Mutation Testing of Quantum Programs

Daniel Antunes Bustorff Fortunato

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Rui Filipe Lima Maranhão de Abreu
Prof. José Carlos Campos

September 2021

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisors Prof. Rui Maranhão and Prof. José Campos for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

This work was supported in part by FCT/MCTES through national funds under Grants PTDC/CCI-COM/29300/2017 (FaultLocker), CMU/TIC/0064/2019 (SecurityAware), UIDB/50008/2020 (IT), UIDB/50021/2020 (INESC-ID).

Abstract

As quantum computing is still in its infancy, there is an inherent lack of knowledge and technology to properly test a quantum program. In the classical realm, mutation testing has been successfully used to evaluate how well a program's test suite detects seeded faults (i.e., mutants). In this thesis, building on the definition of syntactically-equivalent quantum operations, we propose a novel set of mutation operators to generate mutants based on qubit measurements and quantum gates. To ease adoption of quantum mutation testing, we further propose QMutPy, an extension of the well-known and fully automated open-source mutation tool MutPy. To evaluate QMutPy's performance we conducted an empirical study on 24 real quantum programs written in the IBM's QISKit library. Our results and observations revealed several issues (i.e., non-optimal code coverage, low mutation scores, minimal number of test cases) that may lead to future failures. We propose coverage and assertion improvements to current quantum test suites, and show how they can increase mutation scores. QMutPy has proven to be an effective quantum mutation tool, providing insight on the current state of quantum test suites and on how to improve them.

Keywords

Quantum computing; Quantum software engineering; Quantum software testing; Quantum mutation testing.

Resumo

Dada a novidade que é a computação quântica, faltam testes aos programas quânticos. Novos métodos para melhorar os testes e para verficar a qualidade das baterias de testes têm de ser desenvolvidos para abordar o problema. A análise de mutação quântica gera versões defeituosas de programas quânticos, chamados mutantes, para verificar a eficácia das técnicas de teste quântico actuais. Desenvolvemos QMutPy, uma extensão do MutPy, uma ferramente de mutação automática que usamos para analisar os programas quânticos do QISKit, a plataforma quântica da IBM. Definimos novos operadores de mutação, especificamente feitos para programas quânticos, que geram mutantes baseados nas medições dos qubits e do uso de portas lógicas quânticas. Para avaliar o desempenho do QMutPy fazemos um estudo empírico em 24 programas quânticos do QISKit. Os resultados obtidos junto com as nossas observações revelam vários problemas (i.e., cobertura do código insuficiente, baterias de testes pequenas, scores de mutação baixos) que podem levar a futuras faltas. Propomos melhorias à cobertura de código e aos casos de teste e mostramos como aumentam o score de mutação. O nosso trabalho valida o QMutPy como uma ferramenta de mutação quântica eficiente, clarificando o estado da arte das baterias de teste quânticas e como melhorá-las.

Palavras Chave

Computação quântica; Engenharia quântica de software; Testes quânticos de software; Testes quânticos de mutação.

Contents

List of Figures

List of Tables

Listings

[1]https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/qiskit/aqua/algorithms/factorizers/shor.py

Acronyms

AOR	Assignment Operator Replacement
AST	Abstract Syntax Tree
CNOT	Controlled-NOT
COI	Conditional Operator Insertion
GUI	Graphic User Interface
LOC	Lines of Code
LOD	Logical Operator Deletion
QGD	Quantum Gate Deletion
QGI	Quantum Gate Insertion
QGR	Quantum Gate Replacement
QMD	Quantum Measurement Deletion
QMI	Quantum Measurement Insertion
QP	Quantum Program
NISQ	Noise Intermediate Scale Quantum
SCD	Super Calling Deletion

1 Introduction

Contents

Quantum computation uses the qubit — the quantum-mechanical analogue of the classic bit — as its fundamental unit instead of the classic bit. Whereas, in classical computing, bits can take on only one of two basic states (e.g., 0 or 1), in quantum computing qubits can be in a superposition state of 0 and 1 that allows them to take an infinity of possible values. Thus qubits can theoretically hold exponentially more information than the same number of classic bits. As a result, quantum computers can, in theory, quickly solve problems that would be extremely difficult for classical computers. Such computation is possible because of qubit properties such as superposition of both 0 and 1, the entanglement of multiple qubits, and interference [7, 8, 9].

The field of quantum computing is still in its infancy but is evolving at a pace faster than originally anticipated [10]. For example, in March 2020, Honeywell announced[1] a revolutionary quantum computer based on trapped-ion technology with quantum volume 64 — the highest quantum volume ever achieved, twice as the state of the art previously accomplished by IBM. Quantum volume is a unit of measure indicating the fidelity of a quantum system. This important achievement shows that the field of quantum computing may reach industrial impact much sooner than originally anticipated.

While the fast approaching universal access to quantum computers is bound to break several computation limitations that have lasted for decades, it is also bound to pose major challenges in many, if not all, computer science disciplines [11], e.g., software testing. Testing is one of the most used techniques in software development to ensure software quality [12, 13]. It refers to the execution of the software in *in vitro* environments that replicate (as close as possible) real scenarios to ascertain its correct behavior.

Quantum Programs (QPs) are much harder to develop than classic programs and therefore programmers, mostly familiar with the classic world, are more likely to make mistakes in the counter-intuitive quantum programming one. Also QPs are necessarily probabilistic and impossible to examine without disrupting execution or without compromising their results. Thus, ensuring a correct implementation of a QP is even harder in the quantum computing realm [14]. Consequently, despite the fact that, in the classical computing realm, testing has been extensively investigated and several approaches and tools have been proposed [15, 16, 17, 18, 19, 20], such approaches cannot be applied to QPs off-the-shelf. This makes the lack of benchmark tools and programs to assess testing effectiveness notable. To tackle this problem new quantum software testing techniques are being developed [21, 22, 23].

Mutation testing [24, 25] has been shown to be an effective technique in improving testing practices, hence helping in guaranteeing program correctness. Big tech companies, such as Google and Facebook, have conducted several studies [26, 27, 28] advocating for mutation testing and its benefits. The general principle underlying mutation testing is that the bugs considered to create versions of the program represent realistic mistakes that programmers often make. Such bugs are deliberately seeded into the original program by simple syntactic changes to create a set of buggy programs called mutants, each

[1] https://www.honeywell.com/us/en/press/2020/03/honeywell-achieves-breakthrough-that-will-enable-the-worlds-most-powerful-qu

containing a different syntactic change. To assess the effectiveness of a test suite at detecting mutants, these mutants are executed against the input test suite. If the result of running a mutant is different from the result of running the original program for at least one test case in the input test suite, the seeded bug denoted by the mutant is considered detected or killed. One outcome of the mutation testing process is the mutation score, which indicates the quality of the input test suite. The mutation score is the ratio of the number of detected bugs over the total number of the seeded bugs.

Just et al. [29] performed a study on whether mutants are a valid substitute for real bugs in classic software testing and they concluded that (1) test suites that kill more mutants have a higher real bug detection rate, (2) mutation score is a better predictor of test suites' real bug detection rate than code coverage. We have no reason to believe that it would be any different in quantum computing. Thus, and in order to shed light on whether manually-written test suites for QPs are effective at detecting mistakes that programmers might often make, in this thesis, we aim to investigate the application of mutation testing on real QPs.

In our work, we focus our investigation on the most popular open-source full-stack library for quantum computing [30], IBM's Quantum Information Software Kit (QISKit) [31][2]. QISKit was one of the first software development kits for quantum to be released publicly and provides tools to develop and run QPs on either prototype quantum devices on IBM Quantum Experience infrastructure or on simulators on a local computer. In a nutshell, QISKit translates QPs written in Python into a lower level language called OpenQASM [32], which is its quantum instruction language. Many famous quantum algorithms such as Shor [33] and Grover [34] have already been implemented using QISKit's API[3].

In this thesis, we propose QMutPy, a novel Python-based toolset that generates and tests mutants for QISKit's [31] QPs[4]. QMutPy is an extension to the popular mutation tool MutPy [1], it possess 5 novel quantum mutation operators. We compare four relevant Python open-source mutation tools and explain why we chose to extend MutPy. In a nutshell, the reason to extend QMutPy is threefold: (i) it supports Python programs (which is the programming language of the popular frameworks QISKit and Cirq), (ii) making it capable to apply not only classic mutant operators but also quantum ones during mutation testing, and (iii) its popularity helps in making the quantum operators available to a large audience. With QMutPy we conduct an empirical study over 24 QPs selected from QISKit which met our criteria and answer 5 proposed research questions. Furthermore, we discuss and execute improvements to current quantum test suites.

[2]`https://qiskit.org`
[3]`https://github.com/Qiskit/qiskit-aqua/tree/stable/0.9/qiskit/aqua/algorithms`
[4]Qiskit is an open-source framework for quantum computing. It provides tools for creating and manipulating QPs using quantum gate arrays and running them on prototype quantum devices on IBM Quantum Experience or on simulators on a local computer. It is arguably amongst the most popular techniques to create QPs

In the realm of quantum testing, there are other, already published, relevant testing tools: MTQC [5] and Muskit [6]. We discuss and compare with QMutPy their differences, such as supported quantum frameworks, supported mutation operators and if they are fully automated.

1.1 Motivation

There is still a whole new world to discover in quantum computing. Many of its concepts are still to be defined and there is still some controversy around those that are. But what we do know for sure is that quantum computing is here to stay and will most likely change the way we perceive our world and do every day tasks in the upcoming years. This is very exciting for everyone but especially for us, computer scientists, for we will have a key role in the development of the tools that will be available for quantum computers. From a researcher's point of view this is the opportunity of a lifetime for we are on the brink of being able to do things we had no idea we could, not so very long ago. We will be able to develop cutting edge technology and be pioneers of quantum computing.

In this thesis we propose to bring a classical testing technique, mutation testing, to the quantum world and analyze how current QPs test suites fare when faced with mutation. This is very exciting for software testers since we get to design new techniques to test a different form of programming.

1.2 Contributions of this Thesis

The main contributions of this thesis are:

- A set of 5 novel mutation operators, leveraging the notion of syntactically-equivalent gates, tailored for QPs.
- A novel Python-based toolset named QMutPy that automatically performs mutation testing for QPs written in the QISKit's [31] full-stack library.
- An empirical evaluation of QMutPy's effectiveness and efficiency on 24 real QPs.
- A detailed discussion on how to extend test suites for QPs to kill more mutants and therefore detect more bugs.

To the best of our knowledge, the study conducted and evaluated in this thesis is the first comprehensive mutation testing study on real QPs, also QMutPy is the first mutation testing tool fully automated that is capable of exhaustively influencing qubit measurements and mutating all quantum gates, as well as classic operators. Our results suggest that QMutPy is capable in generating fault-revealing quantum mutants and it surfaced several issues in the test suites of the real QPs used in the experiments. We have discussed two improvements to test suites, viz. increasing code coverage and improving the quality of test assertions. Such improvements greatly increase the mutation score of the test suites — hence, leading to QPs of higher quality.

At the time of writing this thesis, a paper-based version of this work is under review at the 44th IEEE/ACM International Conference on Software Engineering (ICSE).

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows. We do a background research of quantum computing and explain its differences compared with classical computing in Chapter 2. We present current available open-source mutations tools and detail the extension done for QMutPy in Chapter 3. We detail how our experiment was conducted and subjects were selected in Chapter 4. We present our results in Chapter 5. We discuss and execute improvements to current quantum test suites and how they were impacted in Chapter 6. In Chapter 7 we mention published works about quantum mutation tools. We conclude our thesis and discuss future work in Chapter 8.

2 Background

Contents

Nowadays quantum computing has become a buzzword. It is often used in an uneducated way or in a poor context by people who are not versed on the subject. But one idea that always seems to be present when talking about quantum computing is its capability and potential to make some world-changing breakthrough that just was not possible before. Quantum computing's main promise is substantial speedups over classical machines for many practical applications and this of course cannot be overlooked.

In this Chapter we want to clarify what quantum computing is, explain key concepts (e.g. qubits and quantum gates), talk about the different existing frameworks available to the public, and define what is a QP.

2.1 Quantum Computing

2.1.1 Qubit

As we know, classic computers process and operate on bits, the fundamental unit of memory that we all know and understand, a number that can only be in two states, 0 or 1. Quantum computers are built upon a similar concept but fundamentally different: the quantum bit or qubit for short. Which begs the question what is a qubit?

A qubit, just like the bit, has a state which can be $|1\rangle$ or $|0\rangle$ but contrary to the bit, those are just two possible states. When expressing quantum states we use '$|\rangle$', the Dyrac notation [35], for it is the standard notation for states in quantum mechanics.

The difference between classic states and quantum states, is that quantum states can be in superposition [9], meaning that it is possible to form linear combinations of states. A qubit can be expressed as follows:

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.1}$$

Take note that α and β are complex numbers but for all intents and purposes, we can think of them as real numbers as we will not use the complex part. A qubit can also be expressed as a two-dimensional complex vector space.

Unlike the bit, in which we can easily determine whether it is in state 0 or 1, we cannot determine a qubit's quantum state [9], we can only measure a qubit, and when we do we obtain either 0 with $|\alpha|^2$ probability or 1 with $|\beta|^2$ probability. From this we can derive that $|\alpha|^2+|\beta|^2= 1$ since it is a probability [9].

A qubit exists in a continuum of states between 0 and 1, until it is observed. This state is not fixed, it can vary and we can vary it. To better visualize a qubit we can transform equation (2.1) into:

$$|\Psi\rangle = cos\frac{\theta}{2}|0\rangle + e^{i\varphi}sin\frac{\theta}{2}|1\rangle \tag{2.2}$$

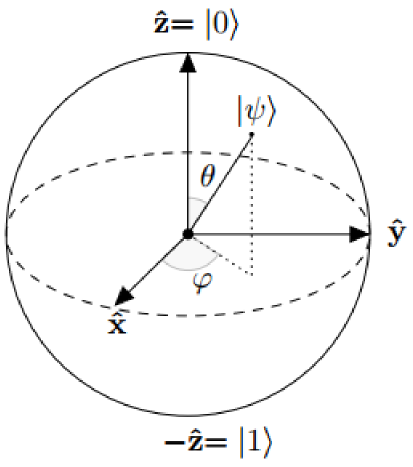


Figure 2.1: Representation of a qubit: the Bloch sphere.

This formula represents the Bloch sphere as shown in Fig. 2.1 and it helps us to perceive what a qubit is, but it can only represent one qubit as there is yet to be a way to visually represent multiple qubits.

Now that we have seen the first important property of a qubit, superposition, we can talk about the second one which is entanglement.

In classical computing we know that bits operate independently from one another, which is not the case in quantum computing. Entanglement is, at the moment, still an ill-defined concept currently being subjected to heavy research, but its main idea is that the state of a qubit affects the state of other qubits in the system [36, 37], meaning that there is a correlation between them. What that correlation is, however, still to be discovered.

Now that we briefly defined what a qubit is and know how information is treated in a quantum computer we can discuss about how we operate on said information.

2.1.2 Quantum computation

A classical computer is built from electrical circuits containing wires and logic gates, similarly a quantum computer is built from quantum circuits containing wires and quantum gates that carry around and operate on qubits. But how does an operation on a qubit (or multiple qubits) work?

The best way to explain this is with an example, so we now introduce some basic quantum gates.

The well known NOT gate that brings a classic bit from 0 to 1 and from 1 to 0 also exists in the quantum world. Simply enough the quantum NOT gate [9] interchanges the weights on α and β. It is represented by the following X matrix:

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.3}$$

If our quantum state is $\alpha|0\rangle + \beta|1\rangle$ the vector notation would be

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{2.4}$$

now if we want to apply the NOT gate to our quantum state our output would be

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}, \tag{2.5}$$

we call this operation a quantum computation, as we operated on a qubit to transform its state. It is worth mentioning that the X matrix (2.3) is the truth table from the classic NOT gate, in which the 1st column is the input and the second the output.

Another very useful quantum gate that we need to mention is the Hadamard gate,

$$H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.6}$$

which turns a $|0\rangle$ into ($|0\rangle$ + $|1\rangle$)/$\sqrt{2}$ that is exactly between the $|0\rangle$ and the $|1\rangle$ state, and turns a $|1\rangle$ state into ($|0\rangle$ - $|1\rangle$)/$\sqrt{2}$ which is also exactly between the $|0\rangle$ and the $|1\rangle$ state. So in fact what the Hadamard gate does is putting a qubit that was initialized at state $|0\rangle$ or $|1\rangle$, or already measured, in a state of perfect superposition where the probability of it being $|0\rangle$ or $|1\rangle$ is exactly 50%. Applying twice the Hadamard gate does of course nothing since H^2 is equal to the identity matrix. Many uses of this gate are shown throughout our work.

Note that apart from the NOT and Hadamard gates there are many single qubit gates [9]. Single qubit gates can be described by two by two matrices. Therefore we can ask ourselves if these matrices have any constraints when being used to represent quantum gates. And yes they have [9]. After the application of a quantum gate we can derive from our equation (2.1) that the result is $|\Psi'\rangle = \alpha'|0\rangle + \beta'|1\rangle$ which must also follow the normalization condition that we established earlier, meaning that $|\alpha'|^2 + |\beta'|^2 = 1$. From that we can say that a matrix *U* describing a single qubit gate must be *unitary*, which means that *U*† *U* = *I*, where *U*† is the *adjoint* of *U*, and *I* is of course the identity matrix. This is the only constraint on quantum gates.

Before moving forward in our work there is another very important gate that we must mention. The Controlled-NOT (CNOT) gate is the prototypical multi-qubit quantum gate, it takes two qubits as input, a control qubit and a target qubit. Why prototypical? Because any multiple qubit quantum gate can be composed of CNOT and single qubit gates [9]. We can describe the CNOT gate as follows:

$$U_{CN} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.7}$$

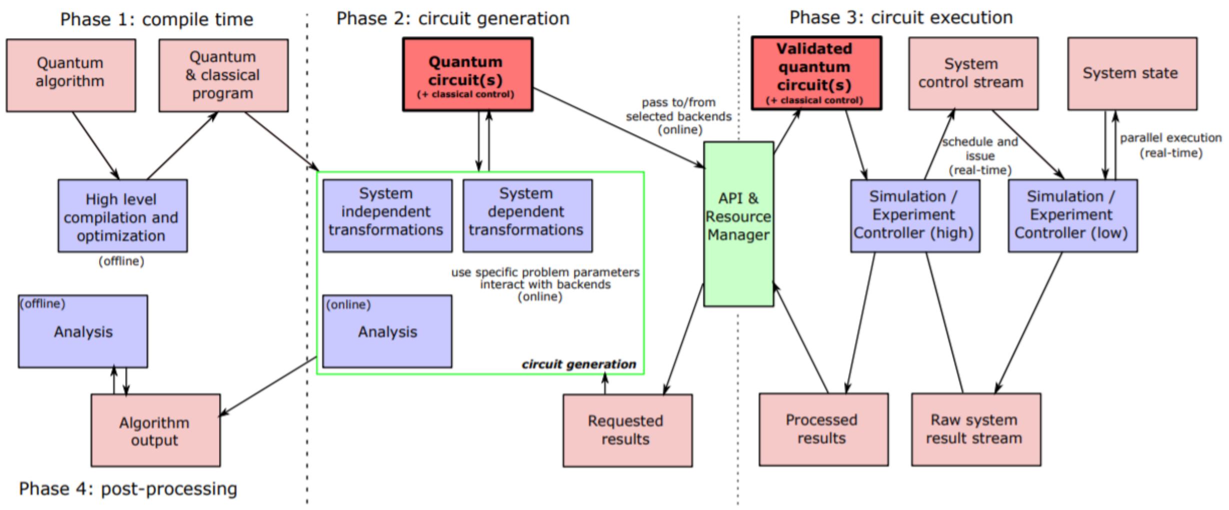

Phase 1: compile time
Phase 2: circuit generation
Phase 3: circuit execution
Quantum algorithm
Quantum & classical program
Quantum circuit(s) (+ classical control)
pass to/from selected backends (online)
Validated quantum circuit(s) (+ classical control)
System control stream
System state
High level compilation and optimization
(offline)
System independent transformations
System dependent transformations
API & Resource Manager
schedule and issue (real-time)
parallel execution (real-time)
Simulation / Experiment Controller (high)
Simulation / Experiment Controller (low)
use specific problem parameters interact with backends (online)
(offline) Analysis
(online) Analysis
circuit generation
Algorithm output
Requested results
Processed results
Raw system result stream
Phase 4: post-processing

Figure 2.2: Block diagrams of processes (blue) and abstractions (red) to transform and execute a quantum algorithm. The API and Resource Manager (green) represents the gateway to backend processes for circuit execution. Dashed vertical lines separate offline, online, and real-time processes.

The CNOT gate does nothing if the control qubit is 0, otherwise the target qubit is flipped. It is very simple and looks like a XOR gate, but fundamentally different. Classic gates are irreversible meaning that from the output of a XOR gate we cannot determine the inputs, this results in an irretrievable loss of information. However, as we have just seen, quantum gates are always invertible since their matrices must be *unitary*. This is crucial for it is a significant source of quantum computation's power.

Now that we have explained how quantum computers function, we explain how a QP is executed.

2.1.3 Quantum Programs

In the work of Cross et al. [32], Green et al. [38] and Svore et al. [39] we get a good description of happens when executing a QP and its different phases of execution.

The execution of a QP can be separated in 4 distinct phases. Figure 2.2 [32] represents a high-level description of all the different steps and necessary abstractions used when running a quantum algorithm, how the algorithm is transformed into executable form, the execution or simulation, and the analysis of the results.

In these different phases intermediate representations are used. The intermediate representation of a computation is a mix between the source language and the target machine instructions. Many intermediate representation may be used in each execution phase for it helps compilers in optimizing and translating programs.

2.1.3.A Compile time.

This first phase happens on a classic computer, in an environment where the program's specific parameters are not yet known and where no interaction with the quantum computer is necessary, meaning it is done offline. The input received in order to compile is the source code of the QPs and the compile time parameters, whereas the output is a combined quantum and classic program expressed using intermediate representations. During this phase we compile classic procedures into object code that does not need complete knowledge of the program's parameters.

2.1.3.B Circuit generation.

The second phase is also done on a classic computer but in a setting where the previously unknown parameters are now available and interactions with the quantum computer are possible and can occur, meaning it is done online if those interactions do happen. The input of the circuit generation is the intermediate representations generated in the last phase and the now known parameters. The output, as one should expect, is a collection of quantum circuits. They can, and usually are, associated with classic control instructions and classic object code needed at run-time. This is expressed as an intermediate representation since further generation of circuits may be possible based on measurement results or classic interactions.

The circuits we generated for our examples, i.e. Figure 2.3 and Figure 2.4, illustrate this. In fact quantum circuits are only straight-line code sequences, and classic control instructions can simply be run-time parameter computations, whereas classic object code can be external algorithms that process measurement outcomes into new conditions or results, or code that generates new circuits on the fly.

2.1.3.C Execution.

The third phase takes place on physical quantum computer controllers and happens in real-time, meaning that the quantum computer is active. The input is the intermediate representation of the previously generated quantum circuits collection, the associated classic control instructions and classic object code. It is then handled to a high-level controller that transforms it into a stream of real-time machine instructions. In simpler terms, this corresponds to the physical operations represented in a low-level format. These are then passed on to the low-level controller and executed. A results stream is also available in case there is some data that needs to be forwarded back to the high-level controller. The output of this whole execution process is a set of measurement results.

2.1.3.D Post-processing.

The fourth and final phase of the program execution is done on a classic computer after all the computation in the real-time environment is over. As we can see in Figure 2.2 the green box receives the set of measurement results obtained from the last phase that are analysed and transformed into the final result of the quantum computation or into intermediate results needed to continue with the program execution.

Now that we explained how a QPs is executed, we turn to the real world and present what tools are already available for computer scientists to explore these different paradigms and how to work with real quantum computers.

2.2 Frameworks

There are many quantum computing frameworks and we could not present them all in our work so we had to make a selection. Keeping in mind that possible contributions to these frameworks was not out of the scope of this work it was necessary that the selected project be open-source and willing to have contributors. Additionally the available documentation had to be complete and clear. Since we want our contribution to have an impact, the project had to be popular and in an advanced stage of development, meaning a high number of pull requests and issues.

Thus after carefully analysing the available open-source projects [40] we decided to present two well known frameworks: QISKit, that is being developed by IBM, and Cirq that is being developed by Google. We give an honourable mention to Microsoft's Q#, since is not qualified as an open-source framework but is a quantum framework comparable to Cirq and QISKit.

Both these frameworks are based on the availability of Noise Intermediate Scale Quantum (NISQ) computers which are devices that allow the use of 50 to 100 qubits and high fidelity quantum gates. To understand how powerful these machines are, the development of algorithms is of utmost importance, because taking full advantage of these machines and their available resources is still a key — though uncertain — procedure [10].

In this section we explain the general layout of these two projects, their goals, what they have already implemented and how they work.

2.2.1 Cirq

2.2.1.A Overview

The framework in development by Google is open-source and available on GitHub[1]. Currently there is much discussion about this framework and it has been the subject of many publications and research. Consequently, it has a great amount of pull requests and issues compared to others [40] however it is relatively new as it was announced in July 2018. Cirq is still in alpha testing and with each new release it is likely that some breaking changes will be released.

Cirq is a Python library for NISQ circuits, enabling developers and researchers to write QPs and run them on quantum computers (not available for public use at the time of writing) or with simulators. This is an important aspect because we can run QPs locally on a simulator, or we can ask to run jobs on the actual computer but this takes more time as jobs need to be queued.

The framework's core goal is to give the user control over quantum circuits, gate usage and behavior, and in scheduling the timing of these gates appropriately within the constraints of the current available hardware.

Cirq's documentation[2] is very clear and contains a whole section that explains how to contribute to their project. This should prove very helpful to contributors for it is considered a good practice to follow pre-established coding guidelines and norms from the beginning of software development. It is also trivial to install since a simple command using pip[3] (the package installer for Python) works. A docker image is available and it can be used in Windows, Linux and MAC OS. This effort to make the framework available in different systems, and easy to install, broadens Cirq's public reach.

Now that we have given an overview of Cirq, we wanted to try out our own example and get the feel of what it is to run a QP with Cirq.

2.2.1.B Example

Cirq represents qubits with a row and column number and as a result visualizes circuits as a grid. The grid can be further divided into *Moments* — all operations for a specific time slice — and *operations* — the application of a gate on a set of qubits.

We already explained what a qubit is and detailed some existing quantum gates. In this next example we put in practice our recently acquired knowledge to create and measure a Bell state [41]. Bell states are 4 entangled quantum states represented by 2 qubits. Listing 2.1 implements this.

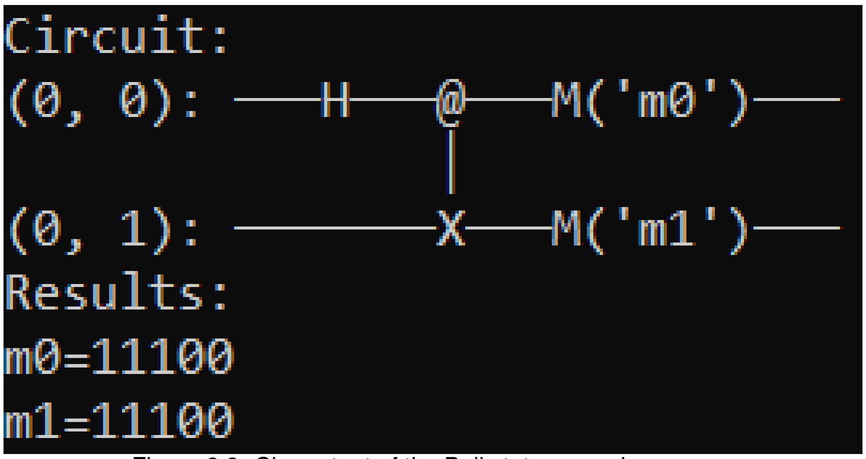
[1]https://github.com/quantumlib/cirq
[2]https://quantumai.google/cirq
[3]https://pypi.org/project/pip/

Listing 2.1: Cirq program implementing a Bell state.

```
import cirq

# Pick qubits.
q0 = cirq.GridQubit(0, 0)
q1 = cirq.GridQubit(0, 1)

# Create a circuit.
circuit = cirq.Circuit(
    cirq.H(q0),               # Apply Hadamard gate.
    cirq.CNOT(q0, q1),        # Apply CNOT gate.
    cirq.measure(q0, key='m0'), # Measurement q0.
    cirq.measure(q1, key='m1') # Measurement q1.
)
print("Circuit:")
print(circuit)

# Simulate the circuit 5 times.
sim = cirq.Simulator()
result = sim.run(circuit, repetitions=5)
print("Results:")
print(result)
```

Cirq by default initializes qubits in state 0, so in this example we apply the Hadamard gate to the first qubit and then the CNOT gate to both (with *q0* as control and *q1* as target), finally measuring both qubits. We remind that the Hadamard gate makes the probability of a qubit to be 0 or 1 exactly 50% when measured and the CNOT gate is the gate that creates a communication between both qubits. If the control qubit is in state 1 it flips the state of the second qubit. Consequently, both qubits will always be in the same state since we only apply the Hadamard gate to *q0* and *q1* is initialized in state 0. In our example we repeated this experiment 5 times as we can see in Figure 2.3. The output of our example confirms the expected result as all 5 measurements for both qubits are the same, in this case 11100 for m0 and m1. This program is executed by a simulator that Cirq provides.

This concludes our presentation of Cirq as we gave a clear overview of Cirq and got to work with the framework demonstrating a powerful quantum concept with a simple program.



```
Circuit:
(0, 0): ───H───@───M('m0')───
               │
(0, 1): ───────X───M('m1')───
Results:
m0=11100
m1=11100
```

Figure 2.3: Cirq output of the Bell state example program

2.2.2 QISKit

2.2.2.A Overview

The framework in development by IBM is also open-source and available on GitHub[4]. QISKit's initial release dates back to March 2017, so it has been in development for more than 4 years now. It was created by IBM to allow software development in their cloud quantum computing service: IBM Quantum[5].

QISKit is a Python library for NISQ circuits, suitable for users without quantum computing expertise to create and manipulate QPs and running them on the available IBM Quantum devices or simulators. It has been used as a testbed for many quantum experiments [42], by researchers and developers, as it is a versatile tool and as a result has been the target of many contributions that advanced its development significantly.

QISKit's goal is to build an easy to use software tool for anyone to use quantum computers. It also aims to be used for solving open issues with current quantum computation problems. The framework allows us to work at the level of circuits, pulses and algorithms.

QISKit's documentation[6] is very oriented to help new users understand the structure of the framework, its components and how they can contribute to the project. It has a section specifying guidelines contributors need to follow and a road map section that has information on what they developed until now and on what they plan on developing for the next period of time (their defined period of time is 12 months). It also instructs how to set up QISKit on the user's machines and explains how to run programs on the actual quantum computers available at IBM Quantum, instead of just using the simulators, and how to create an account and obtaining the necessary credentials in order to do so. All of this informa-

[4] https://github.com/Qiskit/qiskit
[5] https://quantum-computing.ibm.com/
[6] https://qiskit.org/documentation/

tion is easy to follow and has step-by-step instructions making it very easy for everyone to work with and understand the tool. QISKit is available for the usual operating systems: Windows, Linux and MAC OS.

QISKit is a larger project than Cirq currently. Its functionalities are separated in 4 main components which ease the tool's comprehension. Components in QISKit are called elements, the four elements are *Terra*, *Ignis*, *Aqua* and *Aer*. Each element focuses on specific functionalities [43]. For this work the most relevant part of QISKit is *Aqua*[7]; it is where the high-level algorithms for quantum computers are built. It provides the user with high-level interfaces in order to handle quantum hardware or simulators without the necessity of learning how to construct quantum circuits. It is here that we make quantum computing live up to its expectations and develop quantum algorithms that have real-world applications and that surpass classic algorithms. Current algorithms are mostly focused on chemistry, optimization, finance and AI problems, making the most out of the benefits that quantum computers brings by executing specific computational tasks that would be very — perhaps impossibly — heavy for classic computers.

As we did for Cirq, we try out an example to gain some hands-on experience.

2.2.2.B Example

Since we want to compare the use of Cirq and QISKit we also create and measure a Bell state [41]. Listing 2.2 implements this.

Listing 2.2: QISKit program implementing a Bell State.

```
from qiskit import QuantumRegister, ClassicalRegister
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram

q = QuantumRegister(2)     # Pick qubits
c = ClassicalRegister(2)   # Pick corresponding bits
qc = QuantumCircuit(q, c)  # Create circuit

qc.h(q[0])                 # Apply Hadamard gate
qc.cx(q[0], q[1])          # Apply CNOT gate
qc.measure(q, c)           # Measure

# Get Simulator
backend = Aer.get_backend('qasm_simulator')

# Execute job 1000 times
job = execute(qc, backend, shots = 1000)
```

[7]Although QISKit-*Aqua*'s repository (https://github.com/Qiskit/qiskit-aqua) has been deprecated as of April 2021, all its functionalities "are not going away" and have been migrated to either new packages or to other QISKit packages. For example, core algorithms and operators' functions have been moved to the QISKit-Terra's repository. More information in https://github.com/Qiskit/qiskit-aqua/#migration-guide.

```
result = job.result()
counts = result.get_counts(qc)

print("Circuit:")
qc.draw()

print("Results:")
print(counts)

plot_histogram(counts)
```

Creating a circuit in QISKit is not so different from Cirq. Nevertheless, the main differences is that QISKit uses *QuantumRegisters*, that initialize the number of qubits we want, and *ClassicalRegisters* that map each qubit with a corresponding classic bit. To create a circuit a *QuantumRegister* and a *ClassicalRegister* are used as input. In this example *ClassicalRegisters* are used to store the qubits' states when measured which, as we know, is always 0 or 1. Apart from the classic part of the circuit (Figure 2.4), it is very similar to the one we saw earlier with Cirq (Figure 2.3).

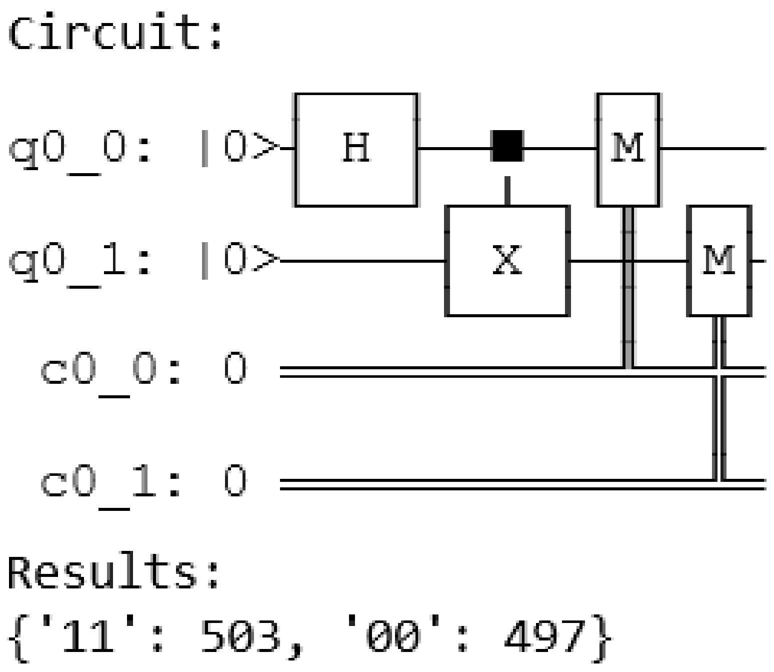
In this example we repeated the experiment 1000 times (we did this 5 times for Cirq) for curiosity's sake and because there exists visualization tools that allow us to do so. As we already discussed the pairs of states we obtain when running this program will always be 00 or 11. We obtained 503 times the pair 11 and 497 times the pair 00, this concurs with what we were expecting since the probabilities of obtaining one of those pairs is 50%.

Looking at this last example we can see that QISKit is more developed and has more functionalities than Cirq, but is as simple! It was easy to obtain the knowledge to write this and while researching we noticed that functions in QISKit were well defined and possessed more information in general than Cirq's. Also, QISKit's visualization tools are somewhat more developed than Cirq's. In the last line of code of Listing 2.2 we plotted a histogram that shows the probabilities of each measurement, as you can see in Figure 2.5. We believe these kind of incorporated tools will be of great help in the development of more complex programs.

2.2.2.C Backends

QISKit, contrary to Cirq, has real-life quantum devices available for public use. We showcase the hardware and simulators made available by IBM to run QISKit programs.

When running a program we choose whether we want to run it on the actual quantum computers or with the simulators. This is an important decision, considering that running them on the actual hardware takes longer since jobs need to be queued.



```
Circuit:
```

```
Results:
{'11': 503, '00': 497}
```

Figure 2.4: Bell State QISKit's example program: ouput.

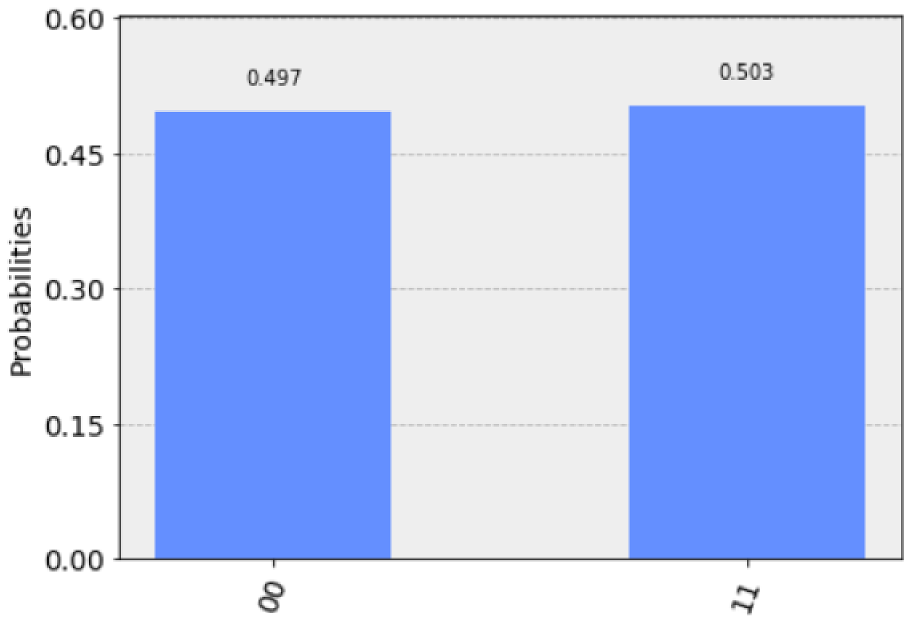


Figure 2.5: Bell State QISKit's example program: histogram.

Backends provide a fixed maximum of available qubits that we can manipulate, so depending on how many we want to use we may want to run our program on a different backend. Looking back to Listing 2.2, we chose our backend with the *get_backend* function and used the simulator. Thus no other action was required, although if we had chosen to run the example on real hardware, we would have had to provide credentials in the program.

IBM initially launched two quantum computers for public use: *IBM QX2* and *IBM QX3*. The first one could manipulate up to 5 qubits and the second one up to 16. But these were, respectively, later revised into *IBM QX4* and *IBM QX5*. Despite that, the number of qubits available for manipulation did not change. More than one of these two types of computers are available throughout the world in different IBM labs.

As we stated in the beginning of our work, quantum hardware is under heavy research but still very immature so it comes as no surprise when the computers can only support elementary single qubit operations and CNOT gates (this information is available in more detail in their backend information repository in GitHub[8], and their IBM Quantum interface). Combinations of these can form other gates.

Current hardware suffers from coupling restrictions, meaning that the user cannot arbitrarily place multiple-qubit gates when creating circuits because they are restricted to some prescribed pairs of qubits [43]. These restrictions are given by a coupling-map that is different in each of the computers and must be taken into account depending on which we want to use. This is one of the issues that the *Ignis* library tries to solve.

Apart from the hardware, a simulator is also available. We used it in QISKit's example, it is called the *qasm_simulator* and it allows the manipulation of up to 32 qubits. When given a series of quantum operations and a quantum state the simulator changes its state. This is simply a sequence of matrix multiplications that given an input state and the operations defined in a circuit, yields a result. When multiplied, vectors and matrices grow exponentially. This limits the power of simulations.

2.3 Summary

In this Chapter we explained what a qubit is and its special properties derived from the quantum paradigm (i.e., superposition, entanglement). We established what a quantum gate is and presented 3 of the most important quantum gates, the NOT, Hadamard and CNOT gates, and how they operate on qubits. We explained how a QP works and its different phases of execution (i.e., compile time, circuit generation, execution and post-processing). We described two well-known quantum frameworks, IBM's QISKit and Google's Cirq, and ran a simple QP for each highlighting their differences.

[8]https://github.com/Qiskit/ibmq-device-information/tree/master/backends

In the next Chapter we will present various Python mutation tools, describe in detail our selected tool, MutPy, and detail the extension we performed.

3 QMutPy

Contents

As we saw in Chapter 2 quantum frameworks (e.g., Microsoft's Q#, Google's Cirq and IBM's QISKit) implement the necessary tools to program quantum computers.

For our work we decided to work with QISKit as it was the first quantum programming language to be released publicly and has already many famous quantum algorithms implemented which favor its use. The number of QPs and their corresponding individual test suites was an important factor in this decision since we wanted to do a mutation analysis in as many QPs as we could (see Chapter 4). We revisit Cirq at a later stage of our work since Cirq's test suites are bundled together and it would require a significant amount of manual work to organize the files for our experiment.

Our purpose is to develop new ways to test QPs. Since many classical testing methods cannot be used without destroying the superposition state of qubits, finding new or improve classical testing methods is not a trivial matter. However, mutation testing can bypass this problem.

Mutation testing is used by software testers to assert the quality of test suites. We propose creating new mutation operators to mutate QPs and check for the quality of existing test suites. This led us to choose a mutation tool we could extend and implement new operators. Considering that most quantum programming frameworks are written in Python [30], we figured the most suitable mutation tool would be one written in Python.

In this Chapter we compare Python mutation tools and explain why we chose to work with MutPy. We do an in-depth analysis of how MutPy works. We define the concept of syntactically-equivalent gates and, finally, describe each mutation operator we added to MutPy and how they work.

3.1 Python-based Mutation Testing Tools

QPs written in Python and using QISKit library are a mix of classic operations (e.g., initialization of variables, loops), as well as quantum operations (e.g., initialization of quantum circuits, measuring qubits). Thus, we foresee that the most suitable mutation tool for QPs would be one that

- Supports Python programs and the two popular testing frameworks for Python: `unittest` and `pytest`.
- Supports various classic mutation operators (e.g., Assignment Operator Replacement (AOR), Conditional Operator Insertion (COI)).
- Supports the creation of a report that could be shown to a developer or easily parsed by an experimental infrastructure (as the one described in Chapter 4).
- Fosters wide adoption, the learning curve to install, configure and use the tool ought to be low.

Mutatest [4], mutmut [2], MutPy [1], and CosmicRay [3] are the most popular mutation testing tools for Python that are available through pip. Table 3.1 reports the most relevant features of each mutation tool

	MutPy [1]	mutmut [2]	Cosmic Ray [3]	Mutatest [4]
Open-Source	✓	✓	✓	✓
Language	Python	Python	Python	Python
Installing/Setup	🪫	🪫	🔋	🪫
Mutation operators	AOD, AOR, ASR, BCR, COD, COI, CRP, DDL, EHD, EXS, IHD, IOD, IOP, LCR, LOD, LOR, ROR, SCD, SCI, SIR	value mutations, decision mutations, statement mutations	Binary Operator Replacement, Boolean Replacer, Break/-Continue, Comparison Operator Replacement, Exception Replacer, Keyword Replacer, Number Replacer, Remove Decorator, Unary Operator Replacement, Zero Iteration For Loop	AugAssign, BinOp, BinOp Bitwise Comparison, BinOp Bitwise Shift, BoolOp, Compare, Compare In, Compare Is, If, Index, NameConstant, Slice
Can select operators	✓	✗	✓	✓
Test framework	unittest, pytest	any	unittest, pytest	pytest
Report	yaml, html	xml	html	—
Fully automated	✓	✓	✓	✓

Table 3.1: MutPy [1] vs. mutmut [2] vs. Cosmic Ray [3] vs. Mutatest [4]. Regarding testing frameworks mutmut supports all test runners (because mutmut only needs an exit code from the test command)

and in the following subsections we describe their advantages and disadvantages. Albeit being open-source, fully automated, and support classic mutation operators, not all tools fulfil all our requirements.

Mutatest [4] only supports `pytests` whereas, e.g., the programs in the QISKit-*Aqua*'s repository[1] require `unittest`. It neither produces a report of a mutation testing session. Thus, any postmortem analysis (e.g., statistical analysis) could not be easily performed.

mutmut defines its mutations in three types, value mutations (string, number mutations), decision mutations (logical, keywords mutations) and statement mutations (removing or changing a line of code). As no real definitions of mutation operators are provided the user cannot choose which operator he wants to use. Thus, a developer that decides to use it would have to wait for all mutants to be analyzed. This can be severely time consuming as a program could have thousands of mutants, and more importantly a developer would not be able to, e.g., only select quantum mutation operators. Thus, using mutmut would be unproductive.

MutPy [1] and Cosmic Ray [3] are similar in nature. Both provide a reporting system, support `unittest` and `pytest`, and allow one to select a subset of mutation operators. However, from our own experience in installing and running the tools, MutPy's learning curve is more gradual than Cosmic Ray. The latter requires a configuration file, which could be useful to long runs of the tool, that requires one to learn how to setup such file. With MutPy, on the other hand, the installation procedure is quite simple and to actually perform mutation testing one would only need to execute a command with no

[1] https://github.com/Qiskit/qiskit-aqua/tree/stable/0.9

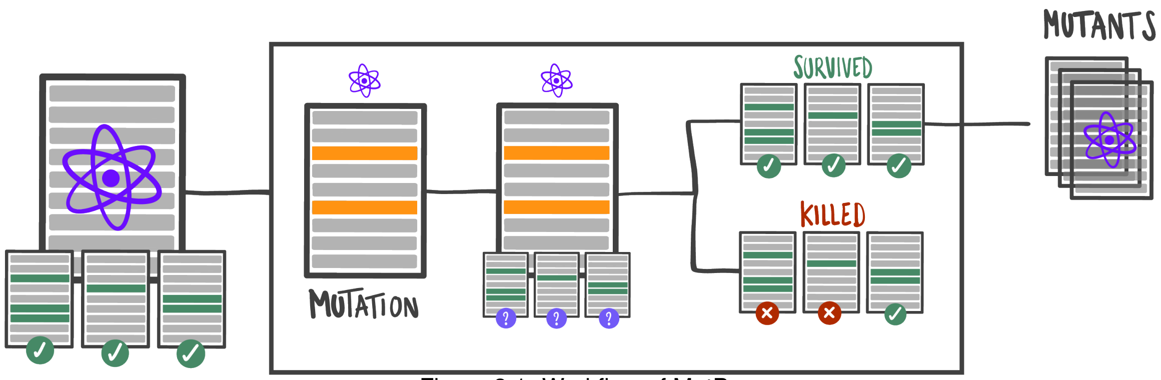


Figure 3.1: Workflow of MutPy

previous configuration. Thus, MutPy [1] is the only tool that fulfils all of the necessary requirements for a mutation tool.

3.2 MutPy

MutPy [1] is an open-source Python mutation testing tool with in-built mutation operator. Installing and using MutPy is simple and straightforward. MutPy's workflow is shown in Figure 3.1 and is divided in four steps. Given a Python program P, its test suite T, and a set of mutation operators M, MutPy's workflow is as follows:

1. MutPy starts the mutation process, it loads P's source code and test suite;
2. Executes T on the original (unmutated) source code;
3. Applies M, generates all mutant versions of P and executes T on each mutant;
4. Provides a summary of the results either as a `yaml` or `html` report.

Since step 1 and 2 are self-explanatory, we will focus on step 3 and 4.

There are currently 20 mutation operators available in MutPy plus 7 experimental mutations. If the user does not specify any given operator MutPy will try all of them by alphabetical order. Examples of some of the most used classic mutation operators would be the Arithmetic Operator Deletion (AOR), Constant Replacement (CRP) or Conditional Operator Deletion (COD).

In step 3, MutPy parses through the code and for each operator checks if there are mutations that it can produce. Mutations in MutPy are done through the Python Abstract Syntax Tree (AST). When a possible mutation is found, the corresponding node from the AST is removed and a mutated node is created and injected into the source code. If at least one mutation is found for an operator, the corresponding node from the AST is removed and a mutated node is created and injected into the source code. MutPy then executes the mutant with the given tests and produces the result. The mutant can produce four types of results: killed, survived, incompetent or timeout (the latter are usually considered killed). Apart from the mutant result, MutPy also shows the code differences that it produced and the time it took to execute the mutation.

In step 4, MutPy shows the aggregate final result: the total number of mutants, of mutants killed, of mutants that survived, of incompetent mutants, of timeout mutants, the mutation score and the time it took to execute all this process.

MutPy is built in such a way that it is straightforward to extend it with new mutation operators. Notwithstanding, addressing the technical challenges to implement the quantum operators, we added the possibility for the tool to mutate AST $Call$s[2].

We propose QMutPy (`https://github.com/danielfobooss/mutpy`), an extension of MutPy. QMutPy has 5 new mutation operators: Quantum Gate Replacement (QGR), Quantum Gate Deletion (QGD), Quantum Gate Insertion (QGI), Quantum Measurement Deletion (QMD) and Quantum Measurement Insertion (QMI).

3.3 Quantum Mutation Operators

In this section, we explain our mutation strategy, i.e. a formal definition of the concept of syntactically-equivalent gates and the five novel mutation operators tailored for QPs, and the implementation details of QMutPy — our proposed Python-based toolset to automatically perform mutation testing for QPs written in QISKit [31].

3.3.1 Syntactically-equivalent gates

From Chapter 2 we learned that similarly to classic programs, a QP is fundamentally a circuit in which qubits are initialized and go through a series of operations that change their state. These operations are commonly known and referenced to as quantum gates. We saw that two of the most popular and used quantum gates are the NOT gate and the Hadamard gate, usually referred in code as the `x` gate and the `h` gate, respectively. They are single-qubit operations, i.e., they change the state of one qubit [44].

At the time of writing, QISKit v0.29.0 provides support to more than 50 quantum gates[3]. This includes single-qubit gates (e.g., `h` gate), multiple-qubit gates (e.g., `cx` gate) and composed gates, or circuits, (e.g., `QFT` circuit). Given their importance on the execution and result of a QP, as a simple typo on the name of the gate could cause bugs that developers may not be aware of, our set of mutation operators to generate faulty versions of QPs is based on single- and multi-qubit quantum gates, in particular, syntactically-equivalent gates.

[2]`https://docs.python.org/3/library/ast.html#ast.Call`
[3]`https://qiskit.org/documentation/apidoc/circuit_library.html`

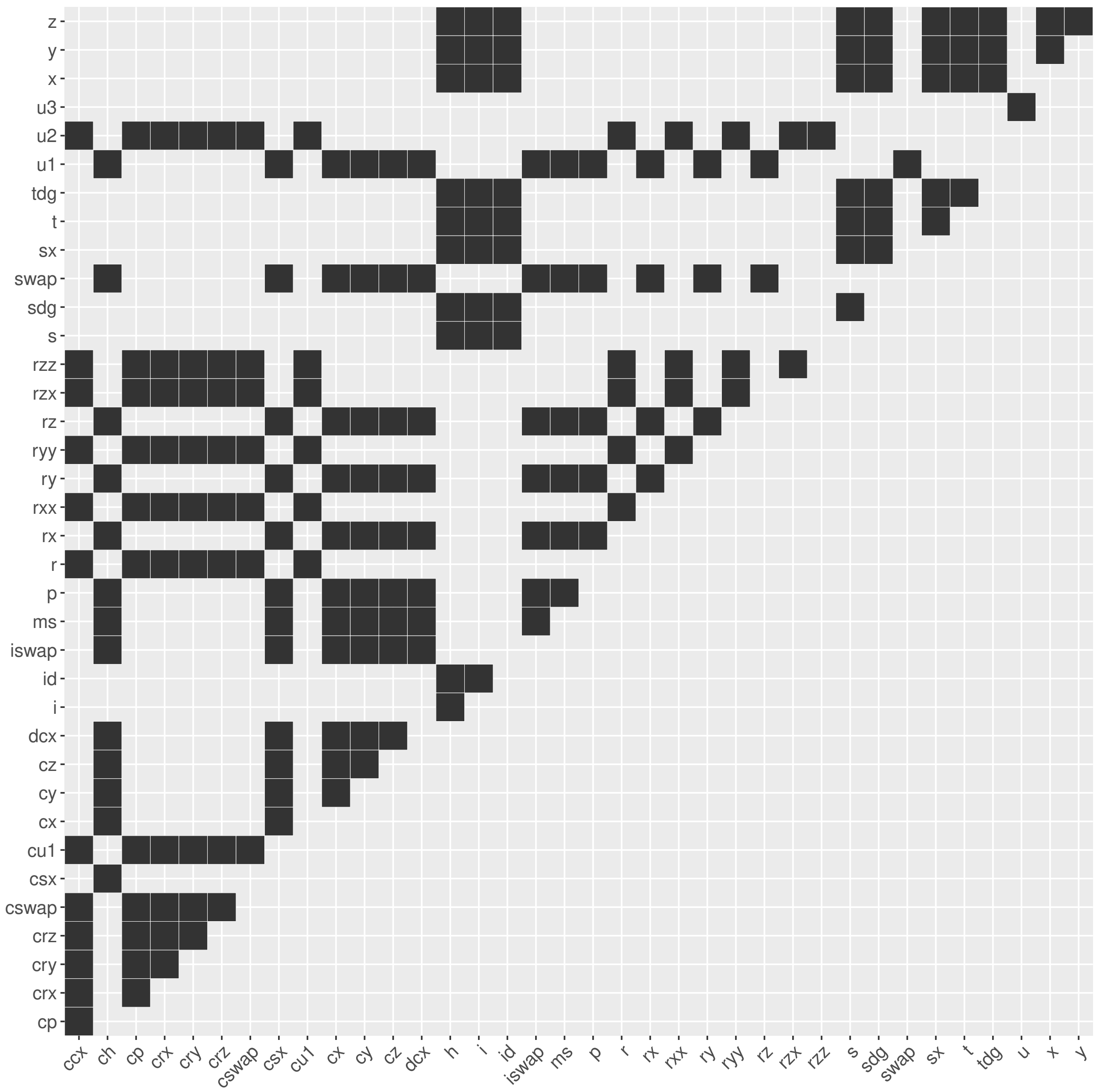


Figure 3.2: Equivalent gates.

Formally, a gate g is considered syntactically-equivalent to gate j if and only if the number and type of arguments[4] required by both g and j are the same. At the time when we performed our experiment, we had identified 40 gates that had syntactical-equivalents. Figure 3.2 lists all gates and their syntactically-equivalent ones. For instance, the `h` gate has 10 syntactically-equivalent gates: `i`, `id`, `s`, `sdg`, `sx`, `t`, `tdg`, `x`, `y`, and `z`. Note that these gates do not perform or compute the same operation; they are simply used in the same manner and required the same number and type of arguments.

3.3.2 Quantum Gate Replacement

QMutPy possesses a new operator: QGR. QGR is an operator that can identify quantum gate functions and change them to a syntactically-equivalent gate, as shown in Figure 3.2.

When identifying a quantum gate we do all possible mutations for every equivalent gate. For example if we encounter the `h` gate we will generate 10 mutants since it has 10 equivalent gates.

Listing 3.1 exemplifies the use of the QGR operator. The code extract contains 3 gates which have syntactically-equivalent gates. Given this code, QMutPy creates the mutant (red line removed, green line added) and runs it against the test suite. After performing all the possible mutations on line 153, it would then pass to line 154, do all the possible mutations and so on.

Listing 3.1: Example of a QGR: extract from `shor`'s source code[5] after a QGR.

```
153 -   circuit.x(qubits[0])
153 +   circuit.h(qubits[0])
154     circuit.cx(qubits[0], ctl_aux)
155     circuit.x(qubits[0])
```

3.3.3 Quantum Gate Deletion

Adding and removing gates from a circuit can have a significant impact on its result. The QGD operator as the name suggests consists in deleting the addition of a gate to a circuit. Listing 3.2 shows the mutated source code.

Listing 3.2: Example of a QGD: extract from `shor`'s source code after a QGD.

```
153 -   circuit.x(qubits[0])
153 +   pass
154     circuit.cx(qubits[0], ctl_aux)
155     circuit.x(qubits[0])
```

[4]Optional arguments are not taken into consideration.
[5]https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/qiskit/aqua/algorithms/factorizers/shor.py

3.3.4 Quantum Gate Insertion

This quantum mutation operator performs the opposite action of the QGD operator. That is, instead of deleting a call to a quantum gate, it inserts a call to a syntactically-equivalent gate. For each quantum gate in the source code, this mutation operator creates as many mutants as the number of each syntactically-equivalent gates. For example, for the `x` gate, which has 10 syntactically-equivalent gates, it creates 11 mutants, one per equivalent gate. Note that the `x` gate itself can be inserted in the source code, counting as a valid mutant. Listing 3.3 shows an example of the use of this operator.

Listing 3.3: Example of a QGI: extract from `shor`'s source code after a QGI.

```
-   circuit.x(qubits[0])
+   __qmutpy_qgi_func__(circuit, qubits[0])
    circuit.cx(qubits[0], ctl_aux)
    circuit.x(qubits[0])
+   def __qmutpy_qgi_func__(circuit, qubit)
+       circuit.x(qubit)
+       circuit.y(qubit)
```

3.3.5 Quantum Measurement Insertion

In quantum computing measuring a qubit breaks the state of superposition, the qubit value becomes either 1 or 0. Therefore adding measurements can alter the behavior of a QP, hence it is considered a mutation. As such we added a new operator to QMutPy: QMI. Following the addition of a gate to a circuit, this operator measures the qubit which the gate was added to (see Listing 3.4).

Listing 3.4: Example of a QMI: extract from `shor`'s source code after a QMI.

```
-   circuit.x(qubits[0])
+    __qmutpy_qmi_func__(circuit, qubits[0])
    circuit.cx(qubits[0], ctl_aux)
    circuit.x(qubits[0])
+   def __qmutpy_qmi_func__(circuit, qubit)
+       circ.x(qubit)
+       measur_cr = ClassicalRegister(circ.num_qubits)
+       circ.add_register(measur_cr)
+       circ.measure(qubit, measur_cr)
```

3.3.6 Quantum Measurement Deletion

Similarly to measurement insertion, if we remove a measurement from a QP, we are purposely keeping the superposition state, and as a consequence do not converge the qubit to either 1 or 0. Thus we added a new operator: QMD. This operator removes qubit measurements that exist in the QP. Listing 3.5 shows the mutant created by QMutPy.

Listing 3.5: Example of a QMD: extract from `shor`'s source code after a QMD.

```
254     up_cqreg = ClassicalRegister(2 * self._n, name='m')
259     circuit.add_register(up_cqreg)
260 -   circuit.measure(self._up_qreg, up_cqreg)
260 +   pass
```

3.4 Summary

In this Chapter we compared 4 Python mutation tools and decided to extend MutPy, given that it fulfilled all of our requirements. We did an in-depth analysis of MutPy showing its workflow. We defined the concept of syntactically-equivalent gates and listed all gates that had equivalents found in QISKit. We presented the 5 novel mutation operators we implemented in QMutPy (i.e., QGD, QGI, QGR, QMI and QMD).

In the next Chapter we will define the methodology and protocols employed for our empirical study and propose 5 research questions to evaluate QMutPy's effectiveness and efficiency at creating quantum mutants.

4

Empirical Study

Contents

In this Chapter we define the settings with which we have conducted an empirical study that evaluates QMutPy's effectiveness and efficiency at creating quantum mutants. We aim to study how well manually-written test suites for QPs detect syntactical changes, i.e., mutants. As such we propose to answer the following research questions:

RQ1: How does QMutPy perform at creating quantum mutants?

RQ2: How many quantum mutants are generated by QMutPy?

RQ3: How do test suites for QPs perform at killing quantum mutants?

RQ4: How many test cases are required to kill or timeout a quantum mutant?

RQ5: How are quantum mutants killed?

As baseline, we have compared the results achieved by QMutPy's quantum mutation operators with MutPy's classic mutation operators[1]. Note that works [5, 6] on quantum mutation are very preliminary and no other classic or quantum mutation tool could have been used in our empirical study as baseline (see Section 3.1 and chapter 7 for more information).

4.1 Experimental subjects

To conduct our empirical study we require (1) real QPs written in the QISKit's framework [31] (as, currently, QMutPy only supports QISKit's quantum operations), (2) QPs written in Python[2], (3) an open-source implementation of each QP, and (4) a test suite of each QP. To the best of our knowledge there are four main candidate sources of QPs that fulfil (1): the QISKit-*Aqua*'s repository[3] itself, the "Programming Quantum framework repository Computers" book's repository[4] from O'Reilly, the "QISKit Textbook Source Code"'s repository[5] from the QISKit Community, and the official "QISKit tutorials"'s repository[6].

QISKit-*Aqua*'s repository provides the implementation of 24 QPs in Python, including the successful Shor [33], Grover [34], and HHL [45], and a fully automated test suite for each program. Hence, it fulfils all our requirements.

O'Reilly's book provides the implementation of 182 QPs, 29 written using the QISKit's framework. However, no test suite is provided for any of the 182 programs. Hence, it does not fulfil (4).

"QISKit Textbook Source Code"'s and "QISKit tutorials"'s repositories provide Jupiter Python notebooks with examples on how to interact with the QISKit's framework. No test suite is available for any of

[1] `https://github.com/mutpy/mutpy#mutation-operators`
[2] Although Jupiter Python notebooks include Python source code, they are not supported by QMutPy.
[3] `https://github.com/Qiskit/qiskit-aqua/tree/stable/0.9/qiskit/aqua/algorithms`
[4] `https://github.com/oreilly-qc/oreilly-qc.github.io/tree/1b9f4c1/samples`
[5] `https://github.com/qiskit-community/qiskit-textbook/tree/3ffedf9`
[6] `https://github.com/Qiskit/qiskit-tutorials/tree/eb189a6`

Algorithm	LOC	# Tests	Time (seconds)	% Coverage
adapt_vqe	151	5	85.66	82.78
bernstein_vazirani	80	33	4.28	98.75
bopes_sampler	91	2	320.51	81.32
classical_cplex	210	1	0.04	81.43
cobyla_optimizer	75	4	1.60	94.67
cplex_optimizer	60	3	0.70	81.67
deutsch_jozsa	85	64	4.18	98.82
eoh	70	2	34.71	100.00
grover	381	593	153.77	95.54
grover_optimizer	197	6	21.14	96.45
hhl	341	21	630.65	93.26
iqpe	231	3	20.38	93.51
numpy_eigen_solver	220	5	0.10	76.36
numpy_ls_solver	56	1	0.00	92.86
numpy_minimum_eigen_solver	73	5	0.24	94.52
qaoa	96	18	49.45	95.83
qgan	226	11	349.72	84.51
qpe	197	3	21.27	94.92
qsvm	303	8	266.19	78.22
shor	265	13	251.76	93.21
simon	89	48	17.21	98.88
sklearn_svm	88	4	0.13	76.14
vqc	443	13	1626.38	85.55
vqe	386	19	811.27	85.49
Average	183.92	36.88	194.64	89.78

Table 4.1: Details of QPs used in the empirical evaluation.
The test suite of each QP was identified and selected based on each program's name. In QISKit, a QP is named after the algorithm it implements and to its test suite is given the prefix "test". For example, the test suite `test_shor.py` corresponds to the program `shor.py`. Code coverage was measured using the `Coverage.py` tool.

the examples. These two sources aim at teaching developers who want to use QISKit for writing QPs, therefore, and to ease the execution (and likely the understanding) of such examples they are provided as notebooks rather than traditional Python files. Hence, it does not fulfil (2) nor (4).

In total, the 24 QPs in the QISKit-*Aqua*'s repository meet our criteria. On average, the considered QPs have 184 Lines of Code (LOC), where the smallest program has 56 LOC (`numpy_ls_solver`) and the largest has 443 (`vqc`). The number of tests and the time required to run all tests differ greatly. The number of tests ranges from 1 test (`classical_cplex` and `numpy_ls_solver`) to 593 tests (`grover`), and the runtime ranges from nearly 0 seconds (`numpy_ls_solver`) to 1627 seconds (`vqc`).

Regarding code coverage, on average, QPs' test suites cover 90% of all LOC. This is in line with best practices [46] and also in line with a previous study conducted by Fingerhuth et al. [30] where ratio of code exercise by QPs' tests was slightly above the industry-expected standard. The QP with the lowest code coverage is `sklearn_svm` with 76.14% and the program with the highest coverage is `eoh` with 100%. Upon further analysis we found that most of the uncovered LOC are error messages and exceptions. For quantum mutants nearly all mutated lines were covered, only two lines were not covered, one in the `vqc` QP and one in the `hll` QP. The same cannot be said for classic mutants.

4.2 Experimental setup

All experiments were executed on a machine with an AMD Opteron 6376 CPU (64 cores) and 64 GB of RAM. The operating system installed on this machine was CentOS Linux 7. We used Python version 3.7.0 in our experiments because it is the version supported by QMutPy and one of the required versions of QISKit. We also used `virtualenv`[7] to create an isolated virtual environment to run all test suites so that (1) others could reproduce and replicate our experiments, and (2) no non-relevant library or incorrect versions of a library would be loaded by Python and invalidate our experiments. To run all experiments in parallel we used the GNU Parallel tool [47].

In our experiments, we ran QMutPy with two configurations: first with classic mutants, and then with quantum mutants. For both configurations we used MutPy's defaults parameters. For example, the timeout factor set to 5 times the time a test takes to execute a non-mutated version of the program under test.

For each QP / test suite we collected the number of mutants generated, the number of mutated LOC and the ratio of mutants per LOC, the number of mutants killed, the number of mutants that survived and were exercised by the test suite and that survived and were not exercised by the test suite, the number of incompetent mutants, the number of timeout mutants, the mutation score calculated with the number

[7] `https://virtualenv.pypa.io`

of survived mutants exercised and not exercised by the test suite and finally the time it took to run all mutants.

4.3 Experimental metrics

To be able to compare the effectiveness of each test suite at killing mutants we first compute its mutation score [24], i.e., ratio of killed mutants to total number of mutants (excluding incompetent mutants, e.g., mutants that introduce non-compiling changes). Formally, the mutation score of a test suite T is given by:

$$\sum_{o \in O} \frac{\frac{|K_o|}{|M_o|-|I_o|}, |M_o| - |I_o| > 0}{|O|} \times 100\% \tag{4.1}$$

where O represents the set of mutation operators and o a single mutation operator, $|M_o|$ the number of mutants injected by o, $|I_o|$ the number of incompetent mutants generated by o, and $|K_o|$ the number of mutants (of o) killed by T.

As some mutants might not be killed by T because the mutated code is not even executed by T, in our empirical analysis we also report a mutation score which ignores mutants that are not executed by T. This score would allow one to assess the maximum mutation score T could achieve. Formally, this score is computed as:

$$\sum_{o \in O} \frac{\frac{|K_o|}{|E_o|-|I_o|}, |E_o| - |I_o| > 0}{|O|} \times 100\% \tag{4.2}$$

where $|E_o|$ represents the number of mutants injected by m and exercised by T.

Regarding time, we compute and report three different runtimes: (1) total time to perform mutation analysis on test suite T which includes the time to create the mutants and run all tests on all mutants (Runtime column in Table 5.1), (2) time to inject a mutant in a non-mutated code (Generate mutant in Figure 5.2), (3) time to create a mutated module after injecting the mutant (Create mutated module in Figure 5.2).

4.4 Threats to Validity

Based on the guidelines reported in [48], we discuss threats to validity.

A – Threats to External Validity: The QPs used in our empirical evaluation might not be representative of the whole QPs population. Moreover, the state of test cases selected for each QP might not be complete (i.e., we may have missed other test cases in QISKit-Aqua that test the QPs' code). To minimize these threats, we selected QPs of various sizes, types, and levels of test coverage. Note that the lack of real-world QPs is a well-known challenge [49, 50]. Another threat is that we compared the results for only one, yet popular, quantum framework (QISKit). Caution is required when generalizing to other frameworks (e.g., Cirq).

B – Threats to Internal Validity: The main threat to internal validity lies in the complexity of the underlying tools leveraged to build QMutPy as well as the ones supporting our experimental infrastructure. To mitigate this threat the authors have peer-reviewed the code before making the changes final.

C – Threats to Construct Validity: The parameters we used for drawing our conclusions may not be sufficient. In particular, by default, MutPy (and as a consequence QMutPy) runs a test case t on a mutant m for 5 times the time t takes to run on the non-mutated version. Increasing this number may lead to different results (i.e., less timeouts).

4.5 Summary

In this Chapter we defined the criteria for selecting QPs from QISKit. We detailed the experimental setup used to perform our experiments. We specified the metrics used and collected in our experiments. We discussed the threats to the validity of our work.

In the next Chapter we will present the results obtained in our experiments answering the 5 proposed research questions.

5 Results

Contents

In this Chapter we present the results for our research questions following the methodology and protocol defined in Chapter 4. Our experiments are available for replication at `https://github.com/jose/qmutpy-experiments`.

5.1 RQ1: How does QMutPy perform at creating quantum mutants?

With this question we wanted to do a self-evaluation of our tool, and how it fared in terms of performance, i.e. if it would be a practical way to generate mutants and was not overly slow.

Figure 5.2 shows a distribution of the time (log scale) QMutPy takes to inject a mutant in the AST in red, and the time to create a mutated model, based on the mutated AST, in blue, for classic and quantum mutation operators. On the one hand, the time taken to remove or inject new nodes into the program's AST is higher on all quantum mutation operators (except QMD) than on classic mutation operators. The latter takes up to a maximum of 2.68s (Super Calling Deletion (SCD)) whereas the former takes up to 5.53s (QGD), 11.36s (QMI), 61.13s (QGR), and 75.04s (QGI). On the other hand, the time taken to create a mutated version, i.e., to convert the mutated AST back to Python code, is relatively small (less than 0.1s) for all classic and quantum mutation operators. According to the plot, there is no runtime difference between creating a mutated version with a classic mutation operator or a quantum mutation operator.

QMutPy takes up to 16x more time to generate quantum mutants than to generate classic mutants.

We hypothesize the following reasons to explain its performance while creating quantum operators:

(1) Mutation operators based on functions calls (i.e., calls to quantum gates). Our set of quantum mutation operators, conversely to the classic ones, are based on function calls (see Listings 3.1 to 3.5). Mutating a function is more complex than mutating, for example, a constant or a logical operator. It is worth noting that classic mutation operators that also modify function calls (e.g., SCD) are also more time consuming than operators that work at, e.g., logical operator level, as the Logical Operator Deletion (LOD).

(2) Search for quantum gates. Quantum mutation operators QGR, QGD, QGI, and QMI first visit all nodes of the AST and for each function call checks whether it is a call to a quantum gate. As the number of function calls in a program is typically high, we estimate that the consecutive checking is time consuming. Possible solutions to address this problem would be to create a new type of operation in the Python AST, analogous to logical or arithmetic operations, but specifically dedicated to quantum gates.

(3) Search for an equivalent gate. In QMutPy's current implementation, once a call to a quantum gate is found, quantum mutation operators QGI and QGR (the two most time-consuming operators) attempt

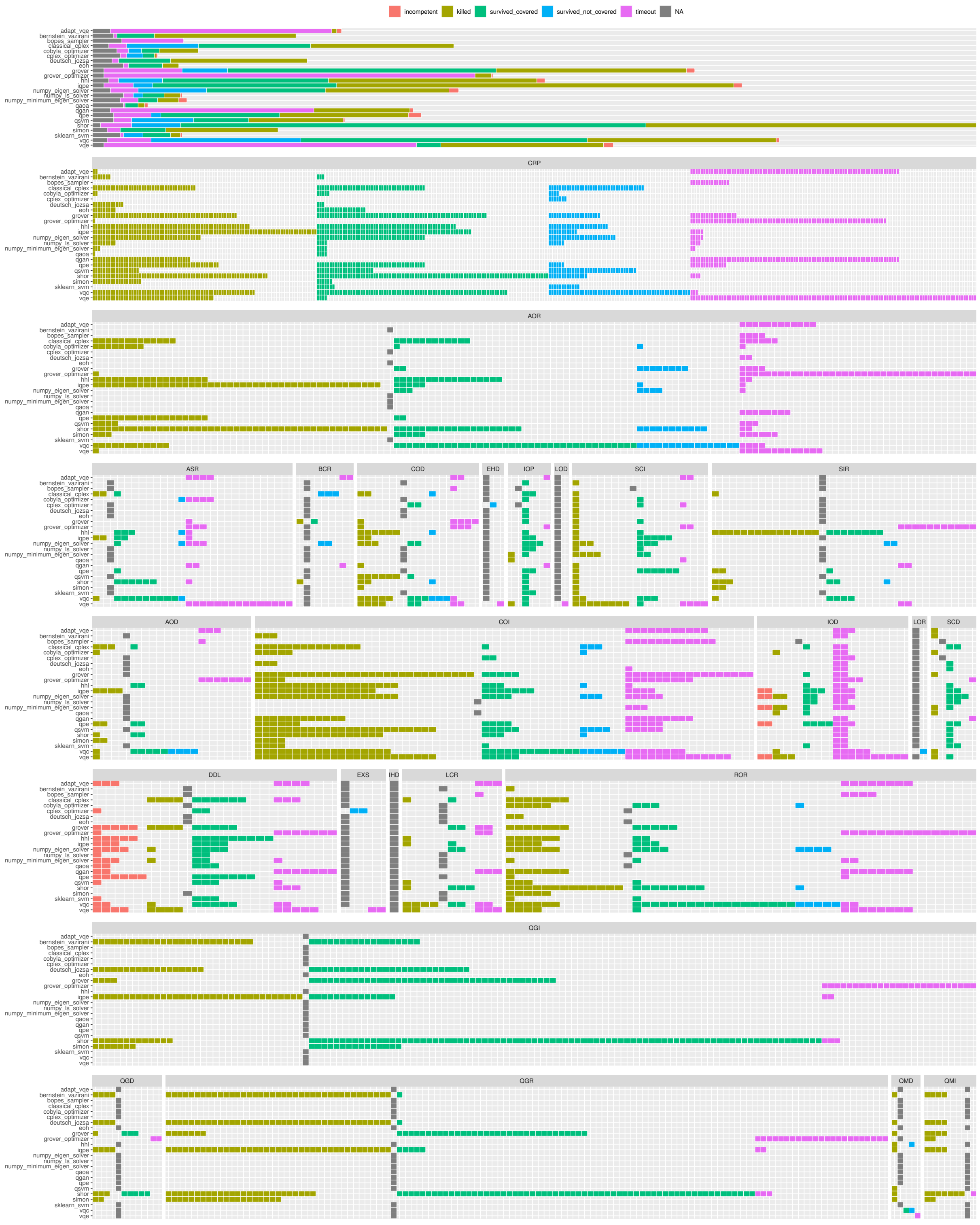


Figure 5.1: Detailed analysis and classification of all mutation operators performed in our study per algorithm and mutation operator.

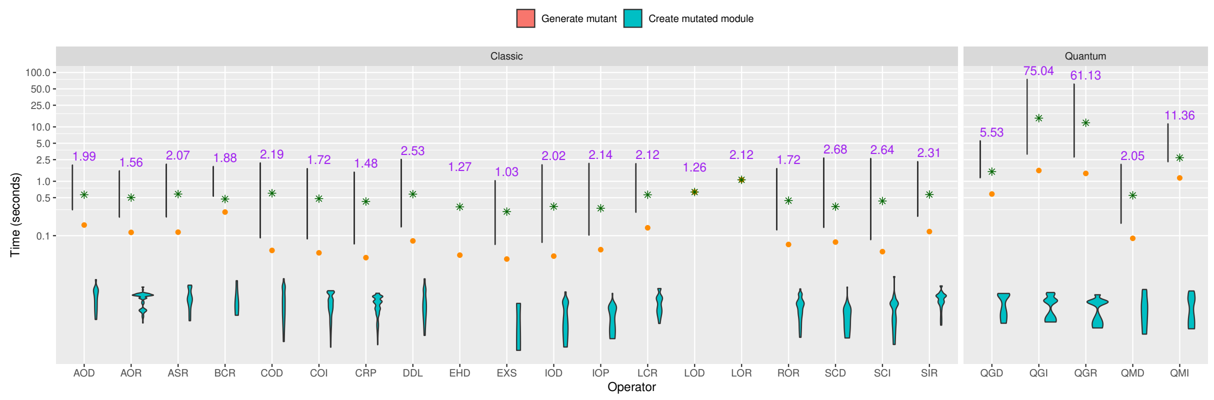


Figure 5.2: Distribution of the time required to inject a mutant and create a mutated target version. For each mutation operator, the purple text reports the maximum time of the 'Generate mutant' phase (the most expensive one), the green star reports the average time a mutation operator takes to generate a mutant and create a mutated module, and the orange circle reports the median time a mutation operator takes to generate a mutant and create a mutated module.

to find a correspondent equivalent gate in the set of available operators. This issue could be mitigated by pre-processing the set of equivalent gates.

(4) Modifying or adding nodes in the AST. Although quantum mutation operators QGR, QMD, and QGD only modify one node of the program's AST (see Listings 3.1, 3.2 and 3.5), QGI and QMI not only modify one node but also add another to the end of the AST (see Listings 3.3 and 3.4). We estimate this to increase the runtime of these operators.

The generation of quantum mutants is more complex to perform than classic mutants and therefore, as expected, more time consuming. Given the low number of quantum mutants we were able to generate (see RQ2), we argue that QMutPy's runtime at generating quantum mutants slightly affects the overall time spent on mutation testing.

5.2 RQ2: How many quantum mutants are generated by QMutPy?

To answer this research question, we analyze our data at two different levels: (i) program level, i.e., how many quantum mutants are generated per program (see Table 5.1), and (ii) mutation operator, i.e., how many mutants are generated by each quantum mutation operator (see Table 5.2). For this research questions, we focus on the columns "# Mutants" and "# Mutated LOC" on both tables. More details on generated mutants are given in Figure 5.1 which analyses and classifies all mutations performed per algorithm and mutation operator.

5.2.1 RQ2.1: How many quantum mutants are generated on each program?

As we can see in Table 5.1 (column "# Mutants"), QMutPy generates at least one quantum mutant for 11 out of the 24 QPs. This means that the remaining programs neither use quantum gates nor measurements. Thus, more quantum mutation operators should be investigated and developed to support those QPs.

On average, QMutPy generated 64 quantum mutants (e.g., 1 mutant for `vqe` and `qsvm` – 207 mutants for `shor`). Given that our set of mutation operators focus on specific function calls which might not occur as often as, e.g., classic arithmetic operations in a program, on average, QMutPy only mutated 4 LOC with an average of 13 mutants per line (see column "# Mutated LOC"). In contrast, at least one classic mutant was generated for all programs. 147 mutants on average (+83) and 64 LOC mutated (+60) with an average of 3 mutants per line (-10). Note that QPs are composed of more traditional programming blocks such as conditions, loops, and arithmetic operations than calls to the quantum API. Thus, and as there are many more LOC that can be mutated using classic mutation operators than using quantum mutation operators, it is expected to have fewer quantum mutants in a QP.

5.2.2 RQ2.2: How many mutants are generated by each quantum mutation operator?

As we can see in Table 5.2 (column "# Mutants"), on average, 140 mutants were generated by our set of quantum mutation operators. The quantum mutation operator that generated fewer mutants is QMD (12 mutants), whereas QGI (328 mutants) is the one generating more mutants. These results show that

- Quantum measurements are not that common in QPs (as only 12 measurements were mutated).
- Out of the 40 quantum gates with at least one syntactical-equivalent gate, 28 appear in the evaluated QPs.
- The insertion and replacement of quantum gates with their syntactical-equivalent ones represent 90% of all quantum mutants. This shows the importance of syntactically-equivalent gates, tailored for QPs, in mutation testing.

Worth noting that the average number of mutants generated by our quantum mutation operators is slightly below the number of mutants generated by classic mutation operators (140 vs. 186, which is highly dominated by CRP). As there are many more QPs that could be targeted by classic mutation operators (e.g., usage of constants) and many more classic operators (18 vs. our set of 5 quantum ones), it is expected that there are more classic mutants than quantum mutants. Nevertheless, the top-2 quantum mutation operators (i.e., QGI and QGR) generated more mutants than 15 out of the 18 classic mutation operators (i.e., AOD, AOR, ASR, BCR, COD, COI, CRP, DDL, EHD, EXS, IHD, IOD, IOP, LCR, LOD, LOR, ROR, SCD, SCI, and SIR), 628 vs. 517 mutants.

For 11 out of 24 QPs, QMutPy mutates 4 LOC and generates 14 different mutants per mutated line. In total, it generates a total of 696 mutants, 140 per mutation operator.

5.3 RQ3: How do test suites for QPs perform at killing quantum mutants?

The goal of this question is to analyze the quality and resilience of test suites designed to verify QPs. As mentioned before, the idiosyncrasies underlying QPs (e.g., superposition, entanglement) makes testing far from trivial. We argue that QMutPy's mutants can be used as benchmarks to assess the quality of tests designed to verify QPs.

Table 5.1 reports the results of performing mutation testing on the 24 QPs described in Table 4.1, whereas Table 5.2 summarizes the results per mutation operator.

As we can see in Table 5.2, out of the 696 mutants generated by our quantum mutation operators, 325 (46.70%) were killed by the programs' test suites. QGI, the mutation operator that generated more mutants, had a ratio of 102 killed mutants, followed by QGR with 170 killed mutants out of 300 generated. The non-killed mutants either survived to the test suites (307, 44.11%), were not even exercised by the test suites (2 QMD mutants, 0.29%), or resulted in a timeout (62, 8.91%). In comparison, out of the 3527 generated by classic mutation operators, 1264 (35.84%) were killed, 971 (27.53%) survived, 353 (10.01%) were not exercised by the test suites, and 885 (25.10%) timeout. Investigating timeout mutants might be something worth doing in the future since it might affect mutation score (e.g., 61.21% of classic mutants generated for `vqe` were timeouts).

These results show that the programs' test suites might have been designed to mainly verify the quantum aspect of each program as

- +10.86% more quantum mutants are killed than classic ones.
- Only 0.29% of all quantum mutants are not exercised the test suites, as opposed to 10.01% (+9.72%) of the classic mutants.

At program level, on average, the mutation score achieved by all programs' test suites was 57.69% if all mutants are considered (Equation (4.1)) and 62.23% if only mutants covered by the test suite are considered (Equation (4.2)). Recall that non-covered mutants would never be killed by any test. The mutation score achieved by each test suite ranged from 0% (`vqc` and `vqe`, more on this in Chapter 6) to 100% (`hhl` and `qsvm`). The mutation score achieved by all programs' test suites on classic mutants was 33.51% on average (considering all programs) and 41.61% if we only consider the same set of 11 programs for which quantum mutation operators were able to generated at least one mutant. That is, the

Quantum Program	# Mutants	#Mutated LOC	# Killed	# Survived	# Incompetent	# Timeout	% Score	Runtime
				Classic mutants				
adapt_vqe	142	64 (2.22)	3	0 / 0	3	136	7.31 / 7.31	1023.66
bernstein_vazirani	19	10 (1.90)	13	4 / 0	0	2	67.14 / 67.14	3.51
bopes_sampler	38	22 (1.73)	0	0 / 0	0	38	0.00 / 0.00	1119.35
classical_cplex	212	82 (2.59)	88	69 / 44	0	11	49.50 / 53.77	4.54
cobyla_optimizer	50	25 (2.00)	24	11 / 8	0	7	51.31 / 55.44	4.35
cplex_optimizer	23	14 (1.64)	1	7 / 10	1	4	4.17 / 4.17	1.96
deutsch_jozsa	27	11 (2.45)	18	5 / 0	0	4	47.50 / 47.50	4.21
eoh	34	14 (2.43)	10	21 / 0	0	3	22.02 / 22.02	36.61
grover	270	137 (1.97)	100	89 / 28	5	48	31.90 / 32.28	1031.75
grover_optimizer	187	73 (2.56)	8	0 / 0	1	178	6.65 / 6.65	329.12
hhl	266	121 (2.20)	127	102 / 26	5	6	39.14 / 41.04	1998.06
iqpe	287	93 (3.09)	162	94 / 12	5	14	43.31 / 43.73	81.05
numpy_eigen_solver	214	94 (2.28)	76	73 / 42	6	17	21.37 / 23.83	5.90
numpy_ls_solver	36	14 (2.57)	10	13 / 6	1	6	14.86 / 17.16	1.60
numpy_minimum_eigen_solver	41	19 (2.16)	13	12 / 0	5	11	35.42 / 35.42	2.28
qaoa	15	9 (1.67)	4	8 / 0	2	1	45.00 / 45.00	29.94
qgan	186	80 (2.33)	59	0 / 0	2	125	23.98 / 23.98	3779.19
qpe	189	68 (2.78)	79	73 / 6	8	23	29.59 / 29.80	82.51
qsvm	141	88 (1.60)	57	34 / 38	1	11	45.94 / 48.50	674.82
shor	331	123 (2.69)	153	136 / 30	0	12	40.78 / 44.99	1011.41
simon	58	21 (2.76)	37	13 / 0	0	8	63.40 / 63.40	23.94
sklearn_svm	38	20 (1.90)	6	17 / 12	1	2	28.75 / 28.75	1.25
vqc	411	181 (2.27)	116	175 / 91	2	27	27.25 / 30.52	8630.39
vqe	312	136 (2.29)	100	15 / 0	6	191	31.87 / 31.87	13419.82
Average	146.96	63.29 (2.25)	52.67	40.46 / 14.71	2.25	36.88	32.42 / 33.51	1387.55
				Quantum mutants				
adapt_vqe	0	—	—	—	—	—	—	—
bernstein_vazirani	93	5 (18.60)	74	19 / 0	0	0	91.32 / 91.32	7.29
bopes_sampler	0	—	—	—	—	—	—	—
classical_cplex	0	—	—	—	—	—	—	—
cobyla_optimizer	0	—	—	—	—	—	—	—
cplex_optimizer	0	—	—	—	—	—	—	—
deutsch_jozsa	93	5 (18.60)	66	27 / 0	0	0	87.68 / 87.68	7.70
eoh	0	—	—	—	—	—	—	—
grover	93	5 (18.60)	17	76 / 0	0	0	50.32 / 50.32	212.24
grover_optimizer	52	2 (26.00)	2	0 / 0	0	50	25.00 / 25.00	118.56
hhl	2	2 (1.00)	1	0 / 1	0	0	50.00 / 100.00	97.70
iqpe	105	5 (21.00)	82	19 / 0	0	4	90.56 / 90.56	31.07
numpy_eigen_solver	0	—	—	—	—	—	—	—
numpy_ls_solver	0	—	—	—	—	—	—	—
numpy_minimum_eigen_solver	0	—	—	—	—	—	—	—
qaoa	0	—	—	—	—	—	—	—
qgan	0	—	—	—	—	—	—	—
qpe	0	—	—	—	—	—	—	—
qsvm	1	1 (1.00)	1	0 / 0	0	0	100.00 / 100.00	47.85
shor	207	9 (23.00)	50	150 / 0	0	7	53.34 / 53.34	779.68
simon	47	3 (15.67)	32	15 / 0	0	0	86.36 / 86.36	13.45
sklearn_svm	0	—	—	—	—	—	—	—
vqc	2	2 (1.00)	0	1 / 1	0	0	0.00 / 0.00	170.21
vqe	1	1 (1.00)	0	0 / 0	0	1	0.00 / 0.00	144.21
Average	63.27	3.64 (13.22)	29.55	27.91 / 0.18	0.00	5.64	57.69 / 62.23	148.18

Table 5.1: Summary of our results per QP. Column "Quantum Program" lists the subjects used in our experiments. Column "# Mutants" reports the number of mutants per subject. Column "# Mutated LOC" reports the number of LOC with at least one mutant and also the ratio of mutants per line of code. Column "# Killed" reports the number of mutants killed by the subject's test suite. Column "# Survived" reports the number of mutants that survived and were exercised by the test suite, and the number of mutants that survived and were not exercised by the test suite. Note that any buggy code or mutant that is not exercised by the test suite cannot be detected or killed. Column "# Incompetent" reports the number of mutants that were considered incompetent, e.g., mutants that make the source code uncompilable. Column "# Timeout" reports the number of mutants for which the subject's test suite ran out of time. Column "% Score" reports the mutation score considering all mutants killed and survived (but excluding incompetents), and also reports the mutation score considering all mutants killed by the test suite and all mutants that survived and were exercised by the test suite. Column "Runtime" reports the time, in minutes, QMutPy took to run on all mutants.

Operator	# Mutants	# Killed	# Survived	# Incompetent	# Timeout
Classic mutants					
AOD	42	15	12 / 4	0	11
AOR	421	169	105 / 41	0	106
ASR	67	5	23 / 4	0	35
BCR	11	2	1 / 5	0	3
COD	63	34	10 / 6	0	13
COI	397	221	53 / 19	0	104
CRP	1860	634	551 / 256	0	419
DDL	147	15	55 / 0	44	33
EHD	2	0	0 / 1	0	1
EXS	4	0	0 / 2	0	2
IHD	0	—	—	—	—
IOD	100	10	17 / 0	10	63
IOP	31	3	25 / 0	0	3
LCR	38	11	11 / 0	0	16
LOD	1	0	0 / 0	0	1
LOR	1	0	0 / 1	0	0
ROR	185	79	47 / 11	0	48
SCD	31	8	21 / 0	0	2
SCI	69	34	25 / 0	0	10
SIR	57	24	15 / 3	0	15
Average	185.63	66.53	51.11 / 18.58	2.84	46.58
Quantum mutants					
QGD	28	18	8 / 0	0	2
QGI	328	102	196 / 0	0	30
QGR	300	170	102 / 0	0	28
QMD	12	8	1 / 2	0	1
QMI	28	27	0 / 0	0	1
Average	139.20	65.00	61.40 / 0.40	0.00	12.40

Table 5.2: Summary of our results per mutation operator.

programs' test suites achieved a higher mutation score on quantum mutants than on classic mutants, +20.62% (62.23% vs. 41.61%). Hence, reinforcing the idea that the test suites may have been designed to mainly verify the quantum characteristics of each QP.

According to current benchmarks the mutation score obtained for these mutations is low and so are the number of test cases [51], usually a higher mutation score in correlation with a significant number of tests cases leads to less faults in the program.

Test suites for QPs achieved a low mutation score on quantum mutants (62.23%), although +28.72% higher than the mutation score achieved on classic mutants. The low number of test cases and mutation score for our set of programs point to poor test suite quality.

5.4 RQ4: How many test cases are required to kill a quantum mutant?

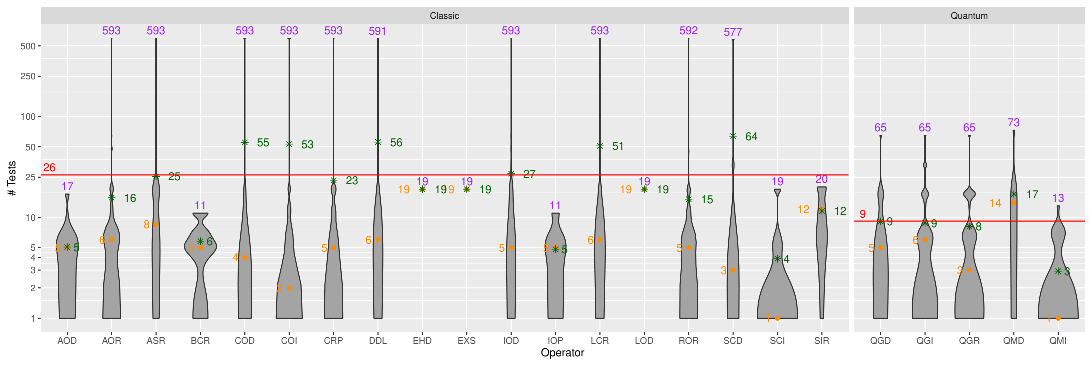
The goal of this question is to understand the effectiveness of current quantum test suites. Figure 5.3 shows the distribution of the number of tests required to kill or timeout each mutant per mutation operator and per QP. The red line represents the average number of tests needed to kill a mutant. The green star represents the mean and the orange circle represents the median of the number of tests needed to kill a mutant.

At the mutation operator level, the average number of tests needed to kill or timeout each quantum mutant is 9 (e.g., 1 test for QMI – 73 tests for QMD). The average number of tests needed to kill or timeout each classic mutant is 26, with 10 out of 18 classic mutation operators executing more than 500 tests.

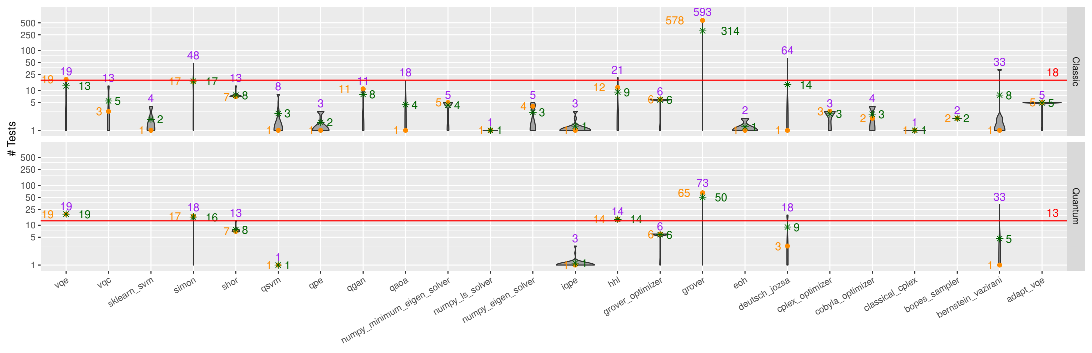
At program level, the average number of tests needed to kill or timeout a quantum mutant is 13 (e.g., 1 test for `bernstein_vazirani`, `iqpe`, and `qsvm`, and 73 for `grover`). Regarding classic mutants, the average number of tests needed to kill or timeout each classic mutant was 18 (considering all programs) or 64 if only the 10 programs for which at least one quantum mutant was generated and killed or timeout are considered.

As fewer tests are required to kill quantum mutants than to kill classic mutants, these results are in line with the assumption that these test suites primarily check quantum-related behavior.

On average, quantum mutants require -65% tests to be killed or timeout than classic mutants (9 vs. 26).



(a) Distribution of the number of tests that must be executed to kill a mutant per mutation operator.



(b) Distribution of the number of tests that must be executed to kill a mutant per program.

Figure 5.3: Distribution of the number of tests that must be executed to kill each mutant. The purple text reports the maximum number of tests needed to kill a mutant, the green star reports the median of the number of tests needed to kill a mutant, and the orange circle reports the average number of tests needed to kill a mutant. The red line represents the overall average number of tests needed to kill a mutant in classical and quantum mutation operations.

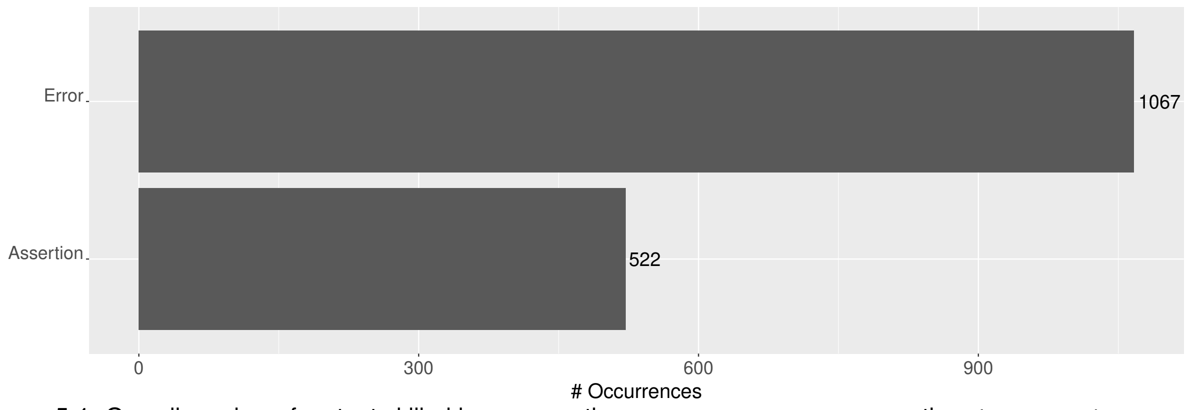


Figure 5.4: Overall number of mutants killed by an assertions or an error, e.g., an exception. In our experiments we found three types of errors thrown by the test suites. (1) Qiskit-related: `AquaError`, `QiskitOptimizationError`, `QiskitError`, and `CircuitError`. (2) Python: `NotImplementedError`, `IndexError`, `ValueError`, `AttributeError`, `IsADirectoryError`, `ZeroDivisionError`, `OverflowError`, `UnboundLocalError`, `RuntimeError`, `NameError`, and `KeyError`. (3) Third-party: `CplexSolverError`[a], `DQCPError`[b], `AxisError` and `LinAlgError`[c].

[a] https://www.ibm.com/docs/en/icos/12.8.0.0?topic=cplex

[b] https://www.cvxpy.org

[c] https://numpy.org

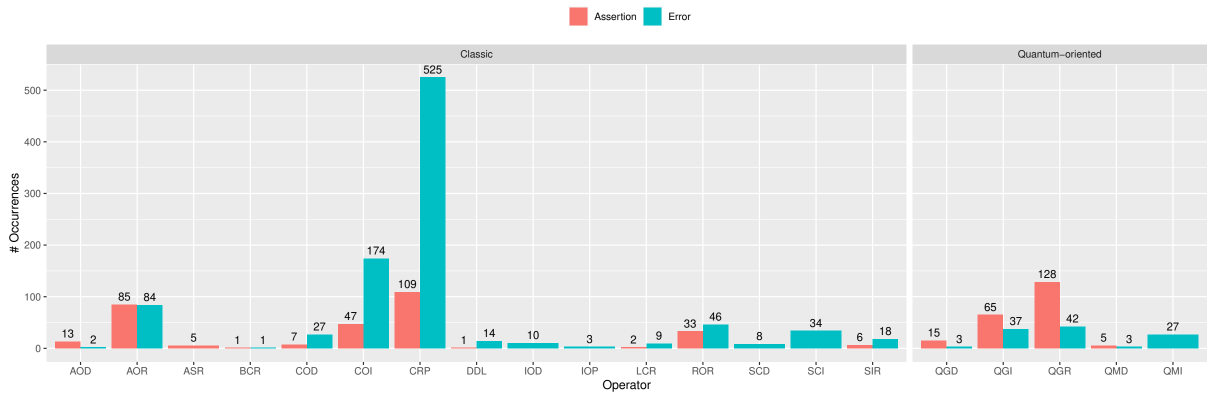


Figure 5.5: Number of mutants killed by an assertions or an error per mutation operator. In our experiments we found three types of errors thrown by the test suites. (1) Qiskit-related: `AquaError`, `QiskitOptimizationError`, `QiskitError`, and `CircuitError`. (2) Python: `NotImplementedError`, `IndexError`, `ValueError`, `AttributeError`, `IsADirectoryError`, `ZeroDivisionError`, `OverflowError`, `UnboundLocalError`, `RuntimeError`, `NameError`, and `KeyError`. (3) Third-party: `CplexSolverError`[a], `DQCPError`[b], `AxisError` and `LinAlgError`[c].

[a] https://www.ibm.com/docs/en/icos/12.8.0.0?topic=cplex

[b] https://www.cvxpy.org

[c] https://numpy.org

5.5 RQ5: How are quantum mutants killed?

With this question we wanted to analyze what kills quantum mutants (e.g. an assertion or an error). Figure 5.4 depicts the overall number of mutants killed by an assertion or an error and Figure 5.5 shows the same but per mutation operator.

We observe that, out of the 1589 killed mutants, two-thirds of mutants are killed by errors (1067) and the other one-third by test assertions (522). Overall, the majority of classic mutants are killed by errors. As already mentioned, we argue that QISKit test suites are mainly designed to check for the correct behavior of QPs. Therefore, they are less resilient to classic mutations and likely to be killed by errors instead of test assertions. This observation does not hold for quantum mutants.

We can see that two thirds of mutants are killed by errors and the other third by assertions. We separate errors in three types: QISKit related errors, Python errors, and third-party errors.

QGD, QGR, QGI, and QMD mutants are killed more often by test assertions than by errors. We also observed that QMI mutants, as expected, are killed by errors only. The reason is that QISKit does not have a fail-safe mechanism for inserting measurements. When a measurement operation is randomly inserted, the circuit may become unprocessable and an error is thrown.

Quantum mutants are mainly killed by test assertions (with the exception of QMI mutants). Classic mutants, on the other hand, are mainly killed by errors.

5.6 Summary

In this Chapter we demonstrated that QMutPy took more time generating quantum mutants than classic mutants and argued that it slightly affected the overall time spent on mutation testing. We showed that more classic mutants were generated than quantum mutants and that, for quantum operators, QGI and QGR were by far the ones that generated the most mutants. We established that on average quantum mutation score was higher than classical mutation score, argued that this was most likely by design (i.e., the purpose of the test suites is testing QPs behaviour) and that the low number of test cases and mutation scores obtained pointed to poor test suite quality. We explained that quantum mutants required less tests to be killed than classic mutants. We demonstrated that quantum mutants are mainly killed by assertions whereas classic mutants are mainly killed by errors.

In the next Chapter we will propose improvements for coverage and assertions of quantum test suites and prove how they increase mutation score for QPs.

6

Improving Quantum Test Suites

Contents

The obtained results reported in Chapter 5 showed that QISKit's test suites performed poorly at killing classical and quantum mutants. For example, we observed that 150 out of the 207 quantum mutants generated for `shor` survived.

From a classical mutation testing point of view the low mutation scores obtained are due to poor test suite quality, this can be easily fixed by adding more tests and improving coverage. However, we realized that maybe the point of these test suites is not to test for traditional bugs but to test for quantum bugs. Therefore this may have been overlooked deliberately, nevertheless we argue that all software should be extensively tested (to a reasonable degree) in all circumstances, it is a software engineering good practice that should not be overlooked.

More importantly the poor performance in killing quantum mutants must be addressed.

Understanding QPs and how they work requires knowledge of quantum physics that most software testers do not possess or even want to possess since it is not their area of expertise. Therefore, for a software tester, it might be difficult to design tests for QPs.

In this Chapter we draw on two hypotheses to guide our discussion on how to improve QPs' test suites to kill more quantum mutants:

h_1 The low mutation score achieved by each test suite is due to their low coverage.

h_2 The low mutation score achieved by each test suite is due to their low number of assertions.

6.1 Improving coverage

Figure 6.1 shows the relation between coverage and mutation score overall and for each mutation operator. We can see that QPs with higher coverage tend to have higher mutation scores; `bernstein_vazirani`, `simon` and `deutsch_jozsa` are three of the QPs with the highest coverage and mutation score. On the other hand `cplex_optimizer`, `adapt_vqe` and `bopes_sampler` are the QPs with lowest coverage and mutation score. Thus, with the first hypothesis we aim to investigate whether increasing the coverage of QPs, e.g., covering mutated LOC that are not exercised by the program's test suite, leads to a higher mutation score.

Table 5.1 shows that two QPs, `hhl` and `vqc`, have one mutation, generated with the QMD operator, done in uncovered methods (Listing 6.1 and Listing 6.3); `construct_circuit` and `get_optimal_vector` respectively. We extended their test suites [1] [2] (Listing 6.2 and Listing 6.4) to cover these methods and added a more specific test assertion to each test. The assertions created verify that the number of combination of qubits measurements is correct, which it would not be if no measurement was performed. We then ran the tests to confirm that all tests were still passing.

[1] https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/test/aqua/test_hhl.py
[2] https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/test/aqua/test_vqc.py

By rerunning the mutation analysis using the augmented test suites, we verified that our hypothesis holds. In both QPs, the mutants that survived our initial mutation analysis were now killed by the augmented test suites. Consequently, the previous mutation score calculated in Table 5.1 for `hhl`'s was increased from 50% to 100% (coverage increased from 86.55% to 89.16%), and `vqc`'s was increased from 0% to 50% (coverage increased from 93.26% to 94.43%). We, therefore, accept h_1 hypothesis and conclude that mutation score of test suites for QPs increases with coverage.

Listing 6.1: Mutant not exercised by `hhl`'s original test suite and therefore not killed.

```
  def construct_circuit(self, measurement: bool = False) -> QuantumCircuit:
      ...
      # Measurement of the ancilla qubit
      if measurement:
          c = ClassicalRegister(1)
          qc.add_register(c)
-         qc.measure(s, c)
+         pass
          self._success_bit = c

      self._io_register = q
```

Listing 6.2: Augmented `hhl`'s test suite.

```
  @data([0, 1], [1, 0], [1, 0.1], [1, 1], [1, 10])
  def test_hhl_diagonal(self, vector):
      ...
      self.log.debug('fidelity HHL to algebraic: %s', fidelity)
      self.log.debug('probability of result: %s', hhl_result.probability_result)
+     qc = algo.construct_circuit(True)
+     result = execute(qc, backend = BasicAer.get_backend('qasm_simulator'), shots = 1000).result()
+     counts = result.get_counts()
+     self.assertTrue(len(counts) == 2)
```

Listing 6.3: Mutant not exercised by `vqc`'s original test suite and therefore not killed.

```
  def get_optimal_vector(self):
      ...
      else:
          c = ClassicalRegister(qc.width(), name='c')
          q = find_regs_by_name(qc, 'q')
          qc.add_register(c)
          qc.barrier(q)
-         qc.measure(q, c)
```

```
+         pass
          ret = self._quantum_instance.execute(qc)
          self._ret['min_vector'] = ret.get_counts(qc)
      return self._ret['min_vector']
```

Listing 6.4: Augmented `vqc`'s test suite.

```
  def test_minibatching_gradient_free(self):
      ...
      self.log.debug(result['testing_accuracy'])
      self.assertAlmostEqual(result['testing_accuracy'], 0.3333333333333333)
+     vector = vqc.get_optimal_vector()
+     self.assertTrue(len(vector) == 4)
```

6.2 Improving test assertions

From Chapter 2 we know of the probabilistic nature of QPs. Suppose a quantum circuit with two qubits. When read, these qubits could either be 00, 01, 10, and 11. Suppose that the correct behavior is to observe two measurement values, 00 with 25% probability and 11 with 75%. If, instead, we observe survived mutants with four measurement values, i.e. 00, 01, 10, and 11 with some probability, then we would have a false negative since the mutants should have been killed.

We argue that asserting the number of measurements in the test suites is necessary to avoid these false negatives — hence, improving the mutation score. To verify this intuition, we augmented `shor`'s test suite[3] (the QP with the most generated quantum mutants, see Table 5.1) with additional test assertions. The added assertions check the correctness of the number of obtained measurement values.

Listing 6.5: Augmented `shor`'s test suite with four additional assertions.

```
  def test_shor_factoring(self, n_v, backend, factors):
      ...
      result_dict = shor.run(QuantumInstance(BasicAer.get_backend(backend), shots=1000))
      self.assertListEqual(result_dict['factors'][0], factors)
      self.assertTrue(result_dict["total_counts"] >= result_dict["successful_counts"])
+     self.assertTrue(result_dict["total_counts"] >= 55)
+     self.assertTrue(result_dict["total_counts"] <= 75)
+     self.assertTrue(result_dict["successful_counts"] >= 10)
+     self.assertTrue(result_dict["successful_counts"] <= 25)
```

[3] https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/test/aqua/test_shor.py

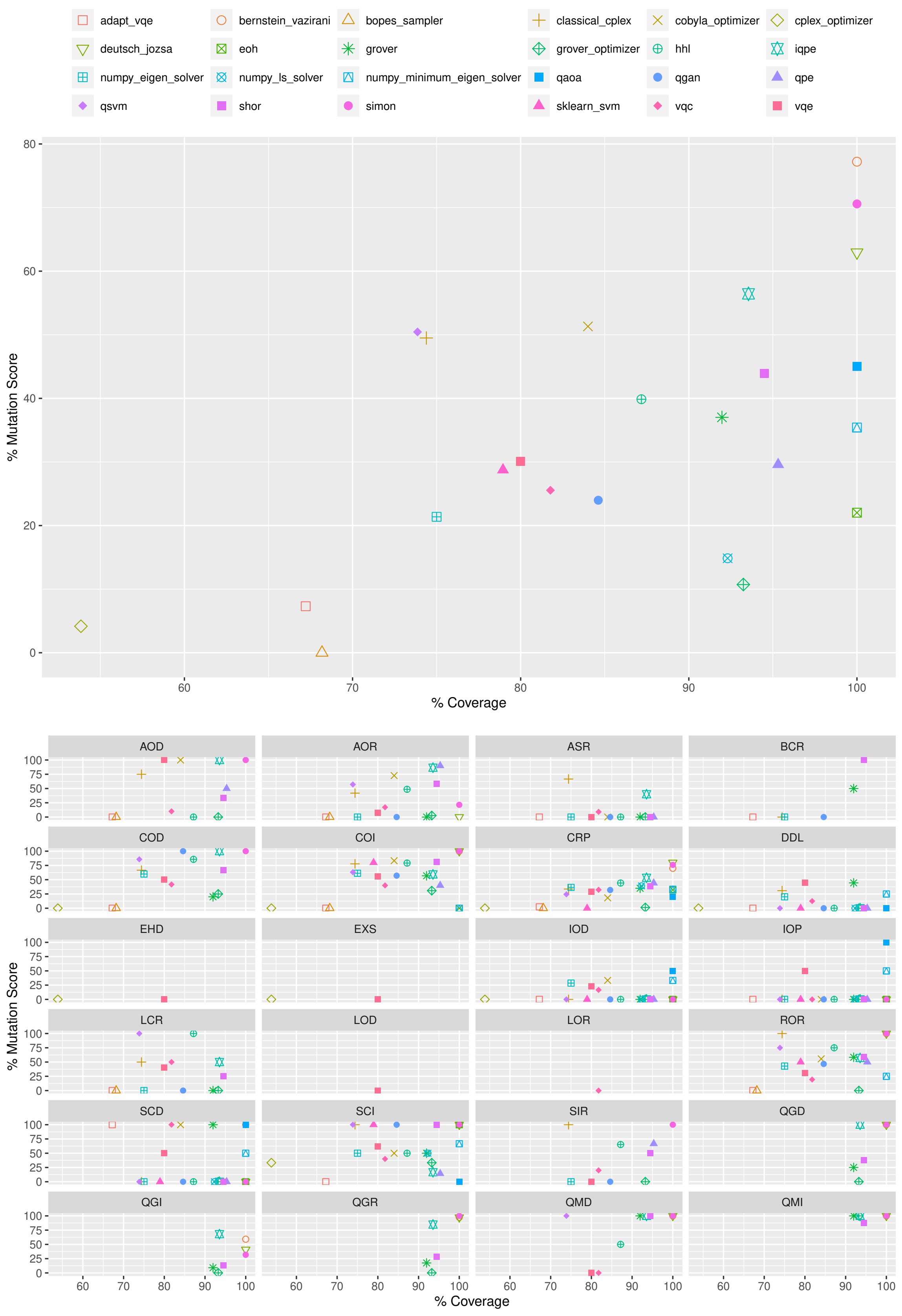


Figure 6.1: % Mutation score vs. % Coverage.

Similarly to h_1, we first ran `shor`'s tests to confirm they were still passing and then re-ran the mutation analysis using the augmented test suite to verify that h_2 holds. Mutation score achieved by the `shor`'s original test suite was 53.34% (50 mutants killed and 150 survived out of 207). The augmented test suite achieved a mutation score of 72.81% (109 mutants killed and 91 survived). In detail, the augmented test suite killed 6 out of 8 QGD mutants (+3 than original test suite), 32 out of 99 QGI mutants (+19), 63 out of 91 QGR mutants (+37), and the same QMD and QMI mutants (1 out of 1 and 7 out of 8, respectively) as the original test suite.

We, therefore, accept h_2 hypothesis and concluded that mutation score of test suites for QPs can be increased by adding more test assertions to test cases.

6.3 Summary

In this Chapter we showed that there seems to be a correlation between coverage and mutation score for our set of QPs. We proposed improvements to two QPs' test suite coverage and one QP test assertions and proved that they increased mutation scores.

In the next Chapter we will discuss other published work related to quantum mutation.

7

Related Work

	QMutPy	MTQC [5]	Muskit [6]
Open-Source	✔	✔	✔
Quantum languages	QISKit	QISKit, Q#	QISKit
Mutation operators	QGD, QGI, QGR, QMD, QMI	QGR[1]	AG, RemG, RepG
Test framework	unittest, pytest	custom	custom
Report	yaml, html	GUI	textual / GUI
Fully automated	✔	✖	✔

Table 7.1: QMutPy vs. MTQC [5] vs. Muskit [6].

In this Chapter we discuss other published work related to quantum mutation.

In the work of Liu et al. [52] the authors show that quantum mutation can be useful to ascertain the correct behavior of QPs. In particular, they propose a compiler technique applicable to QPs in order to simplify them and reduce their execution time while keeping their correctness. The authors found that their stochastic optimization technique was able to simplify many QPs by reducing the number of gates and steps of quantum circuits. This is a significant finding since they were able to provide better implementations of QPs than those provided manually by experts. The proposed stochastic optimization is a search-based technique that generates mutants for a QP and stochastically accepts or rejects that mutant. This technique aims at reducing QPs' runtime while keeping their correctness. They propose 6 mutation operators: insert an operation, remove an operation, swap two operations, replace the gate in an operation, replace qubits in an operation and replace an operation. QGR, QGI and QGD were based on their mutation operators.

Regarding quantum mutation tools, to the best of our knowledge MTQC [5] and Muskit [6] are the only two — preliminary — works that were published before. Next, we describe these two tools and draw a comparison with QMutPy (Table 7.1).

MTQC is a Java-based quantum mutation testing tool that uses a Graphic User Interface (GUI) to perform mutations on either QISKit or Q# QPs. MTQC performs primitive, custom mutations, albeit we can say that the mutation operation it performs is similar to the one QGR performs. However, the concept of equivalent gates is not defined and the gate swaps performed are a subset of our set of equivalent gates. They implement 52 isolated operations, each replacing a gate by another. Compared to MTQC, QMutPy's usability is preferable since it does not require the use of a GUI to do mutations and is fully automated. Furthermore it has a larger set o mutation operators, including classic ones, and is used by the testing community.

Muskit [6] is a Python mutation tool that is provided as a command line interface, a GUI, and a web application. Researchers and software developers might prefer the command line version (to, e.g., run empirical studies, as the one described in this work, in parallel), while for quantum enthusiast the GUI or web app may be more appealing. Muskit supports 19 QISKit gates and can perform three quantum mutation operations: add, remove, and replace gate. These are similar to our QGI, QGD, and QGR mutation operators, respectively. QMutPy, on the other hand, supports 40 gates (+21) and two additional mutation operators. Although Muskit has also been tailored for QISKit programs, it cannot be used out-of-the-box on, e.g., the 24 QPs evaluated in our empirical study. Muskit either uses a manually-written test suite or automatically generates a new suite [50]. Note that both test suites are sequences of test inputs and not complex sequence of code statements (e.g., calls to constructors to instantiate objects, method calls on these objects) as the ones used in our study which are required to test the 24 QPs. Furthermore, Muskit's test analyzer requires a program specification to determine whether a mutant has been killed by a test case. No specification is available for the 24 QPs considered in our study and writing one would require expertise on QISKit and on quantum computing.

[1]MTQC does not implement QGR, it has 52 isolated operations, each replacing one gate by another (e.g. operation Gate-CCX_GateCX)

8 Conclusion

Contents

8.1 Conclusions

In our thesis we started by giving an overview of what quantum computing is; we defined what a qubit is and how it differs from the classic bit, we demonstrated special quantum properties, we explained how operations are performed on qubits (i.e. with quantum gates) and how a QP works.

Due to special quantum properties (e.g. superposition, entanglement) inspecting qubits is forbidden. This makes testing QPs difficult and forbids the use of many classic testing techniques. New testing techniques must be explored and employed to test them. We justify that mutation testing can bypass some of the QPs' limitations test-wise and proposed QMutPy, an extension for MutPy, a famous mutation tool. QMutPy can mutate QPs for QISKit, the IBM quantum framework. Apart from classic mutation operators, QMutPy possesses 5 new quantum mutation operators: QGD, QGI, QGR, QMD and QMI.

To demonstrate the effectiveness of QMutPy, we conducted an empirical study with 24 real QPs selected from QISKit where we highlight the metrics we collected and our setup for running our experiment. We proposed to answer five research questions. How did our tool perform at creating quantum mutants? How many mutants did our tool generate? How do quantum test suites for QPs perform at killing quantum mutants? How many test cases are required to kill a quantum mutant? How are quantum mutants killed?

We found that our tool performed slightly slower in generating quantum mutants than it did generating classic mutants. However, we argued that the time discrepancies were of no significant impact. With our collected results (i.e. mutation score, code coverage, number of test cases) for each QP we observed several issues that may lead to future failures — non-optimal code coverage; low mutation scores; minimal number of test cases. We found that quantum mutants required less cases to be killed than classic mutants and argued that this is likely due to the objective of the designed test suites — checking for the QPs' behavior. This is reinforced by what we found with our last research question: that classical mutations are mainly killed by errors and quantum mutations are mainly killed by assertions.

As a consequence of our observations, we draw on two potential ways to improve test suites: coverage and assertion improvements. We showed how both improvements can increase mutation scores significantly for the QPs considered in our study[1].

Finally, we compared our tool with other quantum mutation tools currently available, highlighting their differences, advantages and disadvantages.

8.2 Future Work

Some of our subjects did not generate quantum mutants. For future work we plan to add new mutation operators to QMutPy, to possibly solve this problem. Furthermore, we plan on extending our current

[1]We are currently discussing with the IBM QISKit developers how to integrate our findings into their codebase.

set of syntactically-equivalent quantum gates to keep up with QISKit's latest releases. Also, we plan to extend QMutPy to other quantum frameworks (e.g., Cirq), and evaluate their test suites.

Automatically generating test suites for quantum programs [22, 23, 53, 21] is also a possibility, as we could use QMutPy to assert the effectiveness of the generated test suites.

It would also be interesting to try and run our mutation analysis with real quantum computers, instead of simulators, and check for potential differences.

Offering our mutation tool in a Continuous Integration / Continuous Delivery (CI/CD) format would be beneficial to anyone who would want to implement new quantum testing mechanisms and would broaden QMutPy's public reach.

Bibliography

[1] K. Hałas, "MutPy: A Mutation Testing Tool for Python 3.x Source Code," https://github.com/mutpy/mutpy, 03 2011, accessed: 2021-01-18.

[2] A. Hovmöller, "Mutmut: a Python mutation testing system," https://github.com/boxed/mutmut, 11 2016, accessed: 2021-01-18.

[3] A. Bingham, "Cosmic ray: mutation testing for python," https://github.com/sixty-north/cosmic-ray.

[4] E. Kepner, "mutatest: Python mutation testing," https://github.com/EvanKepner/mutatest.

[5] J. Pellejero, "MTQC: Mutation Testing for Quantum Computing," https://javpelle.github.io/MTQC, 06 2020, accessed: 2021-01-18.

[6] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue, "Muskit: A Mutation Analysis Tool for Quantum Software Testing," Tech. Rep., 2021.

[7] A. Steane, "Quantum computing," *Reports on Progress in Physics*, vol. 61, no. 2, p. 117, 1998.

[8] N. S. Yanofsky and M. A. Mannucci, *Quantum computing for computer scientists*. Cambridge University Press, 2008.

[9] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[10] J. Preskill, "Quantum Computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, Aug. 2018. [Online]. Available: https://doi.org/10.22331/q-2018-08-06-79

[11] J. Zhao, "Quantum Software Engineering: Landscapes and Horizons," 2020.

[12] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. USA: Cambridge University Press, 2016.

[13] G. Fraser and J. M. Rojas, *Software Testing*. Cham: Springer International Publishing, 2019, pp. 123–192. [Online]. Available: https://doi.org/10.1007/978-3-030-00262-6_4

[14] Y. Huang and M. Martonosi, “Qdb: from quantum algorithms towards correct quantum programs,” *arXiv preprint arXiv:1811.05447*, 2018.

[15] N. Juristo, A. M. Moreno, and W. Strigel, “Guest editors’ introduction: Software testing practices in industry,” *IEEE software*, vol. 23, no. 4, pp. 19–21, 2006.

[16] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, “An industrial evaluation of unit test generation: Finding real faults in a financial application,” in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP ’17. IEEE Press, 2017, p. 263–272. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.27

[17] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, “Effective regression test case selection: A systematic literature review,” *ACM Comput. Surv.*, vol. 50, no. 2, May 2017.

[18] A. Gambi, M. Mueller, and G. Fraser, “Automatically testing self-driving cars with search-based procedural content generation,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 318–328. [Online]. Available: https://doi.org/10.1145/3293882.3330566

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE ’07. USA: IEEE Computer Society, 2007, p. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[20] H. Hemmati, A. Arcuri, and L. Briand, “Achieving scalable model-based testing through test case diversity,” vol. 22, no. 1, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2430536.2430540

[21] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, “Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428218

[22] J. Wang, M. Gao, Y. Jiang, J. Lou, Y. Gao, D. Zhang, and J. Sun, “QuanFuzz: Fuzz Testing of Quantum Program,” 2018.

[23] S. Honarvar, M. R. Mousavi, and R. Nagarajan, “Property-Based Testing of Quantum Programs in Q#,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 430–435. [Online]. Available: https://doi.org/10.1145/3387940.3391459

[24] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[25] G. Petrovi'c, M. Ivankovi'c, G. Fraser, and R. Just, “Does mutation testing improve testing practices?” *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 910–921, 2021.

[26] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, “What it would take to use mutation testing in industry–a study at facebook,” 2021.

[27] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just, “Practical mutation testing at scale: A view from google,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[28] G. Petrović and M. Ivanković, “State of mutation testing at google,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 163–171. [Online]. Available: https://doi.org/10.1145/3183519.3183521

[29] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are Mutants a Valid Substitute for Real Faults in Software Testing?” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 654–665. [Online]. Available: https://doi.org/10.1145/2635868.2635929

[30] M. Fingerhuth, T. Babej, and P. Wittek, “Open source software in quantum computing,” *PLOS ONE*, vol. 13, no. 12, pp. 1–28, 12 2018. [Online]. Available: https://doi.org/10.1371/journal.pone.0208561

[31] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C.-F. Chen, J. M. Chow, A. D. Córcoles-Gonzales, A. J. Cross, A. Cross, J. Cruz-Benito, C. Culver, S. D. L. P. González, E. D. L. Torre, D. Ding, E. Dumitrescu, I. Duran, P. Eendebak, M. Everitt, I. F. Sertage, A. Frisch, A. Fuhrer, J. Gambetta, B. G. Gago, J. Gomez-Mosquera, D. Greenberg, I. Hamamura, V. Havlicek, J. Hellmers, Łukasz Herok, H. Horii, S. Hu, T. Imamichi, T. Itoko, A. Javadi-Abhari, N. Kanazawa, A. Karazeev, K. Krsulich, P. Liu, Y. Luh, Y. Maeng, M. Marques, F. J. Martín-Fernández, D. T. McClure, D. McKay, S. Meesala, A. Mezzacapo, N. Moll, D. M. Rodríguez, G. Nannicini, P. Nation, P. Ollitrault, L. J. O’Riordan, H. Paik, J. Pérez, A. Phan, M. Pistoia, V. Prutyanov, M. Reuter, J. Rice, A. R. Davila, R. H. P. Rudy, M. Ryu, N. Sathaye, C. Schnabel, E. Schoute, K. Setia, Y. Shi, A. Silva, Y. Siraichi, S. Sivarajah, J. A. Smolin, M. Soeken, H. Takahashi, I. Tavernelli, C. Taylor, P. Taylour, K. Trabing, M. Treinish, W. Turner, D. Vogt-Lee, C. Vuillot, J. A. Wildstrom, J. Wilson, E. Winston, C. Wood, S. Wood, S. Wörner, I. Y. Akhalwaya, and C. Zoufal, “Qiskit: An Open-source Framework for Quantum Computing,” Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2562111

[32] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017.

[33] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999. [Online]. Available: https://doi.org/10.1137/S0036144598347011

[34] L. K. Grover, "A Fast Quantum Mechanical Algorithm for Database Search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 212–219. [Online]. Available: https://doi.org/10.1145/237814.237866

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

[36] D. Bruß, "Characterizing entanglement," *Journal of Mathematical Physics*, vol. 43, no. 9, pp. 4237–4251, 2002. [Online]. Available: https://doi.org/10.1063/1.1494474

[37] R. Horodecki, P. Horodecki, M. Horodecki, and K. Horodecki, "Quantum entanglement," *Rev. Mod. Phys.*, vol. 81, pp. 865–942, Jun 2009. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.81.865

[38] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper," *ACM SIGPLAN Notices*, vol. 48, no. 6, p. 333, Jun 2013. [Online]. Available: http://dx.doi.org/10.1145/2499370.2462177

[39] K. M. Svore, A. V. Aho, A. W. Cross, I. Chuang, and I. L. Markov, "A layered software architecture for quantum computing design tools," *Computer*, vol. 39, no. 1, pp. 74–83, 2006.

[40] M. Fingerhuth, T. Babej, and P. Wittek, "Open source software in quantum computing," *PloS one*, vol. 13, no. 12, p. e0208561, 2018.

[41] D. Sych and G. Leuchs, "A complete basis of generalized bell states," *New Journal of Physics*, vol. 11, no. 1, p. 013006, jan 2009. [Online]. Available: https://doi.org/10.1088%2F1367-2630%2F11%2F1%2F013006

[42] G. García-Pérez, M. A. C. Rossi, and S. Maniscalco, "Ibm q experience as a versatile experimental testbed for simulating open quantum systems," 2019.

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

[44] J.-L. Brylinski and R. Brylinski, “Universal quantum gates,” in *Mathematics of quantum computation*. Chapman and Hall/CRC, 2002, pp. 117–134.

[45] A. W. Harrow, A. Hassidim, and S. Lloyd, “Quantum Algorithm for Linear Systems of Equations,” *Phys. Rev. Lett.*, vol. 103, p. 150502, Oct 2009. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.103.150502

[46] M. Ivanković, G. Petrović, R. Just, and G. Fraser, “Code coverage at google,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 955–963.

[47] O. Tange, “Gnu parallel - the command-line power tool,” *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb. 2011. [Online]. Available: http://www.gnu.org/s/parallel

[48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[49] J. Campos and A. 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)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2021, pp. 28–32. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/Q-SE52541.2021.00013

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

[51] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, “Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 537–548.

[52] P. Liu, S. Hu, M. Pistoia, C. R. Chen, and J. M. Gambetta, “Stochastic Optimization of Quantum Programs,” *Computer*, vol. 52, no. 6, pp. 58–67, 2019.

[53] S. Ali, P. Arcaini, X. Wang, and T. Yue, “Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs,” in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 13–23.