRN 4421955*, 2023.

[76] Ajay Kumar. Formalization of structural test cases coverage criteria for quantum software testing. *International Journal of Theoretical Physics*, 62(3):49, 2023.

[77] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[78] Peixun Long and Jianjun Zhao. Testing multi-subroutine quantum programs: From unit testing to integration testing. *arXiv preprint arXiv:2306.17407*, 2023.

[79] Peixun Long and Jianjun Zhao. Testing quantum programs with multiple subroutines. *arXiv preprint arXiv:2208.09206*, 2022.

[80] Peixun Long and Jianjun Zhao. Equivalence, identity, and unitarity checking in black-box testing of quantum programs. *arXiv preprint arXiv:2307.01481*, 2023.

[81] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26:1411–1473, 1997.

[82] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

[83] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

[84] Daniel R Simon. On the power of quantum computation. *SIAM journal on computing*, 26(5):1474–1483, 1997.

[85] various authors. *Qiskit Textbook*. Github, 2023.