UNIVERSITY
OF OSLO

Master's thesis

QCross: Quantum Cross-Platform Testing

Applying differential and metamorphic testing techniques to test IBM Qiskit, Rigetti PyQuil, and Google Cirq

Arfat Salman

Informatics: Programming and System Architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2023

Arfat Salman

QCross: Quantum Cross-Platform Testing

Applying differential and metamorphic testing techniques to test IBM Qiskit, Rigetti PyQuil, and Google Cirq

Supervisors:
Dr. Shaukat Ali and Dr. Christoph Laaber

Abstract

The popularity of quantum computing has led to the creation of various Quantum Software Platforms (QSP), which include various components, such as a standalone or library-based quantum programming language, an optimizing compiler that translates high-level quantum circuit code into gate instructions, a quantum simulator that emulates these instructions on a classical device, and a software controller that sends analog signals to quantum hardware based on quantum circuits.

As their usage continues to increase, the underlying platforms are becoming more complex and capable; however, questions of robustness and correctness still remain. A buggy and non-robust platform could hinder adoption and potentially deter users from quantum computing altogether. For example, Qiskit[1], a popular platform, has 40% bug-labelled issues[2]. Since these platforms are foundational to the quantum computing revolution, it is crucial to test them for both robustness and cross-platform compatibility. Unfortunately, testing these platforms is challenging due to a lack of diverse quantum-specific testing techniques, limited availability of the same quantum programs using different platforms, and varying levels of inter-platform compatibility. Additionally, the oracle problem in testing poses a challenge, as there is a lack of specifications for the expected behavior of programs.

We present QCross, a cross-platform differential and metamorphic testing approach for testing quantum computing platforms. We build on top of MorphQ and attach a converter that translates quantum circuits from one platform to another. Furthermore, we also present a Python library, **bloqs**, that makes Qiskit quantum gates available in PyQuil, and Cirq. By evaluating the approach with 1500+ randomly-generated quantum programs on three platforms (Qiskit, PyQuil, and Cirq), we discovered several new real-world bugs in each platform. QCross expands the limited range of available testing techniques and aims to play an important role in developing a more dependable software stack for this rapidly growing field.

[1]https://github.com/Qiskit/qiskit-terra/issues

[2]Issues are a user-based reporting mechanism that tracks ideas, feedback, tasks, or bugs for a GitHub repository.

Contents

Acknowledgements

I am profoundly grateful to everyone who contributed to this master's thesis journey. First and foremost, I would like to extend my heartfelt appreciation to my supervisors, Dr. Shaukat Ali and Dr. Christoph Laaber. Upon our initial meeting concerning this project, I had doubts about my capacity to effectively engage with the subject matter. Nevertheless, they assuaged my apprehensions and expressed confidence in my ability to succeed. Throughout the thesis, they offered invaluable guidance, steadfast support, and persistent encouragement during our weekly meetings. Their patience, feedback, and expertise were indispensable to the completion of this thesis.

I am also deeply indebted to Simula Research Laboratory for furnishing me with an exceptional research environment, resources, and facilities. I am particularly appreciative of the eX3 super cluster, which facilitated the execution of intensive quantum computing simulations.

I owe an eternal debt of gratitude to my friends and family for their unwavering love, support, and encouragement. I am especially grateful to my friends Wahid and Anum, whose presence made a new country feel like home. And lastly, to Umme, whose unwavering belief in me served as a catalyst for my personal growth. Her steadfast presence in my life and her enduring faith in me has been a comforting constant. Their steadfast belief in me, even in the face of adversity, has served as a constant source of motivation and focus throughout this journey.

Finally, I would like to convey my appreciation to all the individuals who participated in my research, generously offering their time, expertise, and feedback.

Part I

Arrange

Chapter 1

Introduction

A Quantum Computer (QC) is a computational device capable of performing calculations using the principles of quantum physics. It is important to note that even traditional computers, often referred to as classical computers[1], adhere to the laws of physics. However, QCs exploit fundamental quantum-mechanical principles, such as quantum entanglement, to drive computation and achieve results that are unattainable for classical computers within the same time or space constraints. Essentially, it is a distinct type of computer [21]. An unattributed quote exemplifies this:

> We shouldn't be asking 'where do quantum speedups come from?' we should say 'all computers are quantum, [...]' and ask 'where do classical slowdowns come from?'

A common question that arises when learning about QCs is, *"Will they replace traditional computers?"*. The short answer is: *No*. A QC can be regarded as specialized hardware designed to solve a narrow set of problems that pose challenges for classical computers. This concept is akin to a Graphics Processing Unit (GPU), which assists and accelerates the rendering of game graphics and images. In this context, the term "Quantum *Computer*" is somewhat misleading, and "Quantum Co-processor" [21] or Quantum Processing Unit (QPU) might be more appropriate.

Initially proposed in the 1980s by Richard Feynman and Yuri Manin [35], a surge of research activity has since transformed quantum computers from science fiction to reality. Within two decades of its inception, a functional two quantum bit (qubit) quantum computer was constructed in the late 90s, which demonstrated an experimental solution to Deutsch's problem [12]. Today, quantum computers containing 100+ qubits exist, such as in IBM Quantum Experience program.

Owing to their unique computational paradigm, quantum computers can solve certain problems exponentially faster and more efficiently than their classical counterparts. For instance, the well-known Shor's algorithm computes prime factors of a number in polynomial time. Currently, the fastest classical prime factorization algorithm is the **General number field sieve**, which exhibits non-polynomial time complexity. Another

[1]Following the same distinction as Quantum and Classical Physics

Company	Platform Name	Release Date
IBM	Qiskit	March 2017
Rigetti	Forest	June 2017
Microsoft	Q#	December 2017
Google	Cirq	July 2018
Quantinuum	TKET	-

Table 1.1: Partial List of Quantum Platforms

example is Grover's algorithm, which searches for an item in an unsorted database with a $\sqrt{N}$ speedup. Additional examples of potential impacts from quantum computers in fields as diverse as finance and chemistry are presented in [16].

Numerous industries are increasingly recognizing the potential of quantum computing and its capacity to solve problems that are intractable for classical computers. The past decade has witnessed substantial corporate momentum in quantum computing. As of 2022, it is estimated that over 80 companies[2] are involved in QCs in some capacity, including prominent entities such as IBM, Microsoft, and Google. These companies have introduced QC platforms, alternatively referred to as Quantum Software Platforms (QSP) or Quantum Development Kits (QDK), which facilitate the design and execution of quantum programs or quantum circuits, either through emulation or on actual hardware. A partial list of quantum platforms is provided in Table 1.1. For additional options, an extensive list of open-source projects, numbering well over fifty, can be found at [5], and a compilation of quantum computer simulators is available in [26].

Although significant progress has been made over the years, publicly accessible and user-programmable quantum computers remain in their nascent stages and are quite costly. At the time of writing, a 5-qubit system from IBM is available for public use as part of the IBM Quantum Experience program. Amazon, through its Amazon Braket service, offers one complimentary hour of simulation time per month. Nonetheless, the lengthy waiting times and 5-qubit limitations, as in IBM's case, deter many users. Furthermore, quantum hardware is rapidly evolving. It is highly likely that improved hardware, featuring an increased number of qubits, enhanced error-correction capabilities, or a superior native quantum gate set, will emerge within a few years. Despite potential hardware advancements, the software necessary for interfacing with quantum computing technology is expected to remain consistent. Utilizing these new quantum computers is anticipated to be a straightforward process, necessitating only minor adjustments to a few lines of code while preserving the same existing syntax for generating and executing quantum circuits. For instance, in Qiskit, one could simply modify the backend's name (backend refers to the actual quantum hardware or simulator) when executing the circuit:

[2]https://thequantuminsider.com/2022/09/05/quantum-computing-companies-ultimate-list-for-2022/

```
execute( quantum_circuit , backend =... )
```

These quantum software platforms are employed by researchers and programmers alike to efficiently design, develop, execute, and troubleshoot their quantum programs, for research and commercial purposes as outlined above. Consequently, it is imperative that these platforms be free of any bugs or instabilities that could compromise the accuracy of their outputs.

1.1 Motivation

A cursory examination of popular platforms' GitHub repository pages reveals the prevalence of *issues* users encounter[3]. In Qiskit, 40% of the 3800 total issues are labeled as **bugs**. Similarly, more than 15% of Cirq issues bear the bug label while Microsoft's Q# quantum runtime has 30% bug-labeled issues. Empirical studies [32] have conducted comprehensive analyses of these issues, demonstrating that numerous bugs continue to afflict these platforms. Nevertheless, it is evident that these development kits are not without flaws, and testing them further may uncover additional shortcomings.

Moreover, guaranteeing the robustness of these platforms is crucial for the rapid adoption of quantum programs by a broader audience. Ensuring cross-platform compatibility, in this case for Qiskit, Cirq, and PyQuil, is also of great importance since a user should not be bound to a single platform. A quantum circuit should exhibit consistent behavior across multiple platforms, akin to a website being accessible on any browser or a program behaving similarly on an Intel CPU or Apple's M1 CPU (albeit with different compilations).

1.1.1 Challenges in Cross-Platform Testing

Cross-platform testing of **quantum software platforms** presents several challenges for various reasons. These include:

1. **C1:** The requirement for a substantial number of test quantum programs. As the field is still nascent, only a limited number of popular programs are frequently utilized. Furthermore, the same program is needed for all platforms under test to ensure cross-platform compatibility.

2. **C2**: Due to the platforms' varying levels of support and distinct APIs, it is difficult to test the same program without manually porting them from one platform to another.

3. **C3**: The presence of stochasticity. When the outputs of the same program on two different platforms differ, determining the extent to which the difference can be attributed to noise or pure randomness becomes challenging.

[3]at the time of writing

1.1.2 Challenges in testing quantum programs

Apart from the above-mentioned challenges in testing QC platforms, a single quantum program has its own set of challenges. Unlike testing of classical programs, quantum programs pose certain novel challenges:

- Quantum computation is fundamentally probabilistic and non-deterministic due to quantum indeterminacy. Therefore, a **single** known correct output cannot be compared with the output of a quantum circuit. For example, a superpositioned quantum state can either be 0 or 1 with 50% probability each. Running this program multiple times would result in a probability distribution (50% 1 outcomes and 50% 0 outcomes) and not the *same* output at every run. This is different from a classical program which produces a definite answer on every execution. Hence, equality comparisons of circuit outputs become impossible.

- Quantum computers today are noisy and fault-tolerant, that is, fully error-corrected QCs don't exist as of now. Therefore, if an observed probability distribution of qubits differs from the expected probability distribution, then what percentage of that is due to noise and/or natural randomness?

- The No-Cloning theorem [31] prohibits making a copy of an arbitrary unknown quantum state. Therefore, reading or asserting on intermediate quantum states is impossible.

- Simulating quantum programs on classical computers take an exponential amount of memory. For example, consider a system of electrons where electrons can be in any of, say 40 positions. The electrons, therefore may be in any of 2^{40} configurations. To store the quantum state of the electrons in a conventional computer memory would require in excess of 130 GB of memory[35]! For 45 positions, classical computers would require more than 4000 GB. Therefore, testing large quantum programs on simulators is currently impossible.

1.2 Existing Work in Quantum Platform Testing

Given its nascent stage, quantum software testing lags behind traditional software and compiler testing in terms of available software tools and research output. There appear to be only two prior studies in this area: QDiff [41] and MorphQ [33].

QDiff is a differential testing approach in which six existing well-known and pre-written source programs are modified to generate semantically equivalent but distinct programs. These are then executed and tested on various QSSes (Qiskit, Cirq, PyQuil) and IBM's quantum hardware.

MorphQ is a metamorphic testing framework that comprises (i) a program generator that creates a large and diverse set of valid (i.e., non-crashing) quantum programs, and (ii) a set of program transformations that exploit quantum-specific metamorphic relationships to generate equivalent follow-up programs. The metamorphic relationship helps in addressing the Oracle problem of testing. That is, MorphQ does not require the specification of a test program. Since two programs are metaphorically related, they are expected to produce the same output even though the program structures differ from each other. Although QDiff begins with a small set of manually written programs, MorphQ enhances it by generating random programs, thereby improving the test coverage criteria [33].

We believe that both studies have potential shortcomings. As noted in [33], QDiff's non-random programs do not sufficiently stress the system. Moreover, they are less likely to utilize all available gates (or other features) in each of the platforms, as the programs were predefined and pre-written. Consequently, they may not test intricate corner cases that only occur with a (random but valid) combination of specific gates and/or features. MorphQ improved upon these aspects by generating random programs and establishing metamorphic relationships between them. However, MorphQ's limitation lies in generating only Qiskit programs, and it was evaluated solely on that specific platform.

QCross aims to address the limitations of both QDiff and MorphQ. We build upon and combine the differential and metamorphic testing techniques of MorphQ and QDiff in an effort to conduct cross-platform testing, as described further in the "Contributions of the thesis" section.

1.3 Research Question

We have identified these following research questions to focus on:

- **RQ1:** How many syntactically different but correct programs can be translated by QCross's converter?
- **RQ2:** What has QCross found via cross-platform testing of the widely-used QSSes? i.e., how many warnings and errors do QCross produce?
- **RQ3:** How does QCross compare to prior work on testing quantum computing platforms?
- **RQ4:** How useful is Bloqs?

1.4 Contribution of the thesis

The thesis builds on top of MorphQ and attaches a program converter that translates a MorphQ Qiskit circuit to either a Cirq or PyQuil circuit while maintaining the original metamorphic transformations of the source input. The converter can be extended to allow support for other platforms in the future. This covers challenge C1.

We also modify and extend the original 10 metamorphic relations with newer and different relations. For example, we add a QASM 3 (an intermediate representation of quantum programs) round-trip and QPY (a binary serialization format) round-trip for Qiskit. Moreover, as part of translation, we provide concrete techniques and implementation for converting every MorphQ metamorphic relation in all three platforms.

For the challenge C2, we presents an open-source library called *bloqs*. This library helps in achieving parity in a number of quantum gate operations supported by each platform. For example, Qiskit has 50+ named Quantum gates. Cirq and PyQuil have $\sim$25 named gates. We implement more than 20 Qiskit gates in both Cirq and PyQuil, for a total of 40+ specific implementation to achieve quantum gate parity. This is done using the platform's own ability to create a custom quantum gate.

In this thesis, quantum circuits are not executed on real hardware. Therefore, the noise part challenge C3 is not relevant. We only use a noise-less simulator provided by each platform. Hence, we don't have to deal with noise. However, any statistical difference (beyond and irrespective of noise) in outputs of the circuits need to be explained on a per-program basis since it can be purely stochastic, or it may point to some instabilities in the underlying platform(s).

1.4.1 Why not extend MorphQ?

QCross can be seen as an extension of MorphQ. There is, however, a real question of "Why not modify MorphQ itself"? That is, MorphQ can be extended to generate PyQuil and Cirq random programs. It can use the gate sets and features of these platforms to natively emit random Cirq or PyQuil circuits. The reason for having a translator that is separate from MorphQ are as follows:

- Since MorphQ was designed with Qiskit in mind, arguably, equal efforts would've been spent in generalising the MorphQ metamorphic relations so that they work with all three platforms. Trying to retro-fit other platforms within MoprhQ can be more challenging than implementing an external translator as we may find limitations in the structures of MorphQ.
- When generating a circuit within Cirq or PyQuil, we'd be limited by their gate sets or their specific features. For example, Cirq has 25 built-in gates. A random Cirq program would necessarily be limited to these 25 gates unless explicit care is taken to poly-fill extra gates.

Since the spirit of the thesis is to do cross-platform testing, a given Qiskit program should have the same behaviour (after creating an equivalent version) in Cirq or PyQuil. Similarly for other platforms. But that means that Qiskit features need to be translated as well. Currently, the **`bloqs`** library fills in the gaps of available gates in these platforms. Though, in the future, the platforms may converge on some common core set of features. At that time, import statements from **`bloqs`** can be replaced with the built-in gates, and the circuits should still behave the same.

1.4.2 Why choose only these platforms?

As mentioned above, there are many quantum platforms available. We chose Qiskit, Cirq, and PyQuil to test and not anything else for the following reasons:

1. **Liveliness**: Arguably, Qiskit is the most popular and feature-rich quantum software platform. Along with Google's Cirq and Rigetti's PyQuil, these platforms feel "alive". The notion of "liveliness" here refers to active development, quick patching of bugs, timely release of new versions, timely response to community members' issues, etc. These platforms also have huge financial support from their parent organizations which helps the platforms in having a dedicated team of developers.

2. **Python**: All these platforms have first-class support for Python-based SDK. This helps in having a uniform testing environment which can execute all three platforms' code. Also, we can leverage the huge library ecosystem of Python.

3. **Real Quantum Computer**: All three platforms have the ability to connect to a real quantum computer. The QC may be their own or they may have partnered with a hardware institution to allow for cloud-based access via their platform. Though not relevant in this thesis immediately, one can see the benefit of running the same program on real hardware as opposed to a simulator. This can extend the current testing infrastructure and integrate testing frameworks more tightly in the future.

4. **MorphQ**: Given that this thesis builds on top of MorphQ which was coded in Python for Qiskit, it made more sense to continue using the same stack.

Other respectable platforms that were considered but were not included are ProjectQ from ETH Zurich (no real possibility of connecting to hardware), Microsoft Q# (different language C#, and stack), Quantinuum pyTKET (released too recently).

1.4.3 Why not use an existing translator?

Upon searching, we were able to find two converters/translators for quantum circuits. Though they are feature-rich tools, they were not chosen for the reasons listed below.

- **QConvert by Quantum Programming Studio**: QConvert is able to convert programs written in QASM or QUIL (**Qu**antum **I**nstruction **L**angauge) to circuits of various other platforms. However, it cannot convert Qiskit programs. For QConvert to work, Qiskit circuits need to be exported in QASM format. Upon testing, we found that QConvert does not support a lot of the gates that QASM supports. For example, exporting this QASM 2.0 snippet

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
```

```
creg c[2];
dcx qr[2],qr[0];
```

to PyQuil 2.2 [4] produces the following warning:

```
# Export to PyQuil WARNING: unknown gate "dcx".
```

- **Quantinuum pytket**: pytket is a python module for interfacing with tket, a quantum computing toolkit and optimising compiler developed by Quantinuum. It comes with its own syntax but has support for conversion of circuits between platforms using pytket extensions[5]. Using pytket, a Qiskit circuit can be transformed into Cirq (or PyQuil) in the following manner:

  1. Create (or import) a Qiskit circuit
  2. Use the function **`qiskit_to_tk`** available in Qiskit extension to convert the circuit to pytket
  3. Use the function **`tk_to_cirq`** and feed it the circuit from the previous step to get a Cirq circuit.

  There are couple of problems with this approach as listed below.

  1. Translating circuits in this manner won't preserve all metamorphic relations. For example, the execution or simulation backend is not part of a circuit, hence, it can't be translated. But we have a metamorphic relation which enumerates between multiple backends of a circuit. Using pytket, this extra bit would have had to be added manually.
  2. Lack of support for all Qiskit gates[6]. This code snippet throws an error when executed

     ```
     from qiskit import (
         QuantumCircuit,
         ClassicalRegister,
         QuantumRegister
     )
     from qiskit.circuit.library.standard_gates import *
     from pytket.extensions.qiskit import qiskit_to_tk
     ```

[4] highest available PyQuil version at the time of writing

[5] extensions are separate python modules which allow pytket to interface with other quantum providers. Link https://cqcl.github.io/pytket-extensions/api/index.html

[6] executed with pytket-qiskit version 0.28.0, and GitHub issue: https://github.com/CQCL/pytket-qiskit/issues/88

```
qr = QuantumRegister(3, name='qr')
cr = ClassicalRegister(3, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')

qc.append(CSXGate(), qargs=[qr[1], qr[0]], cargs=[])

qiskit_to_tk(qc)
# KeyError: <class '..library.standard_gates.sx.CSXGate'>
```

3. Since the circuit transformation happens via pytket (Qiskit $\rightarrow$ TKET $\rightarrow$ Cirq), it can potentially mask certain platform issues within its own errors.

1.5 Structure of the thesis

The thesis is arranged following the AAA (Arrange, Act, Assert) pattern [7] which is a common way of writing tests. It makes sense that a thesis about testing should follow testing standards! The terms are defined as following:

1. **Arrange** all necessary preconditions and inputs. For our thesis, this constitutes "Part 1" which includes the following chapters: Introduction, Background, Literature Review, and Approach.
2. **Act** on the object or method under test. Essentially, "Act" refers to execution. Hence, it includes Part 2, the chapter on evaluation and experimentation.
3. **Assert** that the expected results have occurred. The final part includes these chapters: Discussion and Conclusion. They assert on the findings of the thesis.

1.6 Open Science

All code and artifacts related to QCross are provided at
`https://github.com/ArfatSalman/qcross`

[7] http://wiki.c2.com/?ArrangeActAssert

Chapter 2

Background

2.1 Quantum Computing

2.1.1 Quantum Bit

A classical computer operates on *classical* bits. A bit can be either 0 or 1, describing respectively the two states of the bit. A QC, on the other hand, operates on a **Quantum Bit** (qubit). Unlike a classical bit, a quantum bit can exist in more states than just exclusively 0 and 1. Mathematically, the state of a qubit is described as a vector in a two-dimensional complex vector space,

$$|\psi\rangle = \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.1}$$

where the α and β are complex numbers, called the *amplitudes*.[1] Using the *Dirac notation*, the $\left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$ can be written as $|0\rangle$ and $\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$ is written as $|1\rangle$[2]. Therefore, the equation (2.1) can be re-written as

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \tag{2.2}$$

The $|\alpha|^2$ denotes the probability of measuring the qubit as 0, and $|\beta|^2$ denotes the probability of measuring the qubit as 1. Due to the Normalization condition, which states that the probabilities must sum to one, $|\alpha|^2 + |\beta|^2 = 1$ (i.e., the sum of modulus squared of amplitudes should be 1)

The special states $|0\rangle$ and $|1\rangle$ are known as computational basis states or also called **standard basis**, and form an ortho-normal basis for this vector space. We can visualize these distinct states, $|0\rangle$ and $|1\rangle$, as the north and south poles of a sphere of radius unit 1 called the Bloch sphere, as shown in Figure 2.1. In fact, a qubit can be any point on the Bloch sphere [44].

Following the standard Physics convention, the x-axis comes out of the page, the y-axis points to the side, and the z-axis is oriented up. Then, since the Bloch sphere has

[1] $|\psi\rangle$ is conventionally used to refer to a generic quantum state.

[2] 0 and 1 enclosed between a vertical bar and an angle bracket called a *ket*

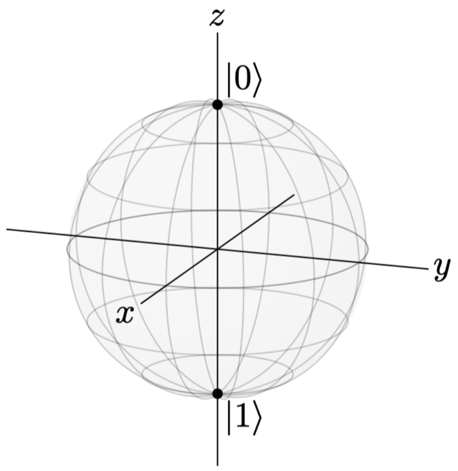


Figure 2.1: Bloch Sphere

radius of 1 unit, $|0\rangle$ corresponds to the (x, y, z) point $(0, 0, 1)$, and $|1\rangle$ corresponds to $(0, 0, -1)$. The $|0\rangle$ and $|1\rangle$ (the standard basis) are also known as Z-basis states. Other basis states are possible as well such as the X-basis

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

and the Y-basis (these following quantum states are also commonly known as the "plus i" state and "minus i" state, respectively)

$$|+i\rangle = \frac{|0\rangle + i|1\rangle}{\sqrt{2}} \quad |-i\rangle = \frac{|0\rangle - i|1\rangle}{\sqrt{2}}$$

2.1.2 Multiple Qubits

Classically, two bits encode (2^2) states. They are: 00, 01, 10, and 11. When we have multiple qubits, we write their states as a tensor product $\otimes$. For example, two qubits, both in the $|0\rangle$ state, are written as $|0\rangle \otimes |0\rangle$. For notational convenience, this can be compressed to $|0\rangle\,|0\rangle$, which can be further compressed to $|00\rangle$. With two qubits, the Z-basis is $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. A general 2-qubit state is a superposition of these basis states:

$$|\psi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle$$

As per the Normalization condition, the total probability of all the modulus squared of amplitudes is 1, just like in the case of one qubit.

$$|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$$

A single quantum state can also be written using the vector form. For example, the two-qubit $|00\rangle$ is

$$|0\rangle \otimes |0\rangle = |00\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ 0 & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Similarly, for 3 qubits there are 8 (2^3) Z-basis states $\{|000\rangle, |001\rangle, |010\rangle, |011\rangle, ...\}$. The previous computation can be generalized to n qubits. With n qubits, there are N = 2^n Z-basis states, where each state has its own amplitude. Thus, for example, if we have just $n = 300$ qubits, then we must keep track of $N = 2^{300} \approx 2.04 * 10^{90}$ amplitudes, which is more than the number of atoms in the visible universe (10^{78} to 10^{82}). Now, it may seem that simulating 300 qubits will be impossible on a classical computer as we'll never be able to fulfill the space requirements. However, it's unknown whether all the amplitudes are necessarily required by a quantum computer [41]. For example, the *Gottesman–Knill* theorem states that a Clifford quantum circuit can be simulated efficiently on a classical computer [22].

2.1.3 Little- vs. Big-endian

As mentioned earlier, with three qubits, there are eight Z-basis states

$$\{|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle\}$$

. For simplicity, these states (binary string) can also be written in decimal numbers $|0\rangle, |1\rangle, ..., |7\rangle$. Given that we have 3 qubits, one possible scheme is to call the right qubit the zeroth qubit, the middle qubit the first qubit, and the left qubit the second qubit, so a Z-basis state takes the form

$$|q_2 q_1 q_0\rangle$$

Then, the decimal representation of this is

$$2^2 q_2 + 2^1 q_1 + 2^0 q_0$$

This convention, where the rightmost qubit is the zeroth qubit, is called *little endian*. Many quantum platforms (such as Qiskit) use little endian. In contrast, the opposite convention, where the leftmost qubit is the zeroth qubit, is called big endian. Platforms, in general, allow for inter-conversion between little-to-big endian systems.

2.1.4 Quantum Computation

Quantum computation is a change in the quantum state (i.e., the state of a qubit). For effecting a change, quantum logic gates are used. These are similar to classical logic gates, where computation is defined as a change in the classical bit. For example, the classical NOT gate can change a 0 to a 1 and 1 to a 0. Analogously, a Quantum NOT gate should transform $|0\rangle$ to $|1\rangle$ and vice-versa while maintaining other quantum properties.

2.1.5 Measuring quantum states

Consider a qubit defined as $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$. What happens if we measure this qubit? *Classically*, we can think of the measurement of a bit as simply a readout: we have a system that encodes the state '0' and '1' and we make a measurement to find out which one it is. Although the laws of quantum mechanics permit superposition of $|0\rangle$ and $|1\rangle$, it also demands that if we measure the qubit, such as at the end of a computation, in order to read the result, we get a single, definite value. This value is either 0 [3] with the probability of $|\alpha|^2$ or 1 with the probability of $|\beta|^2$, not a superposition of values. This loss of superposition (and other quantum properties such as entanglement) is called a *collapse*. That is, the qubit collapses to $|0\rangle$ or $|1\rangle$. If we measure the collapsed qubit again, we get a previously-collapsed result with a probability of 1.

2.1.6 Quantum Gates

Mathematically, a quantum gate is a *unitary* matrix that acts on a quantum state and changes it. As per the Normalization condition, it keeps the total probability equal to 1. In fact, all quantum gates can be thought of as matrices, with the matrix entries specifying the exact details of the gate [4]. Moreover, a gate can act on more than one qubit at the same time. For example, in classical computing, a NOT logic gate acts on one bit, and an AND logic gates act on two bits.

2.1.7 Reversibility

Reversible computing (reversibility) is any model of computation where the computational process, to some extent, is time-reversible. Since a quantum gate U must be unitary, it satisfies

$$\mathrm{U}^\dagger \mathrm{U} = \mathrm{U}\mathrm{U}^\dagger = I$$

where I is an identity matrix. Therefore, a quantum gate U is always reversible, and its inverse is $\mathrm{U}^\dagger$ (pronounced as *U dagger*).

2.1.8 Multi-qubit Quantum Gates

Quantum gates can also operate on two qubits at the same time. For example, the CNOT gate (Controlled-NOT) gate inverts the right qubit if the left qubit is 1:

$$\begin{aligned} CNOT|00\rangle &= |00\rangle \\ CNOT|01\rangle &= |01\rangle \\ CNOT|10\rangle &= |11\rangle \\ CNOT|11\rangle &= |10\rangle \end{aligned}$$

The left qubit is called the control qubit, and the right qubit is called the target qubit. Note that the control qubit is unchanged by CNOT, whereas the target qubit becomes

[3] In Quantum Computer Science, it is customary to label the outcomes '0' for $|0\rangle$ and '1' for $|1\rangle$

$$CX\ q_0, q_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

(a) Little-endian matrix

$$CX\ q_1, q_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Big-endian matrix

Figure 2.2: Little- and big-endian matrices for CNOT quantum gate

the XOR (exclusive OR) of the inputs. In other words, if control qubit is set, the target qubit is inverted, leaving the control unchanged. Also, since the X gate is the NOT gate, the CNOT gate is also called the CX gate or controlled-X gate.

As we saw earlier, with little-endian convention, higher qubit indices are more significant. Therefore, the matrix for CNOT with q_0 as control and q_1 as target is given in Fig. 2.2a. However, in many textbooks [4] (and platforms such as Cirq), controlled gates are presented with the assumption of more significant qubits as control, which in this case would be q_1. Thus a matrix for such a gate is provided in Fig. 2.2b.

When inter-operating between platforms, it is pertinent to remember this as it may lead to different numerical answers for the same circuit. Moreover, this is true for any control gate [31].

2.1.9 Quantum Gates Example

NOT gate

The Quantum NOT gate should, in principle, do the following:

$$NOT(\alpha |0\rangle + \beta |1\rangle) = \alpha |1\rangle + \beta |0\rangle \tag{2.3}$$

As we saw earlier, a qubit is just a vector. Hence, a matrix transformation can affect the change. The NOT matrix is:

$$NOT = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.4}$$

Hadamard gate

A very interesting gate that is not available on classical computers is the Hadamard gate. This gate is responsible for putting a qubit in a *superposition* of states.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.5}$$

[4]such as *Quantum Computation and Quantum Information* by Isaac Chuang and Michael Nielsen

Gate	Action on Computational Basis	Matrix Representation
Identity	$I\|0\rangle = \|0\rangle$ $I\|1\rangle = \|1\rangle$	$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Pauli X	$X\|0\rangle = \|1\rangle$ $X\|1\rangle = \|0\rangle$	$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli Y	$Y\|0\rangle = i\|0\rangle$ $Y\|1\rangle = -i\|1\rangle$	$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli Z	$Z\|0\rangle = \|0\rangle$ $Z\|1\rangle = -\|1\rangle$	$I = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
T	$T\|0\rangle = \|0\rangle$ $T\|1\rangle = e^{i\pi/4}\|1\rangle$	$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Hadamard H	$H\|0\rangle = \frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$ $H\|1\rangle = \frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$	$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
SWAP	$SWAP\|00\rangle = \|00\rangle$ $SWAP\|01\rangle = \|10\rangle$ $SWAP\|10\rangle = \|01\rangle$ $SWAP\|11\rangle = \|11\rangle$	$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Table 2.1: Common Quantum Gates

The H gate puts the quantum states $|0\rangle$ or $|1\rangle$ in superposition such that measuring the qubit will result in either 1 or 0 with 50% probability. The H gate can act generally on any quantum state as described by (2.1).

2.1.10 List of Common Quantum Gates

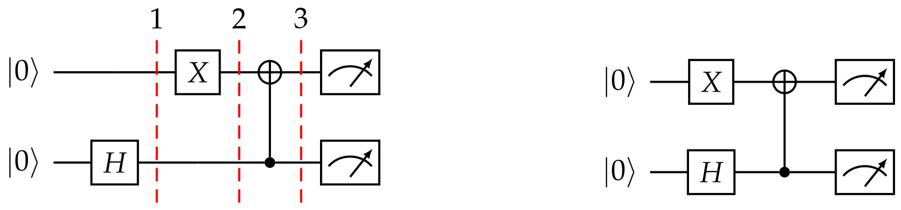
There exists an uncountably infinite number of gates [31]. However, some common gates are named and mentioned in Table 2.1. A full description of these gates can be found in [31].

2.1.11 Quantum Circuit

A combination of quantum logic gates that transforms the input quantum state into the output quantum state is termed as quantum circuit or a quantum program. For example, given two qubits $|00\rangle$, we may apply the H gate to 1st qubit. Then, we apply the X gate to 0th qubit. Then, we apply the CNOT gate where the 1st qubit is the control and the 0th qubit is the target. Finally, measure the outputs.

Circuit diagrams

The previous textual description of a circuit is often not enough to understand what a circuit is doing. Just like we can draw a classical circuit diagram consisting of bits and logic gates, we can draw quantum circuit diagrams consisting of qubits and quantum gates. This helps us visualising the circuit for improved clarity. Here's a standard diagram of the circuit discussed above:



(a) More verbose diagram (b) Compact diagram

Figure 2.3: A quantum circuit that applies H, X, and CNOT gates

As we can see in Fig. 2.3(a), there are two lines starting with labels $|0\rangle$. These lines are conventionally called "quantum wire". The $|0\rangle$ is the initial value of the wire. Next, we see the $\boxed{H}$ symbol. The square box with a letter inside indicates a gate. In this case, it is the Hadamard (H) gate. Other possible options are listed in Table 2.1. We also see red dotted vertical lines. These lines delineate quantum operations from each other. Similarly, X gate ($\boxed{X}$) is applied next. Then, a two-qubit CNOT gate is applied. The solid dot (•) is the control. The cross-hair ($\oplus$) symbol directly above the control is the target qubit. A line connects them to indicate the control-target relationship of the CNOT gate. Finally, the meter symbol ($\boxed{\nearrow}$) indicates a measurement gate.

In general, the operations in a diagram can be written compactly as shown by an equivalent diagram in Fig. 2.3(b). The diagrams generated by a quantum platform can be made more colourful and may add extra information but the main concepts stay the same. We can see a larger and more complex diagram of an example Qiskit circuit (shown in Fig. 2.4) that is using three qubits and two classical bits. There are four main components in this quantum circuit:

1. Initialization and reset: q_0 is initialised to an arbitrary quantum state $|\psi\rangle$ whereas q_1 and q_2 have the default initial value $|0\rangle$.

2. Quantum gates: Two H gates and two CNOT gates are applied.

3. Measurements: The **`crz`** and **`crx`** denote classical bits. Outputs of q_0 and q_1 are stored in **`crz`** and **`crx`** respectively as classical bits.

4. Classically conditioned quantum gates: Single-qubit Z and X quantum gates are applied on the third qubit. These gates are conditioned on the results of the measurements that are stored in the two classical bits. In this case, we are using

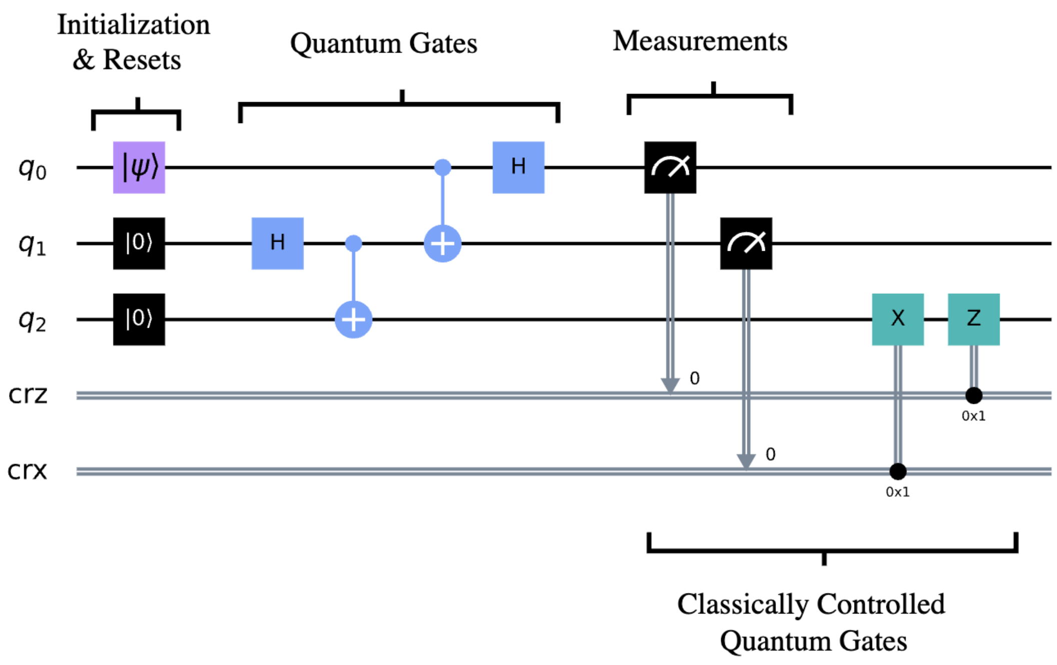


Figure 2.4: A more complex Qiskit circuit diagram presenting quantum teleportation.

the results of the classical computation concurrently in real-time within the same quantum circuit.

2.1.12 Quantum Programs

Like classical circuit diagrams, quantum circuit diagrams are good for understanding small quantum algorithms and gate interactions. However, they become untenable as the number of qubits and/or the number of gates grow. Moreover, they are not executable in the circuit form.

A quantum program is a program that manipulates quantum gates to drive a quantum algorithm, and it is executable. However, this distinction between circuit and program is often ignored, and these words are used interchangeably. Generally, quantum programs are written in "classical" languages such as Python, which then emit instructions in the form of an intermediary code that a quantum computer may understand. For example, IBM has OpenQASM (Open Quantum Assembly language) [15] quantum assembly language which interoperates with their Python framework, Qiskit. Since a classical language is used, we can also use all the features of Python to model a quantum problem. We are not limited to just quantum domains. However, one can use standalone quantum languages which are designed from the ground-up to have first-class support various quantum operations. For example, the Scaffold programming language [1]. QASM[5] is also fairly readable and high-level. Here's an annotated simple program

```
// necessary preamble for all QASM programs
OPENQASM 2.0;
include "qelib1.inc";

// allocate 5 qubits, initially set to 0
qreg q[5];

// allocate 5 classical bits where final measurements are recorded
creg c[5];

// Apply Hadamard gate on all the qubits
h q[0];
h q[1];
h q[2];
h q[3];
h q[4];

// Measure the qubits
measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
measure q[3] -> c[3];
measure q[4] -> c[4];
```

This program generates a **truly** random 5-bit on every run. The H gate puts all the qubits in superposition. Upon measurement, the superpositions collapse, and each qubit becomes 0 or 1 with 50% probability. Quantum physics forbids the measurement of qubits without losing the superposition, thereby making each qubit truly random.

2.2 Quantum Software Platforms

In 1940s and 50s, computers such as ENIAC (Electronic Numerical Integrator and Computer) were used to calculate factors of large numbers, compute ballistic routes, and simulate the decay of neutrons. Highly specialised personnel were required to operate the computer, and it was manipulated by operators at the level of registers and gates. A "modern" programming language (such as C) was 20 years in the future. The state of a QC mirrors this quite closely. That is, we are manipulating QCs at the level of gates and registers and highly specialized training is required to understand a quantum circuit. Moreover, currently, QCs are not used in general-purpose applications, but for

[5]OpenQASM is generally referred to as QASM

specialised use-cases in Chemistry, Finance, and others. However, a major difference between programming QCs today versus CCs of the past is that we can take advantage of the CCs to program a QC using specialised CC-based languages and IDE (Integrated Development Environment).

In this thesis, we define the term **Quantum Computing Platform** (QSP) to refer to the entire apparatus (hardware and software), which is necessary to develop and deploy quantum software applications. It can include a quantum programming language, an optimizing compiler that translates a quantum algorithm written in a high-level language into quantum gate instructions, a quantum simulator that emulates these instructions on a classical device, and a software controller that sends analog signals to very expensive quantum hardware based on quantum circuits.

At the lowest level of the quantum computing stack, a language must instruct the computer which physical operations to perform on which qubits. We refer to these languages, such as Quil in Forest and OpenQASM in Qiskit, as quantum assembly/instruction languages, or occasionally as quantum languages for brevity. On top of quantum languages sit quantum programming languages, which are used to manipulate quantum languages in a more natural and readable way for programmers. Examples of quantum programming languages include pyQuil, which is embedded into the classical "host" Python programming language, or Q#, a standalone quantum programming language resembling the classical C# language. If we abstract the hardware and focus purely on the software, other terms for this collection of modules are **Quantum Software Stack** (QSS) or **Quantum Development Kit** (QDK [6]) or **Quantum Software Platform** (QSP). Since there is no definite meaning of these terms, the terms may be used interchangeably without loss of generality.

A normal workflow of a user that wants to run a Quantum Circuit on a specific QDK is:

1. **Build**: Design a quantum circuit(s) that represents the problem you are considering. This would involve using the platform-specific APIs to create qubit registers and apply gates and measurements.

2. **Compile** or transform: Compile circuits for a specific quantum service, e.g. a quantum system or classical simulator. Platforms generally provide API in a high-level language, such as in Python. The compilation would ensure that the program is ready to execute by a simulator or real hardware.

3. **Run:** Run the compiled circuits on the specified quantum service(s). These services can be cloud-based (often real hardware) or local (often simulator).

4. **Analyze:** Compute summary statistics and visualize the results of the experiments. The results are a distribution of bit values.

Let's take a look the software platforms.

[6]analogous to Software Development Kit or SDK

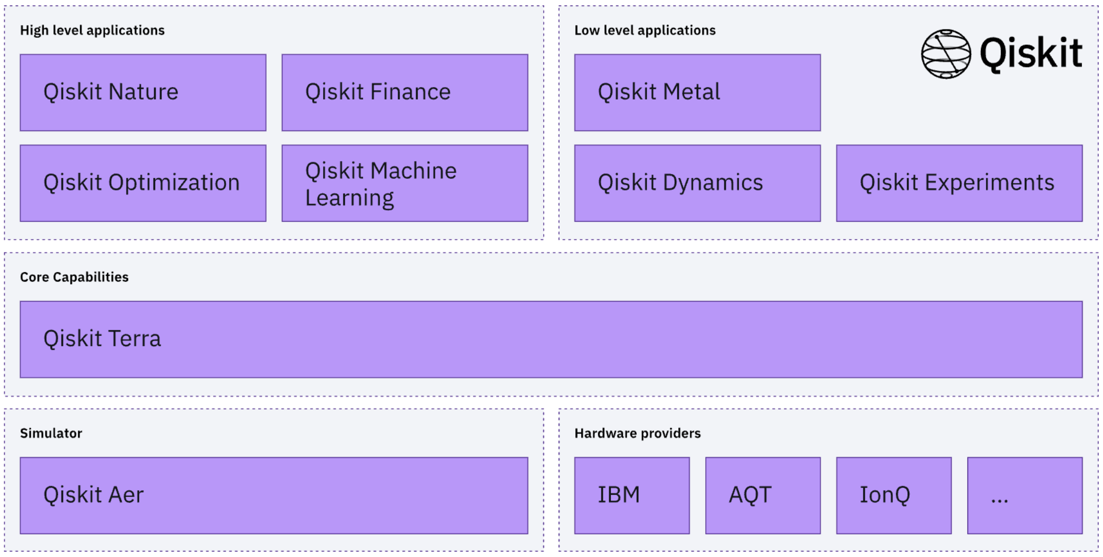


Figure 2.5: Qiskit Components

2.2.1 Qiskit

Qiskit (The Quantum Information Software Kit) [20] is a Python-based open-source software development kit for working with quantum computers at the level of circuits, pulses, and algorithms. Qiskit consists of a few components [34] (also called *Elements*) as shown in Figure 2.5:

- **Qiskit Terra**: It is a core module that handles quantum circuit construction, circuit analysis and transformation, and general-use algorithms, such as VQE (Variational Quantum Eigensolver).
- **Qiskit Aer**: provides high-performance quantum computing simulators with realistic noise models.
- **Qiskit Experiments** (previously called ignis): the module that contains all the required tools to implement error mitigation techniques.
- **Qiskit Aqua**: Another deprecated module. The module was split out to separate application repositories (qiskit-optimization, qiskit-machine-learning, qiskit-nature, qiskit-finance), with the core algorithm and operator function moved to qiskit-terra.
- **Qiskit Dynamics** is for building, transforming, and solving models of quantum systems.
- **Qiskit Metal** is used for the design of superconducting quantum chips and devices.

- **Hardware Providers**: A collection of quantum hardware and software partners that extend and complement the Qiskit ecosystem.

Here's a sample Qiskit program (Listing 1). This code was randomly generated by MorphQ as part of the running of the experiments.

```python

# importing packages
from qiskit import (
    QuantumCircuit,
    ClassicalRegister,
    QuantumRegister,
    Aer,
    transpile,
    execute,
)
from qiskit.circuit.library.standard_gates import *

# Initializing bits and qubits
qr = QuantumRegister(3, name="qr")
cr = ClassicalRegister(3, name="cr")
qc = QuantumCircuit(qr, cr, name="qc")

# Adding Gates
qc.append(
    U3Gate(4.655749679598676, 2.7381706999194857, 2.740795817289426),
    qargs=[qr[0]],
    cargs=[],
)
qc.append(RYYGate(5.171156764260811), qargs=[qr[2], qr[1]], cargs=[])
qc.append(DCXGate(), qargs=[qr[2], qr[0]], cargs=[])
qc.append(U1Gate(4.660569462447812), qargs=[qr[1]], cargs=[])
qc.append(CPhaseGate(5.442036812415247), qargs=[qr[1], qr[0]], cargs=[])
qc.append(RYGate(3.1620892961233205), qargs=[qr[2]], cargs=[])
qc.append(RZGate(2.816396898940768), qargs=[qr[2]], cargs=[])
qc.append(HGate(), qargs=[qr[0]], cargs=[])

# Adding Measurement Gates
qc.measure(qr, cr)

# Transforming the circuit
qc = transpile(qc, optimization_level=2)

```

```
38 # Simulating the Experiment
39 backend = Aer.get_backend("qasm_simulator")
40 counts = (
41     execute(qc, backend=backend, shots=1024)
42     .result()
43     .get_counts(qc)
44 )
45
46 # output
47 # counts =
48 # {'010': 66,
49 #   '000': 198,
50 #   '110': 63,
51 #   '100': 182,
52 #   '001': 202,
53 #   '111': 75,
54 #   '011': 71,
55 #   '101': 167}
56
```

Listing 1: Qiskit Sample circuit

This code listing shows all the necessary components required to understand most Qiskit programs. We begin by importing the packages required to design and execute the circuit at the top of the script (lines 3-11).

- **QuantumCircuit**: is the class that encapsulates a quantum circuit. It's a container for all the quantum operations.
- **ClassicalRegister**: implements a classical register, i.e classical bits.
- **QuantumRegister**: implements qubits.
- **AerSimulator:** is a high-performance circuit simulator.
- **transpile**: transpiles circuits according to some desired transpilation targets. For example, performing optimization passes, or converting the circuit from one gate set to another equivalent gate set.
- **execute**: executes the circuit given a backend (a real quantum hardware or a simulator). It returns the result of the execution.

Line 11 imports all available gates in Qiskit. Normally, one would only import required gates. However, to keep the already-long code listing short, **`import *`** is used.

We create a quantum register with 3 qubits (line 14), a classical register with 3 bits (line 15) to hold the output of the computation and initialize a quantum circuit with both of these registers (line 16). By default, qubits and bits are in $|0\rangle$ and 0 state respectively. The **`name`** argument is a human-readable label that is visible when the circuit is visualised.

Next, statements of the kind

```
qc.append( Gate(...), qargs=[...], cargs=[] )
```

(where **`Gate`** is any quantum gate) add a gate that acts on certain qubit(s) and bits. Some important observations about gates:

- A gate can accept any number of parameters. These parameters are often rotations around axes as depicted in a Bloch sphere. For example, in line 20, **`U3Gate`** is a single-qubit rotation gate with 3 Euler angles.
- A gate may act on single or multiple qubits as defined by the **`qargs`** array argument. For example, on line 20, **`U3Gate`** acts on the 0th qubit of the 3-qubit register initialized on line 14.
- Similarly, **`cargs`** array defines which classical bits to act on. It is generally an empty array as most gates only act on qubits.

Since Qiskit is a huge platform, **`qc.append`** is not the only way to add gates to a circuit. Other APIs such as the following exist. For example, this line

```
qc.append( HGate(), qargs=[qr[0]], cargs=[ ] )
```

is equivalent to

```
qc.h(0)
```

Lines 19 to 30 apply various gates such as **`RYY`** (A parametric 2-qubit $Y \otimes Y$ interaction, rotation about YY), **`DCX`** (Double-CNOT gate), **`CPhase`**(Controlled-Phase gate) and more.

Once all the gates are applied, the measurement gates are added, in this case, on line 33. The **`measure(qr, cr)`** is a shorthand for measuring each qubit identified by an index into the classical bit identified by the same index. That is, **`qr[0]`** measurement will be kept in **`cr[0]`**. Similarly, for **`qr[1]`** and **`cr[1]`** and subsequently for all values up to n where n is qubit register size. One can also measure just a single qubit. Generally, measurement gates are terminal in a quantum circuit as we saw earlier that the act of measurement leads to a collapse of the quantum properties.

Now, the circuit can be (optionally) transformed into an equivalent circuit. In this case, we optimize the circuit to level 2 on line 36. Other possible levels are 0, 1, and 3 where 3 is the maximum level of optimization. Since **`transpile`** returns another circuit, we store it in **`qc`** again, overriding the older circuit.

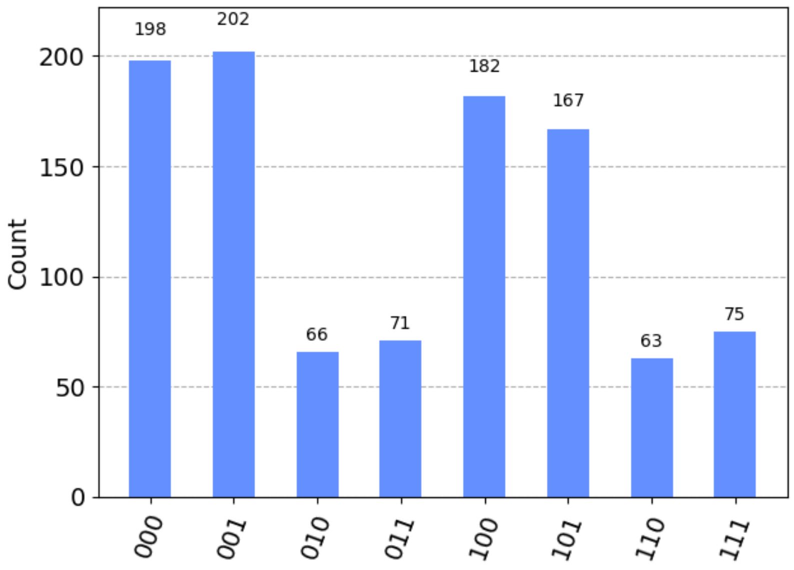


Figure 2.6: Histogram visualisation of Qiskit results

After the transpilation, the circuit is ready to be executed. We create a backend using the **Aer.get_backend** function on line 39. Here, we use the **qasm_simualtor**. However, other options are **statevector_simulator**, **aer_simulator_density_matrix** etc. The **execute** function accepts a circuit, a backend, and an optional **shots** argument. As we saw that a quantum circuit can be probabilistic (for example, due to superposition), the circuit needs to be executed many times to get a distribution of the output. The number of times a circuit is to be executed is referred as *shots*[7]. The **shots** argument in the **execute** function is the same thing. By default, its value is 1024.

Once the simulation (or execution on real hardware) succeeds, we use the **result().get_counts(qc)** to get a distribution of the output bit strings, shown here from lines 48 to 55. The keys are bit strings and values are a number of times it appeared.

The distribution can be visualised as a histogram using the from **plot_histogram** function. For example, given **counts** on line 40 in the above listing, it can be plotted as a histogram using this code snippet:

```python
from qiskit.visualization import plot_histogram
plot_histogram(counts)
```

If you execute this command in a Jupyter Notebook, you'd see something like Figure 2.6. The sum of histogram values will equal 1024 in this case as that was the **shots** value.

[7]The word "shot" comes from experimental physics where an experiment is performed many times, and each result is called a shot.

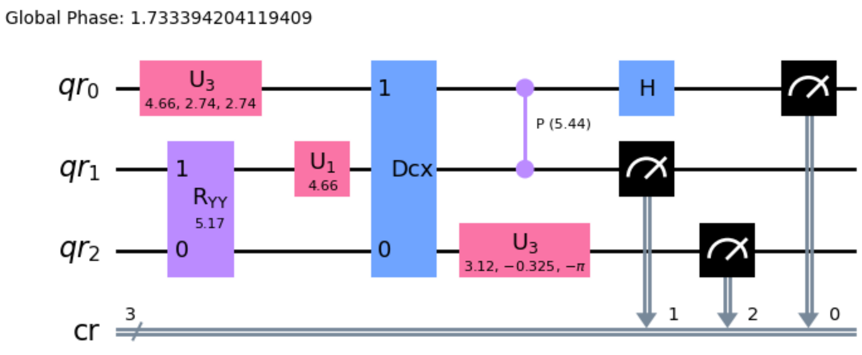


Figure 2.7: Circuit visualisation

Visualising a circuit

As discussed in Section 2.1.11, a circuit can be visualised. Given a Qiskit circuit, we can use the **draw** function to render it. The rendering can be done using pure ASCII letters (when using the command line) or using the rasterized image (for example, when using Jupyter notebooks). The circuit in Listing 1 can be visualised as Fig. 2.7 using the following code snippet.

```
qc.draw(output="mpl")
```

"mpl" value for **output** argument stands for matplotlib[8] library. Other options are **"latex"** and **"latex_source"**. Skipping it would render ASCII diagrams.

OpenQASM

IBM uses the OpenQASM [15] as an intermediate representation that can be used by higher-level compilers to communicate with quantum hardware. There are currently two versions of OpenQASM that Qiskit supports. OpenQASM 2 and OpenQASM 3 [14]. The third version is still under active development. Hence, it's part of Qiskit as a beta version. We already saw an example of QASM 2 circuit in Section 2.1.11. Below is a QASM 3 export of Listing 1.

```
OPENQASM 3;
include "stdgates.inc";

gate ryy_4403967552(_gate_p_0) _gate_q_0, _gate_q_1 {
  rx(pi/2) _gate_q_0;
```

[8]https://matplotlib.org/

```
5    rx(pi/2) _gate_q_1;
6    cx _gate_q_0, _gate_q_1;
7    rz(5.171156764260811) _gate_q_1;
8    cx _gate_q_0, _gate_q_1;
9    rx(-pi/2) _gate_q_0;
10   rx(-pi/2) _gate_q_1;
11 }

12 gate dcx _gate_q_0, _gate_q_1 {
13   cx _gate_q_0, _gate_q_1;
14   cx _gate_q_1, _gate_q_0;
15 }

16 bit[3] cr;
17 qubit[3] _all_qubits;

18 let qr = _all_qubits[0:2];

19 u3(4.655749679598676, 2.7381706999194857, 2.740795817289426) qr[0];
20 ryy_4403967552(5.171156764260811) qr[2], qr[1];
21 dcx qr[2], qr[0];
22 u1(4.660569462447812) qr[1];
23 cp(5.442036812415247) qr[1], qr[0];
24 ry(3.1620892961233205) qr[2];
25 rz(2.816396898940768) qr[2];
26 h qr[0];

27 cr[0] = measure qr[0];
28 cr[1] = measure qr[1];
29 cr[2] = measure qr[2];
```

Listing 2: Qiskit QASM3 exported circuit

It's beyond the scope of this thesis to discuss the differences between QASM versions 2 and 3. However, a brief summary is given below: QASM3 begins with a preamble (lines 1-2) that declares version 3 and required module(s) to import. **`stdgates.inc`** contains most of the gate definitions. As opposed to QASM 2, QASM 3 allows variable declarations, improve custom gate definitions, loop constructs etc. In summary, it can be viewed as a marriage of QASM 2 with additional basic C-inspired syntax such as variable assignments, looping, and control-flow statements. A complete list of differences can be found in [14].

2.2.2 PyQuil

PyQuil is a Python-based framework designed for constructing Quil [9] (**QU**antum **I**nstruction **L**anguage) programs and executing them on either simulated or actual quantum processors [37]. Quil serves a similar purpose as Qiskit's QASM within the context of the Rigetti Quantum Computing platform. Rigetti Computing, the company behind PyQuil, has integrated it as a component of their Quantum Development Kit (QDK), *Forest*. Utilizing PyQuil necessitates the installation of other components within the Forest QDK, specifically the **Quil compiler (quilc)** and the **Quantum Virtual Machine (QVM)**, both of which are employed for simulating quantum computers. PyQuil can also be used to execute programs on real quantum computers through Rigetti's Quantum Cloud Services (QCS).

The Forest QDK encompasses three primary components:

1. **PyQuil:** A high-level Python library dedicated to the generation and execution of Quil programs.
2. **QVM:** A high-performance simulator for executing Quil programs.
3. **quilc:** The Quil Compiler (quilc) facilitates the compilation and optimization of Quil programs to native gate sets.

Both the QVM and the quilc compiler are distributed as separate program binaries, accessible through the command line. They offer support for direct command-line interaction in addition to a server mode. The latter is necessary when working with PyQuil. Let's see a pyQuil code sample.

```python
from pyquil import Program, get_qc
from pyquil.gates import *

qc = Program()

qr = qc.declare("ro", "BIT", 2)

qc.inst(H(0))
qc.inst(CNOT(0, 1))
qc.inst(H(1))
qc.inst(RZ(6.163759533339787, 0))

qc += MEASURE(0, qr[0])
qc += MEASURE(1, qr[1])

qc.wrap_in_numshots_loop(20)
```

[9] also stylized as QUIL

```
12 qvm = get_qc("9q-square-qvm")
13 executable = qvm.compile(qc, protoquil=True)
14 result = qvm.run(executable).readout_data.get('ro')
15 print(result)
```

Listing 3: pyQuil sample code

The syntax of pyQuil is similar to Qiskit, and is fairly readable and succinct. The main element for writing quantum circuits is a **Program** class and can be imported from **pyquil** module. **Program** is equivalent to Qiskit's **QuantumCircuit**. Gate operations can be found in **pyquil.gates** as can be seen on line 2. Unlike Qiskit and Cirq, one does not need to pre-define qubits in pyQuil , rather, they are allocated dynamically. Qubits in this "implicit" qubit register are referred to by bare indexes (0, 1, 2, ...) as we can see on lines 5-8. For example, **H** gate acts on 0th qubit.

However, classical bits do need to be declared. It can be done using the **declare** method available on the **Program** instance (line 4).

In the **declare** method, the **"ro"** is the identifier of the declared variable, which in this case stands for **r**ead**o**ut. The second argument is the type of the declared memory. Other options are: **'REAL'**, **'OCTET'** or **'INTEGER'**. The third argument is the allocated memory size, which is the number of array elements in the declared memory.

Once a **Program** object is instantiated, gates can be applied on it. Lines 5-8 apply the **H**, the **CNOT**, **H**, and a final **RZ** gate with a parameter value of around **6.16**. The **.inst** (for instruction) API is used to add a gate, but the **+=** operator can also be used for the same thing as lines 9-10 show when adding the **MEASURE** gates. The measurement gates are very similar to Qiskit and Cirq. The first argument is the qubit whose measurement value will be stored in the location specified by the second argument.

Once a circuit is ready to be executed, the **wrap_in_numshots_loop** tells how many trials to do for this circuit. This is equivalent to **shots** in Qiskit's **run** method.

The **get_qc** function can be used to get a simulator (qvm) or an cloud-based access to an actual quantum computer. The **"9q-square-qvm"** is a specific kind of simulator (line 12). Other options are **"Aspen-X"** and **"pyqvm-X"** where X is the max number of qubits required by the circuit.

Once a QVM is fetched, we can make the circuit ready to be run on it. This is same thing as Qiskit's **transpile**. However, in case of pyQuil, it's called **compile** (line 13). We give the circuit as argument. An extra argument **protoquil=True** is also given. The optional protoquil keyword argument instructs the compiler to restrict both its input and output to protoquil, that is, Quil code that can be executed on a QPU.

After the circuit compilation, it can be executed. However, before calling the **run** method of the QVM, it is important to ensure that **quilc** and **qvm** components of the Forest SDK are running in server mode. To run the **quilc**, you can use this command:

```
quilc -S \
    --quiet \
    --prefer-gate-ladders \
    --log-level notice \
    --protoquil \
    --enable-state-prep-reductions
```

Similarly, for the QVM component,

```
qvm -S --log-level warning --compile --optimization-level 3
```

A description of these command-line options can be found by running **`quilc -h`** or **`qvm -h`**. After execution, the result can be read by calling the **`readout_data.get('ro')`** method (line 14). A pyQuil result of the above execution may looks something like the following:

```
[
    [0, 1], [1, 0], [1, 0], [1, 1], [1, 1],
    [1, 1], [1, 0], [1, 0], [0, 0], [0, 1],
    [1, 0], [0, 0], [0, 1], [0, 1], [1, 1],
    [0, 1], [1, 0], [0, 0], [0, 0], [0, 1]
]
```

The result is a list of list. The outer list contains a **`num_shot = 20`** inner lists. The inner lists' size is equal to the total number of measured qubits. In this case, the outer list has 20 inner lists and each inner list has two measurements corresponding to **`qr[0]`** and **`qr[1]`** respectively. This is same same ouput as Cirq, but Cirq returns bitstring rather than a list of bits, though it is possible to get a similar list in Cirq as well using the **`.data`** property of the **`Result`** instance.

Visualising a circuit

pyQuil programs may be converted to LaTeX circuit diagrams, or even rendered immediately in a Jupyter Notebook using the **`pyquil.latex`** module. For this to work, one needs two external programs, **`pdflatex`** and **`convert`**, to be installed and accessible via the system shell path along with a number of external latex packages. The visualised circuit will look very similar to Fig. 2.3 in appearance and design.

Quil (Quantum Instruction Language)

The Quil [37] language, analogous to OpenQASM, is what instructs the quantum computer which physical gates to implement on which qubits. The general syntax of Quil is "GATE index" where GATE is the quantum gate to be applied to the qubit indexed by index (0, 1, 2, ...). pyQuil has a feature for generating Quil code from a given program. For example, quil equivalent of the code in Listing 3 is:

```
DECLARE ro BIT[2]
H 0
CNOT 0 1
H 1
RZ(6.163759533339787) 0
MEASURE 0 ro[0]
MEASURE 1 ro[1]
```

2.2.3 Cirq

Google Cirq is a Python software library for writing, manipulating, and optimizing quantum circuits, and then running them on quantum computers and quantum simulators.

Let's see a Cirq code sample.

```
1  import cirq

2  qr = cirq.NamedQubit.range(2, prefix='q')

3  # Create a circuit that applies gates
4  circuit = cirq.Circuit(
5      cirq.H(qr[0]),
6      cirq.CX(qr[0], qr[1]),
7      cirq.H(qr[1]),
8      cirq.rz(6.163759533339787)(qr[0]),
9      cirq.measure(qr[0], key='q0'),
10     cirq.measure(qr[1], key='q1')
11 )

12 # Simulate the circuit several times.
13 simulator = cirq.Simulator()
14 result = simulator.run(circuit, repetitions=20)
15 print("Results:")
16 print(result)

17 # Results:
18 # q0=11011110100010101111
19 # q1=01100101011001110001
```

Listing 4: Cirq sample circuit

Unlike Qiskit and pyQuil, note the single line of import in Cirq on line 1. It is conventional to access everything using this **cirq** object (though we'll also import more things later on). It acts as a single global name-space for everything available in Cirq.

We then create two **NamedQubit** qubits. We can create the qubits in multiple ways, such as this:

```
qr =[cirq.NamedQubit('q' + str(i)) for i in range(2)]
```

However, the **range** built-in method already returns a array of qubits containing the first argument number of elements (Line 2). The **prefix** argument value prefixes each qubit name. In this case, the names of the qubit are **q0** and **q1**.

Cirq supports three topologies of qubits: **NamedQubit**, **LineQubit**, and **GridQubit**. **NamedQubit** is an abstract qubit that only has a name, nothing else. **LineQubit** is a qubit on a 1-d lattice. Some quantum devices have lines of qubits, **LineQubit** can be useful to represent that. **GridQubit** is a qubit that is placed on a grid and is identified by 2D coordinates.

Lines 4 - 10 create a **Circuit** object, an equivalent of **QuantumCircuit** in Qiskit. A **Circuit** is a collection of **Moments**. A **Moment** is a collection of **Operations** that all act during the same abstract time slice. Each **Operation** must be applied to a disjoint set of qubits compared to each of the other **Operations** in the **Moment**. A **Moment** can be thought of as a vertical slice of a quantum circuit diagram. In this case, there can be 4 moments $\{H, CX, Rx + H, M + M\}$ as we can see in Fig. 2.8.

Just like Qiskit, the gates can accept qubit arguments. In the case of **rz**, which takes a rotation parameter and a qubit argument, we first call **rz** with rotation paramter which returns a modified gate. This is, then, given the qubit argument (line 8). Finally, the measurement gates are applied. The **key** argument of **measure** method is the identifier of the readout bit.

Once the circuit is constructed, it can be executed with the help of the **Simulator** (lines 13-14). We construct an object **Simulator** and use the **run** method to execute the circuit. The **repetitions** is the **shots** equivalent of Qiskit.

After the execution, the result can be printed. Lines 18-19 show the measurement of qubits **q0** and **q1** 20 times. This output is unlike Qiskit's output representation. Qiskit clubs the bits and shows their frequency. In Cirq, the bit frequency needs to calculated manually.

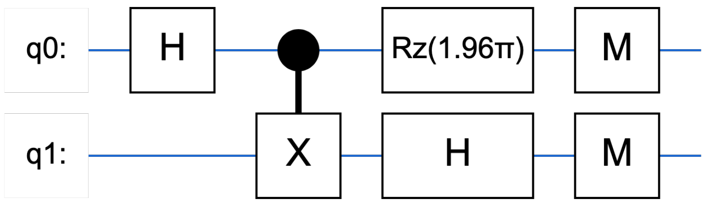


Figure 2.8: Visualisation of Cirq sample circuit

Visualising a circuit

Printing a **`Circuit`** object will render an ASCII diagram of the circuit. However, a better output may be achievend using the **`SVGCircuit`** class which can render high-quality circuit diagrams in SVG. Here's the code snippet to do that:

```python
from cirq.contrib.svg import SVGCircuit, circuit_to_svg
SVGCircuit(circuit)
```

Executing these line in Jupyter notebook with a valid **`circuit`** object will draw a digram similar to Fig. 2.8.

Intermediary Langauge

Cirq does not have its own intermediary language such as OpenQASM or QUIL. Nonetheless, Cirq has basic support for interoperability with QASM 2 and QUIL (with **`cirq_rigetti`** module). To convert a QASM 2 (for example, the circuit in Section 2.1.11) circuit to Cirq, the following functions can be used:

```python
from cirq.contrib.qasm_import import circuit_from_qasm
qasm_str = "..."
circuit = circuit_from_qasm(qasm_str)
```

2.3 Testing

Software testing is the act of examining artifacts and the behavior of the software under test by validation and verification. Two widespread methods of testing are **white-** and **black-box** testing. The former method tests internal data structures and program flow. The latter method tests the functionality, ignoring the inner workings of the software, answering the following question: will we get an expected output for a given input? In this case, we are interested in black-box testing as we are not verifying the internal data structures of the platforms under test.

Quantum Software Testing (and testing at large) is facing two fundamental problems: the *oracle problem* and the *reliable test set* problem. The oracle problem refers to situations where it is extremely difficult, or impossible, to verify the test result of a given test case t (that is, an input selected to test the program). Once a test t executes, it produces a result. Let's call this value *computed*. Now, a test oracle checks the *computed* result with its own value, let's call it, *expected*. There are two possible options: the expected does not match with the *computed* value. We say that t fails and refer to it as a failure-causing test case. Otherwise, in the second option, we say that t succeeds and refer to it as a successful, or non-failure-causing, test case. In several real-world scenarios, though, an oracle might be absent, or even if present, it could be impractical to use due to resource constraints such as time required in the case of Travelling Salesman Problem (TSP). In the case of quantum computing, the oracle problem manifests as such: if we create a **random** quantum circuit, how can we *know* if the output is what it's supposed to be without knowing it *a priori*.

The reliable test set problem means that since it is normally not possible to exhaustively execute all possible test cases, it is challenging to effectively select a subset of test cases (the reliable test set) with the ability to determine the correctness of the program [10].

What can be the oracle?

For testing quantum software platforms, a real quantum hardware could've worked as an oracle. However, quantum hardware are yet to be commercially available on a large scale [10]. They also have poor error correction, fault tolerance, and too much noise in the outputs. Hence, their use case as oracles is limited.

Two testing techniques that are used in this thesis are Metamorphic (as part of MorphQ) and Differential testing. As we'll see later, both techniques can be used to alleviate the oracle problem. Moreover, metamorphic testing also addresses the reliable test set problem.

2.3.1 Differential testing

This technique involves supplying identical input to comparable applications or distinct implementations of the same application and observing discrepancies in their performance and behaviour. It is an effective complement to conventional software testing, as it is particularly adept at uncovering semantic or logical bugs that do not manifest as overtly erroneous behaviors such as crashes or assertion failures. Back-to-back testing or differential fuzzing is an alternate term for differential testing [29].

For example, two different C compilers such as GCC and Clang can be given the same source code. If the produced executable from both compilers differs in behaviour (or in some other observable and meaningful metric), it can potentially point to a bug in either of the system.

This testing method can detect bugs without relying on an oracle, as it looks for differences in behavior rather than trying to verify the correctness of the output. Therefore, differential testing can be an effective way to identify bugs in cases where there is no oracle, or where it is impractical to use one.

2.3.2 Metamorphic testing

Metamorphic testing (MT) is a property-based software testing technique, which can be an effective approach for addressing the test oracle problem and test case generation problem (reliable test set problem). [10] A central element of MT is a set of Metamorphic

[10]at the time of writing

Relations (MRs), which are necessary properties[11] of the target function or algorithm in relation to multiple inputs and their expected outputs. When implementing MT, some program inputs (called source inputs) are first generated as **source test cases**. Then, on the basis of an MR, new inputs are generated as **follow-up test cases**. Unlike the traditional way of verifying the test result of each individual test case, MT verifies the source and follow-up test cases as well as their outputs against the corresponding MR.

For example, consider the function $min(a, b)$ which takes numbers a and b as input and returns the minimum number between them. In this case, an MR can be derived from the following property: If the numbers a and b are swapped, the answer remains unchanged. That is,

$$min(a, b) = min(b, a)$$

Based on this MR, we need two test executions, one with the source test case (a, b) and the other with the follow-up test case (b, a). Instead of verifying the result of a single test execution, we verify the results of the multiple executions against the MR — we check whether the relation $min(b, a) = min(a, b)$ is satisfied or violated. If a violation is detected, we can then say that min is faulty.

2.3.3 Cross-platform testing

Cross-platform testing, also known as multi-platform testing, ensures that a program operates as intended across various supported platforms. This approach is prevalent in testing web applications that need to function across different browsers, such as Google Chrome, Mozilla Firefox, and Apple Safari. It is also relevant for testing software that runs on diverse operating systems like Windows, macOS, and Linux or on mobile devices with Android and iOS.

Cross-platform testing does not dictate the use of any specific technique. For instance, browser-automated end-to-end methods may be employed to test web applications, while unit testing, integration testing, and system testing can also be applied to ensure software compatibility across platforms. In our study, we utilize differential and metamorphic testing to achieve cross-platform testing objectives, focusing on the compatibility and consistency of program behavior in diverse environments.

The importance of cross-platform testing has grown with the increasing variety of devices, operating systems, and user environments. This has become especially relevant with multiple quantum software platforms available today. Ensuring consistent user experience and seamless functionality across platforms is crucial for both user satisfaction and market success. Cross-platform testing helps developers identify and address potential compatibility issues, performance bottlenecks, and user interface inconsistencies, ultimately leading to a more robust and reliable software product.

[11] A necessary property of an algorithm means a condition that can be logically deduced from the algorithm.

Cross-platform testing may look very similar to differential testing. However, differential testing focuses on comparing different implementations of the same functionality, while cross-platform testing ensures consistent behavior and performance across different platforms, operating systems, browsers, or devices. Both methodologies are distinct in their usages and important in achieving a high-quality, reliable software product.

Chapter 3

Literature Review and Related Work

3.1 An Overview of Quantum Computing Platform Bugs

The field of quantum computing platforms is explored in LaRose's work, which provides a detailed comparison of gate-level quantum software platforms, including Qiskit, PyQuil, ProjectQ, and Q# [24]. Multiple studies have been undertaken to understand the nature and patterns of bugs within these platforms. Paltenghi's empirical study identifies ten unique quantum-specific bug patterns [32], while Luo's research investigates 96 real-world bugs along with their fixes across four prevalent quantum programming languages [28]. Sodhi and Kapur further delve into the quality attributes of these platforms and elucidate the challenges faced by platform developers [38]. These comprehensive studies indicate a significant need for testing within the realm of quantum computing platforms, which is the basis for our work.

3.2 Insights into Cross-Platform, Differential, and Metamorphic Testing

Cross-platform testing is a prevalent practice within the field of web and mobile applications [6]. Given the diversity of software and hardware features across various platforms, modern applications are expected to function consistently.

In the context of large software systems, differential testing serves as an effective method to identify bugs [29]. This technique has found applications across various domains, such as SSL/TLS [7], JVM [11], and clone testing [45], amongst others. Researchers have successfully applied differential testing to validate certificate correctness in SSL/TLS [8].

Metamorphic testing, on the other hand, has found applications in compiler testing, though it's not limited to this domain. This is accomplished by employing strategies like code deletion and insertion in program dead zones [25], and domain-specific transformations for graphics shading compilers [17], effective testing is achieved. Beyond compilers, tools such as debuggers can also undergo metamorphic testing [39].

This technique has also been studied in the quantum setting [2, 33].

3.3 Other Testing Techniques for Quantum Programs

While there are a few techniques focused on the testing of quantum computing platforms, there exists a range of testing methods aimed at quantum programs themselves. These methods include search-based techniques [40], statistical assertion checks that aim to minimize the impact on the actual computation [19, 27], combinatorial testing [42], and coverage-based methods [3]. Mutation testing presents another alternative, as it has been extensively applied to quantum programs and existing test suites [18, 30]. In contrast to our study, these techniques primarily examine the specific programs, not the platforms on which these programs run.

Chapter 4

Approach

In this section, we present a detailed overview of the comprehensive cross-platform testing methodology. A visual depiction of the approach can be observed in Figure 4.1. The entire overview consists of two primary components: metamorphic testing and differential testing, which together form the basis of cross-platform testing. We begin with the labeled elements ① and ② in the referenced figure. These elements represent the initial phase: the generation of a random Qiskit source program and a corresponding metamorphic-linked follow-up program (Section 4.1.1). A "program" in this context refers to source code that defines the quantum circuit and its execution settings, such as the type of backend to use or the transpiler's configurations.

Following the generation of the Qiskit programs, we utilize QCross to translate them into equivalent Cirq and PyQuil programs (③). The QCross translator ensures the preservation of any applied metamorphic relations during the translation process. Additionally, QCross employs our custom bloqs library to implement any missing gates in either PyQuil or Cirq. As part of bloqs, we ported over twenty Qiskit gates in PyQuil and Cirq each. Further details on program translation can be found in Section 4.1, while Section 4.5 elaborates on the development of the bloqs library. The six resulting programs form the "input programs" set.

Given the input programs, each program pair (source and follow-up program of a single platform) is executed (④) and the outcomes are documented in a metadata file (⑤). The collection of six results, corresponding to the six input programs, forms the "program output" set. During the execution of a single program, crashes may occur, potentially indicating bugs. Consequently, any exception data is recorded, and the program is directed to the "bug bucket" for further examination (⑩).

In instances where both programs in a pair do not crash, their output pairs undergo an equivalency check (⑦). The testing procedure is explained in section 4.4. This check may yield either a match or a mismatch (referred to as a divergence, ⑧). A mismatch could also signal potential bugs, thus leading to the inclusion of both divergent pairs in the "bug bucket." The absence of divergence implies that the test has "passed" and no further analysis of this program pair is required. This process constitutes the

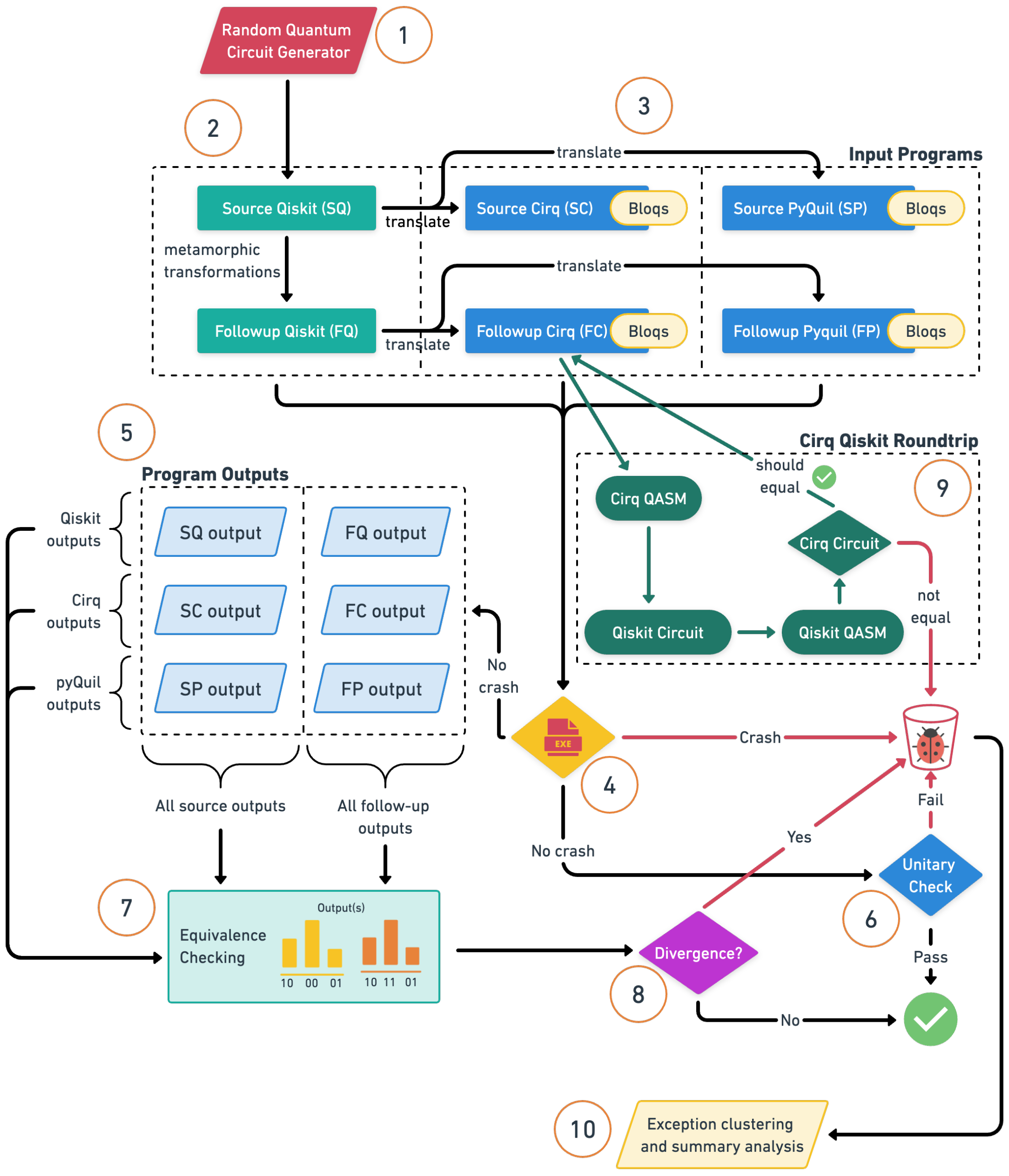


Figure 4.1: Overall approach

metamorphic component of testing, with further details available in Section 4.2.

In addition to the program pair outputs, all source outputs (5) and follow-ups of all platforms are tested for equivalency. A divergence indicates a bug. Moreover, a program pair's unitary matrices are also assessed for equality (6). Unequal unitary matrices may suggest bugs, similar to divergent pairs of outputs. Lastly, a Cirq-Qiskit-Cirq roundtrip test, mediated by QASM, is performed as well (9). The roundtrip should preserve the circuit structure; if it is unsuccessful, this may indicate a potential bug. This process constitutes the differential testing component, described in detail in Section 4.3.

All the aforementioned steps occur for each randomly generated program. For example, executing 1000 programs would involve repeating the entire process 1000 times. Once a sufficient number of programs have been tested, crash messages are clustered into similar groups by removing program-specific information such as line numbers, memory addresses, variable names, etc (10). Upon filtering and clustering the results, they are analyzed to determine whether the crash or divergence is an actual bug. A fuzzy-search investigates relevant GitHub repositories for mentions of the crash message. If no GitHub issues exist, we may have discovered a novel bug. In cases where the crash is unexplained or the bug is novel, a GitHub issue is filed, thereby contributing to the ongoing development and improvement of the respective quantum computing platforms. The author is involved in patching few of the bugs as well.

Having seen a brief overview of the entire approach, we first begin by understanding how a quantum program generated and translated. In subsequent sections involving specific testing approaches, the translation descriptions will become more precise and detailed.

4.1 QCross Program Translation

Achieving successful cross-platform testing requires the availability of the same program in multiple platforms. To address this issue, we developed the QCross program translator. Starting with a MorphQ-generated Qiskit program pair comprising a source and a metamorphically-linked follow-up program, QCross translates this pair into Cirq and PyQuil program pairs, maintaining any metamorphic relations. The following subsections elucidate the general translation process, while Section 4.2 explains how metamorphic relations are transferred across platforms.

4.1.1 Program Generation

All random programs are generated by MorphQ's Qiskit program generation utility. It uses a template-based generation function which fills in the template gaps with random gates which has random qubit arguments and parameter values. The set of gates to use is specified using a YAML[1] configuration file (along with other configuration options). MorphQ ensures that generated programs are valid, i.e., they are not syntactically malformed and do not crash due to misuse of functions/features [33]. An example of such a generated program is given in Code Listing 1.

Each quantum program can be logically divided into six parts:

1. **`import`** statements
2. Declaration of qubits / Classical bit
3. Application of gates and addition of terminal measurements
4. Preparation of the circuit to be executed on simulators
5. Execution of the program
6. Collection of the results

The QCross translator translates the circuit using the above points. Let's see it step-by-step:

4.1.2 Relevant Imports

All circuits begin with the following import statements. Together, they import the modules required for the circuit to be created, transpiled, and executed. More modules are imported as they are needed.

Import statements for Qiskit

```python
from qiskit import (
    QuantumCircuit, ClassicalRegister, QuantumRegister,
    Aer, transpile, execute
)
from qiskit.circuit.library.standard_gates import *
```

Import statements for PyQuil

```python
from PyQuil import Program, get_qc
from PyQuil.gates import *
```

Import statements for Cirq

```python
import cirq
```

As we can see, all available quantum gates are imported from Qiskit and PyQuil.

[1]YAML is a data interchange format like JSON

4.1.3 Declaring qubits and classical bits

As we saw in Listing 1, a Qiskit program requires an instance of **QuantumCircuit** which takes **QuantumRegister** and **ClassicalRegister** as arguments. For example, the following code snippet declares all the three things:

```
qr = QuantumRegister(7, name='qr')
cr = ClassicalRegister(7, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
```

Cirq: The Cirq equivalent of **QuantumCircuit** is **cirq.Circuit()**. Unlike **QuantumCircuit**, **cirq.Circuit()** does not take quantum and classical registers as arguments. In fact, an equivalent of **ClassicalRegister** does not exist. When classical registers are needed, they are used implicitly. However, **QuantumRegister** has three counterparts: **NamedQubit**, **LineQubit** and **GridQubit**.

When converting a Qiskit program to Cirq, QCross uses **NamedQubit** by default. The reason for this is that **NamedQubit** is the least constrained qubit of all three. Therefore, a Cirq equivalent of the above Qiskit code snippet is:

```
qc = cirq.Circuit()
qr = [cirq.NamedQubit('q' + str(i)) for i in range(7)]
```

qr is an array of seven **NamedQubit**s. A **NamedQubit** accepts a name argument as its identifier. In this case, the arguments are **q0**, **q1**, . . . , **q6**.

PyQuil: The following code snippet is a PyQuil equivalent of the above Qiskit code snippet:

```
qc = Program()
cr = qc.declare("ro", "BIT", 7)
```

In PyQuil, the equivalent of **QuantumCircuit** is a **Program**, which also does not take arguments as Qiskit does. Unlike both Qiskit and Cirq, qubits in PyQuil do not need to be declared upfront (see Section ***).

Here's a small table summarizing the differences in these three platforms:

4.1.4 Application of gates

In Qiskit, a quantum gate is applied on a circuit by appending it to the circuit[2]. For example, given **qc** as an instance of **QuantumCircuit** and **qr** as declared qubits, the **ZGate**, **CHGate** and **RYYGate** are applied as following:

```
qc.append(ZGate(), qargs=[qr[2]], cargs=[])
qc.append(CHGate(), qargs=[qr[2], qr[0]], cargs=[])
qc.append(RYYGate(5.398622178940033), qargs=[qr[0], qr[2]], cargs=[])
```

[2]As noted in Section 2.2.1, there are other APIs to add a gate to a circuit

Type	Qiskit	Cirq	PyQuil
Circuit	**`QuantumCircuit`**	**`Circuit`**	**`Program`**
Qubits	**`QuantumRegister`**	**`NamedQubit`** **`LineQubit`** **`GridQubit`**	⋆
Classical bits	**`ClassicalRegister`**	⋆	**`.declare`**

⋆ Not available or implicitly defined.

Table 4.1: Declaration circuit, qubits, and classical bits in quantum platforms

As we saw in Section 2.2.1, measurement gates can be added with **`measure`** method which accepts a qubit register and a classical register.

Cirq: Cirq and PyQuil follow a similar model for gate application. For example, in Cirq, the first gate (**`ZGate`**) of the above Qiskit code snippet can be translated as (assuming appropriately declared **`qc`** and **`qr`**):

```
qc.append(cirq.Z( qr[2] ))
```

Since **`ZGate`** is already available in Cirq, we can use it.

The next gate (**`CHGate`**) can be translated as follows:

```
qc.append(cirq.H.controlled()( qr[2], qr[0] ))
```

Cirq already has **`H`** gate built in. We, therefore, use the **`controlled`** method to get a controlled version of this gate. Then, the qubits are passed to this returned gate value.

However, what about translating the **`RYYGate`**?

We are out of luck as Cirq does not provide the **`RYYGate`** gate out of the box. At this point, we use **`bloqs`** library to import a Cirq-compatible **`RYY`** gate. First, we add the following import statement at the top of the Cirq program.

```
from bloqs.ext.cirq import Gates
```

Then, we can translate the statement involving **`RYYGate`** as the following snippet.

```
qc.append(Gates.RYYGate(5.398622178940033)( qr[0], qr[2] ))
```

Therefore, the entire Qiskit code snippet in Cirq becomes:

```
# qc is an instance of cirq.Circuit()
qc.append(cirq.Z( qr[2] ))
qc.append(cirq.H.controlled()( qr[2], qr[0] ))
qc.append(Gates.RYYGate(5.398622178940033)( qr[0], qr[2] ))
```

For sake of consistency, **bloqs** also exports the **ZGate** and **CHGate** for Cirq, even though **CHGate** can be generated with an extra method call. Therefore, the above code can be simplified to

```
# ZGate is equal to cirq.Z
qc.append(Gates.ZGate( qr[2] ))
# CHGate is equal to cirq.H.controlled()
qc.append(Gates.CHGate( qr[2], qr[0] ))
qc.append(Gates.RYYGate(5.398622178940033)( qr[0], qr[2] ))
```

Similarly, most of Qiskit gates can be translated to Cirq with the help of **bloqs** library. Measurement gates are added using the **cirq.measure** function. They are "appended" (generally terminally) just like any other gate. For example, measuring the qubit **qr[0]** is:

```
qc.append(cirq.measure(qr[0], key="cr0"))
```

PyQuil: A similar case is made for PyQuil. **ZGate** is translated as the following since **ZGate** is already available.

```
# qc is an instance of Program
qc.inst(Z(2))
```

CHGate can be translated using PyQuil's ability to create a controlled version of any gate using the **controlled** method of the gate.

```
# control qubit 2 and target qubit 0
qc.inst(H(0).controlled(2))
```

When translating the **RYYGate**, we face a similar problem as we did in Cirq. PyQuil does not have **RYYGate** natively available. We again use **bloqs** to overcome the problem. We add PyQuil-specific **bloqs** import statement.

```
from bloqs.ext.PyQuil import Gates
```

Then, the entire Qiskit code snippet becomes:

```
qc.inst(Z(2))
qc.inst(H(0).controlled(2))
qc.inst(Gates.RYYGate(5.398622178940033)( 0, 2 ))
```

Just as in Cirq, for sake of consistency, **bloqs** also exports the **ZGate** and **CHGate** for PyQuil. Therefore, the above code can be simplified to

```
# equal to Z(2)
qc.inst(Gates.ZGate( 2 ))
# equal to H(0).controlled(2)
qc.inst(Gates.CHGate( 2, 0 ))
qc.inst(Gates.RYYGate(5.398622178940033)( 0, 2 ))
```

Unfortunately, just importing the **RYYGate** and adding it in a circuit is not sufficient for execution. Since **RYYGate** is a new custom gate, its gate definition also needs to be "added" to the circuit. Luckily, the **bloqs** library provides us with gate definitions as well. To do this, we need to import **get_custom_get_definitions** from **bloqs**. Then, as the following code snippet shows, we add the definition (**ryy_defn**) to the circuit using the **+=** operator. The full translation of the above 3-gate Qiskit code is:

```
# ... imports ...
from bloqs.ext.PyQuil import get_custom_get_definitions

qc = Program()
ryy_defn = get_custom_get_definitions("RYYGate")
qc += ryy_defn

# equal to Z(2)
qc.inst(Gates.ZGate( 2 ))
# equal to H(0).controlled(2)
qc.inst(Gates.CHGate( 2, 0 ))
qc.inst(Gates.RYYGate(5.398622178940033)( 0, 2 ))
```

Measurement gates are added with the help of **MEASURE** gate. Its description can be found in Section 2.2.2.

As we can see in the import statements of PyQuil and Cirq, **bloqs** library is used to import quantum gates as opposed to importing from the module itself. This ensures that a poly-filled version of the gate is imported if the gate in question is missing in either PyQuil or Cirq. Construction and working of the **bloqs** library is provided in Section 4.5.

4.1.5 Preparation of the circuit to be executed on simulators

Platforms that provide connectivity to real quantum devices must necessarily have a means of translating a given circuit into operations the computer can understand. This process is known as *transpilation* or *compilation*, or verbosely *quantum circuit compilation/quantum compilation*. Each hardware has a specific fixed set of available gates (basis gates) and qubit layout. It is the job of the platform to accept a circuit and return an equivalent circuit obeying the basis set and qubit connectivity requirements [24].

Moreover, at this stage, a circuit may be optimized to remove redundant operations (for example two gates that cancel each other out), re-arrange operations as to independently execute two sub-circuits parallelly for faster execution, merge multiple gates into a single different gate (for example, 3 CNOTs can be replaced with a SWAP) or change qubit topology or more.

In Qiskit, the function that does this is called **`transpile`** which accepts many arguments to tune its behaviour. In PyQuil, the **`compile`** method on the QVM instance can be used to optimize a circuit and make it ready for execution. Cirq, on the other hand, does not have a compilation step as Qiskit and PyQuil do. It has different modules such as cirq-riggeti, cirq-ionq to help with the transformation. For just optimization, it has independent functions such as **`drop_empty_moments`**, **`drop_negligible_operations`** etc. that the developers need to pick and apply manually.

4.1.6 Execution of the program

Once the circuit is ready, it is fed to a simulator for execution. In Qiskit, the **`Aer`** module provides different simulators. For example, running **`Aer.backends()`** lists these backends:

```
[
    AerSimulator('aer_simulator'),
    AerSimulator('aer_simulator_statevector'),
    AerSimulator('aer_simulator_density_matrix'),
    AerSimulator('aer_simulator_stabilizer'),
    AerSimulator('aer_simulator_matrix_product_state'),
    AerSimulator('aer_simulator_extended_stabilizer'),
    AerSimulator('aer_simulator_unitary'),
    AerSimulator('aer_simulator_superop'),
    QasmSimulator('qasm_simulator'),
    StatevectorSimulator('statevector_simulator'),
    UnitarySimulator('unitary_simulator'),
    PulseSimulator('pulse_simulator')
 ]
```

Once a simulator is chosen using the **`get_backend`** method on **`Aer`** module, it can be instantiated and passed to the **`execute`** function along with **`shots`** to execute the circuit:

```
qasm_sim_backend = Aer.get_backend('qasm_simulator')
result = execute(qc, backend=qasm_sim_backend, shots=2048)
```

Cirq has two simulators: the **`Simulator`** and the **`DensityMatrixSimulator`**. Just like Qiskit, we instantiate it and use it as follows:

```
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=2048)
```

PyQuil has many simulators. But we are going to use two local simulators: **9q-square-qvm** and **Xq-qvm** where **X** is the number of qubits one needs such as **13q-qvm**. **9q-square-qvm** is not general purpose, but suitable for a small circuit with less than 9 qubits. On the other hand, **Xq-qvm** can be used for an arbitrary circuit for larger qubit sizes. We can get either of the simulators using the **get_qc** function and then, like Cirq, use the **run** method to execute the circuit. Note that the **QVM** and **quilc** have to be running in the background in server mode as described in Section 2.2.2.

```
qc = get_qc("11q-qvm")
result = qc.run(executable)
```

4.1.7 Collection of results

As we saw in Section 2.2, all three platforms have different representations of results. By default, qiskit provides with bit string frequency, Cirq gives us all measured bit-strings, and PyQuil returns an array containing the measured values. Now, there are many methods available in each of the platforms to convert one representation to another. However, in this thesis, Cirq and PyQuil results are converted to a Qiskit-like representation. For this, we use the **get_qiskit_like_output** function available in **bloqs** module. The exact description of how the results are normalised is provided in 4.5. Here's a code sample for Cirq:

```
from bloqs.ext.cirq.utils import get_qiskit_like_output

# ... circuit
qc.measure(qr[0], key='cr0')
qc.measure(qr[1], key='cr1')
qc.measure(qr[2], key='cr2')

result = simulator.run(circuit, repetitions=2048)

# keys are the measurement `key` in measurement gates
output = get_qiskit_like_output(result, keys=['cr0', 'cr1', 'cr2'])
```

and for PyQuil:

```
from bloqs.ext.PyQuil.utils import get_qiskit_like_output

# ... circuit
```

```
result = qc.run(executable)

data = result.readout_data.get('ro')
output = get_qiskit_like_output(data)
```

4.2 Metamorphic Testing Methodology

We utilize metamorphic testing to test intra-platform consistency and robustness. Two different, but meta-moprhically linked, programs are executed on the same platform and a difference in execution is recorded for further analysis for potential bugs. Due to metamorphic testing, we overcome the Oracle problem and can assert on the outputs of the two programs without knowing the expected output prior.

Algorithm 1 gives a high-level overview of the steps taken as part of metamorphic

Algorithm 1: Metamorphic Testing Approach

```
Input: Set $S$ of Qiskit source and follow-up program pairs
       QCross Translator $T$
       Comparison component $C$
Result: Likely bug-revealing pairs $B$ of programs
$B \leftarrow \emptyset$
input_programs $\leftarrow$ []
foreach pair $P$ in set $S$ do
    input_programs.push($P$)
    foreach platform in ["Cirq", "PyQuil"] do
        $T_{src}$ $\leftarrow$ T.translate(platform, $P_{src}$)
        $T_{foll}$ $\leftarrow$ T.translate(platform, $P_{foll}$)
        input_programs.push(T)
    end
end
foreach pair $P$ in input_programs do
    try
        $out_{src}, out_{foll}$ $\leftarrow$ C.execute($P_{src}$, $P_{foll}$)
    catch Execution Error
        B.add($P$)
    end
    if C.compareDivergence($out_{src}$, $out_{foll}$) == divergent then
        B.add($P$)
    end
end
return $B$
```

Metamorphic Relation	**Qiskit**	**Cirq**	**PyQuil**
Circuit Transformation			
Change Qubit Order	Y	Y	Y
Inject null-effect operation	Y	Y	Y
Add quantum register	Y	N[a]	-
Inject parameters	Y	N[a]	Y
Partitioned execution	Y	Y	Y
Representation transformation			
Intermediary language roundtrip	Y	Y[b]	Y[c]
Roundtrip conversion via QASM3[†]	Y	-	-
Serialization roundtrip	Y	Y[b]	-
Execution transformation			
Change of coupling map	Y	Y	N[a]
Change of gate set	Y	N[a]	N[a]
Change of optimize. level	Y	Y	Y
Change of backend	Y	Y	Y

[†] This is a new transformation.
[a] No API support for this feature.
[b] Cirq supports limited QASM2 and has its own JSON-based serialization format.
[c] PyQuil does not inter-operate with Qiskit's QASM, but has its own assembly format called `quil`.

Table 4.2: Metamorphic Relations

testing. As a first step, MorphQ generates a random Qiskit program, called the *source program*. It also generates a *follow-up program* that is meta-morphically linked to the source program by one or more metamorphic relations. As noted earlier, a follow-up program may have multiple metamorphic relations applied to it. These programs comprise a set and is one of the input of Algorithm 1. The QCross translator converts the source and the follow-up programs to equivalent Cirq and PyQuil programs (lines 3-9). In the process, it preserves any metamorphic transformations that were originally applied to the follow-up program.

The **bloqs** library does the heavy-lifting of polyfilling[3] the gates using the custom gates API of each platform as described in Section 4.5.

Once all 3 programs are ready, execution begins. The 3 pairs are executed one by one (line 13). If one of the programs crashes, it is recorded as part of the metadata. Programs that crash form the "crash bucket" (line 15). They may potentially point to a bug. If the program ran successfully, the results are recorded. Due to metamorphic relation, each pair's result should be equivalent. We test this equivalency using the

[3]a piece of code used to provide modern functionality on systems that do not natively support it

Kolmogorov–Smirnov test (line 17), as is the norm in testing quatum programs [41]. If the results are not equivalent, it's a divergence. A divergent pair may also point to potential bugs. Hence, it also forms part of the crash bucket.

After the execution, the crash messages are clustered into similar groups. Once the results are filtered and clustered, they are analyzed to see if the crash or the divergence is *actually* a bug or not.

Table 4.2 summarizes all the metamorphic relations that are part of this testing methodology. Subsequent sections will describe each relation and provide details for its translation in different platforms.

The metamorphic relations can be categorized into three categories [33]:

- **Circuit transformations**: These transformations exploit the properties of the gate model of computation, such as the entanglement of qubits, the presence of registers and the properties of reversible computing.
- **Representation transformations**, which change the intermediate representation used to represent the circuit.
- **Execution transformations**, which affect the execution environment, e.g., by changing the backend, the optimizer, the coupling map, or the target gates to use.

4.2.1 Change Qubit Order

This transformation maps the qubit indices of source program to new positions and then creates a follow up program by adapting the sequence of gates to the newly mapped qubit indices.

For example, consider the source circuit of Figure 4.2(a) (visualised in 4.2(c)), which has a two-qubit gate between qubit 1 and qubit 2. Applying the transformation with the bijective qubit mapping $m = \{0 \to 2, 1 \to 0, 2 \to 1\}$ results in Figure 4.2(b) (visualised in 4.2(d)), where the two-qubit gate now is between qubit 0 and qubit 1. The final measurement gates are not affected by the qubit mapping. Instead, the approach applies a function to all the output bit-strings of the follow-up program that applies the inverse of m to the order of measured qubits. In the example, suppose we obtain an output bit-string 001 by the follow-up program. The approach will turn it into a bit-string 100, because the bit at index 2 in the follow-up program corresponds to be at index 0 in the source program. After this re-mapping of the measurements, the two resulting output distributions are expected to be equivalent.

When translating this in Cirq and PyQuil, nothing extra is required. A follow-up program is treated in the same way as a source program by the QCross translator, except the qubits are different.

4.2.2 Inject null-effect operation

As we saw earlier in Section 2, quantum computing is an instance of reversible computing, i.e., performing any operation or gate, with the exception of the measurement gate, on a set of qubits never looses any information, and hence, can be reverted back with a suitable inverse operation. This metamorphic transformation exploits this property by inserting into the main circuit a sub-circuit that performs a sequence of gate operations followed by its inverse, so that the overall effect is null. The sub-circuit may include an arbitrary number of gates and act on an arbitrary number of available qubits except gates which cause quantum states to collapse such as measurement gate.

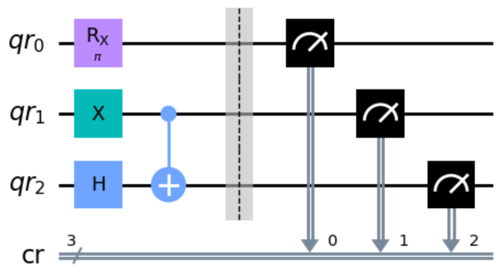
Here's an example of a sub-circuit in a Qiskit program.

```python
qr = QuantumRegister(
    3, name='qr'
)
cr = ClassicalRegister(
    3, name='cr'
)
qc = QuantumCircuit(
    qr, cr, name='qc'
)

qc.rx(math.pi, 0)
qc.x(1)
qc.h(2)
qc.cx(1, 2)

qc.measure(qr, cr)
```

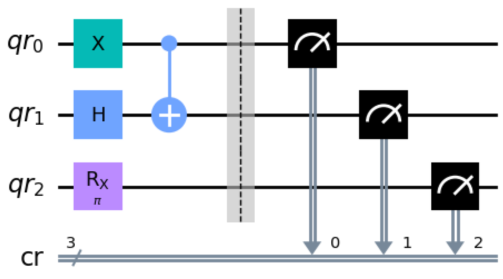
(a) Code of source program for Change Qubit Order metamorphic relation

```python
qr = QuantumRegister(
    3, name='qr'
)
cr = ClassicalRegister(
    3, name='cr'
)
qc = QuantumCircuit(
    qr, cr, name='qc'
)

qc.rx(math.pi, 2)
qc.x(0)
qc.h(1)
qc.cx(0,1)

qc.measure(qr, cr)
```

(b) Code of follow-up program for Change Qubit Order metamorphic relation



(c) Visualisation of source program



(d) Visualisation of follow-up program

Figure 4.2: Example of "Change Qubit Order" transformation

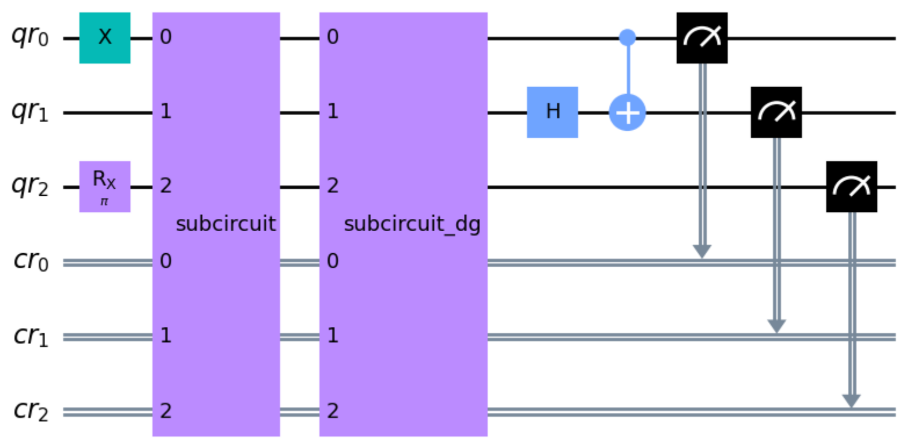


Figure 4.3: Example of code inserted by the "inject null-effect operation" transformation.

```
1  qr = QuantumRegister(3, name='qr')
2  cr = ClassicalRegister(3, name='cr')
3  qc = QuantumCircuit(qr, cr, name='qc')
4
5
6  qc.rx(math.pi, 2)
7  qc.x(0)
8
9  subcircuit = QuantumCircuit(qr, cr, name='subcircuit')
10 subcircuit.append(HGate(), qargs=[qr[0]], cargs=[])
11
12 qc.append(subcircuit, qargs=qr, cargs=cr)
13 qc.append(subcircuit.inverse(), qargs=qr, cargs=cr)
14
15 qc.h(1)
16 qc.cx(0,1)
17
18 qc.measure(qr, cr)
```

Lines 9-10 construct a sub-circuit with the same **qr** and **cr** as the original circuit. Then, the circuit is appended like a normal gate to the circuit in line 12. Immediately afterwards, it's inverse is appended to nullify the effect in line 13 using the **.inverse** method available on gates and circuits. When visualising this in Qiskit, it looks like Figure 4.3. In the figure, the block *subcircuit* denotes an arbitrary sub-circuit and *subcircuit_dg* denotes its inverse (dg = dagger).

A similar strategy can be applied when translating this relation to Cirq and PyQuil. Cirq and PyQuil, both, allow for appending of a sub-circuit.

Cirq: Cirq exports a function named **.inverse** which accepts a Cirq circuit and returns an inverted version of the circuit.

The following example shows a code snippet of how it can achieved:

```
qc = cirq.Circuit()
# add gates to qc

subcircuit = cirq.Circuit()
# add gates to sub-circuit

qc.append(subcircuit)
qc.append(cirq.inverse(subcircuit))
```

PyQuil: PyQuil behaves exactly the same, the only difference being that the inverse function is called **`dagger`** and is available on a gate instance or a **`Program`** instance. The following code example illustrates it:

```
qc = Program()
# add gates to qc

subcircuit = Program()
# add gates to sub-circuit

qc.inst(subcircuit)
qc.inst(subcircuit.dagger())
```

QCross translate the relevant sub-circuit in this case and uses the appropriate API to invert it. The sub-circuit translation is identical to a circuit translation for QCross.

4.2.3 Add quantum register

Enlarging the set of available qubits by adding a new and unused quantum register should not affect the computation on the existing qubits. This transformation exploits this property by randomly adding new quantum registers to the circuit of the follow-up program. This transformation cannot be performed when the coupling map has been specified before via the Change of coupling map transformation, since the addition of a register would make the coupling map too small.

In Qiskit, the method **`add_register`** can be used to dynamically extend the register size further in the circuit (for example, after the measurement gates). The following code snippet demonstrates this:

```
unused_register = QuantumRegister(5, name='extra_registers')
qc.add_register(unused_register)
```

A comparable API does not exist in Cirq and PyQuil. Cirq's and PyQuil's qubit registers are not bound to a circuit, and in PyQuil specifically, there is no need to declare qubits to begin with. Hence, this relation does not extend to all platforms tested in this thesis. It's a Qiksit-exlucsive transformation.

4.2.4 Inject parameters

Parameterized quantum circuits are quantum circuits that contain one or more parameters (for example, the angle provided to **`RXGate`**) that can be adjusted without changing the overall structure of the circuit. These parameters are typically represented by continuous variables, such as angles, that control the behavior of specific gates in the circuit. The reason for parameterization is that this flexibility allows us to optimize the parameters in our circuit for a particular task or application without having to rebuild the entire circuit each time the parameters change. Given the recent interest in quantum machine learning, quantum computing platforms offer abstractions to support the parameterization of quantum circuits [36]. They can be employed in variational quantum algorithms like the Variational Quantum Eigensolver (VQE) and the Quantum Approximate Optimization Algorithm (QAOA) [31].

Before we can execute a parameterized circuit, we have to bind the parameter to a specific value or an array of values. This is very similar to having a function definition with parameters and then using the function with concrete argument at the time of function invocation. Moreover, creating a function that accepts arguments to generate a quantum circuit is indeed an alternative approach to using parameters in a quantum circuit. However, there are some advantages to using parameters:

- **Efficiency**: When using parameters, the circuit structure remains fixed, and only the parameter values change. This allows for more efficient optimization, as the circuit doesn't need to be reconstructed from scratch every time a parameter changes.
- **Circuit Compilation**: Parametrized circuits can be transpiled and optimized before the parameters are bound to specific values. This allows for more efficient circuit execution, as the optimization can be performed once and then reused for different parameter values.

In Qiskit, a parameter is an instance of **`Parameter`** class which accepts a user-defined name. Once an instance is created, it can be used instead of the parameter value in a gate. Before execution, the concrete values can be supplied using the **`bind_parameters`** method available on a **`QuantumCircuit`** instance. The following code snippet demonstrates all the steps for Qiksit:

```python
from qiskit.circuit import Parameter

theta = Parameter('theta')
gamma = Parameter('gamma')
# lambda is a reserved keyword in Python
_lambda = Parameter('lambda')
```

```
7
8  # initialize qc, qr, cr
9
10 qc.append(RZGate(theta), qargs=[qr[0]], cargs=[])
11 qc.append(U2Gate(theta, 2.12), qargs=[qr[2]], cargs=[])
12 qc.append(CRZGate(gamma), qargs=[qr[1], qr[0]], cargs=[])
13 qc.append(RZGate(_lambda), qargs=[qr[1]], cargs=[])
14
15 qc.measure(qr, cr)
16
17 # before execution
18 qc = qc.bind_parameters({
19     theta: 4.2641612072511235,
20     gamma: 2.5163050709890156,
21     _lambda: 2.586208953975239,
22 })
```

Line 1 imports the **Parameter** class and lines 3, 4, and 6 construct three parameters: *theta*, *gamma*, *_lambda*. Like float values, parameters are used as values in gates as shown in lines 10-13. Note line 11. Parameters and concrete values can be used at the same time. Now, the circuit can be transpiled / optimized. Before the execution, lines 19-23 provide concrete values for the parameters in the form of a dictionary. The keys are **Parameter** instances that were defined on lines 3-6.

In terms of steps, Cirq and PyQuil are same: declare parameters, use them in gates, and bind values.

Cirq: In contrast to Qiskit and PyQuil, Cirq uses SymPy [4], a external symbolic mathematics package, to define parameters. The above Qiskit circuit example can be ported to Cirq as following (note that U2Gate is missing as Cirq has a bug in supporting Parameters in **QasmUGate** which is used by bloqs to implement U2Gate in Cirq):

```python
import cirq
from sympy import Symbol

theta = Symbol('theta')
gamma = Symbol('gamma')
# lambda is a reserved keyword in Python
_lambda = Symbol('lambda')

# initialize qc, qr, cr, and import bloqs

qc.append(cirq.rz(theta)(qr[0]))
qc.append(Gates.CRZGate(gamma)( qr[1], qr[2] ))
```

[4]https://www.sympy.org/en/index.html

```
13 qc.append(cirq.ry(_lambda)(qr[1]))
14
15 # measurement gates
16
17 # before execution
18 qc = cirq.resolve_parameters(qc, {
19     "theta": 4.2641612072511235,
20     "gamma": 2.5163050709890156,
21     "_lambda": 2.586208953975239,
22 })
```

Line 2 imports the **Symbol** class and lines 4-7 construct three parameters: *theta*, *gamma*, *_lambda*. These are used in gates in lines 11-13. Note the use of a bloqs gates on line 12. All Cirq gates in bloqs support parameters. Now, the circuit can be transpiled / optimized. Before the execution, lines 19-23 provide concrete values for the parameters in the form of a dictionary. The function **resolve_parameters** expects a circuit and a dictionary of concrete values. The keys are, unlike Qiskit, string values that were passed to Symbols in line 4-7.

PyQuil: PyQuil is different than the rest in its implementation. In PyQuil, parameters are defined as declared memory region, which are then replaced with actual values. The following programs is a port of the above example in PyQuil (note that U2Gate is missing as PyQuil has a bug in supporting Parameters in custom gates which is used by bloqs to implement U2Gate in PyQuil):

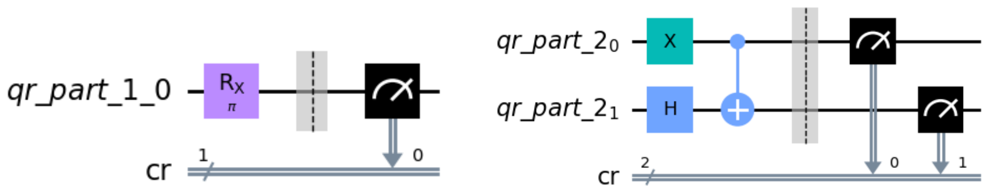
```
from PyQuil import Program

qc = Program()

theta = qc.declare('theta', 'REAL')
gamma = qc.declare('gamma', 'REAL')
_lambda = qc.declare('_lambda', 'REAL')

# import native gates, import bloqs
qc.inst(RZ(theta, 0))
qc.inst(Gates.CRZGate(gamma, 1, 2 ))
qc.inst(RY(_lambda, 1))

# measurement gates

# before execution
params = {
    "theta": 4.2641612072511235,
    "gamma": 2.5163050709890156,
20         "_lambda": 2.586208953975239
21     }
22
23     for param, value in params.items():
24         qc.write_memory(region_name=param, value=value)
```

Lines 5-7 construct three parameters: *theta*, *gamma*, *_lambda*. The **`.declare`** method is used with **`'REAL'`** as the type of memory to store floating values. These values are used in gates from line 10 to 12. Most PyQuil gates in bloqs support parameters (full support is missing due to a bug in PyQuil, explained in Section 5). Now, the circuit can be transpiled / optimized. PyQuil has a method **`write_memory`** which accepts a **`region_name`** (the name supplied to the declared memory, lines 3-5), and the concrete value. Given a dict with names as keys from lines 17 to 21, a simple loop "binds" the parameters in lines 23-24.

4.2.5 Partitioned execution

Some generated source programs might have two subsets of qubits that never interact with each other, i.e., there is no gate operation that involves the qubits of the two subsets. In this case, the source program performs two completely independent computations that can be executed in parallel. Given such a source program, this transformation separates the circuit into two sub-circuits, executes them individually, and then post-processes the result of the two sub-circuits to derive the distribution of the overall program. The post-processing of the result is a Cartesian product of the output distributions of the two independent sub-circuits, described in detail in [33].

For example, Figure 4.2(c) can be partitioned as the following two-part circuits shown in Figure 4.4(a) and (b).



(a) Part 1 (b) Part 2

Figure 4.4: Example of "Partitioned execution" transformation

4.2.6 Intermediary language roundtrip

OpenQASM2 [15], commonly abbreviated as QASM or QASM2, has become the widely accepted de-facto assembly language for quantum programs. A majority of quantum computing platforms provide API calls for converting to and from this language. Ensuring accurate conversion to and from QASM is essential for maintaining the interoperability of various quantum computing platforms. This transformation involves converting the quantum circuit into QASM format, followed by parsing the QASM code to reconstruct the original circuit.

A Qiskit circuit can be transformed into QASM2 format utilizing the **qasm** method available on the circuit instance. Subsequently, the generated QASM2 code can be converted back into a Qiskit circuit using the **from_qasm_str** method. Consequently, a roundtrip can be represented as follows:

```
qc = qc.from_qasm_str(qc.qasm())
```

An example of QASM 2 code can be seen in Section 2.1.11.

PyQuil has its own quantum assembly language called QUIL (see Section 2.2.2 for details). A PyQuil program object can be converted to QUIL instructions using the **out** method. Converting QUIL back to a circuit requires the use of **parse** function available in **PyQuil.parser** module. Therefore, a roundtrip can be performed as

```
from PyQuil.parser import parse
# ...
quil_str = qc.out()
qc = parse(quil_str)
```

Cirq, in contrast to Qiskit and PyQuil, does not possess a dedicated intermediary quantum assembly language. Nevertheless, it provides limited support for both QASM and QUIL formats. This support will be leveraged in our differential testing methodology, as discussed in Section 4.3.

4.2.7 Round-trip conversion via QASM3

In the case of Qiskit, the QASM 3 format is anticipated to supersede QASM 2 in the near future (refer to Section 2.2.1). Consequently, it is logical to conduct a QASM 3 roundtrip in addition to the QASM 2 roundtrip. It is important to note that QASM 3 is currently under active development, and Qiskit provides beta-level support for QASM 3 at the time of writing. Nevertheless, the expectation is that this transformation in QASM 3 can expedite the bug discovery process and help the development process in releasing a stable version sooner. This is a Qiskit-only transformation and it's not possible in Cirq and PyQuil.

Unlike QASM 2, QASM 3 APIs can be found in the **qiskit.qasm3** module. This module includes a **dumps** function for converting a circuit to QASM 3 and a **loads** function for transforming QASM 3 code back into a Qiskit circuit. The code snippet below demonstrates this process:

```
from qiskit.qasm3 import loads, dumps
qc = loads(dumps(qc))
```

An example of QASM 3 code can be seen in Listing 2.

4.2.8 Serialization roundtrip

Serialization is defined as the process of converting the state of an object into a form that can be persisted or transported. Qiskit supports a binary serialization format called QPY. It's a serialization of QuantumCircuit and ScheduleBlock objects and is designed to be cross-platform, Python version agnostic, and backwards compatible moving forward. QPY should be used if we need a mechanism to save or copy between systems a QuantumCircuit or ScheduleBlock that preserves the full Qiskit object structure (except for custom attributes defined outside of Qiskit code). Therefore, QPY roundtrip should result in the same circuit, just like QASM roundtrip. In case of Qiskit, QPY differs from QASM in that QASM is not meant to be a serilization format and at time QASM can result in a loss of information contained in the original circuit [5].

The QPY functionality is exported as part of the **qiskit.qpy** module, which behave a lot like Python's pickle module. The following code shows the roundtrip:

```
from qiskit.circuit import QuantumCircuit
from qiskit import qpy

qc = QuantumCircuit(2, name='Bell', metadata={'test': True})
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

with open('bell.qpy', 'wb') as fd:
    qpy.dump(qc, fd)

with open('bell.qpy', 'rb') as fd:
    new_qc = qpy.load(fd)[0]
```

The **qpy.dump()** function allows us to include multiple circuits in a single QPY file, and therefore, **load** returns a list of circuits. If there's a single circuit, we retrieve it using the 0th index as shown in the last line of the code **qpy.load(fd)[0]**.

Cirq has JSON-based serialization format. It serializes a Cirq circuit to a Cirq-compatible JSON representation. Many objects in Cirq can be serialized as JSON and then shared between collaborators, stored as a text file for transfer, storage, or for posterity.

The serialization can be done with **cirq.to_json** function. This will return a string that contains the serialized JSON given an object that may be Cirq circuits, moments, gates, operations, or other Cirq constructs. To take JSON and turn it back into a Cirq object, the protocol **cirq.read_json** can be used.

PyQuil does have any defined serialization format. It is expected that QUIL intermediary language be used for any serialization.

[5]https://qiskit.org/documentation/apidoc/qpy.html

4.2.9 Change of coupling map

A coupling map is defined as a representation of the connectivity between the qubits in a quantum computing device. It describes which qubits are physically connected and can directly interact with each other through two-qubit gates, such as the controlled-NOT (CNOT) gate. When designing quantum circuits for execution on real quantum hardware, it is crucial to consider the coupling map of the target device. If a circuit requires interactions between qubits that are not directly connected according to the coupling map, additional gates or operations, such as SWAP gates, may be necessary to adjust the circuit to the device's connectivity constraints.

In Qiskit, a coupling map is a simple list of pairs of integers. Note that MorphQ generates a random coupling map and enforces the it to be fully connected, i.e., no qubit is isolated, to ensure that every qubits can eventually interact with all the others at least indirectly via intermediate connections. An example of linear coupling map for our program in Figure 4.2(c) is: **`[[0,1],[1,2]]`**, where the qubits 0 and 1 are connected with each other, whereas there is no connection between qubits 0 and 2. Here's an example in Qiskit, specifying a custom coupling map:

```
1  from qiskit import QuantumCircuit, transpile
2
3  qc = QuantumCircuit(4)
4  qc.h(0)
5  qc.cx(0, 1)
6  qc.cx(1, 2)
7  qc.cx(2, 3)
8
9  # Define a custom coupling map
10 custom_coupling_map = [[0, 1], [1, 2], [2, 3]]
11
12 # Transpile the circuit using the custom coupling map
13 transpiled_qc = transpile(qc, coupling_map=custom_coupling_map)
```

The **`coupling_map`** map keyword-argument to the **`transpile`** function specifies the custom coupling map defined on line 10.

Cirq: Cirq allows for custom qubit connectivity, however, it's not as easy as Qiskit's API of passing an edge list to **`coupling_map`** argument. In Cirq, a connectivity graph (instance of **`nx.Graph`**[6]) needs be built by the user. The graph nodes are instances of either **`NamedQubit`**, **`LineQubit`**, or **`GridQubit`**. The connection between these nodes establishes the qubit connectivity. Once the graph is built, it can be passed to **`cirq.RouteCQC`** which instantiate a router instance. Calling this router on a circuit returns a routed circuit that is made of the device's physical qubits and contains only 2-qubit operations that are between physically adjacent qubits.

[6]Networkx (short-form nx) is an external Python library https://networkx.org/

We provide a function **edge_list_to_cirq_graph** as part of QCross that can generate a Cirq graph given an edge list. The implementation is:

```python
import networkx as nx
import cirq

def edge_list_to_cirq_graph(edge_list, nodes=None):
    if nodes is None:
        num_qubits = len(
            set(item for sublist in edge_list for item in sublist)
        )
        nodes = [
            cirq.NamedQubit("q" + str(i)) for i in range(num_qubits)
        ]

    graph = nx.Graph()
    for n in nodes:
        graph.add_node(n)

    for e in edge_list:
        graph.add_edge(nodes[e[0]], nodes[e[1]])
    return graph
```

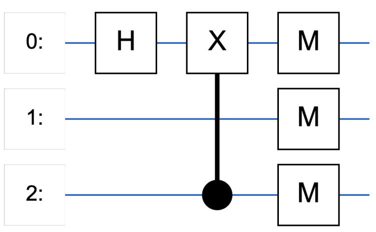
We can see this in an example. How can the following circuit be executed? Note that CNOT is applied to qubit 2 and 0. Hence, ideally they need to be connected. However, we can provide a qubit connectivity graph where qubit 0 is connected with 1, and 1 is connected to 2. Note that there is no direct connection between 0 and 1, hence, the circuit can't be executed as is.

```python
q = cirq.LineQubit.range(3)
qc = cirq.Circuit()
qc.append(cirq.H(q[0]))
qc.append(cirq.CNOT(q[2], q[0]))

qc.append(cirq.measure(q[0], key="cr0"))
qc.append(cirq.measure(q[1], key="cr1"))
qc.append(cirq.measure(q[2], key="cr2"))
```

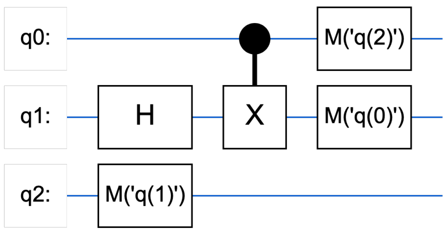
We can use the **edge_list_to_cirq_graph** to construct a re-routed circuit as following:

```python
connected_edges = [
    [0, 1],
    [1, 0],
    [1, 2],
    [2, 1],
]

routed_qc = edge_list_to_cirq_graph(qc)
```

When visualised, the circuits looks as Figure 4.5. Note the change in measurement nodes to reflect the re-routing.



(a) Before re-routing



(b) After re-routing

Figure 4.5: Example of "Change of coupling map" transformation

PyQuil does not offer straightforward APIs for defining custom coupling maps. While it is feasible to create custom devices (with **`_get_qvm_with_topology`** function) with restricted qubit connectivity, implementing custom devices may introduce challenges beyond testing. Therefore, in the context of this thesis, we don't consider this a valid PyQuil transformation.

4.2.10 Change of gate set

During transpilation, a given quantum program is converted to be compatible with a specific target device, and this often involves translating the program gates to the natively supported gates. This transformation exercise this translation step by replacing the circuit gates in the program with a universal gate set, such as **`["rx", "ry", "rz", "p", "cx"]`** gates which has been shown to be universal [43].

In Qiskit, a custom gate-set can be provided using the **`basis_gates`** keyword argument to the **`transpile`** function. Here's an example:

```
from qiskit import QuantumCircuit, transpile

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)

custom_basis_gates = ['rx', 'ry', 'rz', 'p', 'cx']
transpiled_qc = transpile(qc, basis_gates=custom_basis_gates)
```

Similar to the previous section 4.2.9 , there isn't a straightforward API in Cirq and PyQuil for converting between arbitrary universal gate sets. While options exist, such as implementing custom devices, implementing your own gate decomposition logic in Cirq via the **`cirq.CompilationTargetGateset`** class, or loading configurations of existing quantum computers in the case of PyQuil, these methods do not adequately encapsulate the essence of this transformation. As a result, like the transformation discussed in section 4.2.9, this also remains a Qiskit-exclusive feature.

4.2.11 Change of optimization level

The current and the next transformations are inspired by work on compiler testing [9]. Similar to modifying the optimization level of a traditional compiler, we can change the optimization level of the quantum transpilation process. This change is not expected to affect the final output of a program, just like traditional compilers.

Qiskit has four possible levels of optimizations: 0 (default), 1 (light optimization), 2 (medium optimization), and 3 (heavy optimization). Like coupling map and basis gates, we can pass **`optimization_level`** to the **`transpile`** function with an appropriate integer value denoting the level.

Both Cirq and PyQuil allow users to perform optimizations on their quantum circuits. However, the approach to optimization and the specific optimization levels might not be as directly comparable to Qiskit's optimization levels (0 to 3). Below, we outline how optimization can be achieved in both Cirq and PyQuil.

Cirq: Cirq provides various optimization passes that can be applied to circuits. These are called circuit transformers. These passes can be combined and applied in sequence to achieve different optimization levels. Some of the available optimization passes in Cirq are:

- **`cirq.align_left`** / **`cirq.align_right`**: Align gates to the left/right of the circuit by sliding them as far as possible along each qubit in the chosen direction.
- **`cirq.defer_measurements`**: Moves all (non-terminal) measurements in a circuit to the end of circuit by implementing the deferred measurement principle.
- **`cirq.drop_empty_moments`** / **`cirq.drop_negligible_operations`**: Removes moments that are empty or operations that have very small effects, respectively.
- **`cirq.eject_phased_paulis`**: Pushes X, Y, and PhasedX gates towards the end of the circuit, potentially absorbing Z gates and modifying gates along the way.
- **`cirq.eject_z`**: Pushes Z gates towards the end of the circuit, potentially adjusting phases of gates that they pass through.
- **`cirq.expand_composite`**: Uses cirq.decompose to expand gates built from other gates (composite gates).
- **`cirq.merge_k_qubit_unitaries`**: Replaces connected components of unitary operations, acting on <= k qubits, with op-tree given by rewriter(circuit_op).

- **`cirq.stratified_circuit`**: Repacks the circuit to ensure that moments only contain operations from the same category.
- **`cirq.synchronize_terminal_measurements`**: Moves all terminal measurements in a circuit to the final moment, if possible.

When translating, QCross uses all the transformations listed above for a Qiskit circuit with an optimization level of 1, 2 and 3. No transformations are applied for the level 0. The following function shows the code for the same:

```python
from cirq.testing import \
assert_circuits_with_terminal_measurements_are_equivalent

default_context = cirq.TransformerContext(deep=True)

def apply_transformations(circuit, context=default_context):
    optimized_circuit = cirq.expand_composite(circuit, context=context)

    optimized_circuit = cirq.defer_measurements(
        optimized_circuit, context=context
    )

    optimized_circuit = cirq.merge_k_qubit_unitaries(
        optimized_circuit,
        k=2,
        rewriter=lambda op: op.with_tags("merged"),
        context=context,
    )

    optimized_circuit = cirq.drop_empty_moments(
        optimized_circuit, context=context
    )

    optimized_circuit = cirq.eject_z(
        optimized_circuit, eject_parameterized=True, context=context
    )

    optimized_circuit = cirq.eject_phased_paulis(
        optimized_circuit, context=context, eject_parameterized=True
    )

    optimized_circuit = cirq.drop_negligible_operations(
        optimized_circuit, context=context
    )

    optimized_circuit = cirq.stratified_circuit(
        optimized_circuit, context=context
    )

    optimized_circuit = cirq.synchronize_terminal_measurements(
        optimized_circuit, context=context
    )

    # Finally, as sanity check, assert the original
    # and optimized circuit are equivalent.
    assert_circuits_with_terminal_measurements_are_equivalent(
        circuit, optimized_circuit
    )

    return optimized_circuit
```

Every transformer in Cirq accepts a **`cirq.TransformerContext`** instance, which stores common configurable options useful for all transformers. The **`deep`** argument recursively runs a transformer on every nested sub-circuit wrapped inside a **`cirq.CircuitOperation`**.

PyQuil: PyQuil does not have optimization levels as Qiskit and does not have passes as Cirq. However, it's **`compile`** function accepts an **`optimize`** argument that can be set to true or false. By default, it is set to true. As part of translation, QCross sets **`optimize=False`** for Qiskit optimization level 0.

4.2.12 Change of backend

Different simulators typically have completely different implementations, such as one based on state vectors or density matrices. A single simulator often offers two variants, running on a CPU and GPU respectively, which can be treated as two separate backends as well. In analogy to testing traditional compilers, changing the backend roughly corresponds to comparing the behavior across different target platforms. As noted in Section 4.1.6, Qiskit has various backends. Cirq has two simulator backends: **`cirq.Simulator`** and **`cirq.DensityMatrixSimulator`**. PyQuil supports multiple simulator backends as well (apart from real quantum hardware): They have **`Aspen-X`** which can be simulated and is based on the Aspen quantum computer. **`Xq-qvm`** and **`9q-square-qvm`** are simulators that are not based on real hardware but are available with different constraints.

4.3 Differential Testing Methodology

We use differential testing as it serves as a robust method for evaluating inter-platform compatibility. This technique is primarily executed through two distinct approaches. The first approach involves the verification of consistent output generation by a singular program across all examined platforms. The second approach, conversely, entails the consumption of artifacts generated by one platform as input to another, thereby examining compliance in a cross-platform context.

Algorithm 2 describes the differential testing approach of QCross. The inputs to the

Algorithm 2: Differential Testing Approach

```
Input: Set $S$ of all source and follow-up program pairs $P$
       Unitary Calculator $G$
       Comparison component $C$
Result: Likely bug-revealing pairs $B$ of programs
1  $B \leftarrow \emptyset$
2  foreach pair $P$ in $S$ do
3      $Q_{src}, C_{src}, P_{src} \leftarrow P_{sources}$
4      $Q_{foll}, C_{foll}, P_{foll} \leftarrow P_{follow-ups}$
5
6      $U_{Qiskit}, U_{Cirq}, U_{PyQuil} \leftarrow$ G.computeUnitary($Q_{foll}, C_{foll}, P_{foll}$)
7      if not ($U_{Qiskit} == U_{Cirq} == U_{PyQuil}$) then
8          add P to B                              /* Unitary check failed */
9      end
10
11     cirq_generated_qasm $Q_1 \leftarrow$ cirq.qasm($C_{foll}$)
12     qiskit_circuit $C_1 \leftarrow$ qiskit.QuantumCircuit.from_qasm_str($Q_1$)
13     qiskit_generated_qasm $Q_2 \leftarrow C_1$.qasm()
14     cirq_circuit $C_2 \leftarrow$ cirq.circuit_from_qasm($Q_2$)
15     if $C_{foll} \neq C_2$ then
16         add P to B
17     end
18
19     if C.compareDivergence($Q_{src}, C_{src}, P_{src}$) is divergent then
20         add P to B
21     end
22     if C.compareDivergence($Q_{foll}, C_{foll}, P_{foll}$) is divergent then
23         add P to B
24     end
25 end
26 return B
```

algorithm are a set S of all source and follow-up program pairs P, a unitary calculator G, and a comparison component C (input). We initializes an empty set B to store the likely bug-revealing pairs (line 1). We then iterates through each pair P in the set S (line 2), extracting the source and follow-up programs (lines 3-4). These programs were generated as part of the metamorphic testing, described in section 4.2.

In the next step, we compute the unitary matrix of all follow-up programs (line 6). If the unitary matrix for all three circuits are not equal, the program pair is added to the set B as a likely bug-revealing pair (lines 7-9). Next, we perform a Cirq-Qiskit roundtrip. We generate QASM2 output from Cirq's follow-up, feed it to Qiskit, and get Qiskit's version of the QASM 2 code. This is then finally converted to a Cirq circuit (lines 11-14). If the original Cirq follow-up circuit and the converted Cirq circuit are not equal, the program pair is added to the set B (lines 15-17). Finally, we check output divergence for source and follow-up results of all platforms (lines 19-25). These results were generated as part of metamorphic testing. If either comparison results in divergence, the program pair is added to the set B. After iterating through all the program pairs, the algorithm returns the set B containing likely bug-revealing pairs (line 26).

4.3.1 Unitary check

As previously mentioned, a quantum gate can be represented by a unitary matrix. Furthermore, the application of multiple quantum gates preserves unitarity. When two quantum circuits possess the same gates and an equal number of qubits, their respective unitary matrices will be equivalent, modulo a global phase. It is important to note that quantum states differing only by a global phase are essentially identical, as they correspond to the same point on the Bloch sphere [44]. This fundamental property is leveraged to verify the correctness of gate implementation and application across different quantum computing platforms. During the computation of unitary matrices, it is important to consider that these matrices can grow significantly in size, leading to substantial memory consumption. Consequently, to mitigate potential resource constraints, only 10% of the evaluated programs undergo unitary verification in the testing process.

Qiskit: In Qiskit, we can generate unitary matrix of a given circuit using the **`unitary_simulator`** backend[7]. It is essential to note that a Qiskit circuit must not contain measurement gates to be compatible with the unitary simulator. Another crucial consideration is the differing endian conventions used by Qiskit (little-endian) and Cirq (big-endian). To address this discrepancy, one platform must convert its matrix representation to match the other. In this case, the Qiskit circuit is converted to big-endian using the **`reverse_bits`** method. This conversion results in the circuit being "vertically" flipped. If a circuit is defined over multiple registers, the subsequent circuit will maintain the same registers, albeit with their order reversed. The following code snippet demonstrates this check:

[7] Other approaches such as **`Operator`** may also be used.

```python
from qiskit import Aer, transpile, execute

backend = Aer.get_backend('unitary_simulator')
result = execute(qc.reverse_bits(), backend=backend).result()
UNITARY = result.get_unitary(qc).data
```

Cirq and **PyQuil**: The unitary matrix of a Cirq circuit can be obtained using the **`cirq.unitary`** function. PyQuil has a function **`program_unitary`** in the **`PyQuil.simulation.tools`** that computes unitary matrices of PyQuil programs.

Once the matrices are computed, the numpy function **`allclose`**[8] is used for strict equality check, or alternatively, the function **`cirq.equal_up_to_global_phase`** is employed when matrices differ solely by a global phase. If either of these checks passes for any pair of matrices, they are deemed equivalent.

4.3.2 Cirq Qiskit Rountrip

QASM is the de-facto standard today for transferring circuit from one supported platform to another. Qiskit has full support for QASM 2. Cirq has limited but growing support for QASM 2 interoperability. Therefore, this rountrip tests Cirq QASM's generation capabilities, and Qiksit's foreign QASM ingestion capabilities. The full roundtrip consists of 4 steps:

1. Cirq's follow-up program is converted to QASM using the **`cirq.qasm`** function.

2. It is ingested by **`QuantumCircuit.from_qasm_str`** to create a Qiskit circuit.

3. The Qiskit circuit is then exported to QASM using the **`qasm`** method.

4. Finally, Qiskit's QASM is fed into Cirq to generate a Cirq circuit using the **`circuit_from_qasm`**.

The circuits from steps 1 and 4 should exhibit equivalence. As both circuits are represented in Cirq, the **`assert_circuits_with_terminal_measurements_are_equivalent`** function is employed to verify their equivalence. In QCross, the Cirq circuit is expanded before performing step 1 using the **`expand_composite`** function. This causes complex gates (such as ones exported by bloqs) to decompose into their constituents gates. Apart from having more number of simpler gates, this doesn't affect the round-trip in any other meaningful way.

4.3.3 Outputs Equivalence Check

Traditionally, differential testing involves comparing the outputs of the same program on different platforms. We employ a similar approach for this check. Since we can't do a strict equality check of quantum program outputs, we do a pairwise comparison

[8] **`array_equal`** is not used as strict floating point equality is not possible

for all source programs and all follow-up programs. Given that each set contains three programs, we carry out $\binom{3}{2} = 3$ pairwise assessments per set, resulting in a total of six comparisons. If any of these comparisons exhibit divergence, the specific pair in question is scrutinized more closely to identify potential issues.

4.4 Comparing Execution Behavior

When a pair (source and follow-up programs, for example) exposes different behaviors, we add them to the set of likely bug-revealing pairs of programs. The initial level of comparison is when one program in a pair crashes, but the other does not, it points to a potential bug. Using MorphQ terminology, this crash is termed as a crash difference.

The second level applies when both programs run without any crash, where we compare the measured output bits. Due to the probabilistic nature of quantum programming, precisely comparing the output bit-strings would be misleading. Instead, the platform repeatedly executes each circuit for the specified number of shots and then returns the output distributions. We then compares the distribution of two programs. We use the Kolmogorov-Smirnov test [23] to assess the statistical significance of the difference between the two distributions, as done in previous work [41], using a significance level of $\alpha = 5\%$. We call any pair of programs with a p-value below α a statistically significant distribution difference.

4.5 bloqs

There is a lack of one-to-one correspondence among quantum gates implemented in well-established frameworks such as Qiskit, Cirq, and PyQuil. Despite this discrepancy, the universality of quantum computation gates[9] ensures that, provided all platforms incorporate a universal gate set, any quantum gate can theoretically be adapted and ported to another platform [31]. In response to this observation, we have developed the bloqs library, which leverages this fundamental property to enable the porting of Qiskit standard quantum gates to the PyQuil and Cirq ecosystems, thereby fostering cross-platform compatibility and enhancing overall functionality. The library is open-sourced and is available via the **`pip`** package installer.

Qiskit implements around ~50 named quantum gates[10] as part of its standard library. Cirq and PyQuil have around 25 gates. Qiskit also happens to be a superset in terms of gates for Cirq and PyQuil. Therefore, when converting a Qiskit program to Cirq or PyQuil, one of the following approaches needs to be taken:

1. Replace the non-existent gate with an equivalent gate or set of equivalent gates which **are** available in the given platform.

[9]A set of universal quantum gates is any set of gates to which any operation possible on a quantum computer can be reduced, that is, any other unitary operation can be expressed as a finite sequence of gates from the set.

[10]as of writing

2. Construct the non-existent gate using the custom gate API of the platform (since every quantum gate is effectively a unitary matrix).

We implemented the **bloqs** library to solve this problem. bloqs takes the latter of the two approaches. The reasons for this are:

- Approach 1 does not lead to better abstraction. For example, **Z** gate is equivalent to two **S** gate (single qubit S gate is square root of Z). Always using two **S** gates will never introduce a more complex gate such as **Z**. Arguably, this will limit the creation of better abstraction models.
- **bloqs** can outlive this thesis as it's effectively a quantum gate and utility library.
- It's possible to add **new** gates instead of just providing Qiskit gates and provide new utility functions that lets a user inter-operate between platforms (as in the case of changing result representation).

We chose the name **bloqs** as it's a mixture of the terms **q**uantum and **blo**cks. The goal of the library is to provide the building blocks of quantum programs. We've made the source code available at **https://github.com/ArfatSalman/bloqs** and the library can be installed using the following command:

```
pip install bloqs
```

4.5.1 Usage and Structure

bloqs package has two modules: **ext** and **common**. The **ext** module further contains two sub-modules: **cirq** and **PyQuil**. Each module contains gates and utilities respective to their platform. All gates can be accessed via the **Gates** name-space of the sub-module. The **common** module contains interfaces common to all modules in **ext**.

All gates in **bloqs** use Qiskit names, even when imported in Cirq and PyQuil. This is done to ensure consistency and to not remember 3 different names for each platform.

A Cirq gate (or other utility functions) in bloqs can be accessed in such a manner:

```python
from bloqs.ext.cirq import Gates

# Cirq gate
Gates.CXGate
```

Similarly, in PyQuil:

```python
from bloqs.ext.PyQuil import Gates

# PyQuil gate
Gates.CXGate
```

4.5.2 Gates

Table 4.3 shows the list of Qiskit gates that **bloqs** has made available in Cirq and PyQuil. As mentioned above, some gates already have their counterparts in these platforms, such as **HGate** and **RXGate**. However, all empty cells denote a missing gate. "Missing gate" here strictly means that there is not a *single* importable name that is equivalent to the Qiskit gate itself. This strict definition is important as we'll see later that platforms provide simple APIs to create new gates out of existing ones.

Here are a few gate name conventions to be aware of when reading Qiskit gates:

- A gate name starting with **C** is a controlled version of the same gate without the **C**. For example, **CHGate** and **HGate**.
- A plural number of Cs in the name means that many qubits are controls. For example, **CCXGate** has two control qubits (two Cs), **C3XGate** has 3 control qubits (C3 = 3 Cs).
- A gate name having a lowercase **dg** means inverse (dagger) of the gate without the **dg**. For example, **SdgGate** and **SGate**.
- **R**-prefixed gates are rotation gates.
- **S**-prefixed gates are square roots of the non-S prefixed gates. **SXGate** and **SGate**.
- These characters can be combined and stacked. For example, **SXdgGate** is the inverse of the square root of **X** gate.

#	Qiskit	Cirq	PyQuil
1	**CUGate**		
2	**iSwapGate**	**cirq.ISWAP**	**ISWAP**
3	**RXGate**	**cirq.rx**	**RX**
4	**RYYGate**		
5	**CSXGate**		
6	**SXGate**		
7	**PhaseGate**		**PHASE**
8	**SdgGate**		
9	**C3SXGate**		
10	**RZZGate**		
11	**YGate**	**cirq.Y**	**Y**
12	**DCXGate**		
13	**RYGate**	**cirq.ry**	**RY**
14	**SXdgGate**		
15	**ECRGate**		
16	**CRYGate**		

#	Qiskit	Cirq	PyQuil
17	**RC3XGate**		
18	**U3Gate**		
19	**CHGate**		
20	**RVGate**		
21	**CRZGate**		
22	**RZXGate**		
23	**U1Gate**		
24	**SGate**	**cirq.S**	**S**
25	**RCCXGate**		
26	**RZGate**	**cirq.rz**	**RZ**
27	**C3XGate**		
28	**TdgGate**		
29	**CYGate**		
30	**XGate**	**cirq.X**	**X**
31	**HGate**	**cirq.H**	**H**
32	**CSwapGate**	**cirq.CSwapGate** **cirq.CSWAP** **cirq.FREDKIN**	**CSWAP**
33	**ZGate**	**cirq.Z**	**Z**
34	**C4XGate**		
35	**CSdgGate**		
36	**U2Gate**		
37	**CRXGate**		
38	**RGate**		
39	**TGate**	**cirq.T**	**T**
40	**UGate**	**QasmUGate**[11]	
41	**IGate**	**cirq.I**	**I**
42	**CCXGate**	**cirq.CCX** **cirq.CCNOT** **cirq.TOFFOLI**	**CCNOT**
43	**RXXGate**		
44	**CU3Gate**		
45	**SwapGate**	**cirq.SWAP**	**SWAP**
46	**CXGate**	**cirq.CX**	**CNOT**
47	**CPhaseGate**		**CPHASE**
48	**CZGate**	**cirq.CZ**	**CZ**
49	**CU1Gate**		

Table 4.3: Qiskit gates that are made available in Cirq and PyQuil by **bloqs**

[11] It's a compatibility gate for importing QASM circuits into Cirq

4.5.3 How are new gates created?

bloqs tries to leverage existing APIs already available in Cirq and PyQuil to create equivalent Qiskit gates. In general, both platforms provide methods to create:

- Controlled version of an existing gate
- Inverse of an existing gate
- Custom gates using decomposition
- Custom gates using a unitary matrix

Apart from similarities, Cirq and PyQuil have a few differentiating features as well. Cirq, for example, has APIs to raise a gate to a certain exponent t in the form of **PowGate**s such as **XPowGate**. PyQuil has the concept of a FORKED gate. The FORKED modifier allows for a parametric gate (gate accepting parameters such as **RXGate**) to be applied, with the specific choice of parameters conditional on a qubit value.

When trying to create a new gate, **bloqs** first tries to use built-in APIs or a combination of APIs of the platform itself. Only as a last resort is a unitary matrix used as a custom gate. It is because using platforms' own APIs is more future-proof.

Let's see how some of the non-available gates of Table 4.3 can be coded in Cirq and PyQuil.

4.5.4 Controlled version of gates

Cirq: Cirq has a **cirq.ControlledGate** class which augments existing gates to have one or more control qubits. It accepts a Cirq gate and a number of control qubits as arguments (along with other things). It returns an instance of a gate which accepts qubits and/or parameters. However, the class is generally not used. Objects of this class are typically created via **.controlled()** method of the gate. An alternate (but not discussed) API **controlled_by** is available as well. Using the above APIs, the following gates are created:

```python
# pass the rotation angle
def CRXGate(angle):
    return cirq.ControlledGate(cirq.rx(angle), num_controls=1)


def CRYGate(angle):
    return cirq.ControlledGate(cirq.ry(angle), num_controls=1)


def CRZGate(angle):
    return cirq.ControlledGate(cirq.rz(angle), num_controls=1)
```

```
CHGate = cirq.ControlledGate(cirq.H, num_controls=1)

# equivalently using the `.controlled` method
# instead of `ControlledGate` class
C3XGate = cirq.X.controlled(num_controls=3)

C4XGate = cirq.X.controlled(num_controls=4)
```

In Cirq, the starting qubits are controls for the controlled gates. For example, in **`C3XGate(qr[0], qr[2], qr[3], qr[1])`**, the **`qr[0], qr[2], qr[3]`** are controls.

PyQuil: PyQuil also has a **`.controlled`** method on its gates. However, its application is slightly different than Cirq's **`.controlled`** method. In **`H(1).controlled(0)`**, the gate *first* accepts the target qubit as a parameter and then the **`.controlled`** accepts the control qubit. The **`.controlled`** calls can be chained to emulate the **`num_controls`** argument of Cirq. Using this API, the following gates are created:

```
def C3XGate(a, b, c, d):
    return CCNOT(b, c, d).controlled(a)

def C4XGate(a, b, c, d, e):
    return CCNOT(c, d, e).controlled(b).controlled(a)

def CHGate(a, b):
    return H(b).controlled(a)

def CRXGate(angle, a, b):
    return RX(angle, b).controlled(a)

def CRYGate(angle, a, b):
    return RY(angle, b).controlled(a)

def CRZGate(angle, a, b):
    return RZ(angle, b).controlled(a)
```

`a, b, c, ...` represents qubit arguments. In the above gates, the functions ensure that starting qubits are controls, just like Cirq and Qiskit.

4.5.5 Inverse of gates

Cirq: Cirq gates can be raised to an exponent t, just like a number. The system internally uses Python's operator overloading to construct a new gate. For example, **`cirq.T ** (-1)`** constructs the inverse of **`T`** gate by raising the gate to -1 power. Using this API, these gates can be constructed:

```
SdgGate = cirq.S ** (-1)

CSdgGate = cirq.ControlledGate(cirq.S ** (-1), num_controls=1)

TdgGate = cirq.T ** (-1)
```

PyQuil: In PyQuil, the **`.dagger`** method on gates constructs a dagger-ed version. Using this API, these gates can be constructed:

```
def SdgGate(a):
    return S(a).dagger()

def CSdgGate(a, b):
    return Sdg(b).controlled(a)

def TdgGate(a):
    return T(a).dagger()
```

4.5.6 Exponent of gates

As we saw earlier, Cirq gates can be raised to power. What this means mathematically is that all real exponents of unitary matrices are also unitary matrices, and since all quantum gates are unitary matrices, all exponents of gates are also valid quantum gates. This logic can be used to create the square root of gates easily.

```
SXGate = cirq.X ** 0.5

CSXGate = cirq.CX ** 0.5

SXdgGate = cirq.X ** (-1 / 2)

C3SXGate = cirq.X.controlled(num_controls=3) ** 0.5
```

Unfortunately, there is no equivalent API in PyQuil. Hence, in PyQuil, these need to be created via providing their matrices in the form of a custom gate as described in Section 4.5.7.

4.5.7 Custom gates

Cirq and PyQuil both allow custom gates, either by providing a unitary matrix or a decomposition in terms of simpler gates.

Cirq: Gates are classes in Cirq. To define custom gates, we inherit from a base gate class **`cirq.Gate`** and define a few important methods. Some of them are:

- **_num_qubits_**: This returns the number of qubits this gate acts on. A similar method is **_qid_shape_** which is not discussed here.
- **_circuit_diagram_info_**: It tells Cirq how to display the gate in a circuit when rendered in ASCII.
- **_decompose_**: This method **yield**s[12] the operations which implement the custom gate. It can also return a list of operations instead of a generator.
- **_unitary_**: returns a matrix that describes the gate.
- **__pow__**: Optionally, this method **__pow__** implements exponentiation of a gate. Specifically, an exponent $e = -1$ implements the inverse of a gate.

As mentioned, methods such as **_unitary_** which we have seen are known as "magic methods." Much of Cirq relies on "magic methods", which are methods prefixed with one or two underscores and used by Cirq's protocols or built-in Python methods.

The general pattern with custom gate definitions is to:

1. Inherit from **cirq.Gate**.
2. Define **_num_qubits_** (or alternatively, **_qid_shape_**).
3. Define one of the **_unitary_** or **_decompose_** methods since both arrive at the same result via different means.
4. If the gate takes in parameters (such as rotation angles), it is passed to the constructor (**__init__**).

Using the above custom gates API, we can define the **RYYGate** gate:

```python
class RYYGate(cirq.Gate):
    def __init__(self, theta):
        super(RYYGate, self)
        self.theta = theta

    def _num_qubits_(self):
        return 2

    def _unitary_(self):
        """Return a numpy.array for the RYY gate."""
        theta = float(self.theta)
        cos = math.cos(theta / 2)
        isin = 1j * math.sin(theta / 2)
        return numpy.array(
```

[12] yield as the Python keyword in generators

```
15              [
16                  [cos, 0, 0, isin],
17                  [0, cos, -isin, 0],
18                  [0, -isin, cos, 0],
19                  [isin, 0, 0, cos],
20              ],
21          )
22
23      def _circuit_diagram_info_(self, args):
24          return "@", "RYY"
25
26      def _resolve_parameters_(self, param_resolver, recursive):
27          return RYYGate(
28              cirq.resolve_parameters(
29                  self.theta,
30                  param_resolver,
31                  recursive=recursive
32              ),
33          )
34
35      def __pow__(self, power):
36          if power == -1:
37              return RYYGate(-self.theta)
38          return super().__pow__(power)
```

Listing 5: **`RYYGate`** in Cirq

We implemented this gate using a unitary matrix as defined by the **`_unitary_`** method from lines 9 to 21. Instead of using a normal Python array, most platforms work with numpy arrays. Hence, we return numpy arrays. The actual unitary matrix is (reproduced here from [13]) is:

$$R_{YY}(\theta) = \exp\left(-i\frac{\theta}{2}Y \otimes Y\right) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & 0 & 0 & i\sin\left(\frac{\theta}{2}\right) \\ 0 & \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) & 0 \\ 0 & -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) & 0 \\ i\sin\left(\frac{\theta}{2}\right) & 0 & 0 & \cos\left(\frac{\theta}{2}\right) \end{pmatrix} \tag{4.1}$$

Since **`RYYGate`** accepts a parameter θ, the constructor takes a **`theta`** as an argument (lines 2-4) in Code Listing 5. The extra optional method **`_resolve_parameters_`** is used when the parameter passed is not a float number but instead is a different type (such as Symbol). A more thorough discussion of this is presented in Section 4.2.4.

Next, we can implement the DCX (Double CX) gate using gate decomposition as shown below:

```python
class DCXGate(cirq.Gate):
    def __init__(self):
        super()

    def _num_qubits_(self):
        return 2

    def _decompose_(self, qubits):
        a, b = qubits
        yield cirq.CX(a, b)
        yield cirq.CX(b, a)

    def _circuit_diagram_info_(self, args):
        return "@", "DCX"
```

Listing 6: **DCXGate** in Cirq

The **_decompose_** method gets the qubits as a tuple whose size is the value returned by **_num_qubits_**). Extracting the two qubits (a and b), we apply (**yield**) **cirq.CX** twice with alternating controls.

PyQuil: New gates can be added to Quil programs using the **DefGate** function which is a part of the **PyQuil.quil** module. All we need is a matrix representation of the gate. Let's assume **mat** to be a valid matrix. Then, we can define a new gate by passing a user-supplied gate name and the matrix to **DefGate** as shown below:

```python
gate_defn = DefGate("Gate-X", mat)
```

DefGate constructs a gate *definition* instance. It can't be used directly as a gate. The **gate_defn** needs to be added to a **Program** instance before its use. Given **qc** as a **Program** instance, it can be done as follows:

```python
# Get the Quil definition for the new gate
qc += gate_defn
```

The actual gate is returned by the **get_constructor** method of the **DefGate** instance. Here's sample program to show the complete flow:

```python

gate_defn = DefGate("Gate-X", mat)
X_Gate = gate_defn.get_constructor()

qc = Program()

qc += gate_defn

qc += X_Gate(0)
```

Parametric gates may exist in certain situations. In contrast to Cirq, where parameters are directly passed to constructors, PyQuil uses the **Parameter** class for handling parameters. This class allows users to provide a name for distinguishing among potential multiple parameters. Subsequently, these parameters are supplied to **DefGate** as the third argument, formatted as an array, as illustrated below.

```
DefGate("Gate-X", mat, [ Parameter("theta") ])
```

The current tt font is: macro:->\protect \tt

Moreover, unlike Cirq, we can't use **math.cos** or **math.sin** with **Parameter** type values. PyQuil provides a wrapper for these functions which are compatible with **Parameter**. Parametrized functions we can use with PyQuil are: **quil_sin**, **quil_cos**, **quil_sqrt**, **quil_exp**, and **quil_cis**. All these functions and classes are available in **PyQuil.quilatom** module.

Let's take the matrix defined in equation 4.1 for the RYY gate. Combining all of the above, it can be implemented as follows:

```
def RYYGate():
    theta = Parameter("theta")
    cos = quil_cos(theta / 2)
    isin = 1j * quil_sin(theta / 2)
    mat = numpy.array(
        [
            [cos, 0, 0, isin],
            [0, cos, -isin, 0],
            [0, -isin, cos, 0],
            [isin, 0, 0, cos]
        ],
    )
    return DefGate("RYY", mat, [theta])
```

Listing 7: **RYYGate** in PyQuil

The **RYYGate** function returns a **DefGate** instance, which, as we saw earlier, can't be used directly. We need to call the **get_constructor** method to get the actual gate.

For the sake of consistency with Cirq and Qiskit, **bloqs** exports *actual* PyQuil gates. However, since we also need the definitions, the **bloqs** function **get_custom_get_definitions** returns a list of gate definitions given string gate names as ***args** variable arguments.

We can also see how a decomposition custom gate works in PyQuil. A complex gate can be thought of as just a sub-circuit within a bigger circuit. This is the strategy in PyQuil. For example, here's an implementation of the DCX gate using the above approach:

```python
def DCXGate(a, b):
    p = Program()
    p += CNOT(a, b)
    p += CNOT(b, a)
    return p
```

Listing 8: **DCXGate** in PyQuil

The function **DCXGate** can be used like any other PyQuil gate. This approach has the advantage that we don't need to use **DefGate** or its associated methods to create a gate.

Using the techniques described in Section 4.5.4 to Section 4.5.7, we can convert all gates in Table 4.3.

4.5.8 Result Normalization

When the following Qiskit circuit is simulated:

```python
# ... imports ...

qr = QuantumRegister(3, name='qr')
cr = ClassicalRegister(3, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
```

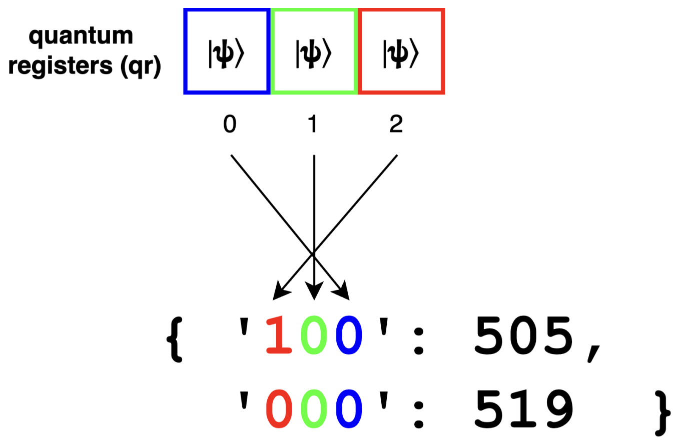


Figure 4.6: Little-endian nature of Qiskit results

```
qc.append(HGate(), qargs=[qr[2]], cargs=[])

qc.measure(qr, cr)

# ... backend selection ...

counts = execute(qc, backend=backend, shots=1024)
    .result()
    .get_counts(qc)
```

The results are:

```
{'100': 505, '000': 519}
```

Note that H gate acts on **qr[2]**. Qubits **qr[0]** and **qr[1]** are untouched and remain 0. Then, we should measure **qr[2]** as 0 and 1 with 0.5 probability each. As we can see from the frequency results above, that seems to be the case.

However, **qr[2]** seems to correspond to the first bit, **qr[1]** to the middle bit, and **qr[0]** to the third bit. This can be seen pictorially in Fig. 4.6. The three blocks denote three qubits, and the numbers below are the indices. In general, the first bit corresponds to the last index, the second bit to the second last index, and so on.

Another way to understand the result is to imagine the indices in reverse (**2 1 0** instead of **0 1 2**) and consider them as powers of 2. When presented like this, it is nothing but little-endian nature as described in Section 2.1.3.

Cirq by default prints the bitstring and has a preference for big-endian interpretation (for example, when using the **histogram** method of the **Result**).

In PyQuil, the measurement has to be specified individually for each qubit. Therefore, we, the user has to make a choice. QCross adds the measurement gates in a big-endian manner in PyQuil.

Since, Cirq and PyQuil (due to QCross) qubits are measured in a big-endian manner, it needs to be converted to a little-endian for equivalence comparison. The function **get_qiskit_like_output** does that conversion. Two versions of this function are available: one in **bloqs.ext.PyQuil.utils** and another in **bloqs.ext.cirq.utils**. They have the same name, but different signatures and implementations due to differences in their platforms.

Part II

Act

Chapter 5

Evaluation

Our evaluation focuses on the following research questions(as introduced earlier in the Section 1):

- **RQ1:** How many syntactically different but correct programs can be translated by QCross's converter?
- **RQ2:** What has QCross found via cross-platform testing of the widely-used QSSes? i.e., how many warnings and errors does QCross produce?
- **RQ3:** How does QCross compare to prior work on testing quantum computing platforms?
- **RQ4:** How useful is bloqs, the custom gates library?

5.1 Tools and Testbed

QCross is implemented using Python 3.9 and tested on the latest versions of various platforms at the time of performing the evaluation. The exact versions of these platforms are specified in Table 5.1. The QCross implementation extends and uses MoprhQ for important tasks such as Qiskit program generation, and output divergence comparison.

All experiments were run on an Apple M1 Pro 14-inch (2021 model) machine. It has ten cores (eight high-performance and two energy efficient), and 16 GB RAM. The OS at the time of evaluation was MacOS Ventura 13.3.1 (22E261).

To begin the comprehensive cross-platform testing methodology, QCross requires Qiskit program pairs to translate and execute. This can be provided in two ways: (i) Over a two-day period, MorphQ generated more than 50 000 Qiskit source and follow-up programs as part of their testing process and made them freely accessible as supplementary material. QCross can leverage these programs to expedite the testing procedure. (ii) Alternatively, QCross can employ MorphQ's random program

Name	Version	Release Date
python	3.9	Oct. 2020
qiskit	0.41.0	Jan 31 2023
qiskit-terra	0.23.2	Feb 23
qiskit-qasm3-import	0.1.0	Nov 10 2022
cirq	1.1.0	Dec 21 2022
PyQuil	3.3.3	Jan 2023
quilc	1.23.0 [e6c0939]	Oct 27 2021
qvm	1.17.1 [cf3f91f]	Jun 24 2021

Table 5.1: Tools and their versions as part of QCross's evaluation

generation capabilities to create new random pairs of Qiskit programs, which can then be passed to QCross for translation.

Throughout QCross's evaluation, a combination of existing and newly generated programs was utilized. The pre-existing MorphQ test data was employed to save time on program generation and to facilitate a regression test on Qiskit, as MorphQ had used Qiskit 0.19.0 while our implementation utilized Qiskit 0.41.0. Furthermore, since Cirq and PyQuil platforms exhibit similarities in construction and structure, it is plausible that programs which exposed flaws in Qiskit might reveal identical issues in these platforms as well. Additionally, new programs were generated because the original MorphQ's Qiskit gate-set was expanded by incorporating these extra newly available gates: **`CCZGate`**, **`CSGate`**, **`CSdgGate`**, **`RGate`**, and **`RVGate`**. Consequently, the newly generated programs can take advantage of a broader gate-set.

5.2 Distribution Difference of Program Outputs

During the execution of a program pair, two potential outcomes are possible: either both programs successfully execute and their output divergences are measured, or one of the programs experiences a crash. These outcomes are referred to as "output differences" and "crash differences," respectively. When examining these differences, it becomes apparent that crash differences typically indicate genuine issues within the tested platform, while output differences often correspond to false positives. The false positives arise from statistical tests misreporting differing distributions even when they are same, which is anticipated to occur in a small percentage of all cases.

As the manual inspection of differences and discerning their root causes necessitate considerable human effort, our detailed evaluation is primarily focused on crash differences. Identifying an effective method for detecting distribution differences that are likely true positives constitutes an intriguing avenue for future research, which could then be readily integrated into the existing methodology.

To further substantiate our emphasis on crash differences, we reference the findings of QDiff [41] and MorphQ [33], the most closely related existing works. Out of

	Qiskit		Cirq		PyQuil	
	#	%	#	%	#	%
Tested program pairs	1594	100%	1594	100%	1594	100%
Metamorphic Testing						
↪ Source program crash	0	0%	0	0%	214[a]	13.42%
↪ Followup program crash	430	26.97%	115	7.21%	437	27.41%
↪ Successful executions	1164	73.02%	1479	92.78%	1157	72.58%
↪ Distribution differences	40	3.43%	18	1.21%	33	2.85%
Differential Testing						
↪ Crashes in Unitary Testing[b]	49	40.83%	9	7.5%	110[c]	91.66%
	Tested		**Crashed**		**Divergent**	
↪ Cirq Qiskit Roundtrip	398	100%	32	8.04%	9	2.26%
↪ All sources comparison	1380	100%	-	-	160	11.59%
↪ All followups comparison	899	100%	-	-	144	16.01%

[a] Mainly due to **`quilc`** timing out.
[b] Only ∼10% (120 total) of total tested programs.
[c] High due to a PyQuil bug where it can't compute unitary of custom gates.

Table 5.2: Distribution of warnings and crashes produced by QCross

the 33 divergent cases identified by QDiff, the authors were only able to detect and report bugs for four crash differences. All divergent cases resulting from distribution differences were ascribed to potential hardware instabilities, a common occurrence given the nascent stage of real quantum computers. MorphQ exclusively concentrated on crash differences for the same rationale, and we believe this approach to be valid. Consequently, by focusing solely on crash differences, we are adhering to the frameworks established by previous works.

5.3 RQ1: Programs translated by QCross

Over the span of a week, the QCross framework was executed multiple times in batches. More than 1000 Qiskit programs were converted to Cirq and PyQuil. The summary of our executions is presented in Table 5.2. All of the Qiskit circuits were successfully translated and executed. We did not record any crashes during the translation process. However, when executing, many follow-up programs crashed. These were recorded as part of manual bug analysis. Source programs of PyQuil also crashed due to bugs or lack of support of certain features in these platforms as explained later in Section in 5.4.

Answer: The program converter successfully translates only valid quantum programs, ensures that any possible metamorphic relation is maintained, and QCross is effective in producing numerous warnings and crashes in follow-up programs of different platforms when executed.

5.4 RQ2: Real-world bugs

Once the testing process is finished, all recorded crashes were clustered by error message into common groups after removing program-specific information such as line number, memory location, argument values etc. The clustering process was automatic and utilized fuzzy-matching of strings. For example, **`'46,7: type error'`** and **`'32,9: type error'`** were assigned to the same cluster. Once the groups were created, we began the manual analysis by selecting one program of each group. Manual analysis was feasible since after clustering we had less than 20 groups to check. We ran the selected program to reproduce the crash. Once the crash was reproduced, we reduced the program to minimal operations which exhibited the crash. Consequently, a GitHub issue was filed in the relevant repository.

Tables 5.4, 5.3, and 5.5 present the outcomes of our manual examination for Qiskit, Cirq, and PyQuil, respectively. For each crash, we provide the reference to the bug report[1], its current status, whether it was a new or duplicate bug report, the crash message, and the necessary causes (metamorphic relations, differential testing) to trigger the bug.

In the course of this investigation, we have uncovered a total of 14 bugs and identified 2 potential issues within the examined quantum programming platforms. It is important to note that there may be an overlap between the Qiskit bugs discovered by MorphQ and QCross, as both employ the same Qiskit program pairs during the evaluation process. To prevent redundancy, the study ensures that any bugs previously identified by MorphQ are not included in our discovery set. Consequently, all bugs presented in the subsequent tables represent novel findings in relation to the MorphQ analysis. The largest number of bugs was detected in Qiskit, followed by PyQuil, and then Cirq. The prevalence of bugs in Qiskit can likely be attributed to factors such as its considerable size, maturity, and extensive feature set. Furthermore, metamorphic testing proved to be a more effective method for bug discovery, accounting for 10 out of the 14 detected bugs, compared to differential testing, which identified only 2 out of the 14 bugs. The remaining 2 PyQuil bugs were observed as the platform's inability to execute specific circuits, independent of any transformations.

As of the present date, developers have verified the authenticity of ten of these bug reports. Among these, five have been confirmed as novel bugs discovered through our testing methodology. The remaining bugs have been reported, and their novelty is pending confirmation by the platform maintainers. Subsequent sections of this paper provide an in-depth analysis and several illustrative examples of the encountered

[1]The GitHub issue number

Table 5.3: Bugs in Google Cirq

ID	Report	Status	Novelty	Crash Message	Causes
1	#5959	Confirmed	New	Incorrect unitary of controlled QasmUGate?	**Diff. Test.**: Unitary Check
2	#5985	Confirmed	New[a]	Support for Symbols in QasmUGate?	**Met. Test.**: Inject Parameters

[a] We filed related issues to this bug such as #5984.

crashes across the evaluated platforms.

5.4.1 Cirq

We discovered several problems in Cirq. After consulting with the maintainers, some of these issues were identified as false positives (refer to Section 5.4.4). Nonetheless, we uncovered two new instances of bugs mentioned in Table 5.3. The first issue involved an incorrect unitary in the controlled variant of **`QasmUGate`**. The second issue pertained to the absence of symbolic parameter propagation in **`QasmUGate`** as well. The root cause of these issues in **`QasmUGate`** stems from its non-native status, as it is a compatibility gate provided by Cirq to interface with Qiskit QASM code.

5.4.2 Qiskit

We identified 10 additional bugs that were not part of MorphQ's result set, summarized in Table 5.4. However, since bugs #1, #2, and #3 are the same bugs with different manifestations, we club them as one bug. Therefore, we have 8 additional bugs. The maintainers of the platform have confirmed all of these findings as genuine. The first four bugs (numbers 1 - 4) are regression-related, while the subsequent five (numbers 5 - 10) are bugs uncovered due to the augmentation of the MorphQ gate set and the inclusion of a QASM 3 round-trip. Three of those bugs are novel bugs. The final two issues (numbers 11-12) arise from the absence of support for specific Qiskit features in QASM 3. Since QASM 3 is currently in beta, these cannot be classified as bugs. Nevertheless, the testing procedure sparked a discussion regarding the potential inclusion of these features in the exporter. The corresponding discussion can be found in the attached GitHub issues.

Maximum allowed dimension exceeded: (Bug #1) Transpiling a circuit with level 2 or 3 can create matrices too large for the system to handle. The OS may kill the program as well. This bug happened due to the following code snippet in Qiskit's **`commutation_checker.py`** file:

Table 5.4: Bugs and issues found by QCross in IBM Qiskit

ID	Report	Status	Novelty	Crash Message	Cause
1	#9197	Confirmed	Duplicate[c]	Maximum allowed dimension exceeded	Inject parameters, Change of opt. level
2	#9197	Confirmed	Duplicate[c]	array is too big; arr.size * arr.dtype.itemsize	Inject parameters, Change of opt. level
3	#9693	Confirmed	Duplicate[c]	Excessive memory usage while transpiling at optimization level 2	Change of opt. level
4	#9627	Confirmed	Duplicate	Parameter is not defined in the current context	Inject parameters
5	#9559	Confirmed	Duplicate	Cannot find gate definition for 'csdg'	QASM 2 Roundtrip
6	#9721	Confirmed	Duplicate	Cannot find gate definition for 'ccz'	QASM 2 Roundtrip
7	#9722	Confirmed	Duplicate	Cannot find gate definition for 'cs'	QASM 2 Roundtrip
8	#10059	Confirmed	New[b]	Duplicate declaration for gate 'r'	Inject null-effect operations and QASM 2 Roundtrip
9	#10060	Confirmed	New[b]	Duplicate declaration for gate 'rv'	Inject null-effect operations and QASM 2 Roundtrip
10	#6	Confirmed	New	type error	QASM 3 roundtrip
11	#9609	Confirmed	No support[a]	name '$*' is not defined in this scope	QASM 3 roundtrip
12	#8	Confirmed	No support[a]	Node of type "SubroutineDefinition" is not supported	QASM 3 roundtrip

[a] QASM3 Converter is in beta as of this writing.
[b] The problem of duplicate declarations was known in general. However, specific instances of 'r' and 'rv' gate issues were new.
[c] Even though these three appear as three bugs, the underlying cause is the same. Hence, we assume this to be a single bug.

```
def _identity_op(num_qubits):
 """Cached identity matrix"""
   return Operator(
     np.eye(2**num_qubits), input_dims=(2,) * num_qubits,
     output_dims=(2,) * num_qubits
   )

# _identity_op invoked later in the code
extra_qarg2 = num_qubits - len(qarg1)
if extra_qarg2:
    id_op = _identity_op(2**extra_qarg2)
```

The **`_identity_op`** function accepts number of qubits. However, it was actually called with **`2 ** extra_qarg2`** as shown in the code above. This created an identity matrix with **`2 ** extra_qarg2`** rows when we just needed **`extra_qarg2`** rows, thereby causing excessive memory usage. The fix was to remove the **`2 **`** part and just call it with **`extra_qarg2`**.

5.4.3 PyQuil

We submitted four issues to PyQuil's GitHub (**`rigetti/PyQuil`**) repository. Unfortunately, at the time of writing, we have not received any confirmation regarding the novelty of these issues from the maintainers. Table 5.5 enumerates all the reports. Two of the issues (#2 and #4) appear to be caused due to PyQuil's inability to execute certain programs, irrespective of any transformations. We assume these to be source crashes, as opposed to follow-up crashes where the cause is generally a metamorphic transformation or differential testing. Therefore, we label the "Cause" columns for these issues as

Table 5.5: Bugs in Rigetti PyQuil

ID	Report	Status	Novelty	Crash Message	Cause
1	#1523	Reported	-	KeyError: 'SQRT-X' (custom gates)	**DT:** Unitary
2	#1524	Reported	-	Too many SWAP instructions selected in a row	*NA*
3	#1530	Reported	-	Condition CL-QUIL::COMPILER-DOES-NOT-APPLY was signalled	Inject Parameters
4	#1532	Reported	-	assertion failed: state update should occur from waiters' queue	*NA*

[*] DT is Differential Testing

Table 5.6: False Positives

ID	Report	Crash Message	Platform
1	#5710	Cannot represent circuits with unbound parameters in OpenQASM 2.	Qiskit
2	#7641	Cannot unroll the circuit to the given basis	Qiskit
3	#9607	Unable to map source basis	Qiskit
4	#5986	Different output for the same seed in Simulator?	Cirq

NA.

Bugs: The first bug, hereby referred to as Bug #1, is triggered during the computation of a unitary matrix belonging to a quantum circuit that incorporates a custom gate. This issue seems to emerge from the complex interactions between user-defined gates and the underlying matrix computation algorithm in PyQuil. The second anomaly, or Bug #2, manifests when executing a quantum circuit that incorporates a multi-qubit gate on a simulator or physical device that lacks direct connections to all qubit arguments of the said multi-qubit gate. In such cases, a SWAP gate is generally required. For example, when using a gate such as CNOT on qubits (control and target qubits) that are not directly connected, a SWAP is needed. However, an abnormal behavior has been observed with PyQuil, wherein it generates an excessive number of SWAP gates, occasionally exceeding 1000, for specific quantum programs. The cause of this over-generation is not yet clear and warrants further investigation.

Bug #3 is revealed when a custom gate is employed in conjunction with a PyQuil **`Parameter`** value. The PyQuil compiler, quilc, appears to encounter difficulties when attempting to map **`Parameter`** values back to gate arguments, specifically in the case of user-defined custom gates. This limitation hampers the flexibility of custom gate usage and can pose significant obstacles for complex quantum algorithms. The fourth anomaly, Bug #4, is a sporadic occurrence, with a higher likelihood of manifestation during the execution of large quantum circuits repetitively using PyQuil's Quantum Virtual Machine (QVM). Under these conditions, the QVM exhibits a tendency to become unresponsive, displaying a specific error message and necessitating a restart of the QVM process. This unpredictable behavior presents a significant hurdle to the stability and reliability of quantum simulations.

In terms of usability, we found PyQuil to be the slowest and least feature-rich platform among the three evaluated platforms. The external **`quilc`** compiler frequently froze following a program crash, rendering further testing of subsequent programs infeasible without terminating and restarting the process. Furthermore, PyQuil often required over 30 seconds to complete the simulation of programs that took only 1 second on Qiskit and Cirq. In fact, some programs exceeded the sixty-second **`timeout`** argument, ultimately timing out with no result.

5.4.4 False Positives

We identify certain crashes as false positives. These occurrences may not necessarily indicate bugs; rather, they could stem from inherent limitations within a particular component or from situations where certain metamorphic relations' assumptions are not upheld. For instance, the error message `Cannot represent circuits with unbound parameters in OpenQASM 2` arises from the OpenQASM 2 exporter's inability (by design) to represent unbounded parameters. QASM 3, however, does have this ability. Another example is when the following circuit is transpiled:

```
qc.append(CHGate(), qargs=[qr[1], qr[0]], cargs=[])
qc.append(CCXGate(), qargs=[qr[5], qr[9], qr[7]], cargs=[])

from qiskit import transpile
qc = transpile(qc, basis_gates=['ccx', 'h'])
```

We encounter the following error when executing the above circuit: `Unable to map source basis {('ccx', 3), ('ch', 2)}`. It is because even though the mapping of **`CCX`** into the basis **`[ccx, h]`** is trivial, but it may not be possible to construct a controlled **`H`** using just those two. At the very least, it will require special synthesis techniques that temporarily use ancilla qubits, since **`CCX`** is a 3-qubit gate while **`CH`** is only 2-qubit gate. This is an example where the "Change of gateset" metamorphic relation does not hold. The transformation assumes that that any circuit can be transformed into an equivalent circuit that uses only gates inside one of the universal gate sets. However, in practice, exploring all possible sequences to find an equivalent sequence of gates is computationally expensive and impractical.

In a final example of a false positive from the Cirq platform, it is anticipated that when a circuit is simulated with an identical **`seed`** value, the resulting numerical output should be precisely the same as any previous run. This expectation emulates the seeding process of random number generators, where the same seed produces an identical set of random values. However, one pair of circuits, which were metamorphically linked through the Inject Null-Effect Transformation, produced different outputs despite having the same seed value. The Cirq maintainers provided clarification on this matter:

> ... by default, **`cirq.Simulator`** has a **`split_untangled_states`** flag set to **`True`**, which tries to optimize the simulation of circuits which can be split into un-entangled states. ... This results in different number of calls to the random number generator and therefore a slight variance in the output results.

5.4.5 Additional bugs and issues

While developing QCross and the bloqs library, we identified few incorrect code comments, documentation issues, and unexpected behaviour of certain functions.

Table 5.7: Bugs and issues found during development

ID	Report	Status	Novelty	Crash Message	Platform
1	#8990	Confirmed	New[a]	Possibly incorrect docstring in DCXGate	Qiskit
2	#9003	Reported	-	Possibly incorrect docstring in C4XGate	Qiskit
3	#5928	Confirmed	New	Different outputs for U3Gate and QasmUGate	Cirq
4	#6016	Confirmed	New	_qasm_() got an unexpected keyword argument 'qubits'	Cirq

[a] Fixed by the author

They are summarized in Table 5.7. Three of the four issues were confirmed by the maintainers.

Answer: QCross identified 14 bugs and 2 potential issues across quantum programming platforms, with Qiskit (8) having the most bugs, followed by PyQuil (4) and Cirq (2). Metamorphic testing proved more effective than differential testing, and ten of the reported bugs have been confirmed as novel by developers.

5.5 RQ3: Comparison with Prior Work

In our study, we primarily contrast our work with MorphQ [33], though some comparison can also be drawn with QDiff [41]. Both QDiff and MorphQ conducted their experiments within a fixed time budget of two days. QDiff utilized a pre-written set of programs, while MorphQ generated random programs. In contrast, QCross was executed in multiple batches over a seven-day period, using random programs generated by MorphQ.

Furthermore, QDiff incorporated quantum hardware in their analysis, as opposed to relying solely on simulation, which is absent in both MorphQ and our own analysis. We consider our work to be a successor and an extension of the research initiated by MorphQ, which exclusively evaluated the Qiskit platform. Without QCross, bugs discovered in PyQuil and Cirq would not have been identified as part of MorphQ's investigation. Additionally, the inclusion of extra gates and QASM 3 roundtrip further exposed more Qiskit-related bugs.

Answer: We regard QCross as an "evolutionary successor" to MorphQ and QDiff. It has uncovered new bugs that were not detected in the previous works, demonstrating its complementary value to the existing research.

5.6 RQ4: Utility of bloqs

In order to address the limitation in the availability of native gate sets of quantum programming platforms Cirq and PyQuil, as compared to the more extensive gate set available in Qiskit, we have created the bloqs library. This important library serves as a crucial tool for enabling the successful translation of Qiskit programs to both Cirq and PyQuil, ensuring gate compatibility across these platforms. Significantly, our analysis reveals that approximately 94% of all translated programs necessitated the utilization of the bloqs library, highlighting its indispensable role in this process.

For this research question, a Qiskit quantum program translation is considered to be independent of bloqs if all the gates within the program can be effectively mapped onto the pre-existing and readily available gate sets within the PyQuil or Cirq quantum computing frameworks. This implies that the program can function adequately without any need for additional or custom gate constructions.

Answer: The bloqs library was developed to overcome gate set limitations in Cirq and PyQuil, enabling successful translation of Qiskit programs. The library proved essential, as it was required in approximately 94% of translations.

5.7 Threats to validity

Several factors pose potential threats to the validity of our results and the conclusions drawn from them. Firstly, the non-deterministic and randomized nature of the MorphQ program generator and the selection of transformations could impact the outcomes. In fuzz testing, prolonged experimentation periods typically reveal more errors or novel program execution paths, as indicated by [9]. We address this concern by conducting long-running experiments, compensating for any bias that may emerge with a limited number of generated programs. Consequently, our experiments spanned a total of seven days.

Secondly, the dependency on the bloqs library might introduce inadvertent errors unrelated to the platforms under investigation. To mitigate this risk, we rigorously tested the bloqs library on a gate-unitary basis. All gates constructed for Cirq were evaluated on a unitary basis, while the majority of PyQuil gates were assessed using unitary and state-vector comparison techniques. Nevertheless, a few gates remain untested due to a bug in PyQuil that hinders the generation of unitary matrices for custom gates.

In the process of program translation, particularly with respect to metamorphic relations, multiple approaches may yield equivalent results. In order to mitigate potential unpredictability, our selection criterion for application programming interfaces (APIs) and techniques is based on the highest degree of similarity to the Qiskit API.

Concerning the verification of output equivalence, the Kolmogorov-Smirnov (KS) test was utilized in our study, which is based on [33]. It should be noted that alternative methodologies, such as the cross-entropy test, may result in varying quantities of divergent cases, either reducing or increasing the overall count.

In conclusion, the methodology delineated in this research is anticipated to be transferable to alternative platforms such as ProjectQ or pytket. Nevertheless, as observed during the translation of metamorphic relations, certain transformations may prove infeasible on specific platforms. Consequently, we refrain from making generalized assertions concerning all quantum platforms. Despite these limitations, it is our belief that the techniques presented herein should prove beneficial in the evaluation of other circuit-and-gate based quantum platforms in a similar manner.

Part III

Assert

Chapter 6

Discussion

The evaluation of QCross demonstrates its effectiveness in addressing the research questions posed, offering important insights into the abilities and limitations of quantum programming platforms. The following discussion highlights the key observations and implications of our study.

Firstly, QCross successfully translates a large number of syntactically different but correct programs between Qiskit, Cirq, and PyQuil platforms. This is largely facilitated by the **`bloqs`** library, which bridges the gap in gate compatibility between these platforms. Our analysis indicates that bloqs plays an indispensable role, being required in approximately 94% of all translated programs. The seamless translation of programs enables comprehensive cross-platform testing, which helps in identifying bugs and other issues that might otherwise remain undetected.

Our evaluation also reveals that metamorphic testing is more effective in uncovering bugs than differential testing. Out of the 14 bugs discovered, 10 were identified using metamorphic testing, while only 2 were found through differential testing. This suggests that future research should continue to explore and refine metamorphic testing techniques to further enhance the bug detection process.

Additionally, the evaluation results point towards the presence of a relatively larger number of bugs in Qiskit compared to Cirq and PyQuil. This can be attributed to the platform's extensive feature set, maturity, and size. However, it is important to note that the maintainers of these platforms have been responsive in addressing the reported issues, confirming the authenticity of 10 out of the 14 bug reports filed.

An interesting observation from the evaluation is the difference in performance between the platforms. PyQuil was found to be the slowest and least feature-rich platform among the three, with some programs taking significantly longer to simulate than on Qiskit and Cirq. This information may be valuable to developers and researchers when selecting a quantum programming platform for their projects.

Lastly, our study highlights the importance of long-running experiments in fuzz testing. By conducting experiments over a seven-day period, we were able to uncover a more comprehensive set of bugs and reduce potential biases introduced by a limited number of generated programs.

In conclusion, the evaluation of QCross presents valuable insights into the strengths and weaknesses of the quantum programming platforms Qiskit, Cirq, and PyQuil. The custom bloqs library, metamorphic testing techniques, and the emphasis on long-running experiments all contribute to the effectiveness of the QCross framework in identifying and reporting bugs. The outcomes of this study can guide the development and improvement of quantum programming platforms, ultimately benefiting the broader quantum computing research community.

Chapter 7

Conclusion and Future Works

Quantum computing has emerged as a promising computing paradigm with significant advantages over classical computing. In response to the growing popularity of quantum computing and the limited techniques for testing its software stack, this thesis introduces the cross-platform testing approach for testing quantum computing platforms, with two key contributions: a program translator that translates a diverse set of non-crashing Qiskit quantum programs into PyQuil and Cirq, and a library, bloqs, to fill in the gaps between multiple platforms in terms of quantum gates.

Our evaluation demonstrates QCross's effectiveness, such as the detection of 8 bugs in Qiskit, 2 in Cirq and 4 in PyQuil, and we foresee that our contributions will enable future work beyond QCross by utilizing either quantum hardware or testing on other platforms. For instance, the program translator could serve as a starting point for translating to other platforms such as Q# and ProjectQ. Even though QCross is not the first to re-invent differential testing in a quantum setting or establish quantum metamorphic transformations, it is the first to merge differential with metamorphic testing in an effort to perform cross-platform testing. Furthermore, we also, for the first time, concretely implement metamorphic transformation in the three tested platforms and provided a methodologies for differential testing such as unitary checking, and Cirq-Qiskit-Cirq round-trip. Overall, this work represents an important step towards enhancing the reliability of software in the nascent field of quantum computing.

Given the nascent stage of the field, we believe there is a huge potential for substantial work to be done. In no specific order, we list a few things for the future:

- Generate better random programs that utilize more APIs of the platforms to increase the coverage area and find potential integration faults. Ensure the generated programs are more "real-world" programs.
- Extend the number of tested platforms to include non-python platforms as well.
- Execute the programs on quantum hardware to establish a source of truth or to find bugs in the hardware.

- Devise a method to better test the divergence of program outputs such that false positives are minimized.
- Analyse the existing divergent programs to find out the source of divergence.

Bibliography

[1] Ali J Abhari et al. *Scaffold: Quantum programming language*. Tech. rep. Princeton Univ NJ Dept of Computer Science, 2012.

[2] Rui Abreu et al. 'Metamorphic Testing of Oracle Quantum Programs'. In: *2022 IEEE/ACM 3rd International Workshop on Quantum Software Engineering (Q-SE)*. IEEE. 2022, pp. 16–23.

[3] Shaukat Ali et al. '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.

[4] Michael Nielsen Andy Matuschak. *Quantum computing for the very curious*. Accessed: 2022-06-30. URL: https://quantum.country/qcvc.

[5] *Awesome List of Quantum Software*. Accessed: 2022-06-30. URL: https://github.com/qosf/awesome-quantum-software.

[6] Nader Boushehrinejadmoradi et al. 'Testing cross-platform mobile app development frameworks (t)'. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 441–451.

[7] Chad Brubaker et al. 'Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations'. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 114–129.

[8] Chu Chen et al. 'Rfc-directed differential testing of certificate validation in ssl/tls implementations'. In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 859–870.

[9] Junjie Chen et al. 'A survey of compiler testing'. In: *ACM Computing Surveys (CSUR)* 53.1 (2020), pp. 1–36.

[10] Tsong Yueh Chen et al. 'Metamorphic testing: A review of challenges and opportunities'. In: *ACM Computing Surveys (CSUR)* 51.1 (2018), pp. 1–27.

[11] Yuting Chen et al. 'Coverage-directed differential testing of JVM implementations'. In: *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 85–99.

[12] Isaac L. Chuang, Neil A. Gershenfeld and Mark Kubinec. 'Experimental Implementation of Fast Quantum Searching'. In: *Physical Review Letters* 80 (1998), pp. 3408–3411.

[13] *Circuit Library*. 2023. URL: https://qiskit.org/documentation/apidoc/circuit_library.html.

[14] Andrew Cross et al. 'OpenQASM 3: A broader and deeper quantum assembly language'. In: *ACM Transactions on Quantum Computing* 3.3 (2022), pp. 1–50.

[15] Andrew W Cross et al. 'Open quantum assembly language'. In: *arXiv preprint arXiv:1707.03429* (2017).

[16] Ronald De Wolf. 'The potential impact of quantum computers on society'. In: *Ethics and Information Technology* 19.4 (2017), pp. 271–276.

[17] Alastair F Donaldson et al. 'Automated testing of graphics shader compilers'. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–29.

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

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

[20] Qiskit community IBM Research. *Qiskit: An Open-source Framework for Quantum Computing*. 2021. DOI: 10.5281/zenodo.2573505.

[21] Eric R Johnston, Nic Harrigan and Mercedes Gimeno-Segovia. *Programming Quantum Computers: essential algorithms and code samples*. O'Reilly Media, 2019.

[22] Robert Koenig and John A. Smolin. 'How to efficiently select an arbitrary Clifford group element'. In: *Journal of Mathematical Physics* 55.12 (Dec. 2014), p. 122202. DOI: 10.1063/1.4903507. URL: https://doi.org/10.1063%5C%2F1.4903507.

[23] Andrej N Kolmogorov. 'Sulla determinazione empirica di una legge didistribuzione'. In: *Giorn Dell'inst Ital Degli Att* 4 (1933), pp. 89–91.

[24] Ryan LaRose. 'Overview and comparison of gate level quantum software platforms'. In: *Quantum* 3 (2019), p. 130.

[25] Vu Le, Mehrdad Afshari and Zhendong Su. 'Compiler validation via equivalence modulo inputs'. In: *ACM Sigplan Notices* 49.6 (2014), pp. 216–226.

[26] *List of QC simulators*. Accessed: 2022-06-30. URL: https://quantiki.org/wiki/list-qc-simulators.

[27] Ji Liu, Gregory T Byrd and Huiyang Zhou. 'Quantum circuits for dynamic runtime assertions in quantum computation'. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1017–1030.

[28] Junjie Luo et al. ‘A comprehensive study of bug fixes in quantum programs’. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 1239–1246.

[29] William M McKeeman. ‘Differential testing for software’. In: *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[30] Eñaut Mendiluze et al. ‘Muskit: A mutation analysis tool for quantum software testing’. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 1266–1270.

[31] Michael A Nielsen and Isaac L Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.

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

[33] Matteo Paltenghi and Michael Pradel. ‘MorphQ: Metamorphic Testing of Quantum Computing Platforms’. In: *arXiv preprint arXiv:2206.01111* (2022).

[34] *Qiskit documentation*. Accessed: 2022-06-30. URL: https://qiskit.org/documentation/.

[35] *Quantum computing history and background*. Accessed: 2022-06-30. URL: https://docs.microsoft.com/en-us/azure/quantum/concepts-overview.

[36] Maria Schuld, Ilya Sinayskiy and Francesco Petruccione. ‘An introduction to quantum machine learning’. In: *Contemporary Physics* 56.2 (2015), pp. 172–185.

[37] Robert S. Smith, Michael J. Curtis and William J. Zeng. *A Practical Quantum Instruction Set Architecture*. 2016. arXiv: 1608.03355 `[quant-ph]`.

[38] Balwinder Sodhi and Ritu Kapur. ‘Quantum computing platforms: assessing the impact on quality attributes and sdlc activities’. In: *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. IEEE. 2021, pp. 80–91.

[39] Sandro Tolksdorf, Daniel Lehmann and Michael Pradel. ‘Interactive metamorphic testing of debuggers’. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 273–283.

[40] Jiyuan Wang, Fucheng Ma and Yu Jiang. ‘Poster: Fuzz testing of quantum program’. In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2021, pp. 466–469.

[41] Jiyuan Wang et al. ‘QDiff: Differential Testing of Quantum Software Stacks’. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 692–704.

[42] Xinyi Wang et al. ‘Application of combinatorial testing to quantum programs’. In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2021, pp. 179–188.

[43] Colin P Williams and Colin P Williams. ‘Quantum gates’. In: *Explorations in Quantum Computing* (2011), pp. 51–122.

[44] T.G. Wong. *Introduction to Classical and Quantum Computing*. Rooted Groove., 2022. URL: https://books.google.no/books?id=WBvtzgEACAAJ.

[45] Tianyi Zhang and Miryung Kim. 'Automated transplantation and differential testing for clones'. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 665–676.