Testing Framework For Networked Quantum Applications

Ravisankar Ashok Kumar Vattekkat

Technische Universiteit Delft

TU Delft

Testing Framework For Networked Quantum Applications

by

Ravisankar Ashok Kumar Vattekkat

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Friday July 2, 2021 at 10:30 AM.

Student number:	5136407	
Project duration:	October 1, 2020 – June 1, 2021	
Thesis committee:	Prof. dr. ir. A. van Deursen,	TU Delft, supervisor
	Prof. dr. ir. S. Wehner,	TU Delft, supervisor
	Dr. Przemysław Pawełczak,	TU Delft
Daily Supervisor:	Dr. Wojciech Kozlowski,	TU Delft

This thesis is confidential and cannot be made public until January 1, 2022.

TU Delft

Abstract

Quantum software development is the process of conceiving, specifying, designing, programming, documenting, and testing executable quantum programs that are meant to run on practical quantum hardware. Even though quantum software development research has gained traction over the years, it is still mainly focused on problem analysis, language design, and implementation. Software testing, which is the process of executing a program or application with the intent of finding faults, and verifying that the software product is fit for use is yet to receive substantial attention in quantum software development. This work answers the questions of to what extent are classical software testing techniques transferable to quantum programs, how effectively can they be used, what would an application independent theory for testing quantum software look like, and how practical is it with the current state of physical hardware. Using the principles of spectrum based fault localization, we show that we are able to properly detect and localize bugs in the state teleportation application and blind quantum computing application, even with the presence of noise in the hardware.

Preface

It is with immense pleasure that I get to present this thesis which has been the greatest accomplishment of my academic life. Spanning over 9 months, counting over 100 pages, and a bucket load of online meetings with people from all around the world, this has been without a doubt the highlight of my young research career. Working in Stephanie's group was one of the main reasons why I wanted to join TU Delft, and I am extremely happy with all that has transpired since. It has not been lost on me that I get to work in a cutting-edge field amongst the best in the field, and I am truly grateful for their help along the way.

As part of this endeavour, I was lucky enough to be under the guidance of two greatly accomplished titans in Prof. Arie van Deursen, and Prof. Stephanie Wehner. My deepest gratitude towards them for giving me the opportunity to work in this topic and learn as I grow. Not only did they provide me with a proper sense of direction, but their invaluable feedback and reaffirmation of what I have been doing helped me strive towards a better result. Immense gratitude towards my daily supervisor Dr. Wojciech Kozlowski who has been exemplar, to say the least. I have learned so much on how to conduct research from him, and I will cherish our Monday morning meetings for years to come. I am also grateful to Dr. Przemysław Pawełczak for being part of the final thesis committee. I would also like to extend my regard to the QNodeOS group who has been gracious enough to help me out, especially Bart van der Vecht for his constant support for everything related to SquidASM.

And as always, to my parents Ashok and Sreelatha.

Ravisankar A V
Delft, June 2021

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

Quantum computation and quantum Internet propose significant theoretical advantages over their classical counterparts. Quantum technology is known to quickly solve problems that are *believed* to be slow on classical computers [43] [3] [20] and promises unconditional security through applications like quantum key distribution [44] [31]. Further success in practically realizing quantum technology will depend on the progress made in both the software and hardware branches. Although there has been a steady progress in the development of hardware for quantum computers for over 20 years [6] [52] [26], software development for practical quantum computers is fairly recent [45][14][28] [51], with the most commercially successful and actively developed projects dating back just under half a dozen years at the time of writing this thesis.

Quantum software development (QSD) is the process of conceiving, specifying, designing, programming, documenting, and testing executable quantum programs that are meant to run on practical quantum hardware. Even though QSD research has gained traction over the years [33][38][37][32], a historical roadmap of active software repositories shared online [19][9][2] shows us that it is still mainly focused on problem analysis, language design, and implementation. Software testing [1], which is the process of executing a program or application with the intent of finding faults, and verifying that the software product is fit for use is yet to receive substantial attention in quantum software development.

In general, software testing involves the execution of a software component to evaluate one or more properties of interest. These properties include, but are not limited to, the extent to which the component or system under test satisfies the requirements that were planned out in the design and development phase, whether the software behaves as expected for all input cases we are interested in, achieves the result in a suitable amount of time, and can be installed and run in its intended environments.

Theoretically speaking, the number of tests that can be conducted on a given piece of code could be infinite. Therefore, the onus is on the tester to strategically decide which tests are suitable and feasible for the limited available time and resources. As a result, software testing generally attempts to ”break” the application in any way possible with the intent of reducing the risk that the software system will not behave as intended in

a production environment.

As quantum technologies become more mainstream, it is crucial that we develop tests for the application that we want to run. Studies conducted on the importance of testing classical software show how much money we can save from testing and finding crucial faults early on in the development stage itself - with one estimate ranging from $22.2 to $59.5 billion [39]. Once quantum technology becomes mainstream, it important to not strain the already monumental economic cost even more by developing inadequate software. This thesis studies the gap left in the quantum software development cycle and takes a step towards realizing the goal of developing a software testing framework.

1.1. Motivation

It is important to develop a consensus on how to properly test a quantum application. Although methods exist to debug programs based on assertion techniques [24] [29], [30] and property-based techniques [23], they are limited in terms of either being highly input conditioned, or unable to work under practical noisy conditions. A detailed discussion on how the current testing techniques vary is given in Chapter 4. Since we are in the early stage of this field, no consensus on how to properly test quantum programs has been formulated yet. Hence, most of the testing is done ad hoc; either through brute-force calculation and comparing the results of different test cases or by manually going through the code and trying to find bugs.

Hence the goal of this thesis is to examine an alternate path, wherein we can devise a theoretically sound approach to detect and localize bugs in our code even in noisy practical hardware settings, irrespective of the initial configuration (for example, the input state of the qubit). Apart from the advantage of having an underlying theory that is platform and application-independent, it can also be used to test the usability of the hardware for our application in consideration. Such results would bring us one step closer to developing a complete software development approach for quantum applications.

The question of developing a test suite for quantum applications is also an interesting challenge from a software engineering theoretical point of view. Unlike in the classical software setting, quantum programs often do not have the luxury for breakpoints, nor can we inspect data objects whenever we want. This is because stopping execution midway and examining the state of variables will cause the qubits to decohere and inevitably ”collapse” the computation. We will be exploring more about this in Chapter 4.

Moreover, even though there exist benchmarking tests [50][47] that check whether or not the hardware components are up to the mark, we do not have a similar standard testing scheme for quantum software.

It is equally interesting to study how to test networked quantum applications. Standalone quantum applications do not tackle network issues and are not exposed to subroutines like entanglement generation calls [15], that require more than one quantum node. Thus, it is important to create testing schemes for networked quantum applications themselves.

1.2. Scope and Relevance

The main purpose of the testing framework that is devised for this thesis is to detect and localize bugs present in the application code. Even though the theory behind the testing framework can be transferred to other quantum programming stacks, the experiments and results discussed throughout this thesis work on the NetQASM SquidASM [49] stack which are used to run quantum programs using the Python scripting language. The applications that we conduct our tests on are what we would classify as near-term quantum network applications [40], and are feasible on the Noisy-Intermediate Scale Quantum (NISQ) [40] hardware. Particularly, the applications we are interested in are quantum state teleportation [34] and quantum blind computation [11].

Since we recognize that near-term quantum hardware is not completely error-free, we conduct the tests using hardware that have less than optimal entanglement fidelity and gate fidelity rates (Chapter 3). For the blind quantum computing application, we consider the noise produced due to host latency and instruction processing time. We do not consider any other noise than the ones mentioned here, and we assume that the corresponding components are perfect.

Since this is the first of many iterations for creating a software testing framework, we try to establish what works and what does not by applying a heuristic approach. This is done by running the testing methods that are available to us right now and listing out the shortcomings we faced, along with ideas on how to address them. We hope that this work gives a better sense of direction and paves a way forward for more research in quantum software testing.

1.3. Research questions

Now that we have set up the stage for the thesis, we discuss the problem statements we are going to tackle. Since the field of classical software testing has already enjoyed much success over the years due to rigorous research worldwide, we review the techniques that exist already so that we do not reinvent the wheel. The first research question is:

RQ1 : *To what extent are classical software testing techniques transferable to quantum programs, and how effectively can they be used?*

Once we have set a testing method, we must see that the theory behind it can be transferred to any application we would like to test and devise an architecture for it.

RQ2 : *What would an application independent quantum testing framework look like? What components does it have?*

After that, we find out how powerful our testing method actually is. Can it detect and localize bugs even in the presence of noise?

RQ3 : *How powerful is our testing frameworks? How many bugs can our testing framework detect, what kind of bugs, and how bad can our hardware get before we cannot properly detect and localize bugs?*

It is also important to evaluate our testing framework in relation to an already existing quantum software testing framework. Thus, the final research question we would like to ask is:

RQ4 : *Given a buggy application, can our testing framework detect and localize bugs better than the standard testing technique?*

Finally, we take a critical look at our testing method and examine its shortcomings. A thorough analysis of the results we got for each test is carried out. Possible solutions to unanswered questions and future areas to explore are discussed.

1.4. Main Results and Contributions

The main outcome of this thesis is the development of a novel framework to detect and localize bugs in quantum applications using the spectrum based fault localization technique. Using Bayesian reasoning, the test suite outputs the probability that each component has a bug associated with them. This form of fault localization using such a regression analysis is believed to be the first of its kind in the quantum software engineering domain and has been able to properly detect and localize bugs in state teleportation and blind quantum computing application better than the standard testing framework.

The main advantage of such a test framework is that the theory behind it is application-independent and works even in the presence of noise. As explained in Chapter 7, this framework was able to localize fault in the teleportation application even in noisy conditions. Moreover, all the four research questions presented above were answered.

1.5. Structure of Thesis

With regard to the research questions posed above, the thesis is divided into following chapters:

- ***Chapter 2*** *Quantum Information Background*: This chapter serves as the reference point for everything related to Quantum Information Science (QIS) that will be used throughout the thesis. The author assumes the reader has limited knowledge of QIS and hence explains the required basics. The different quantum applications used for testing are introduced in this chapter.
- ***Chapter 3*** *Quantum Software Engineering on Realistic Hardware*: This chapter provides a summary on the current state of quantum software engineering and a high level overview on what we can expect if we were to run a quantum program on realistic hardware. For this, we introduce the SquidASM environment using which we implement a blind quantum computing application that is run on a simulated model of realistic quantum hardware. This is done so as to get an intuition

about what it means to run a networked quantum application in a practical setting that has imperfect devices and noises. The chapter also studies how we can interpret the results we get from the simulation.

- ***Chapter 4*** *Current Methods for Testing Quantum Programs*: This chapter reviews the main software testing techniques that are used for classical programs. From these, we study whether they can be used for testing quantum programs. This section answers research question **RQ1**. Furthermore, a study on recently devised testing methods for quantum programs is also carried out.
- ***Chapter 5*** *Framework for Testing Quantum Application*: In this chapter, we create a testing framework for networked quantum applications. The requirements for the testing framework are listed out and from that, the architecture for the framework is discussed and the sub components are explained. The theory of spectrum based fault localization is explained in this chapter, as it is central to the framework. This section answers research question **RQ2**.
- ***Chapter 6*** *Testing Specific Quantum Programs*: The testing framework is put to test by running it on two specific quantum network applications - state teleportation, and blind quantum computing application. One or more bugs are introduced to each application's codebase and we see if the framework correctly detects them and localizes them to relevant part of the code. This section answers parts of research question **RQ3**.
- ***Chapter 7*** *Evaluation of Techniques and Testing Framework*: The testing framework is made to run of practical setting and studied if we are still able to correctly localize the bug. The framework is compared with the system level testing method. This section answers the remaining part of research question **RQ3** and **RQ4**.
- ***Chapter 8*** *Conclusion and Outlook*: In this chapter we summarize the key learnings by reflecting on the results presented in previous chapters. Answers to the research questions are also provided.

2

Quantum Information Background

This chapter provides relevant quantum information literature that is required to comprehend the rest of the thesis. It begins with a quick introduction to the theory of quantum information science and qubits, and then goes ahead to explain the relevant protocols that will be used in this thesis.

2.1. Principles of Quantum Information

A computer is a physical device that helps us process information by executing algorithms. An algorithm is a well-defined procedure with a finite description for realizing an information-processing task. The information presented to the algorithm is encoded in the hardware using the notion of bits - a two-state system represented by 0/1 (or OFF/ON or false/true). The idea of an algorithm is to manipulate these bits so that the result will represent the configuration we are looking for.

Physical realization of bits can be achieved through various methods - it could be, for example, done using voltage levels or electric switches. If the switch is in the OFF state, we represent the bit as state 0 and similarly, we represent the ON state as state 1. This implies that a given bit will always be in either one of the two logical states - 0 or 1. When we are given a set of 100 switches all promised to be in the same state, we can be sure that when we take a look at the state of each switch (henceforth referred to as measuring the bit), all of them will be in the ON state (bit 1) or all of them will be in the OFF state (bit 0). For the promised condition mentioned before, it will never be the case that some $x\%$ of the switches are in the ON state (bit 1) and the rest $(100 - x)\%$ switches are found in the OFF state (bit 0).

Quantum particles, like photons and electrons, can also be used to represent bits. We say that a photon is vertically polarized if it oscillates in the vertical axis, and we say it is horizontally polarized if it oscillates in the horizontal axis (Fig 2.1). We can use the polarization axis as a possible abstraction for bits: a horizontally polarized photon can be configured as bit 0 and a vertically polarized photon particle as bit 1. Thus, a **qubit** (or quantum bit) is the quantum mechanical analogue of a classical bit. Measuring the qubit thus means finding out which axis the photon is oscillating, and hence, figuring out the corresponding bit value it holds. If we are interested in finding out whether the photon is oscillating in the horizontal or the vertical axis, we term the process as "measuring a qubit in the computational basis".

Mathematically, we can express the qubits as a column vector. For example, qubit 0, expressed as $|0\rangle$ (pronounced ket 0) is commonly represented as $\begin{bmatrix}1\\0\end{bmatrix}$ and bit 1 is represented as $\begin{bmatrix}0\\1\end{bmatrix}$. Bit operations like the NOT gate are represented as a 2×2 matrix $X = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$. It is evident that $X|0\rangle = |1\rangle$ and $X|1\rangle = |0\rangle$.

Measuring a vertically polarized photon in the computational basis always shows us that the photon is vertically polarized, and hence the bit value is 1. The corresponding fact also applies for horizontally polarized photons, which gives us bit 0. However, this may not be true for every polarization configuration. Consider a photon that is diagonally polarized; that is, its axis of oscillation is diagonal to the horizontal and vertical axis (Fig 2.1). When we measure a set of N diagonally polarized photons in the standard computational basis, we find that half of them are horizontally polarized (bit 0) and the other half is vertically polarized (bit 1) - and we cannot know for sure which bit state it will be in beforehand itself. The only way to find that out is to measure it. This shows us that a diagonally polarized photon does not exist in one definite com-

putational bit state - it is a **superposition** of the two states. This is in stark contrast to the transistor switches we have in classical computing - no matter what measurement basis we have, a set of bits in the same configuration will always provide the same measurement result.

We can mathematically incorporate the concept of superposition using probability theory. A Qubit, thus can not only exist in one definite basis $|0\rangle$ or $|1\rangle$, but can also exist as a superposition $\alpha|0\rangle + \beta|1\rangle$. The values α, β are called the amplitudes of the basis state and the modulo squared of it gives us the probability that when measured, the qubit will be in that corresponding basis state. Thus for the example mentioned before, when we measure the qubit, it will be in state $|0\rangle$ with probability $|\alpha|^2$, and state $|1\rangle$ with probability $|\beta|^2$. Following the rules of probability, we also have the condition $|\alpha|^2 + |\beta|^2 = 1$. Thus for the diagonally polarized photon mentioned before, we can mathematically represent it as $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}$.

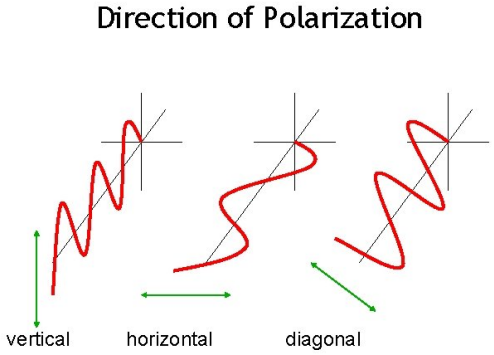


Figure 2.1: Direction of Polarization

When we have more than one qubit, we represent it as the tensor product of the individual qubit states. That is, for two qubits $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and $|\phi\rangle = \gamma|0\rangle + \delta|1\rangle$, the joint representation is:

$$|\psi\phi\rangle = |\psi\rangle \otimes |\phi\rangle = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle = \begin{bmatrix}\alpha\gamma\\ \alpha\delta\\ \beta\gamma\\ \beta\delta\end{bmatrix}$$

As we can see, a n qubit system spans 2^n space. Qubit systems under noisy conditions are better represented using the density operator notation. For an ensemble of quantum states $\{|\psi_i\rangle\}$ with $\{p_i\}$ being their corresponding probability, the density operator is:

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$$

The goal of a quantum computer is to use clever algorithms to manipulate a set of qubits to increase the net probability of the right state and decrease the corresponding value of the wrong state, so that once we measure the final system we will get the right

answer with a very high probability. That is, if we want our end result to be in the state $|1\rangle$, our quantum algorithm devises a series of gate operations that when applied to our qubit will try to drive the value of α towards 0 and β towards 1.

An important fact regarding the process of measurement is that it destroys the state we had before measuring. That is, if we get the measurement result of the diagonally polarized state as bit 0, corresponding measurement results *on that particular qubit in the computational basis* will always produce bit 0. The superposition is destroyed, and that particular qubit will now behave as a normal $|0\rangle$ qubit.

An equally important fact regarding measurement is that it is highly basis-dependent. The result we get after measuring in the computational basis might not be the same as measurement results in the diagonal basis. For example, as seen before, measuring $|+\rangle$ in the computational basis produces bit result 0 or 1 with equal probability. But measuring it in a basis (which we will call the diagonal basis) whose axis makes an angle of $+45$ degrees with the computational basis axis always produces the bit result corresponding to the positive eigenvector. This also implies that it's hard to distinguish two qubits by measuring just one basis alone. For example, $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ produce the same results in computational basis measurement, but we can distinguish them clearly if we were to measure them in the diagonal basis for measurement of $|+\rangle$ produces bit corresponding to eigenvalue of positive eigenvector and $|-\rangle$ produces bit corresponding to eigenvalue of negative eigenvector. To properly distinguish between two qubit states, we have to perform full tomography and calculate the fidelity from the corresponding density matrices. This is explained in Chapter 4.

2.2. Quantum Network and Communications

Qubits are not only useful for solving problems on a standalone quantum computer; they also have a wide range of applications in distributed quantum settings and security enhancements in the networking domain. The proof of concept examples used to demonstrate the power of quantum networks and communication are the key distribution protocols that are theoretically proved to be unconditionally secure [12], and networking applications like blind quantum computing [11] [7]. Quantum networks have also been used to test theoretical results, like the Loophole-free Bell inequality violation experiment [21].

One of the main techniques used to power such applications is the concept of quantum entanglement. Quantum entanglement is the phenomenon between more than one qubit that helps them establish a powerful correlation between them. Once the entanglement is established, the correlation remains intact even if the qubits are separated by large distances. A more detailed discussion on how entanglement is used for quantum computation is prescribed in [34].

Another important feature of qubits that helps us in the security aspect of quantum networking is the No-Cloning Theorem [34], which states that it is impossible to create a copy of a qubit whose state is unknown. We can transfer the state of a qubit from one to another, but it is impossible to create an identical copy of a qubit without destroying the first one (unless we know the state of the original qubit). This again is in stark

contrast with classical bits, where we can easily measure the bits midway through our execution and also copy them for further analysis. In quantum computing, we do not have the luxury to examine the state of the qubits without collapsing it, and neither can we create a copy of an unknown qubit state to examine it on the side.

2.3. Protocols and Algorithms

The principle of a quantum algorithm is to increase the probability of the correct answer state and decrease the probability of the incorrect answer state. This makes sure that once we measure our final qubit set, it will be in the state that we want our answer to be in with a very high probability.

The algorithm specifies the set of operations to be applied on the qubit set. The most common representation that describes which operations are to be executed when is through the circuit model. Circuits are networks composed of wires that carry qubit values to gates that perform elementary operations on the qubits. The circuits we consider will be acyclic, meaning that the bits linearly move through the circuit, and the wires never feedback to a prior location in the circuit. The operation of all applications that we will be discussing in this thesis will be specified using the circuit model.

2.3.1. Quantum State Teleportation

One of the applications that we will be testing is the quantum state teleportation protocol [34]. It is the protocol by which the state of a qubit is transferred from one qubit to another using an entangled pair and two classical bits of information. The circuit diagram for state teleportation protocol is shown in Fig 2.2. The idea of state teleportation is the following: the sender who has the top two qubits in the circuit would like to transfer the state of their first qubit to the receiver, who has the last qubit. In order to do this, the sender and receiver generate an entangled pair which is represented by the EPR box in the circuit. Once the entanglement is established, the sender applies the CNOT gate between their two qubits followed by an H gate on the first qubit and finally measures them in the computational basis. The measurement result is then sent to the receiver in the form of 2 classical bits. The receiver then applies the X and Z gates only if the corresponding bit value from the sender is set to 1. These operations reconstruct the original qubit that the sender had into the receiver's EPR qubit.

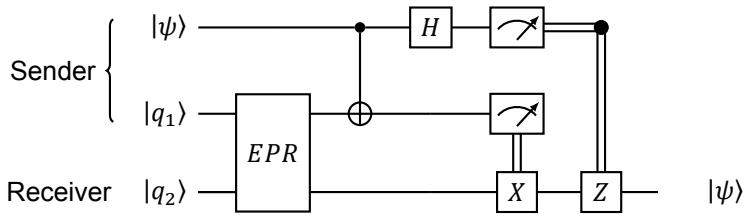


Figure 2.2: Quantum Circuit for State Teleportation

The EPR box generates the 2 qubit entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.The matrix repre-

sentation of the gates used in the protocol are given below:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

2.3.2. Blind Quantum Computation

Another quantum network application that is of interest is Blind Quantum Computation (BQC) [18]. The principle of BQC is to delegate a set of operations that a client requires on to another quantum computer (called the server) that can be accessed through quantum networking, along with the added feature that the server is blind to both the type of operation the client requires and the input/output configuration of the end result.

For example, the client requires the operation specified in Fig 2.3 to be executed in a different server [27].

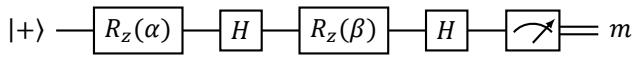


Figure 2.3: Blind Quantum Computation Effective Computation Circuit

The state $|+\rangle$ is represented in the computational basis as $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The matrix representation of the gates used in the protocol are given below:

$$R_z(\theta) = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix} \quad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The computation specified above can also be done with two qubits. The corresponding circuit for the above operation is as follows:

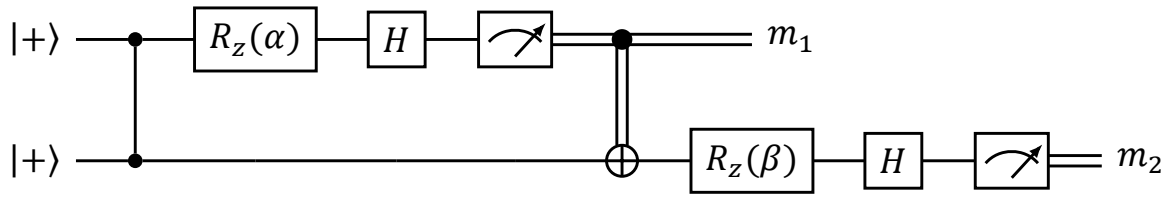


Figure 2.4: Two Qubit Circuit Implementation for the Circuit in Fig 2.3

The first two qubit gate is the CPHASE gate (also called CZ gate). It applies a Z gate to the second qubit (target qubit) if the first qubit (control qubit) is in state $|1\rangle$. The matrix representation of the gate is:

$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

The second two qubit gate applies the X gate on the second gate if the result of m_1 is 1. Else, it doesn't do anything. The end result m_2 corresponds to the m result we get from the circuit in Fig 2.3.

Now, consider the case when the client is hesitant to give the values of α and β directly to the server. Moreover, the client would also like to make sure that the server actually prepares the two qubits in the right state.

In such a case, the concept of BQC [18] is used. The equivalent blind quantum computation circuit [27] that satisfies the client's new requirements in presented in Fig 2.5. From the circuit we see that instead of giving α and β directly, the client calculates $\delta_1 = \alpha - \theta_1 + p_1 \cdot \pi$, $\delta_2 = (-1)^{m_1} \cdot (\beta - \theta_2 + p_2 \cdot \pi)$ and transfers them to the server. The values for θ_1, θ_2 are randomly selected by the client.

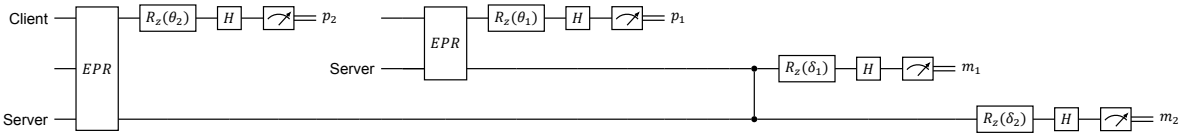


Figure 2.5: Blind Quantum Computation Circuit

The EPR box generates the 2 qubit entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. The rest of the gates used in this circuit is defined before.

3

Quantum Software Engineering on Realistic Hardware

The chapter is written to give the reader an idea on what it means to program a quantum application with the current software and hardware setting. This is done by providing a summary on the current state of quantum software engineering and a high level overview on what we can expect if we were to run a quantum program on realistic hardware that is available to us right now. For this, we introduce the SquidASM environment using which we implement the State Teleportation Protocol and the Blind Quantum Computing application that is run on a simulated model of realistic quantum hardware. This is done so as to get an intuition about what it means to run a networked quantum application in a practical setting that has imperfect devices and noises. The chapter also studies how we can interpret the results we get from the simulation. Even though the theory behind the work presented in this chapter is well established, the actual implementation in a practical setting presented here including the text, code, the math, the results, and the visualization are the author's original work.

3.1. Quantum Software Development

Quantum Software Development is the process of conceiving, specifying, designing, programming, documenting, and testing executable quantum programs that are meant to run on practical quantum hardware. Traditionally, we can divide the software development process into 6 main stages [42]: Planning, Analysis, Design, Implementation, Testing, and Maintenance.

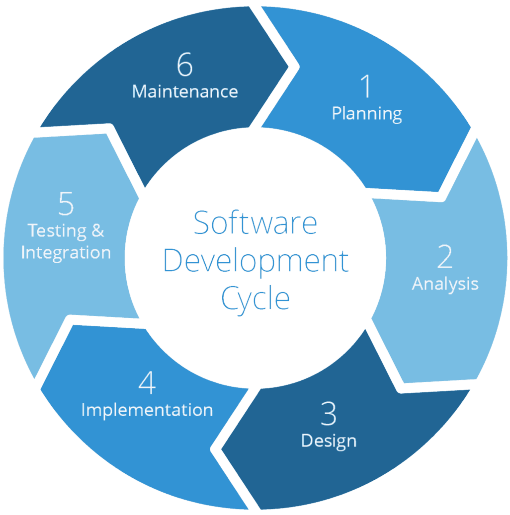


Figure 3.1: Software Development Life Cycle

In the planning phase, we map out the requirements and timeline of our project: for a given problem that can be solved using software engineering, we lay out a high-level solution that describes the system for development and also carry out a feasibility study. Based on these two outcomes, we plan the timeline of our project. For example, if our problem at hand is to develop a secure key distribution system that will be used for cryptographic protocols, the planning phase concerns figuring out which quantum key distribution protocol will solve this problem and conduct a feasibility study on what it takes to develop that product.

The analysis phase concerns itself with defining the objectives and goals. Once we have a high-level solution, it is time to scientifically write down what we are hoping to achieve. Based on the requirements and feasibility study we carried out in the planning phase, we get to decide the performance we expect after each milestone in the timeline. In our example quantum key distribution project, this would entail picking the right protocol and writing down the objective as ”Implementing a BB84 protocol library that is able to distribute x bits of the key within t units of time in an environment that has entanglement fidelity of e.

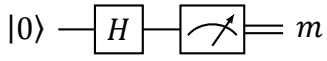
The design phase is the part where we engineer the architecture of our product, its corresponding documentation, and specify the interfaces and interactions between each component of our architecture. Finally in the implementation phase we get to bring our software to life by writing out the code.

Even though Quantum Software Development research has gained traction, it is still mainly focused on problem analysis, language design [46][45][14], and implementation [19][2][28], which only covers the first four fundamental phases. Software testing, which is the process of executing a program or application with the intent of finding failures, and verifying that the software product is fit for use is yet to receive substantial attention in quantum software development. Thus, currently, quantum software development is incomplete.

Before we examine how we can test quantum software, it is important to understand the environment we write our programs in. Thus in the following section, we introduce the SquidASM environment that is used for this thesis and see how we can use it to simulate quantum programs running on realistic hardware.

3.2. SquidASM Environment

As described in the previous chapter, all the quantum applications we have right now are defined using circuits that specify what gates are to be applied for each qubit. Thus quantum programming right now is akin to classical assembly language code where we specify what gate to apply to each qubit line by line. The instruction set architecture which we will be using for this thesis that allows one to control quantum network controllers and run applications on a quantum network is called **NetQASM**. A simple NetQASM program that initializes a qubit to $|0\rangle$ state, applies an H gate to it, measures it, and returns the result looks like the following:




```
# NETQASM 0.5
# APPID 0
# DEFINE op h
# DEFINE q Q0
# DEFINE m M0
set q! 0
qalloc q!
init q!
op! q!
meas q! m!
EXIT:
ret_reg m!
```

Listing 3.1: Corresponding NetQASM Code

Writing assembly-like code for quantum applications is not the preferred software development method. Hence, the NetQASM framework provides an SDK that takes in high-level code written in Python and converts it to NetQASM code, which is then sent to lower-level quantum hardware. The high-level Python code for the application mentioned above looks like the following:

```
from netqasm.sdk import Qubit
from netqasm.sdk.toolbox import set_qubit_state

def main():
    user = NetQASMConnection(app_name='user')
    with user:
        q = Qubit(user)
        set_qubit_state(q, 0, 0)
        q.H()
        m = q.measure()
    return {"m" : m}
```

Listing 3.2: High Level Python Code

Since actual physical hardware is tough to procure, we will be using **SquidASM** [49]: it is a framework that utilizes the NetQASM compiler to convert the high-level python Code to NetQASM code and then send it to real quantum hardware or a hardware simulator. The hardware simulator that we will be using is called NetSquid. It is a network simulator that mocks the execution of real hardware running a network program. If we were to use a real physical hardware for networking purposes, then we would need a software stack to communicate between different nodes and the underlying hardware itself. QNodeOS [48] does this work for us. The architecture of our software stack that ranges from the high-level code written by the end-user down to the physical hardware is shown in Fig 3.2.

The goal of this thesis is to detect and localize bugs that might be generated while creating the high-level quantum program in Python. We assume the software stacks used to simulate the quantum program (SquidASM and NetQASM) are bug-free, and we will only be running our experiments on a simulated setting using the NetSquid simulator. Hence, we are only concerned with the quantum software that is written in the application layer.

3.2.1. Executing Quantum Network Protocols

Now that we have an idea of the environment in which we create our quantum programs, this section will provide the structure for creating quantum network programs, specifically the state teleportation and blind quantum computing applications described before. To run a networked quantum program in the SquidASM environment, we require two main things: one, the high-level Python code that runs on each node, and two, a network.yaml file that specifies the network (how many nodes there are, properties of qubits in each node, properties of link in the network, etc).

Thus, the directory structure for a teleportation protocol looks like this:

```
application
├─app_sender.py
├─app_receiver.py
└─network.yaml
```

The corresponding high level code is shown in listings 3.3 and 3.4.

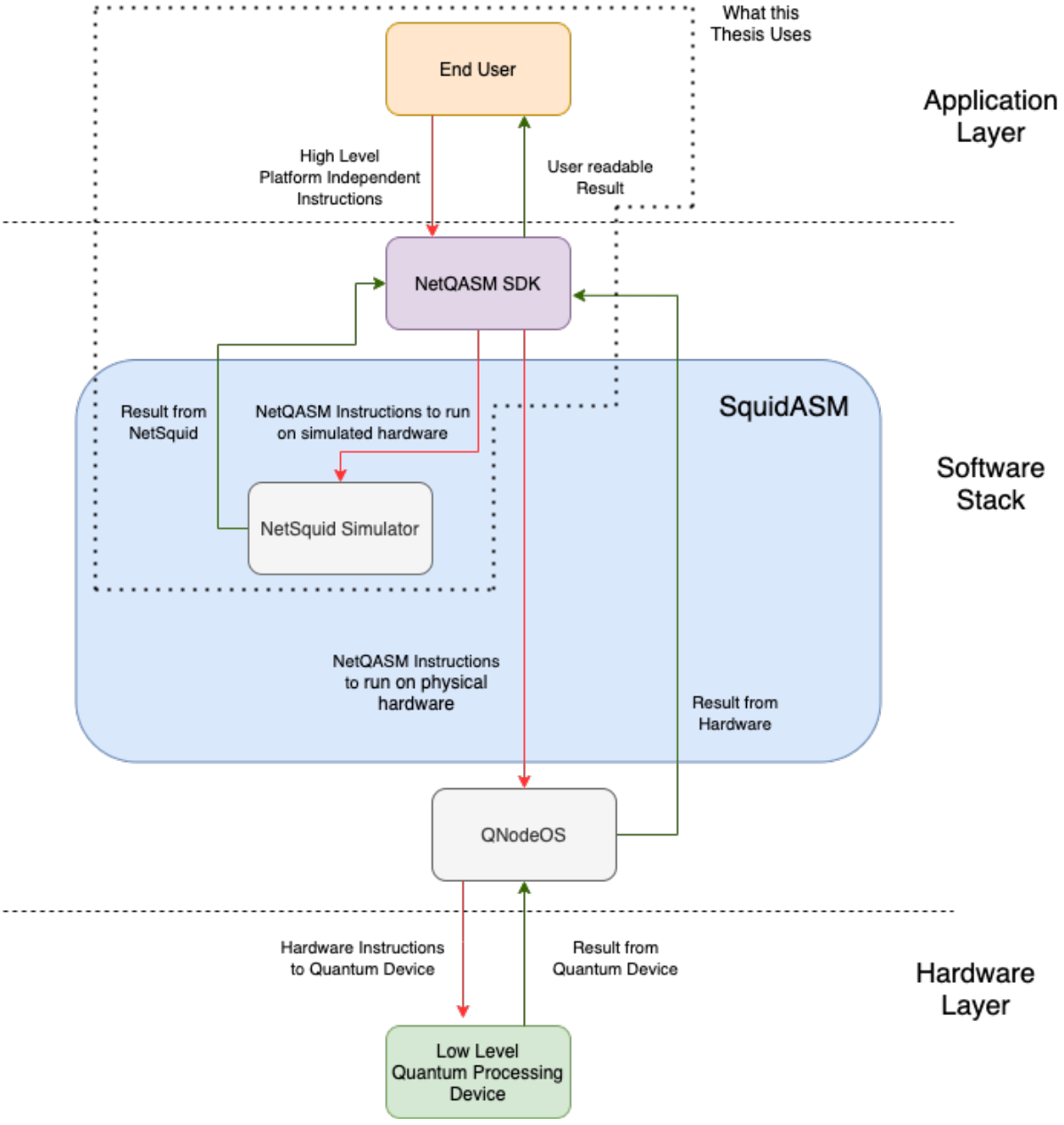


Figure 3.2: SquidASM Environment along with Application and Hardware Layer

```python
from netqasm.sdk import Qubit, EPRSocket
from netqasm.sdk.external import Socket,
    NetQASMConnection

def main():
    epr_socket = EPRSocket("receiver")
    socket = Socket("sender", "receiver")
    sender = NetQASMConnection(
        app_name='sender',
        epr_sockets=[epr_socket]
    )
    with sender:
        qubit = Qubit(sender)
        epr = epr_socket.create()[0]
        qubit.cnot(epr)
        qubit.H()
        m1 = qubit.measure()
        m2 = epr.measure()
    socket.send(str([m1, m2]))
```

Listing 3.3: app_sender.py

```python
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import Socket,
    NetQASMConnection

def main():
    epr_socket = EPRSocket("sender")
    socket = Socket("receiver", "sender")
    receiver = NetQASMConnection(
        app_name='receiver',
        epr_sockets=[epr_socket]
    )
    with receiver:
        epr = epr_socket.recv()[0]
        m1, m2 = socket.recv()
        m1, m2 = int(m1), int(m2)
        if m2 == 1:
            epr.X()
        if m1 == 1:
            epr.Z()
```

Listing 3.4: app_receiver.py

Once the Python code is written, it is transferred to the SquidASM environment where it is compiled to NetQASM code and ran on the simulator. As mentioned before, this thesis is concerned with detecting bugs present in the high-level code, like the one listed above.

3.3. Extracting Information after Executing Quantum Programs

Quantum applications often give us probabilistic answers. Hence, we won't be able to get the required answer in a single shot. We run it many times and collect the statistics after N runs. The type of statistics we collect and how we can interpret them depends on what kind of output we receive from the application. The output could be in the form of a qubit, or as classical bits.

3.3.1. Classical Outputs

Consider a quantum program that produces classical information as output. For example, it could be a program that measures two qubits in the computational basis and returns the two bit result. The corresponding listing is shown in 3.5.

```python
def measureAndReturn(q1, q2):
    m1 = q1.measure()
    m2 = q2.measure()
    return m1, m2
```

Listing 3.5: Measure and return

Based on the input state of the two qubits $q1$ and $q2$, we could have four different outcomes from this function: $(m1, m2) = \{(0,0), (0,1), (1,0), (1,1)\}$. If the input qubits were in the state $|0\rangle, |1\rangle$, we would get $(m1, m2) = (0,1)$ every time we run the function. That is, $Pr(0,1) = 1$ and $Pr(x,y) = 0$ for all other configurations of (x,y). However, if our input qubits were in the state $|+\rangle, |+\rangle$ then all four possible outcomes are equally probable, as each qubit produces a measurement result of $(0,1)$ with equal

probability. Therefore, $\Pr(x,y) = \frac{1}{4}\forall(x,y)|x,y \in \{0,1\}$. Therefore, we cannot know the correct probability distribution of the output from our function if we only run it once - we have to run it many times and conclude what distribution it is from the samples run.

Since we have to run such probabilistic functions an infinite number of times to correctly get the distribution with no error, we employ Maximum Likelihood Estimation (M.L.E) to estimate the distribution from the result we get by running it only a finite N number of times. The idea of M.L.E is to estimate the probability of each outcome ($P(x,y)$ for the example above) using the information we get after running the function only N times.

Suppose there exist k different outcomes for a probabilistic function. Let us denoted them by $X = (x_1, x_2, \dots, x_k)$. After running the probabilistic function N times, suppose we get the following distribution: $D = (n_1, n_2, \dots, n_k)$. That is, we got the outcome x_1 n_1 number of times, x_2 n_2 number of times, etc. We have, $n_1 + n_2 + \dots n_k = N$. The goal of M.L.E is to find the probability associated with each outcome $P = (p_1, p_2, \dots, p_k)$ where $\Pr(x_k) = p_k$, given the distribution D.

The output corresponds to a Multinomial Distribution. The corresponding likelihood for p_i from the given count n_i is then:

$$L(p_i, n_i) = \binom{N}{n_i} p_i^{n_i} (1-p_i)^{N-n_i}$$

Taking the log likelihood,

$$l = \log L(p_i, n_i) = \binom{N}{n_i} (n_i \log p_i + (N - n_i) \log (1 - p_i))$$

The maximum of this likelihood is achieved when

$$\frac{dl}{dp_i} = 0 \implies \frac{n_i}{p_i} - \frac{N - n_i}{1 - p_i} = 0 \implies p_i = \frac{n_i}{N}$$

Therefore, the probability of a particular outcome is just its count divided by the total number of times we ran the function.

The next question is to find out how many samples N we need to get to make sure that the MLE result corresponds to the actual result. Evidently, the larger the number of runs, the better will be our estimation, but now we would like to mathematically prove the bounds so that the end-user can decide how many runs they need on their own.

Suppose we would like to estimate the parameters $(p_1, p_2, \dots, p_k)$ within ϵ error, with probability at least $1 - \delta$. The user provides the values of (ϵ, δ) and now we would like to calculate the minimum number of runs required that will satisfy the above promise condition.

Let us denote the mean of the ”true” distribution as μ and the mean of the distribution we get after running MLE on N samples as μ_N. Mathematically, we can define the user requirement as $Pr(|\mu_N - \mu| > \epsilon) \leq \delta$.

Hoeffding's work [22] has proved an upper bound on the inequality mentioned above. Its main result is that

$$Pr(|\mu_N - \mu| > \epsilon) \leq 2exp(-2N\epsilon^2)$$

Equating with the user requirement inequality, we have

$$\delta = 2exp(-2N\epsilon^2) \implies \log\frac{\delta}{2} = -2N\epsilon^2 \implies \epsilon = \sqrt{\frac{1}{2N}\log\frac{2}{\delta}}$$

Thus,

$$|\mu_N - \mu| < \epsilon = \sqrt{\frac{1}{2N}\log\frac{2}{\delta}}$$

This inequality tells us that as N increases, the difference between the correct mean and the mean after running it N times gets closer and closer. Rearranging to bring out the N term,

$$N = \frac{1}{2\epsilon^2}\log\frac{2}{\delta}$$

For all purposes in this thesis, we assume that the user has set the required maximum error margin as $\epsilon = 0.03$ and probability as $0.95 = 1 - \delta \implies \delta = 0.05$. Plugging these values in to the equation, we find that the minimum number of runs required to correctly run MLE with the given user prescribed constraints is $N \approx 1000$.

Therefore, in summary, we will run every test case $N = 1000$ times and collect the statistics for MLE that will then describe the output distribution of our function.

Comparing Classical Output

To compare two classical distributions, for the purposes of checking whether they are the same or not, we use the Hellinger distance [8]. It is a metric to measure the difference between two probability distributions. It is the probabilistic analog of Euclidean distance.

For two discrete probability distributions $P = (p_1, \dots, p_k)$ and $Q = (q_1, \dots, q_k)$, the Hellinger distance is defined as:

$$h_{PQ} = \frac{1}{\sqrt{2}} \cdot \left\| \sqrt{P} - \sqrt{Q} \right\|_2 = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^{k} (\sqrt{P_i} - \sqrt{Q_i})^2}$$

Rearranging, we find that

$$1 - h_{pq}^2 = \sum_{i=1}^{k} \sqrt{P_i \cdot Q_i} \implies h_{pq} = \sqrt{1 - F_{pq}}$$

where $F_{pq} = \sum_{i=1}^{k} \sqrt{P_i \cdot Q_i}$ is the classical fidelity of the two distributions.

The greater the value of this parameter, the more dissimilar the two distributions P and Q are, and hence the easier to distinguish. Two advantages of using the Hellinger Distance are that it forms a bounded metric on the space of probability distributions, and that it is symmetric: $h_{pq} = h_{qp}$. It is also easy to visualize and use the metric value for further analysis.

The algorithm used to compare two probability distribution is given in 1.

Algorithm 1 Probability Distribution Distance (PDD) Algorithm

Input: Two Distributions $P = (p_1, p_2, \dots, p_k), Q = (q_1, q_2, \dots, q_k)$
Output: Hellinger Distance between P, Q

1: Normalize P and Q such that $\sum p_i = \sum q_i = 1$
2: $F_{PQ} = 0$
3: **for** i in 1...k **do**
4: $\quad F_{PQ} = F_{PQ} + \sqrt{p_i \cdot q_i}$
5: **end for**
6: $h_{PQ} = \sqrt{1 - F_{PQ}}$.
7: Return h_{PQ}

For example, suppose $P = (1000, 0)$ and $Q = (500, 500)$. Normalizing the distribution, $P = (1, 0)$ and $Q = (0.5, 0.5)$. Calculating, $F_{pq} = \sqrt{1 \cdot 0.5} + \sqrt{0 \cdot 0.5} = \sqrt{0.5} = 0.707$ and thus $h_{pq} = \sqrt{1 - 0.707} = 0.54$.

Algorithm 1 is used to test whether a software component that outputs a distribution as result has passed or failed. If the distance between the probability distribution is greater than a threshold distance, we flag the test run as failure.

3.3.2. Quantum Outputs

Now consider a quantum program that produces quantum information (qubits) as output. As mentioned in Chapter 2, we cannot extract all information about a qubit using just one measurement in a particular basis. Hence, to extract information about a qubit, we have to perform measurements in all 3 basis: $Z, X,$ and Y.

From the measurement results of $Z, X,$ and Y basis, we get to reconstruct the density matrix ρ of our qubit using the formula [34]:

$$\rho = \frac{tr(\rho)I + tr(X\rho)X + tr(Y\rho)Y + tr(Z\rho)Z}{2}$$

Where X, Y, Z are the Pauli Matrices defined before and $tr(M)$ calculates the trace of matrix M.

Comparing Quantum Output

From the reconstructed density matrix, we can calculate the fidelity of two qubits using the formula [34]:

$$F(\rho, \sigma) = (tr\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}})^2$$

Here, ρ and σ are the density matrix representation of the two qubits we are interested in. If this fidelity value $F(\rho, \sigma)$ is less than a threshold value set by the user, then we say that resultant qubit does not sufficiently represent the ideal qubit we are after, and hence the output is different from the test case output.

3.4. Noise and Quantum Software Applications

Noise is an indispensable part of quantum computers right now. Hence, we should take that into account, otherwise our results would be incomplete and unsatisfactory. Different noise that we will take into account are entanglement fidelity, gate fidelity, T2 times for carbon and electron. We also study how our test framework behaves in a networking environment that has host latency and instruction processing times. The fidelities will be modelled using a depolarizing channel. When we say that the entanglement fidelity is f, the net density operator for our qubit system becomes [34]:

$$\rho' = f\rho + \frac{1-f}{3}(X\rho X + Y\rho Y + Z\rho Z)$$

Thus when we have unit fidelity, it means that there is no noise and hence our qubit system retains the state ρ. In the opposite extreme case when $f = 0$, we get the maximally mixed state that is of no importance for most of the applications. Similarly when gate fidelity is 1, we have a perfect noiseless gate. For the noisy implementation ρ of gate σ, we say that the fidelity of gate σ is f if $f = (tr\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}})^2$. As the value of gate fidelity decreases, gate σ and its noisy implementation ρ will share little resemblance between them. The final noise factor we are interested in is the T2 time, also called the dephasing time. It is the time after which an initial state $|+\rangle$ will evolve into an equal classical probabilistic mixture of $|+\rangle$ and $|-\rangle$ states, so that it becomes harder to confidently predict which state the qubit is in.

In this section, we will run a particular noise parameter for each of the 3 BQC stages mentioned in Chapter 2, and show that as the complexity of our application increases, the more impact the noise has, thereby deviating the result from the ideal case. For the circuits shown in Fig 2.3, 2.4, and 2.5 we provide two different input values for (α, β) as $(\pi/2, \pi/2)$ and $(\pi/2, -\pi/2)$. The corresponding correct outputs are 0 and 1.

In our first experiment, we run Circuit 2.3 with different instruction processing time (IPT) $N = 1000$ times each and then we collect the output distribution for each IPT. It is then compared with the ideal output probability distribution case ($(1, 0)$ for Test Case 1 and $(0, 1)$ for Test Case 2) by calculating h_{pq} as prescribed in Algorithm 1. The result is then plotted in Figure 3.3.

For the case when IPT=5ms, when we run Circuit 2.3 1000 times, we get 0 as the result 896 times and 1 as the result 104 times. From M.L.E, we get the corresponding probability distribution as $(0.896, 0.104)$. In order to see how close this answer is to ideal case $(1, 0)$, we compute the Hellinger distance using Algorithm 1. We find that the noisy output and the ideal output differ by a distance of 0.23. However, as IPT increases, for our particular 1000 runs, we get 0 as the result only 562 times and 1 as the result 438 times, thereby increasing the Hellinger distance to 0.5.

As we can see, as our development software takes more time to process each instruc-

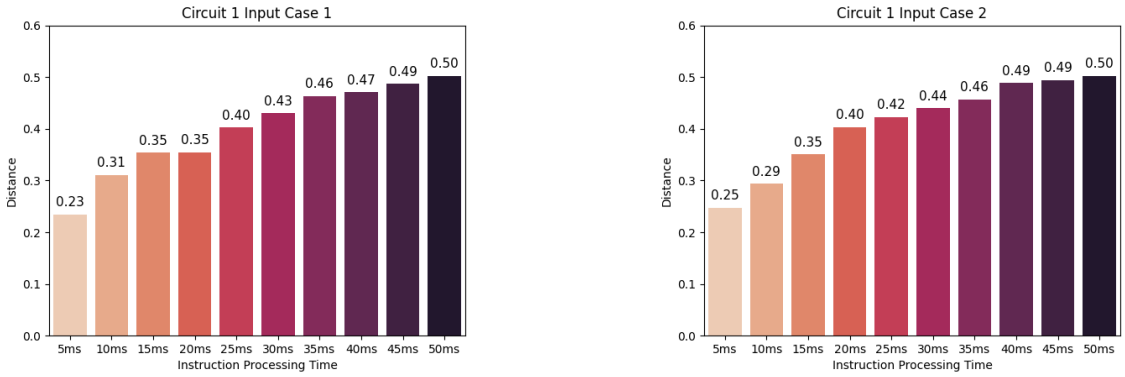


Figure 3.3: IPT v/s Distance for BQC Circuit 1

tion, we get answers that are farther and farther away from the ideal distribution. This is because our qubits are not good enough to withstand the increasing processing time. The T2 time for our qubits is set to 150ms and there are 5 instructions in Circuit 2.3. Thus around the IPT=30ms mark, the qubits start decaying towards the end of the application and produces a result that shares little resemblance with the ideal case. Hence it becomes physically impossible to get the ideal result using the current set of qubits we have. Therefore, when we run our testing framework, our test cases should clarify in what environment we got our results - because different settings of instruction processing time provide us with their own version of ”right” answer.

This is more evident as the application becomes more complex, and when we have more qubits. For the next experiment we will run Circuit 2.4. Not only do we have more qubits and have to deal with already existing instruction processing delays, we also have host-to-host communication, thereby introducing communication host latency. Hence, an IPT of $5ms$, even though it only produces a distribution that is 0.23 distant from the ideal distribution in the first circuit, produces a distribution that is 0.36 distant from the ideal distribution in the second circuit (Fig 3.4).

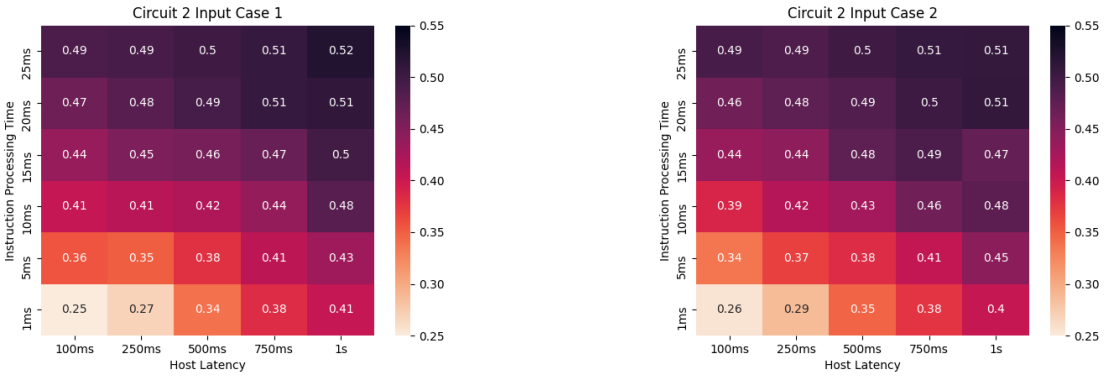


Figure 3.4: Effect of Host Latency and IPT on Distance for BQC Circuit 2

Comparing the distribution between the ideal case and the real distribution for the full application given in circuit 2.5, we see results in the h_{pq} value as plotted in Fig 3.5. These results should be expected for circuit 2.5 because it uses more qubits and gate instructions than the previous two. This is why Circuit 2.5 produces diverging results

even in the few hundred microsecond range, while the previous circuit was able to produce similar distance results in the millisecond range.

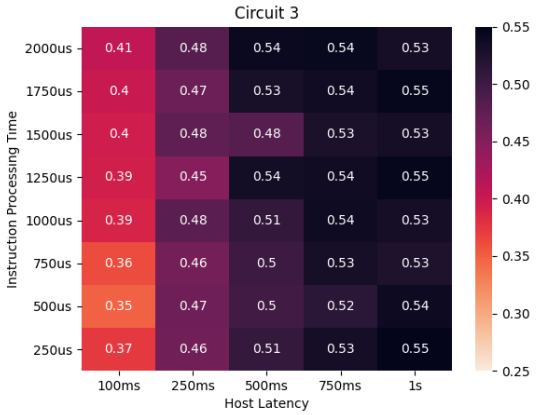


Figure 3.5: Effect of Host Latency and IPT on Distance for BQC Circuit 3

Another factor which we haven't yet considered so far, is the noise produced due to Entanglement Fidelity. For Figure 3.5, we assumed that the entanglement fidelity is a perfect 1. However, practically, this is hard to achieve and hence we should also test how entanglement fidelity affects the end result of our computation. This is presented in Fig 3.6 for varying entanglement fidelity values of $(0.9, 0.8, 0.7, 0.6)$.

The plot presented in 3.6 gives us an idea on the impact of degrading entanglement quality on the end result. As the quality of entanglement fidelity diminishes, even the low IPT,HL cases diverge more and more towards a result that shares little resemblance with the ideal case. These results, however, are not a result of bugs in our program. It is due to the fact that the noise plays a huge role in deciding what result is physically possible for the current setting. Thus, when we test applications, we should work with the test cases for the corresponding practical settings because that is the "right" answer due to the limitations imposed by the physics of hardware. Otherwise, we are testing with the wrong test cases.

(a) Entanglement Fidelity = 0.9

(b) Entanglement Fidelity = 0.8

(c) Entanglement Fidelity = 0.7

(d) Entanglement Fidelity = 0.6

Figure 3.6: Effect of Entanglement Fidelity on Distance for BQC Circuit 3

4

Current Methods for Testing Quantum Programs

In order to create a testing framework for quantum applications, it is important to look into the already existing testing techniques used for classical programs. Along with discussing how effective such techniques would be in a quantum setting, this chapter also summarizes the main techniques that have been recently developed to test quantum applications. The chapter explores the shortcomings and practicalities of these methods and summarizes the state of software testing for quantum applications we have right now. Although the methods discussed in this chapter are already brought up in other works, the author's assessment of said methods (specifically, the Assertion Techniques) is entirely new and an original work.

Program faults are an inevitable consequence of writing code. Faults occur for various reasons: typing errors, misunderstanding software requirements, wrong values assigned to variables, or absence of code to cope with some unpredicted condition. During the testing phase or in the field, a fault is revealed when a program presents an unexpected behavior (known as failure). Once a failure occurs, the two-step debugging process begins. First, the developer inspects the code to locate the failure's cause, and then they fix the fault.

Fault localization is a costly activity in the software development process. Testing and debugging can account for up to 75% of development costs [5]. In practice, fault localization is performed manually; developers observe failing test cases and then search the source code for faults. They use their knowledge of the code to investigate excerpts that may be faulty. The most frequent debugging practices include inserting print statements and breakpoints, checking the stack trace, and verifying failing test cases. Since these manual processes can be expensive and ad-hoc, approaches that automate fault localization are valuable for software development cost reduction.

4.1. Limitations of Classical Testing Techniques

Normally in classical software testing, for debugging purposes we can use techniques like breakpoints to inspect a program's state and the actual values of variables. We stop the execution of program and assert if we have the right value for our data structure. This is not straightforward in quantum setting because of the following main problems:

- Measurement collapses the state of qubits and thus the further execution.
- We do not have the luxury of top grade qubits that allow us to pause the execution midway, examine it, and continue on with our execution.
- It does not make sense to examine the state of the qubits midway through an execution because the result we are after might be distorted by the noise in our hardware. Hence, it becomes harder to assert whether the result we get is due to noise in the system or because of a possible bug.

4.2. Standard System Level Testing

General black box testing that is common right now is to run the whole application from the system level and check if the output matches to that of the test case output. For quantum programs that output classical results, this is done by comparing the probability distribution distance between the output result and test case output. If this distance is greater than a threshold distance set by the tester, then they can say that there is a discrepancy between the result and hence, the test run has failed (meaning, something has gone wrong in our implementation). However for quantum programs that output qubits as their result, in order to find what state the qubit is in, we have to do full tomography [34]. If the fidelity result from that calculation is less than the threshold fidelity set by the user, then the test has failed. The full procedure is described in Algorithm 2.

Algorithm 2 Standard Testing Algorithm

Input: Application to Test A, Quantum Fidelity Threshold θ, Probabilistic Distance Threshold δ
Output: PASS/FAIL

Generate TestCases (i_j, o_j) where i_j in the input and o_j is the correct output of TestCase j.
for each TestCase j **do**
    $r_j = A(i_j)$ # Run Application with input i_j and store result in r_j
    if datatype(o_j) == Qubit **then**
        $f_j = QSF(o_j, r_j)$ # QSF calculates Quantum State Fidelity
        if $f_j < \theta$ **then**
            Return **FAIL**
            break
        end if
    else
        $d_j = PDD(o_j, r_j)$ # Probability Distribution Distance is defined in Algorithm 1
        if $d_j > \delta$ **then**
            Return **FAIL**
            break
        end if
    end if
end for
Return **PASS**

Full tomography is the process of measuring the set of qubits on all possible basis configurations and reconstructing the density matrix. For a single qubit system, as mentioned in Chapter 2, to fully learn about the amplitudes α and β, we need to perform measurement in all 3 basis. When we have 2 qubits as result, the permutation becomes $XX, YY, ZZ, XY, \ldots, IX, \ldots, ZI$ where each measurement AB means we are measuring the first qubit in basis A and second qubit in basis B. This sums up to 15 different measurements that we have to do. So in general, we need to have $4^N - 1$ measurements to fully understand the state of N qubit output so that we can do black box testing. As evident, this does not scale very well. Quantum state tomography algorithms used to calculate fidelity are presented in [35][41][17].

Hence, even though tomography is a very good black box testing technique that would correctly identify whether we have a correspondence between out input and output test case, it is not practical when we have large number of qubits.

4.3. Assertion Techniques

Assertion testing is the process of asserting whether our program is in the right state at a particular point in execution. There has been several papers [30][29][23] published recently that apply this technique to quantum programs. It is mainly through two techniques:

- Syndrome Measurement using Ancilla Qubits: Also referred to as dynamic assertions [30]. The idea of syndrome based assertions is to measure an ancilla qubit that is not part of the main computation to find out the state of qubits that are used for the actual computation.

  For example, we would like to test whether a qubit $|\psi\rangle$ we have is in the state $|0\rangle$.

Rather than measuring the said qubit, we could add one more qubit to the circuit and perform the following operation:

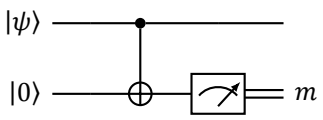


Figure 4.1: Assertion Using Syndrome Measurement

If the input qubit $|\psi\rangle$ was indeed in the $|0\rangle$ state, the measurement result would produce 0 always. However, if something went wrong and we were not given $|0\rangle$, then there is a non zero probability that we will get 1 as the measurement result. Hence, we were able to assert if we have a qubit is state 0 without directly measuring it.

One of the main disadvantage of this method is that it is highly input dependent. The circuit used for syndrome measurement is dependent on the state of qubit that we want to assert the state on. For this reason, to correctly assert the state of the qubit, we might need more than one ancilla qubit. Thus, this method is short in expressive power.

Another shortcoming of this method is that it is not highly responsive to noise. For example if our qubit is initialized in a noisy state, then the corresponding syndrome measurement will not always provide a zero measurement result; which make us to believe that something has gone wrong in our program. However, this non-zero measurement result is not due to a bug in our program, but rather a limitation imposed by our physical hardware. Hence, it is challenging to apply this principle on current hardware.

- Projective Measurement: Checking if the qubits have the right result by applying projective measurement in between the code. If we have the right result, we reconstruct it again.

  Projective assertions implemented in [29] are shown in the circuit figure 4.2.

  Similar to the case of syndrome measurement these projective measurements are highly input dependent. If we were to change the input values then the qubit values we assert in between the computation also change.

  Moreover, calculating the results at each point of the circuit and adding in the corresponding projective measurement computation into the circuit is practically the same as doing an ad-hoc read through to find the correspondence between our code and circuit.

  Projective measurement also has a disadvantage that it is not very susceptible to noise. When we are creating our protective measurement circuits, we should only project them to the noisy subspace we think it would. This would also add an undue burden on the tester to come up with the correct subspace for the given noise parameters.

Although they are limited because of the disadvantages listed before, they could still be used along with more powerful testing techniques as quick subroutines that check the intermediatory results. For example, if we are certain about the state of a qubit at

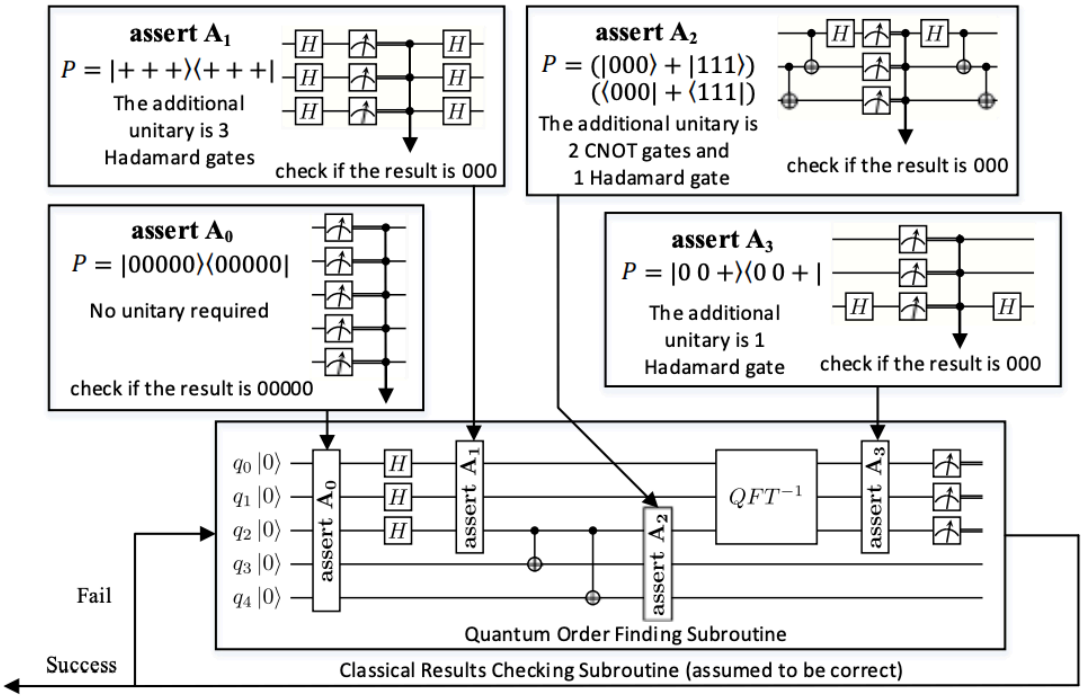


Figure 4.2: Projective Assertion

a particular point in the circuit, then we could add the assertion there itself and check if the qubit is in that state.

4.4. Testing Networked Applications

Having listed out a few testing methods currently devised for quantum programs, it is worth mentioning that none of them directly discuss its applicability to test networked quantum programs. It is worth asking this question because unlike standalone quantum programs, networked quantum applications are run on multiple quantum computers connected together. Hence subroutines such as entanglement generation and transmission are not found in quantum applications that are run standalone.

Due to such extra subroutines, the performance of our whole application changes because our qubits now need to stay alive for the entirety of entanglement generation and transmission. It will be also worth looking into how host to host communication latency will also affect our testing framework's ability to detect and localize bugs.

It is also important to create a testing method that is able to run on a distributed setting because our application will surely be distributed. Rather than making assumptions on a standalone computer that will simulate a networked application, it is better to provide a complete picture for testing production level quantum network application. This will give us more confidence on our testing method and hopefully the ability to detect and localize bugs.

5

Framework for Testing Network Quantum Applications

In this chapter, we first discuss the theory behind test framework and the different types of testing frameworks. From this, we list out what we are looking for in a quantum testing framework. Spectrum-based Fault localization (SFL) technique stands out as a good candidate that satisfies the requirements we have set. Thus, we explore its working and then discuss how SFL can be used to test quantum applications. Then, a general architecture of a framework for testing networked quantum applications is devised.

5.1. Testing Framework

As discussed before, breakpoints and inspections can be a tiring process, and some of them (like breakpoints) are not compatible in the quantum setting (discussed in Chapter 4). Assertions are input dependent and they rely on the tester to come up with the correct subspace for the projective measurement. Easing out the strain put on the tester calls for the use of test automation and a supporting framework. Test automation is the use of software (under a setting of test preconditions) to execute tests and then determine whether the actual outcomes and the predicted outcomes are the same (or sufficiently similar when results are probabilistic). A testing framework is an execution environment for automated tests. It is defined as the set of assumptions, concepts, and practices that constitute a work platform for automated testing. The process of testing itself can be broadly divided into three levels [10]:

- Unit testing: These tests verify the functionality of a section of the code (known as the ”unit”). Usually, this unit comprises a single function, or an entire interface, such as a class. Unit testing each component of the software does not verify the functionality of the whole system, but can be used as a measure to ensure that the constituent blocks of the project work correctly and independently from each other. Thus, we cannot conclude the correctness of our software project using unit testing alone.
- Integration testing: These tests verify the functionality of a group of software components combined together. In this level, the tester exposes the defects in the interfaces and interaction between the integrated components. However, as the number of components used for integration testing increases, it becomes harder to localize the fault, if the tests do fail.
- System testing: System testing tests a completely integrated system to verify that the system meets its requirements [1]. In this level, testing is done with the integrated components, as well as for the system as a whole. The problem of localization is more prominent in this level than the previous one.

Once the level is set, software testing undergoes 4 main phases:

1. Test suite design: In this phase, test cases are designed. Based on the application in hand and the level of testing, test cases are generated and prioritized.
2. Test execution: The generated test cases are put in to action by running the application. A Pass/Fail verdict is formed after comparing the correct output and the output after executing the test script.
3. Fault localization: For the failed test cases, the next step is to identify which part of the code has the bug in them. But like we have explained before, as we go up the level of testing, it becomes harder to correctly localize the bug.
4. Debugging correction: Once the fault is localized, it is now the onus of the programmer to repair the faulty component.

The practicality and usefulness of a testing framework according to each level is shown in Figure 5.1, and is based on the test pyramid model as discussed in [13].

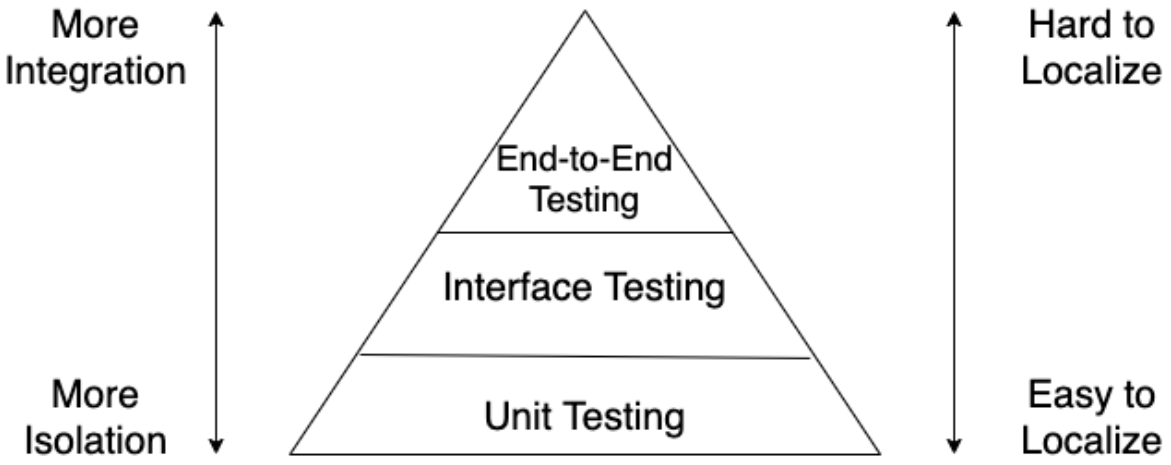


Figure 5.1: Test Pyramid Model

Thus, the testing framework that we design should have a healthy mix of testing from all levels. This makes sure that we are able to test each component in isolation as well as in an integrated form, and that it is also feasible to localize bugs, if any failures arise.

5.2. Lessons Learned: What we are looking for in a Quantum Testing Framework

Having talked about the difficulties in testing a quantum program and the shortcomings of the current testing schemes, we lay out what we are looking for in a quantum testing framework that tries to solve the problems mentioned in the previous chapters, and can also be implemented in a realistic hardware setting that we have today.

The main things we are looking for in a network application testing framework are:

- Input independent.
- Noise tolerant.
- Carry out both unit testing and integration testing.
- Fault localization: Ability to pinpoint where the bug resides in our code base.
- Ability to test networking applications that are run on multiple quantum computers.

5.3. Spectrum based Fault Localization

The Spectrum-based Fault Localization (SFL) technique [4] [16] is a promising candidate that satisfies the requirements mentioned before. The general idea behind SFL is to to divide our protocol circuit to small components and run a subset of them at a time. This is done for various permutations, and using the end result, we create a matrix and try to localize which component has chance of having a bug.

A program entity is a part of a program. It comprises any granularity of a program, from a statement to a subprogram. Program entities include statements, blocks, branches, predicates, definition-use associations, components, functions, program elements, and program units. Program spectra, also known as code coverage, can be defined as a set of program entities covered during test execution. Spectrum-based fault localization uses information from program entities executed by test cases to indicate entities more likely to be faulty. There are several synonyms of program spectrum used in the literature, such as code coverage, execution trace, execution path, or path profile.

Spectrum-based Fault Localization techniques propose several strategies to pinpoint faulty program entities. Most of them rank suspicious entities by using ranking metrics or statistical techniques. Metric-based techniques are those that use ranking metric formulas to pinpoint faulty program entities. Most of the SFL techniques use or propose ranking metrics to improve fault localization. To determine correlations between program entities and test case results, these ranking metrics use program spectrum information derived from testing as input. Each program entity receives a suspiciousness score that indicates how likely it is to be faulty. The rationale is that program entities frequently executed in failing test cases are more suspicious. Thus, the frequency in which entities are executed in failing and passing test cases is analyzed to calculate its suspiciousness score. Statistical techniques have also been applied in fault localization, and are used in the same manner as metric-based techniques. One such technique is the Spectrum-based Reasoning concept (SR) [5] [36].

SR reasons about observed system executions and their outcomes to derive diagnoses that can explain faulty behavior in software. In SR, the following is given:

- A finite set $C = \{c_1, c_2, \ldots, c_M\}$ of M system components. Components can be any source code artifact of arbitrary granularity such as a class, a method, a statement, or a branch.
- A finite set $T = \{t_1, t_2, \ldots, t_N\}$ of N system transactions, which can be seen as records of a system execution, such as, e.g., test cases.
- The outcome of system transactions is encoded in the error vector $e = \{e_1, e_2, \ldots, e_N\}$, where $e_i = 1$ if transaction t_i has failed, and $e_i = 0$ otherwise.
- An $M \times N$ activity matrix A, where each A_{ij} encodes the involvement of component c_i in transaction t_j.

The pair (A, e) is commonly referred to as a spectrum. Several types of spectra exist. The most commonly used is called a hit-spectrum, where the activity matrix is encoded in terms of binary hit (1) and not hit (0) flags, i.e., $A_{ij} = 1$ if c_i is involved in t_j and $A_{ij} = 0$ otherwise.

Prior approaches using spectra were based on a so called similarity coefficient to find a correlation between a component c_i's activity and the observed transaction outcomes encoded in error vector e. SR relies instead on a reasoning approach that leverages a Bayesian reasoning framework to diagnose the system. SR was also shown to outperform similarity-based approaches.

The two main steps of SR are candidate generation and candidate ranking:

- **Candidate Generation**: The first step in SR is to generate a set $D = d_1, d_2, \ldots, d_k$ of diagnosis candidates. Each diagnosis candidate d_k is a subset of C, and d_k is said to be valid if every failed transaction involves at least one component from d_k. A candidate d_k is minimal if no valid candidate d' is contained in d_k. We are only interested in minimal candidates, as they can subsume others of higher cardinality. We rely on heuristic approaches to finding these minimal candidates, which is an instance of the minimal hitting set problem, thus NP-hard.
- **Candidate Ranking**: For each candidate d_k, their fault probability is calculated using the Naive Bayes rule.

$$Pr(d_k|(A,e)) = Pr(d_k) \cdot \prod_{j \in 1 \ldots N} \frac{Pr((A_j, e_j)|d_k)}{Pr(A_j)}$$

Let A_j be short for $(A_{ij}|i \in 1 \ldots M)$ — i.e., the jth column of matrix A, represented by a set encoding all component involvements in test t_j . The denominator $Pr(A_j)$ is a normalizing term that is identical for all candidates and is not considered for ranking purposes.

In order to define $Pr(d_k)$, let p_i denote the prior probability that a component c_i is at fault. The prior probability for a candidate d_k is given by

$$Pr(d_k) = \prod_{i \in d_k} p_i \cdot \Pi_{i \in C \setminus d_k}(1 - p_i)$$

$Pr(d_k)$ estimates the probability that a candidate, without further evidence, is responsible for erroneous behavior. $Pr((A_j, e_j)|d_k)$ is used to bias the prior probability taking observations into account. Let g_i (referred to as component goodness) denote the probability that a component c_i performs nominally.

$$Pr((A_j, e_j)|d_k) = \begin{cases} \prod_{i \in (d_k \cap A_j)} g_i, & \text{if } e_j = 0. \\ 1 - \prod_{i \in (d_k \cap A_j)} g_i, & \text{otherwise.} \end{cases}$$

In cases where values for g_i are not available they can be estimated by maximizing $Pr((A_j, e_j)|d_k)$ — i.e., maximum likelihood estimation (MLE) for the Naive Bayes classifier — under parameters $\{g_i|i \in d_k\}$. This work uses MLE to estimate component goodness.

5.4. SFL on Quantum Programs

Before we explain how SFL technique can be implemented in a quantum setting, we discuss the challenges SFL faces while adopting it to a quantum setting, and provide solutions for it.

For classical programs, the input values decide which component is called upon and hence they do most of the heavy lifting to generate the activity matrix. Quantum Programs, on the other hand are directly a result of the circuit it is working on. These

circuits are mostly linear in nature and for most of the applications that we work with, all the components will be activated every time no matter what input value we have. Thus, ability of test cases to drive the activity matrix generation is limited in quantum setting.

A possible solution, therefore, is to lay the responsibility to generate each transaction and hence the activity in the hands of the tester themselves. The tester decides on how to properly divide the whole circuit into small components and then plan out which subset of components should be activated for a given transaction.

For example, consider the circuit shown in Figure 5.2:

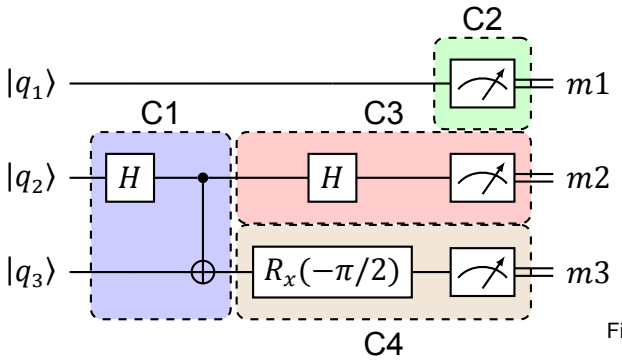


Figure 5.2: Example Quantum Circuit

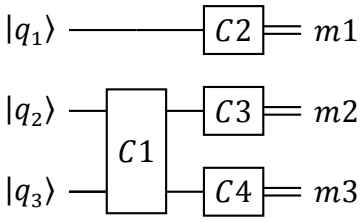


Figure 5.3: One Possible way to Divide of Circuit 5.2

We can divide the Circuit into 4 Components: an EPR Pair Generation Component C1 that uses qubit $|q_2\rangle$ and $|q_3\rangle$, a Z basis measurement Component C2 on $|q_1\rangle$, a X basis measurement Component C3 on $|q_2\rangle$, and a Y basis measurement Component C4 on $|q_3\rangle$. This is detailed in Figure 5.3.

From the Circuit 5.3, we find that no matter what kind of input value $|q_1\rangle$, $|q_2\rangle$, $|q_3\rangle$ have, all the 4 different components will be activated every time. Hence, all the transactions in the activity matrix will be same and equal to (1,1,1,1). Although this cannot be generalized to all quantum circuits, for most of the quantum programs, input test cases are not driving factors that generate the activity matrix. Hence, it is on the onus on the tester to decide which components should be run on each transaction. A possible Activity Matrix for the Circuit 5.3 is shown in Table 5.1. Conceptually, a $'0'$ associated to a component in a particular transaction implies we are removing that component when we test run the application, and are only utilizing the components that are activated by $'1'$. That is, we are isolating the activated components and testing them separately.

	T1	T2	T3	T4
C1	1	1	0	0
C2	0	1	0	1
C3	1	0	1	0
C4	1	0	1	0

Table 5.1: Possible Activity Matrix for Circuit 5.3

The reason behind choosing the transaction set defined above is the following: component C2,C3, and C4 measures a qubit in the Z, X and Y basis respectively. Thus, we test its correctness by running it as a standalone subroutine (unit testing through

transactions T3, T4), and also along with component C1 (integration testing through transactions T1, T2). Thus, we are able to test our components in isolation, as well as with more integration, and from the error vector generated after running each transaction, we get to localize the bug (if it exists) more clearly using SFL.

An advantage of letting the tester decide which component goes into each transaction is that they get to decide the level of testing they want - unit testing is possible by having standalone components in a transaction, integration testing is carried out by having more than one component activated in the transaction, and the system level testing is carried out by activating all the components in that transaction. As explained before, the onus is on the tester to create such practical transaction sets, and not fall into combinatorial explosion and run every one of the 16 possible combinations of enabling four components C1,C2,C3,C4.

Once the transaction set for the activity matrix is generated, each transaction is run one by one. A set of test case is defined for each transaction and the output of the transaction is compared with the test case.

For each qubit participating in a transaction, we can initialize it to a state from the set $S = \{|0\rangle, |1\rangle, |+\rangle, |-\rangle, |+i\rangle, |-i\rangle\}$. For this thesis, we will refer to this set as the standard input set, because any qubit can be represented as a linear combination of either $(|0\rangle, |1\rangle)$, or $(|+\rangle, |-\rangle)$, or $(|+i\rangle, |-i\rangle)$. Thus, the initial input set for each qubits participating in a transaction set can be summarized in Table 5.2. For example, transaction T1 has 36 different input cases because it uses two qubits, each of which takes one value from set S. Similarly, transaction T2 activates component C1 that needs two qubits, thus again 36 different input values. Transaction T4 however only needs one qubit as component C2 uses just one qubit. Thus, the 6 different input states given in S is used for T4.

	T1 (36 Cases)	T2 (6 Cases)	T3 (36 Cases)	T4 (6 Cases)
Qubit 1	Not Used	S	Not Used	S
Qubit 2	S	S	S	Not Used
Qubit 3	S	Not Used	S	Not Used

Table 5.2: Input Test Case for Example Circuit

The correct output for each of the input case is calculated by an oracle before running the transaction test and the final test suite is designed. While running each input case for each transaction, if there is a mismatch between at least one of the results and oracle test case result, the error vector logs a 1 for that particular transaction (e=1). The mismatch check is carried out using fidelity (θ) and probabilistic distance threshold (δ) values that the tester sets. If the end result after a test run does not satisfy either of θ or δ threshold (lines 9 and 14 in Algorithm 3), we mark the run as failure. Once the error vector is fully generated after running every transaction, we generate the Candidate Set and then rank them using Bayesian reasoning. The candidates with the higher probabilities are chosen and their constituent components are debugged.

Algorithm 3 Q-SFL Algorithm

Input: Transaction Set T, Quantum Fidelity Threshold θ, Probabilistic Distance Threshold δ
Output: Set of Candidates that has high probability for having bugs.

Generate Activity Matrix A from Transaction Set T
for each t in T **do**
    Generate TestCases (i_j, o_j) where i_j in the input and o_j is the correct output of TestCase j.
    $ErrorVecotor(t) = 0$
    for each TestCase j **do**
        $r_j = t(i_j)$ # Run Transaction t with input i_j and store result in r_j
        if datatype(o_j) == Qubit **then**
            $f_j = QSF(o_j, r_j)$ # QSF calculates Quantum State Fidelity
            if $f_j < \theta$ **then**
                $ErrorVecotor(t) = 1$
            end if
        else
            $d_j = PDD(o_j, r_j)$ # Probability Distribution Distance is defined in Algorithm 1
            if $d_j > \delta$ **then**
                $ErrorVecotor(t) = 1$
            end if
        end if
    end for
end for
Generate Candidates from Activity Matrix A and Error Vector.
Rank Candidates using Bayesian Reasoning.
From the candidates that has the highest probabilities, output their constituent components.

5.5. SFL Architecture for Testing Networked Quantum Applications

A distinction between testing standalone quantum programs and networked applications is necessary when we are testing an application in practical setting. This is because when we are testing a networked application, a mock of such application running on a standalone system will not consider the whole practical setting like host latency. Moreover, if we were to test entanglement generation between two different nodes, then substituting it by a subroutine that transfers both the qubits from the EPR pair to a single node does not really capture the essence of what we are after.

Therefore we need a distributed testing framework that is able to coordinate with different nodes in the application and decide whether or not there is a bug in our application. For that, we employ a centralized server that coordinates with the different nodes participating in the application. The architecture for testing a networked quantum application using the SFL technique is described in 5.4.

We assume that the required code is made available to all the nodes participating in the testing process. The centralized script runs the Q-SFL Algorithm 3 with just one difference - while running each transaction, the centralized script decides the operations performed by each node. For example, if the transaction calls for generating an EPR Pair, the centralized node decides which node utilizes the component that calls the EPR generating function, and which node utilizes the component that calls the EPR receiving function. Once each node has completed its tasks, its individual result is sent back to the centralized node. These results are then compared with the test cases that the centralized node already has, and then the error vector is generated.

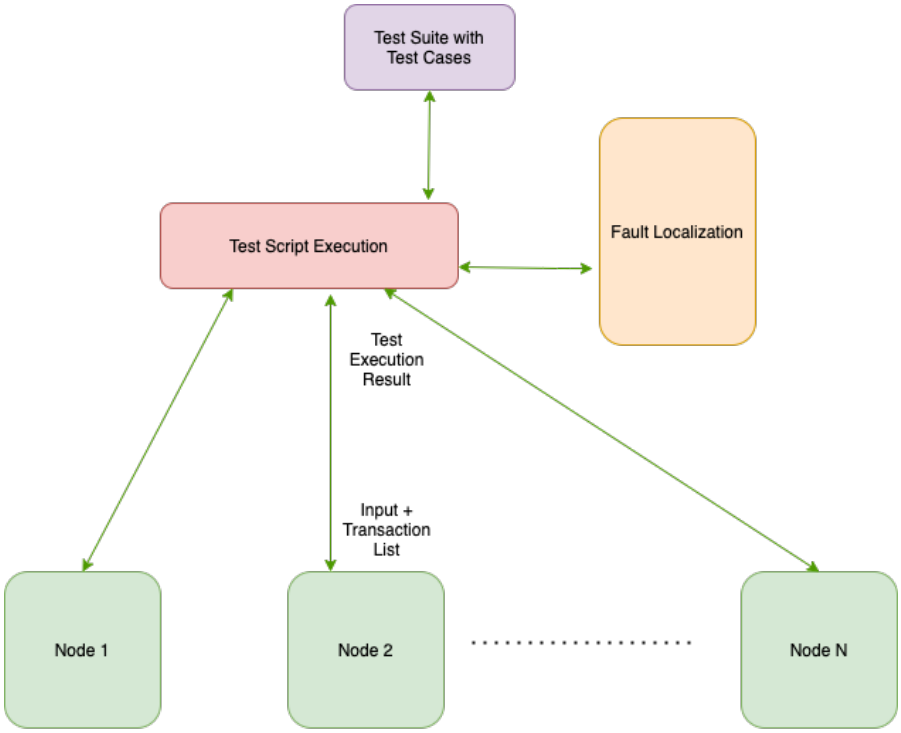


Figure 5.4: SFL Architecture for Networked Quantum Applications

So taken as a whole, the distributed architecture works logically equivalent to the case where we have just one computer, for the centralized script merely delegates the functioning of each transaction to the various nodes involved in the application. However for practical purposes this distinction is necessary.

6

Testing Specific Quantum Programs

The testing framework is put to test by running it on two specific quantum network applications - state teleportation, and blind quantum computing application. One or more bugs are introduced to each application's codebase and we see if the framework correctly detects them and localizes them to relevant part of the code.

6.1. Running SFL on State Teleportation Protocol

For the purpose of Spectrum Based Fault Localization, the circuit in Fig 6.1 can be divided into 3 main Components:

- C1 : EPR Generation, which generates entangled pair $|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$
- C2 : Bell State Measurement (BSM), which performs the BSM operation and returns the measurement result.
- C3 : Pauli Correction applies the X or Z gate according the measurement result from C2.

The complete code that utilizes all three components is given in Appendix A.1.

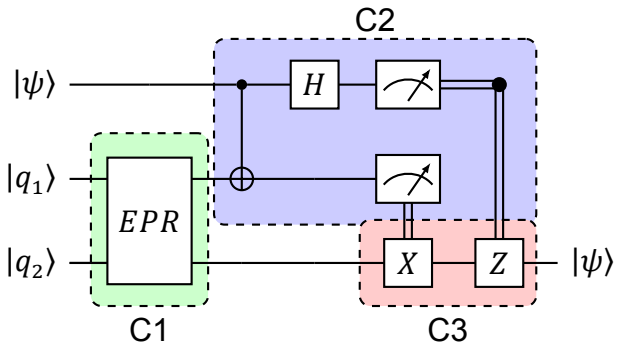


Figure 6.1: State Teleportation Circuit

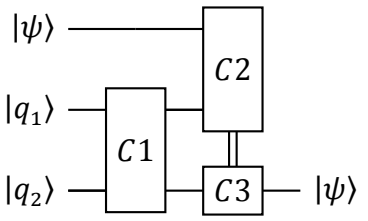


Figure 6.2: Component Division for SFL

According to this division, a possible activity matrix having 4 transaction is the following:

	T1	T2	T3	T4
C1	1	0	0	0
C2	0	1	0	1
C3	1	0	1	1

Table 6.1: Transaction Matrix for Testing State Teleportation

Here, we test the BSM component and Pauli correction component in isolation (unit testing through transactions T2 and T3), and also in an integrated manner through transactions T1 (that tests whether EPR generation can be correctly integrated with other component) and T4 (that tests the interface between BSM and Pauli correction). The operation of each transaction, along with input initialization is specified in Appendix A.3.

The input test case for qubits associated with a transaction is specified in Table 6.2. The oracle computes the correct output for each of these input cases and the final test suite is designed. As before, set $S = \{|0\rangle, |1\rangle, |+\rangle, |-\rangle, |+i\rangle, |-i\rangle\}$ is the standard input set and set $G = \{I, X, H, R_x(\pi/2)\}$ is the gate set that applies a gate from it to the qubit in consideration. Therefore, transaction T1 has 4 different input cases (no gate operation applied on $|q_1\rangle$, it will retain its EPR state), T2 has 36 different input cases (6×6 different input configuration for $|\psi\rangle, |q_1\rangle$), T3 has 6 different input

cases and T4 has 64 different input cases. During the testing phase, if there is a mismatch between the test result and the oracle result for any one of the input case corresponding to a transaction, then the whole transaction will be marked as an error (ErrorVector=1). The operation of each transaction, along with input initialization is specified in Appendix A.3.

	T1 (4 Cases)	T2 (36 Cases)	T3 (6 Cases)	T4 (64 Cases)
$\|\psi\rangle$	Not Used	S	Not Used	G
$\|q_1\rangle$	EPR	S	Not Used	G
$\|q_2\rangle$	G	Not Used	S	G

Table 6.2: Input Test Case for Testing State Teleportation

6.1.1. Localizing Bugs in BSM and Pauli Correction

We will look at a particular type of bug that is logically equivalent to correct implementation of the application, but the constituent components are implemented incorrectly. Thus, we will not be able to detect this discrepancy if we run the application as a whole. The incorrect implementation of C2 and C3 components are shown in listing 6.1, 6.2 respectively.

The bug in the Bell state measurement function is that instead of returning the measurement result of qubit 1 followed by qubit 2, we return the measurement result of qubit 2 and then qubit 1. The bug in Pauli Correction is that instead of checking the value of m2 and applying the X axis, we check the value of m1 and then apply the X gate gate. The same mistake is repeated in Line 4: Instead of checking the value of m1 and then applying the Z gate, we check the value of m2 and then apply the Z gate.

```
1 def bellStateMeasurement(qubit, epr):
2     qubit.cnot(epr)
3     qubit.H()
4     m1 = qubit.measure()
5     m2 = epr.measure()
6     return m2, m1
```

Listing 6.1: BSM Result is swapped

```
1 def pauliCorrection(qubit, m1, m2):
2     if m1 == 1:
3         qubit.X()
4     if m2 == 1:
5         qubit.Z()
```

Listing 6.2: Order of Pauli Correction is Swapped

As mentioned before, we run this buggy code as part of the teleportation application, we won't detect any error because it is logically equivalent to the case when we don't have a bug. The reason is as follows: the bug in the Pauli correction method negates the bug in the Bell State Measurement method. Since both the operations are swapped, the net result is as if we are running a code that is logically consistent to our requirement. Having said that, the implementation of each of these two individual functions is not according to the specification of the library that has been agreed upon before. Moreover, the specification also states that BSM should actually output the measurement result of Qubit 1 first, and then Qubit 2, rather than the other way around.

Now that we have seen that this bug is undetectable even if we run the whole application, it's worth checking if SFL can detect it. The error vector after running each transaction is calculated by running code listed in Appendix A.3 and checking if the

result we get from that code coincides with the result from the oracle. The the Activity Matrix with the error vector is shown in Table 6.3.

	T1	T2	T3	T4
C1	1	0	0	0
C2	0	1	0	1
C3	1	0	1	1
e	1	1	1	0

Table 6.3: Activity Matrix for Testing State Teleportation Example

From the activity matrix, we find that the failed transactions are T1, T2, and T3; meaning at least one input configuration corresponding to each transaction produced a result that does not coincide with the oracle result. The corresponding minimal candidate set is (c_2, c_3). Since there are no other candidates, we do not need to use the candidate ranking algorithm to calculate probability.

In this example, the sole candidate generated is in fact enough. It was able to correctly localize the bug to one of (c_2, c_3), which is exactly what we want. Thus, SFL was able to localize multiple bug, when even running the whole application and checking the end result could not have done it. This has certainly given us a promising start for our testing framework.

6.2. Running SFL on Blind Quantum Computation Application

For the purpose of Spectrum Based Fault Localization, the circuit in Fig 2.5 can be divided into 5 main Components. The division of quantum operations is shown in Figure 6.3.

- C1 : EPR Generation, which generates entangled pair $|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$
- C2 : Remote State Preparation (RSP), which applies the rotation around the Z axis gate by an angle θ, followed by a H gate and then measures the qubit.
- C3 : CPHASE Gate
- C4 : Classical component which calculates $\delta_1 = \alpha - \theta_1 + p_1 \cdot \pi$
- C5 : Classical component which calculates $\delta_2 = (-1)^{m_1} \cdot (\beta - \theta_2 + p_2 \cdot \pi)$

The complete code that utilizes all five components is given in Appendix B.1.

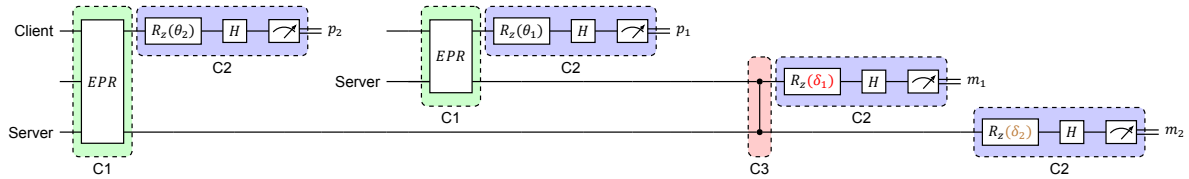


Figure 6.3: BQC Division

According to this division, a possible activity matrix with 6 transactions is shown in Table 6.4.

	T1	T2	T3	T4	T5	T6
C1	0	0	1	1	0	1
C2	1	1	0	1	1	1
C3	0	0	1	0	1	0
C4	1	0	0	1	1	0
C5	0	1	0	1	1	0

Table 6.4: Transaction Matrix for Testing BQC Application

The reason for developing such a transaction set, and thus the activity matrix is the following: in transaction T1 and T2, we test RSP with δ_1 calculation and δ_2 calculation respectively. This works in our favor because RSP component requires an angle as one of its input, and components C4 and C5 produces angle as its output. Thus, it is worth looking into how these three components behave when they are integrated together. Transaction T3 uses CPHASE gate, and the two qubits as its input are provided by the C1 component, which produces the EPR pair. In transaction T6, integration test between EPR generation component and remote state preparation (with input qubits from EPR generation) is carried out. Transaction T4 undergoes tighter integration test by running two RSP procedures in parallel (with angles from components C4 and C5), where the input qubits are produced by component C1. Similarly, T5 applies the CPHASE gate to two qubits that have finished their remote state preparation procedures. The reason why we didn't choose any standalone component in our activity matrix is because we want to test the power of Q-SFL to localize bugs even when we are conducting tightly coupled integration tests.

The input test case associated with a transaction is specified in Table 6.5. The oracle computes the correct output for each of these input cases and the final test suite is designed. As before, set $S = \{|0\rangle, |1\rangle, |+\rangle, |-\rangle, |+i\rangle, |-i\rangle\}$ is the standard input set, set $G = \{I, X, H, R_x(\pi/2)\}$ is the gate set that applies a gate from it to the qubit in consideration, and set $T = \{\pi/2, -\pi/2\}$ contains the two input values used for δ_1 and δ_2 calculations. During the testing phase, if there is a mismatch between the test result and the oracle result for any one of the input case corresponding to a transaction, then the whole transaction will be marked as an error (ErrorVector = 1). The operation of each transaction, along with input initialization is specified in Appendix B.3.

	T1 (12 Cases)	T2 (12 Cases)	T3 (16 Cases)	T4 (16 Cases)	T5 (24 Cases)	T6 (16 Cases)
Qubit1	S	S	G	EPR	S	G
Qubit2	Not Used	Not Used	G	G	$\|0\rangle$	$\|0\rangle$
Theta	T	T	Not Used	T	T	T

Table 6.5: Input Test Case for Testing BQC Application

6.2.1. Localizing a Single Bug

Let us consider the case when there is just one bug in our application. Specifically: a bug in the C3 component. The bug in our code is that instead of the correct CPHASE gate, we apply the CNOT gate.

The error vector after running each transaction is calculated by comparing it with the oracle result for each test case associated with the transaction, and the resulting ac-

tivity matrix is shown in Table 6.6.

	T1	T2	T3	T4	T5	T6
C1	0	0	1	1	0	1
C2	1	1	0	1	1	1
C3	0	0	1	0	1	0
C4	1	0	0	1	1	0
C5	0	1	0	1	1	0
e	0	0	1	0	0	0

Table 6.6: Activity Matrix for Single Bug BQC Application

From the activity matrix, we find that the only transaction that has failed is T3. That is, atleast one of the 16 input case corresponding to transaction T3 produces a result that does not coincide with the correct result we get from oracle. The corresponding candidate is (c_1, c_3). Since there are no other candidate, we do not need to use the Candidate Ranking Algorithm to find the probability. In this example, the sole candidate generated is in fact enough. It was able to correctly localize the bug to one of (c_1, c_3), which is exactly what we want.

6.2.2. Localizing Multiple Bugs

In this case, we consider bugs in all 5 components. The different bugs in each component is listed as follows:

- C1 : Generating $|\psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$ instead of $|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$
- C2 : Applying rotation by angle $-\theta$ instead of $+\theta$ in the $Rot\_Z()$ command.
- C3 : Applying CNOT gate instead of CPHASE Gate
- C4 : Mathematical Error in Calculating Delta1
- C5 : Mathematical Error in Calculating Delta2

The error vector after running each transaction is calculated by executing the code listed in Appendix B.3 and comparing its result with test case result that is provided by the oracle. The Activity Matrix with the error vector is presented in Table 6.7.

	T1	T2	T3	T4	T5	T6
C1	0	0	1	1	0	1
C2	1	1	0	1	1	1
C3	0	0	1	0	1	0
C4	1	0	0	1	1	0
C5	0	1	0	1	1	0
e	1	1	1	1	0	0

Table 6.7: Activity Matrix for Multiple Bug BQC Application

From the candidate generation step, the minimal candidate set are:

- $d_1 = \{c_1, c_2\}$

- $d_2 = \{c_2, c_3\}$
- $d_3 = \{c_1, c_4, c_5\}$
- $d_4 = \{c_3, c_4, c_5\}$

Now that we have more than one minimal candidates, we start ranking each candidate according to their probability of bug.

Assuming the initial probability that a component has a bug is the same $(= \frac{1}{5})$ for all component, the prior probability of having bug for each candidate is:

- $Pr(d_1) = (\frac{1}{5})^2 \cdot (1 - \frac{1}{5})^3$
- $Pr(d_2) = (\frac{1}{5})^2 \cdot (1 - \frac{1}{5})^3$
- $Pr(d_3) = (\frac{1}{5})^3 \cdot (1 - \frac{1}{5})^2$
- $Pr(d_4) = (\frac{1}{5})^3 \cdot (1 - \frac{1}{5})^2$

The next step is to bias the prior probability taking the Activity Matrix in to account:

- $Pr((A, e)|d_1) = (1 - g_2)^2 \cdot (1 - g_1) \cdot (1 - g_1 g_2) \cdot g_2^2 g_1$
  Where (g_1, g_2) is the Goodness parameter of Component (c_1, c_2)
- $Pr((A, e)|d_2) = (1 - g_2)^3 \cdot (1 - g_3) \cdot g_2^2 \cdot g_3$
  Where (g_2, g_3) is the Goodness parameter of Component (c_2, c_3)
- $Pr((A, e)|d_3) = (1 - g_4) \cdot (1 - g_5) \cdot (1 - g_1) \cdot (1 - g_1 g_4 g_5) \cdot g_1 g_4 g_5$
  Where (g_1, g_4, g_5) is the Goodness parameter of Component (c_1, c_4, c_5)
- $Pr((A, e)|d_4) = (1 - g_4) \cdot (1 - g_5) \cdot (1 - g_3) \cdot (1 - g_4 g_5) \cdot g_3 g_4 g_5$
  Where (g_3, g_4, g_5) is the Goodness parameter of Component (c_3, c_4, c_5)

In order to estimate the value of $Pr((A, e)|d_k)$, we use M.L.E. Thus, maximizing for each g_i in each d_i, we find that:

- $Pr((A, e)|d_1) = 0.01213629629$
- $Pr((A, e)|d_2) = 0.00864$
- $Pr((A, e)|d_3) = 0.0138522$
- $Pr((A, e)|d_4) = 0.012224$

Thus, using the Bayes Theorem, we have:

- $Pr(d_1|(A, e)) = Pr(d_1) \cdot Pr((A, e)|d_1) = 0.0002485$

- $Pr(d_2|(A,e)) = Pr(d_2) \cdot Pr((A,e)|d_2) = 0.0001769$
- $Pr(d_3|(A,e)) = Pr(d_3) \cdot Pr((A,e)|d_3) = 0.00007092$
- $Pr(d_4|(A,e)) = Pr(d_4) \cdot Pr((A,e)|d_4) = 0.00006258$

Normalizing,

- $Pr(d_1|(A,e)) = 0.44$
- $Pr(d_2|(A,e)) = 0.32$
- $Pr(d_3|(A,e)) = 0.13$
- $Pr(d_4|(A,e)) = 0.11$

From this, we find that the probability of candidates d_1 and d_2 are way higher than that of d_3 and d_4. This justifies an inspection on the components in those candidates (which are ($\{c_1, c_2\}$) and ($\{c_2, c_3\}$)). Thus, the SFL technique has successfully localized bugs in components C_1, C_2, C_3.

This does not however localize all the bugs we have. In order to see whether there are any bugs left, we have to run SFL once again. Only when all the transactions pass can we stop. Therefore, after correcting the bugs in C_1, C_2, C_3 components, in the next iteration of SFL, we have the following Activity Matrix Table 6.8:

	T1	T2	T3	T4	T5	T6
C1	0	0	1	1	0	1
C2	1	1	0	1	1	1
C3	0	0	1	0	1	0
C4	1	0	0	1	1	0
C5	0	1	0	1	1	0
e	1	1	0	1	0	0

Table 6.8: Activity Matrix for Multiple Bug BQC Application, Second Iteration

After Candidate Generation step, the minimal candidate set are:

- $d_1 = \{c_2\}$
- $d_2 = \{c_4, c_5\}$

Now be begin the candidate ranking step.

Assuming the initial probability that a component has a bug $= \frac{1}{5}$ for all component, the prior probability of having bug for each candidate is:

- $Pr(d_1) = (\frac{1}{5})^1 \cdot (1 - \frac{1}{5})^4$
- $Pr(d_2) = (\frac{1}{5})^2 \cdot (1 - \frac{1}{5})^3$

The next step is to bias the prior probability taking the Activity Matrix in to account:

- $Pr((A,e)|d_1) = (1-g_2)^3 \cdot g_2^2$
  Where g_2 is the Goodness parameter of Component c_2
- $Pr((A,e)|d_2) = (1-g_4) \cdot (1-g_5) \cdot (1-g_4g_5) \cdot g_4g_5$
  Where (g_4, g_5) is the Goodness parameter of Component (c_4, c_5)

In order to estimate the value of $Pr((A,e)|d_k)$, we use M.L.E. Thus, maximizing for each g_i, we find that:

- $Pr((A,e)|d_1) = 0.03456$
- $Pr((A,e)|d_2) = 0.0489$

Thus, using the Bayes Theorem, we have:

- $Pr(d_1|(A,e)) = Pr(d_1) \cdot Pr((A,e)|d_1) = 0.00283$
- $Pr(d_2|(A,e)) = Pr(d_2) \cdot Pr((A,e)|d_2) = 0.00100$

Normalizing,

- $Pr(d_1|(A,e)) = 0.74$
- $Pr(d_2|(A,e)) = 0.26$

The end result is that SFL is still telling us that component C_2 has the highest probability of all components to have a bug. However, candidate d_2 that has the components C_4, C_5 still has a probability of 0.26 of having a bug in it.

The reason why we are getting a high probability for C_2 is because in our transactions, whenever either of C_4 or C_5 is used, C_2 is also used. Thus, C_2 has a higher chance of having a bug in it solely because its used along with C_4 and C_5. However, given that we have already debugged C_2 component, and since the probability of bug for candidate d_2 is not miniscule, we should also look into the components inside the d_2 candidates, which are C_4 and C_5. Thus, using this reasoning, we are able to localize the remaining two bugs.

Thus, we were able to localize all 5 bugs with two iterations. Once all 5 components are debugged, we should run SFL once again and see if there are any error vectors. The activity matrix of this run in shown in Table 6.9.

We see that no transaction is flagged as an error due to the fact that the results after running all input configuration for all transaction coincides with that of the corresponding result from the oracle. Thus, all of our tests have passed and we weren't able to detect any bugs in our application, which for this case is true indeed.

	T1	T2	T3	T4	T5	T6
C1	0	0	1	1	0	1
C2	1	1	0	1	1	1
C3	0	0	1	0	1	0
C4	1	0	0	1	1	0
C5	0	1	0	1	1	0
e	0	0	0	0	0	0

Table 6.9: Activity Matrix for Multiple Bug BQC Application, Third Iteration

7

Evaluation of Techniques and Testing Framework

The Q-SFL Technique is evaluated on practical settings by studying if it can detect and localize different kinds of bugs for the quantum state teleportation and BQC Application.

Along with that, the effectiveness of the testing framework is compared with an ad-hoc testing scheme for the same practical setting. We would like study whether the framework presented is better at detecting and localizing bugs than the ad-hoc testing method.

Results presented in the previous chapter show us that the SFL technique theoretically works well. Next, we look into how SFL works in a practical setting that has noisy hardware. By changing the hardware noise parameters, we check whether SFL is able to detect and localize bugs for different values. It also gives us a hint of how bad our hardware can get after which we cannot detect or localize bugs. Another inference we can make is on the correspondence between the hardware quality and our application: if the hardware gets so bad that the result we get from the correct code is indistinguishable from the buggy code, we shouldn't be using that hardware in the first place.

The second part of the study evaluates the SFL testing framework by comparing it with the standard ad-hoc testing framework (based on algorithm 2). This is crucial to study whether the new technique devised actually performs better than the already existing ad-hoc method. We find out the performance of standard testing framework and see if it can detect different types of mutants in a perfect noiseless cases. Then, we check how effective it will be on noisy case when we run it on degrading hardware parameters.

For both these evaluations, we utilise mutation testing [25]. In mutation testing, the tester purposefully introduces a fault in the code to see if the testing framework is able to detect and debug it. Mutation testing is employed to evaluate the quality of existing software tests and also to devise good test cases. The ”buggy” code that is purposefully introduced by the tester is referred to as a mutant. If the testing framework is able to detect bugs in the mutant, we say that the framework has ”killed” the mutant. Ideally, the testing framework should be able to kill all mutants in the software.

7.1. Evaluating Q-SFL Technique on State Teleportation

We now evaluate the performance of our Q-SFL Technique on state teleportation protocol on varying hardware noise parameters. The parameters we are interested in are gate fidelity and entanglement fidelity, both of which take a value from the set $(0.6, 0.7, 0.8, 0.9)$. Thus, for entanglement fidelity e_i and gate fidelity g_i, (where $e_i, g_i \in (0.6, 0.7, 0.8, 0.9)$) we run Q-SFL for that particular parameter setting and see how many bugs are localized for a particular mutant. The mutants we consider and the ability of Q-SFL to localize bugs is presented in the following sections. The input values and activity matrix we consider are the same as we have defined in section 6.1.

7.1.1. Mutant 1: Fault in the Bell Measurement Component

In the first type of mutant we are interested in, the order of the CNOT Operation in the Bell Measurement Component is wrong. The correct code and the mutant code is listed below, with the bug in Line 2:

```
1 def bellStateMeasurement():
2     qubit.cnot(epr)
3     qubit.H()
4     m1 = qubit.measure()
5     m2 = epr.measure()
6     return m2, m1
```

Listing 7.1: Correct Code in Component C2

```
1 def bellStateMeasurement():
2   epr.cnot(qubit)
3   qubit.H()
4   m1 = qubit.measure()
5   m2 = epr.measure()
6   return m2, m1
```

Listing 7.2: Buggy Mutant in Component C2

The bug in the mutant exists in the C2 component and SFL ideally should be able to detect it for each of the hardware parameter value (e_i, g_i). However, this is not the case as shown in Fig 7.1.

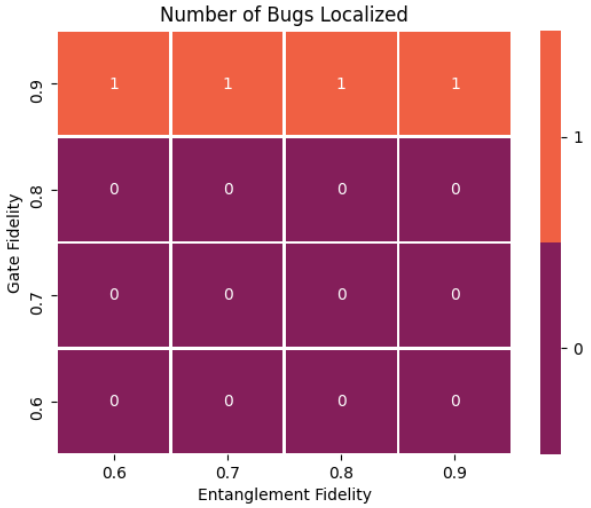


Figure 7.1: Number of Bugs Localized in Mutant 1 for the State Teleportation Application

We find that when the gate fidelity is up to 0.9, we are able to correctly detect and localize the bug. However, when the gate fidelity is 0.8 and below, the SFL technique is not able to localize the bug. The noise produced by gate fidelity impacts the results of each transaction so much that it is difficult to distinguish between the result from buggy code and the correct code. Since this generates an error vector of 0 in the activity matrix, SFL technique is not even used for these particular settings.

7.1.2. Mutant 2: Fault in the Pauli Correction

In this mutant, the correction operation in Component C3 is wrong. Instead of a X gate in Line 3, the bug is that we apply a Z gate.

```
1 def pauliCorrection(qubit, m1, m2):
2   if m2 == 1:
3     qubit.X()
4   if m1 == 1:
5     qubit.Z()
```

Listing 7.3: Correct Code in Component C3

```
1 def pauliCorrection(qubit, m1, m2):
2     if m2 == 1:
3       qubit.Z()
4     if m1 == 1:
5       qubit.Z()
```

Listing 7.4: Buggy Mutant in Component C3

The result for different values of (e_i, g_i) is presented in Figure 7.2. Since there is only

one defect in this mutant that is present in Component C2, we find that we are able to localize it for the given values of entanglement fidelity and gate fidelity.

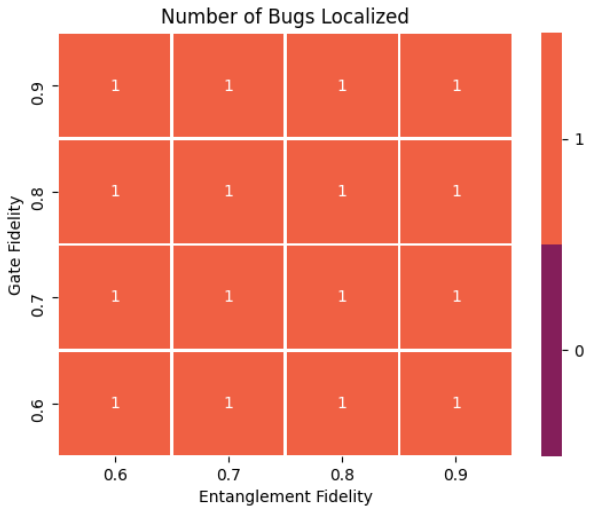


Figure 7.2: Number of Bugs Localized in Mutant 2 for the State Teleportation Application

7.1.3. Mutant 3: EPR error

In this mutant, instead of the EPR state $|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, we generate the EPR state $|\psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$. This bug is present in component C1, and the correct code and the mutant code for that component is listed in 7.5, 7.6:

```
1 def eprGen(epr_socket):
2   qubit = epr_socket.create()
3   return qubit
```

Listing 7.5: Correct Code in Component C1

```
1 def eprGen(epr_socket):
2   qubit = epr_socket.create()
3   qubit.X()
4   return qubit
```

Listing 7.6: Buggy Mutant in Component C1

The number of defects localized for different values of (e_i, g_i) is presented in Figure 7.3. Once again, we are able to localize the bug to the C1 component for the given values of entanglement fidelity and gate fidelity.

To see how bad our hardware parameter should get for SFL to not localize this bug, we conducted the test by which we degrade the entanglement fidelity value form 0.9 to 0 by keeping the gate fidelity constant and equal to 0.9. The result of this test is presented in Table 7.1:

	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1	0.0
Does it localize?	YES	YES	YES	YES	YES	YES	YES	YES	NO	NO

Table 7.1: Localizing Mutant M3 with Different Entanglement Fidelity for State Teleportation

We see that SFL is able to differentiate between the result from the buggy code and correct code until the entanglement fidelity drops to 0.1, after which the results are hard to distinguish and hence we are not able to detect or correctly localize the bug.

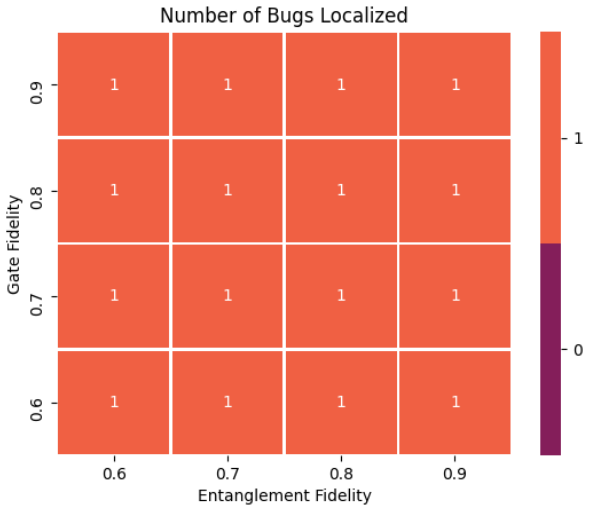


Figure 7.3: Number of Bugs Localized in Mutant 3 for State Teleportation Application

7.1.4. Mutant 4: BSM Result & Pauli Correction Order Swapped

For this section, we test the performance of SFL on the same bug as discussed in 6.1.1 in which the order of result returned after running Component C2 and the Pauli Correction order in Component C3 are swapped. Although it was theoretically possible to localize the bugs to both of these components, it will be interesting to see the role noise plays in its performance. This result is presented in Figure 7.4.

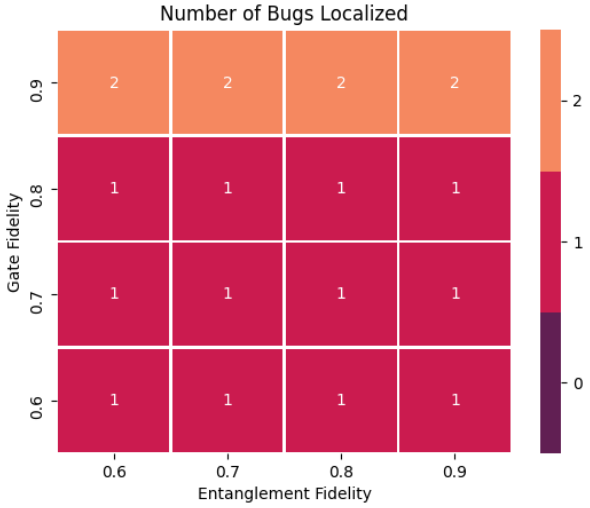


Figure 7.4: Number of Bugs Localized in Mutant 4 for State Teleportation Application

We find that we are able to localize the bugs in both the components when the gate fidelity is 0.9. For gate fidelity $g_i \in \{0.8, 0.7, 0.6\}$, the Activity Matrix produced is presented in Table 7.2.

From the activity matrix, we find that the candidate generation algorithm produces the sole candidate (c_3). Hence, we are able to localize the bug in that component only. For $g_i < 0.9$, we are not able to detect or localize bugs in Component C2 because it is hard to distinguish between the results from correct code and mutant code.

	T1	T2	T3	T4
C1	1	0	0	0
C2	0	1	0	1
C3	1	0	1	1
e	1	0	1	0

Table 7.2: Activity Matrix for Testing State Teleportation Mutant 4 with Gate Fidelity less than 0.9

7.2. Evaluating SFL Technique on Blind Quantum Computation Application

We now evaluate the performance of our Q-SFL Technique on the BQC application by varying hardware noise parameters. The parameters we are interested in are entanglement fidelity e_i, gate fidelity g_i, both which takes a value from $(0.6, 0.7, 0.8, 0.9)$, and host latency hl, which takes values (100ms, 250ms, 500ms, 750ms, 1s). We run SFL for a particular parameter setting (e_i, g_i, hl_i) and see how many bugs are localized for a particular mutant. The mutants we consider and the ability of SFL to localize bugs are presented in the following sections. The input values we consider are the same as we have defined in section 6.2.

7.2.1. Mutant 1: Bug in CPHASE Command

Here, the bug is that we apply a CNOT gate instead of the CPHASE gate in Component C3. The correct code and mutant code of C3 is presented in Listing 7.7,7.8, with fault in Line 2.

```
1 def cphaseGate(qubit1, qubit2):
2   qubit2.cphase(qubit1)
```

Listing 7.7: Correct implementation of C3 Component

```
1 def cphaseGate(qubit1, qubit2):
2   qubit2.cnot(qubit1)
```

Listing 7.8: Incorrect implementation of C3 Component

When $g_i = 0.9$, we get the Activity Matrix presented in Table 7.3 for every value of (e_i, hl_i) listed above.

	T1	T2	T3	T4	T5	T6
C1	0	0	1	1	0	1
C2	1	1	0	1	1	1
C3	0	0	1	0	1	0
C4	1	0	0	1	1	0
C5	0	1	0	1	1	0
e	0	0	1	0	0	0

Table 7.3: Activity Matrix for BQC Application, Mutant 1, Depolarization Probability Until 0.8

We find that transaction 3 always fails for all values of (e_i, hl_i). The corresponding candidate is (c_3, c_5) and hence the bug is localized to either one of those components. Since this is indeed correct, we are able to achieve the optimal result when $g_i = 0.9$.

However, when $g_i \in \{0.8, 0.7, 0.6\}$, none of the transactions fail because we are not even able to distinguish between the result of the buggy code and mutant code. Thus, we are not able to localize this particular bug when $g_i \in \{0.8, 0.7, 0.6\}$. The number of

bugs localized for varying values of (e_i, g_i, hl_i) is presented in Figure 7.5

A cross sectional view for one particular value of entanglement fidelity is presented in Figure 7.6. An interesting find from this experiment is that the change in entanglement fidelity to ones we are interested in right now does not seem to affect the ability to localize the bug. Even after degrading the fidelity value, we are able to differentiate between correct output and mutant output. This however may not hold for even lower fidelity values.

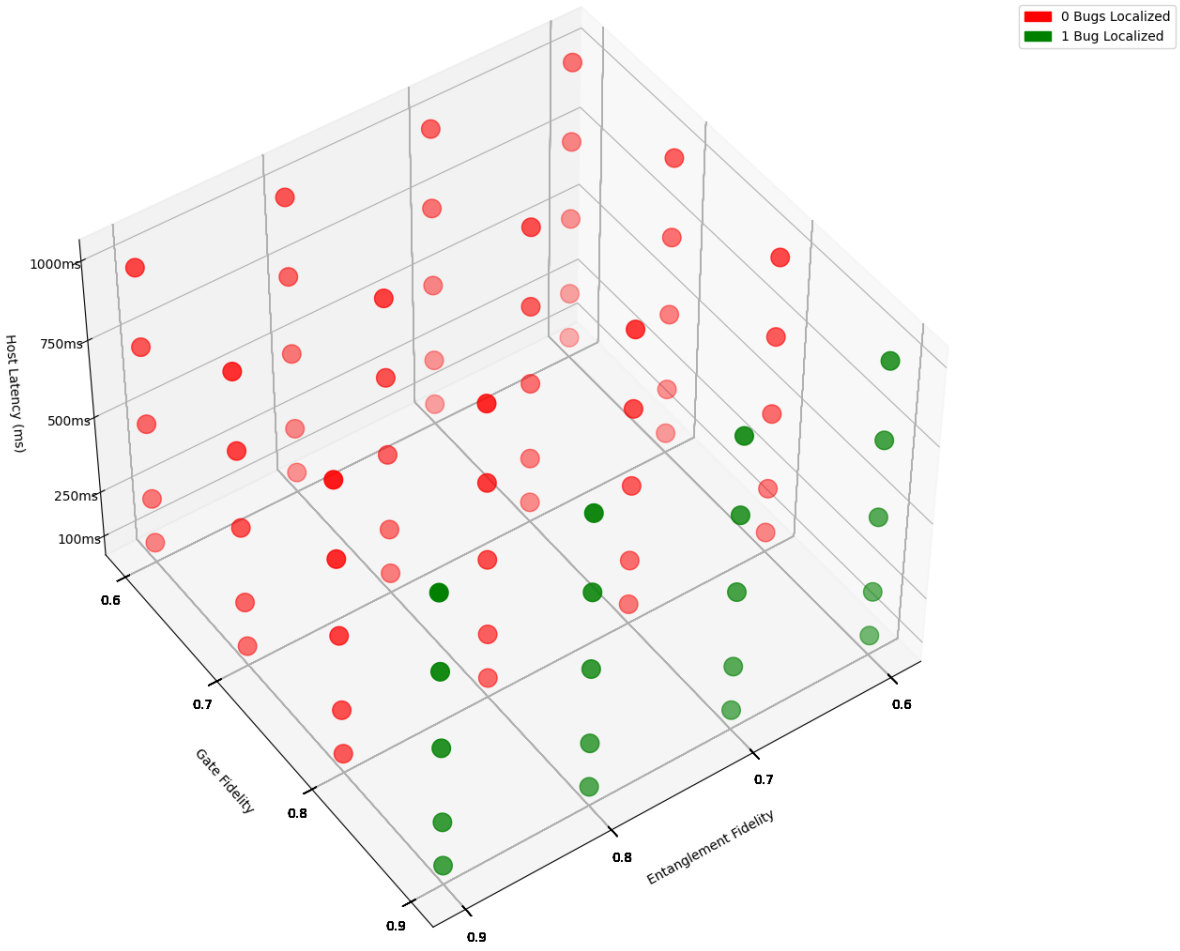
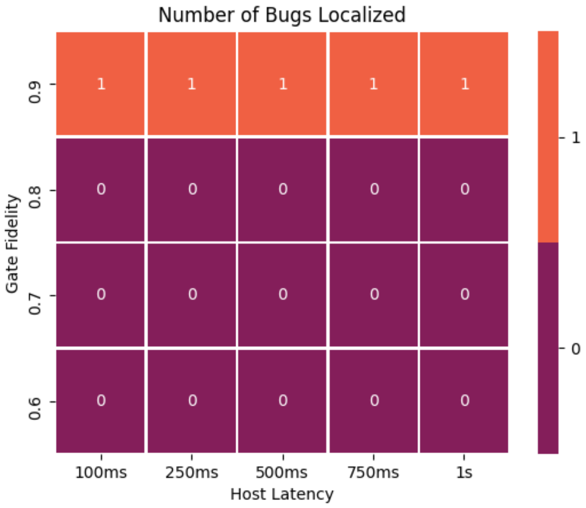


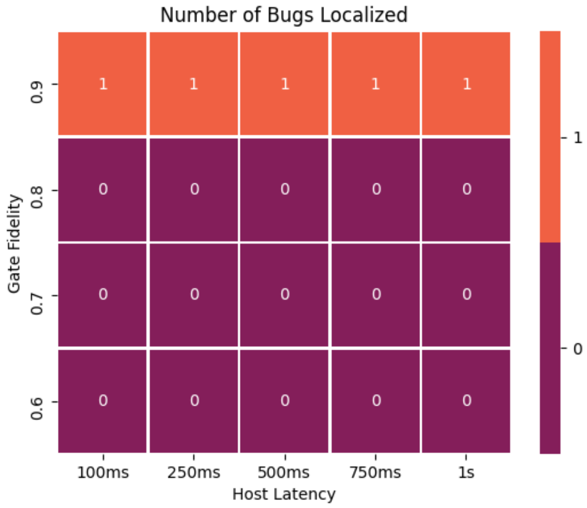
Figure 7.5: Number of Bugs Localized in Mutant 1 for the BQC Application

7.2.2. Mutant 2: Bug in Remote State Preparation

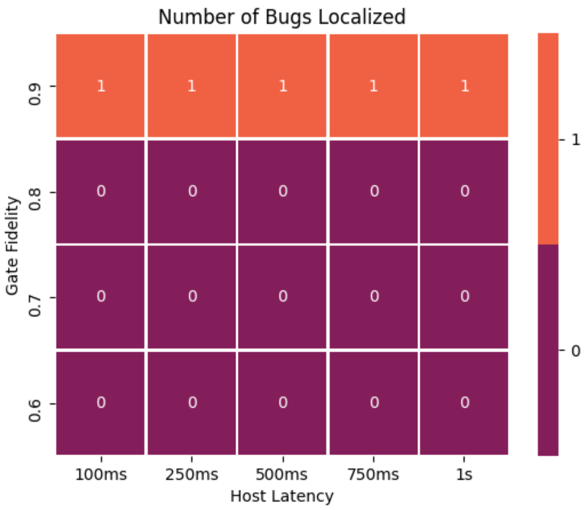
In this mutant, a bug is present in the C2 Component. Specifically, while applying the rotation by the Z axis operation, we negate the angle passed in to the function. The correct code and the mutant is listed below, with bug in Line 2.



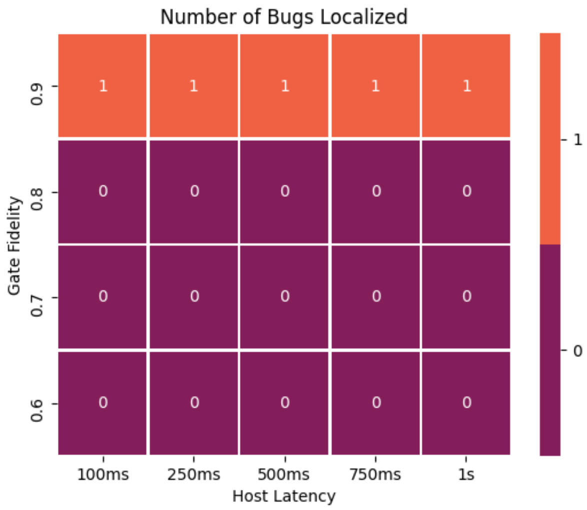
(a) Entanglement Fidelity = 0.9



(b) Entanglement Fidelity = 0.8



(c) Entanglement Fidelity = 0.7



(d) Entanglement Fidelity = 0.6

Figure 7.6: Cross Sectional View of Mutant 1, BQC Application

```
def rsp(qubit, angle):
  qubit.rot_Z(angle=angle)
  qubit.H()
  m = qubit.measure(store_array=False)
  return m
```

Listing 7.9: Correct implementation of C2 Component

```
def rsp(qubit, angle):
  qubit.rot_Z(angle=-angle)
  qubit.H()
  m = qubit.measure(store_array=False)
  return m
```

Listing 7.10: Incorrect implementation of C2 Component

The number of bugs localized for the different values of (e_i, g_i, hl_i) is presented in Figure 7.7. From this figure, and from cross sectional view for a particular entanglement fidelity value as presented in Figure 7.8 we find that the main factor that prevents the SFL technique to detect and localize the bug is the host latency. For host latency values greater than 500ms, we are not able to distinguish between the correct result and mutant result. This is because since the T2 time we have set is 300ms, a high value of host latency implies our qubits would more likely decay. Thus in cases when $hl \geq 500ms$, our correct code produces a result which is indistinguishable to that of results from faulty code.

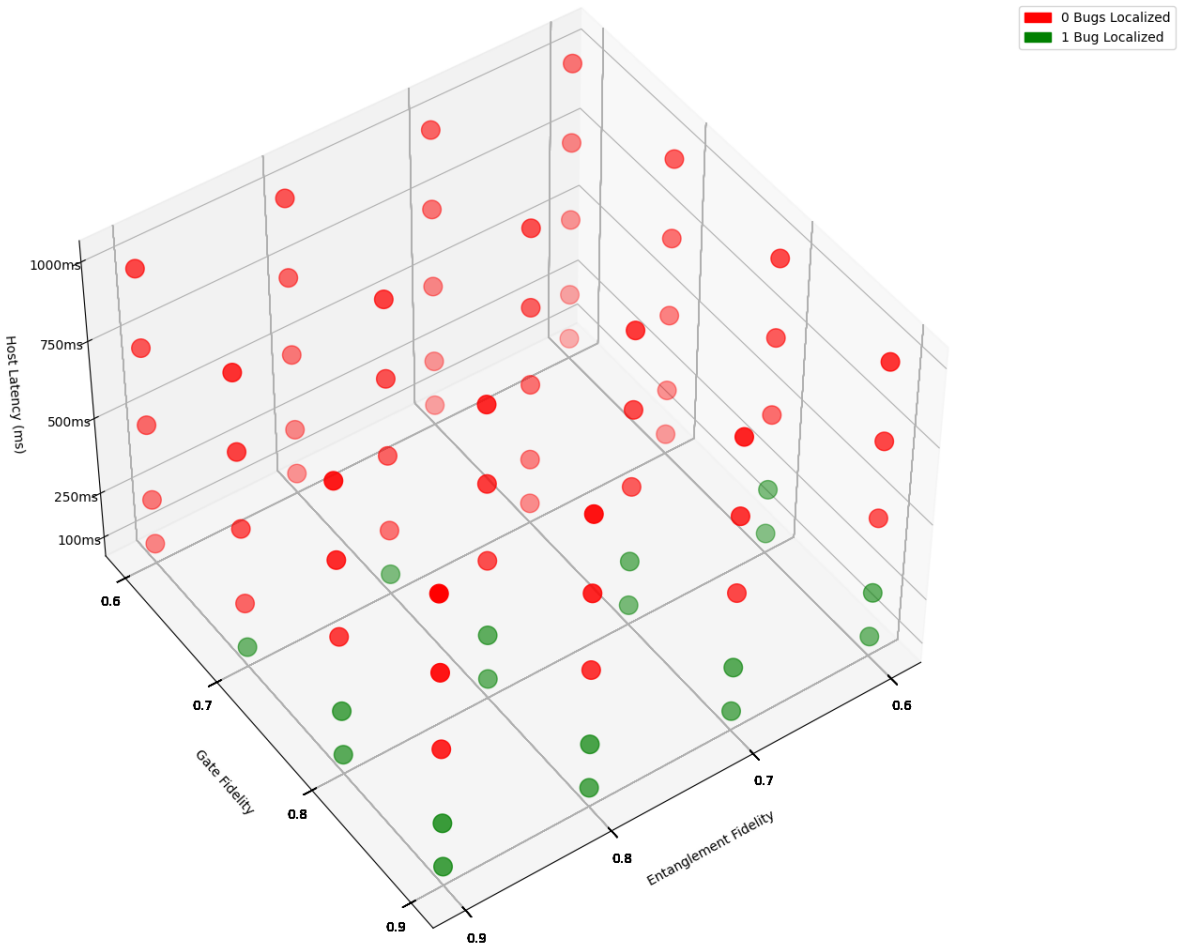
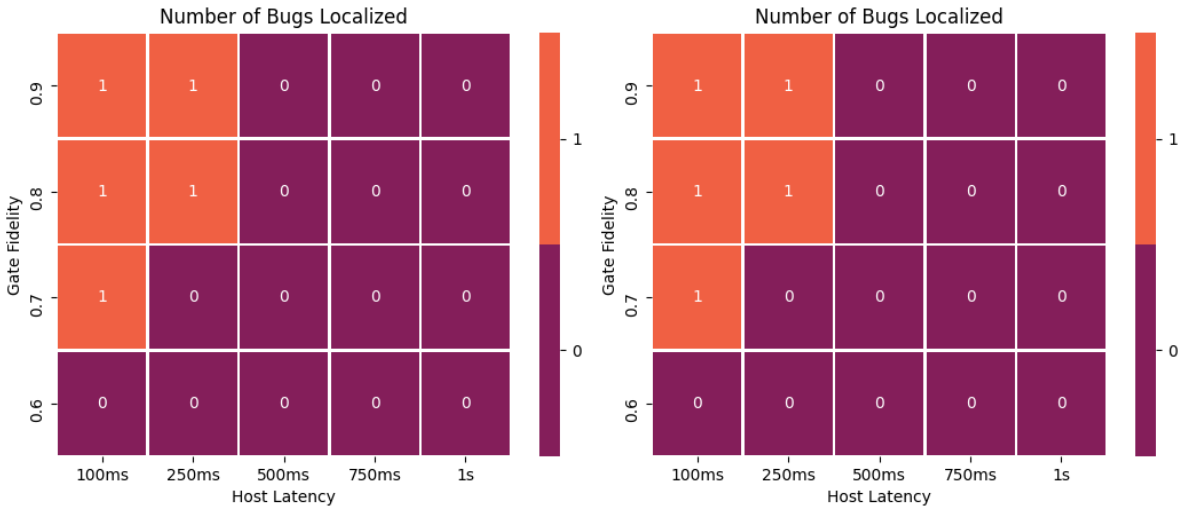
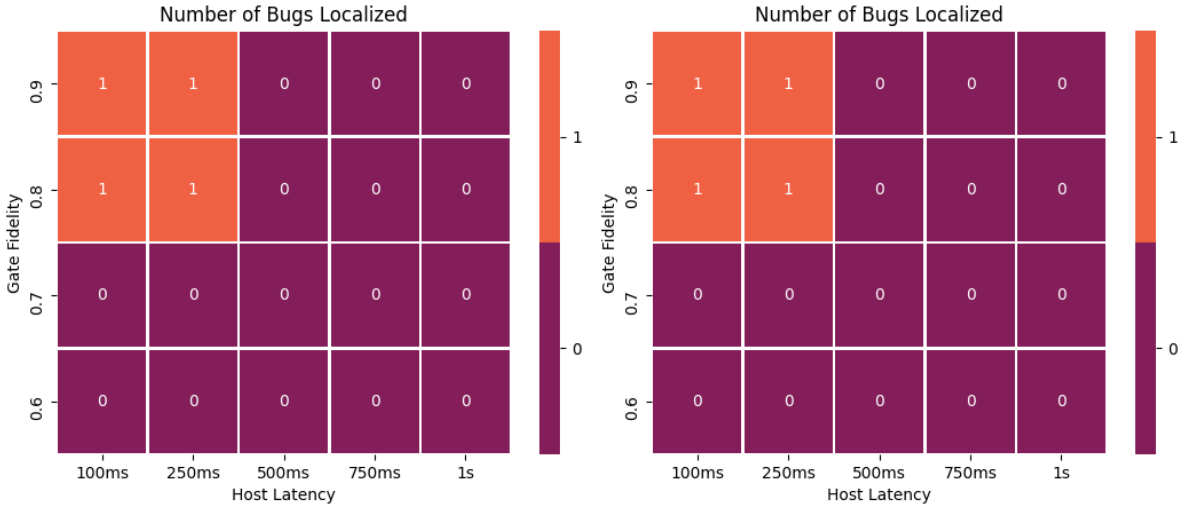


Figure 7.7: Number of Bugs Localized in Mutant 2 for BQC Application



(a) Entanglement Fidelity = 0.9

(b) Entanglement Fidelity = 0.8



(c) Entanglement Fidelity = 0.7

(d) Entanglement Fidelity = 0.6

Figure 7.8: Cross Sectional View of Mutant 2, BQC Application

7.2.3. Mutant 3: Bug in EPR Pair Generation

In this mutant, a bug is present in the C3 Component. Similar to Mutant 3 as discussed in 7.1.3, the bug is that we produce the EPR Pair $|\psi^+\rangle$ instead of $|\phi^+\rangle$. The number of bugs localized for different values of (e_i, g_i, hl_i) is presented in Figure 7.9.

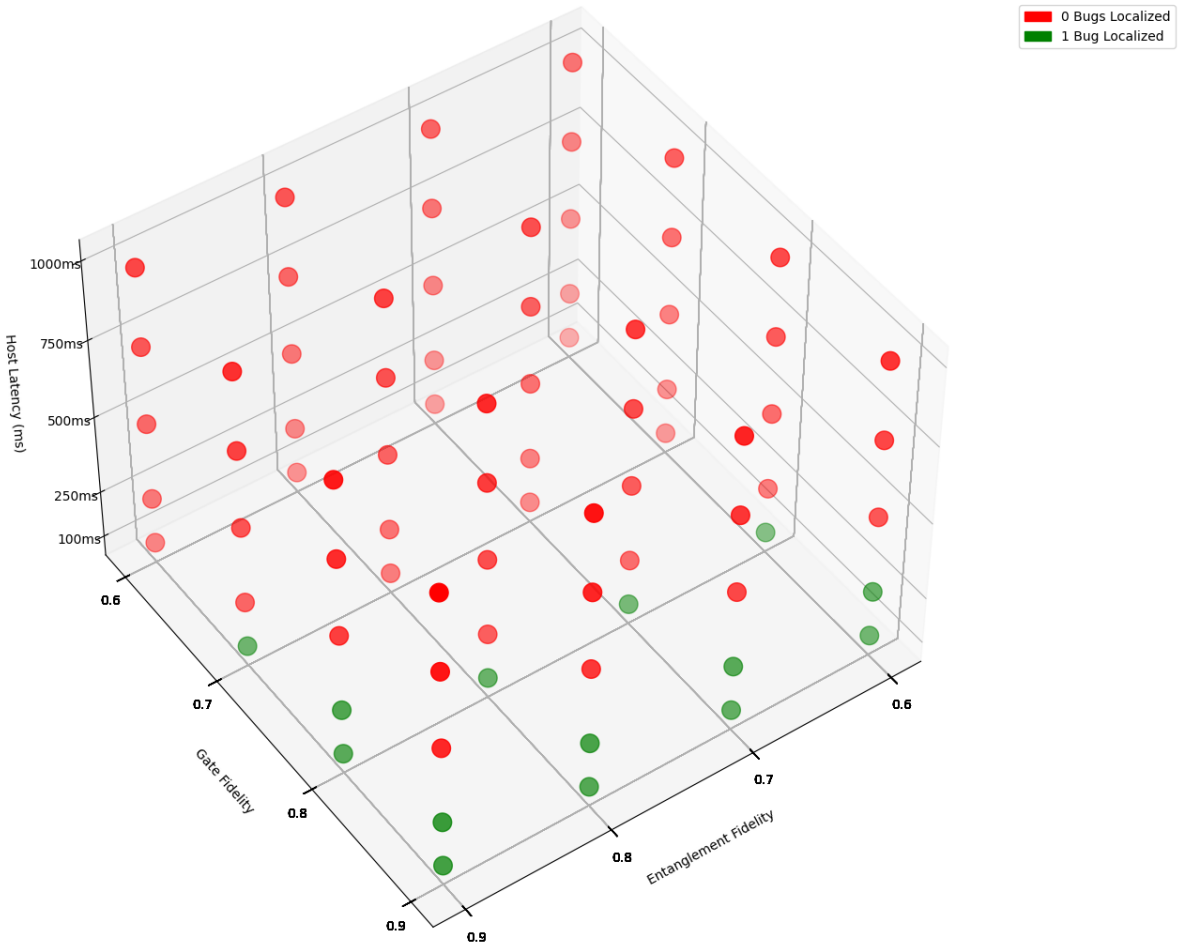
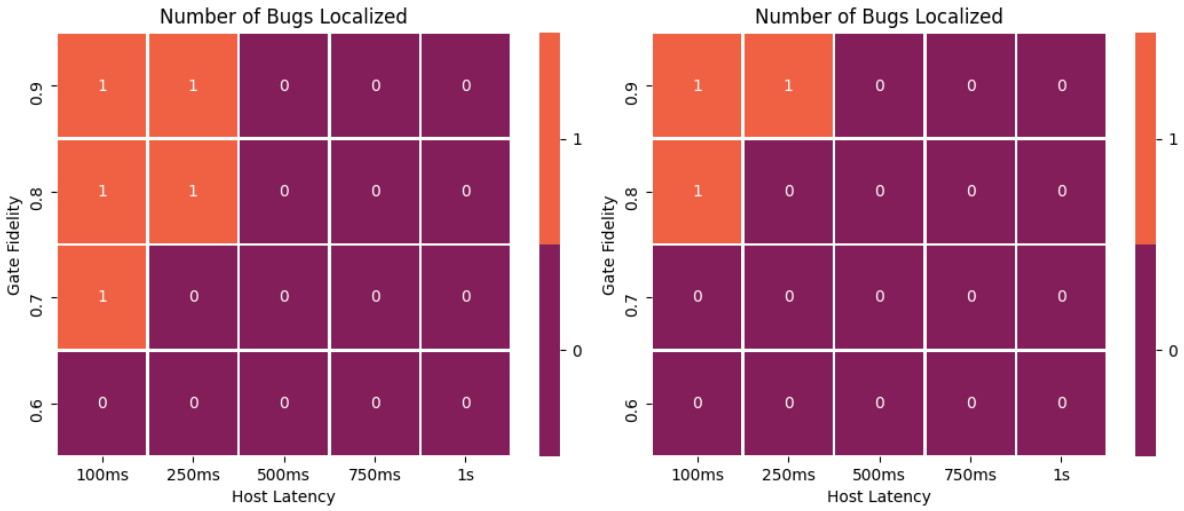


Figure 7.9: Number of Bugs Localized in Mutant 3 for BQC Application

Similar to the case of Mutant 2, the testing technique fails to detect bugs when the host latency is greater than or equal to 500ms. As shown in the cross sectional view for each entanglement fidelity value presented in Figure 7.10, we find that degrading entanglement fidelity also impacts the testing technique's ability to detect bugs.

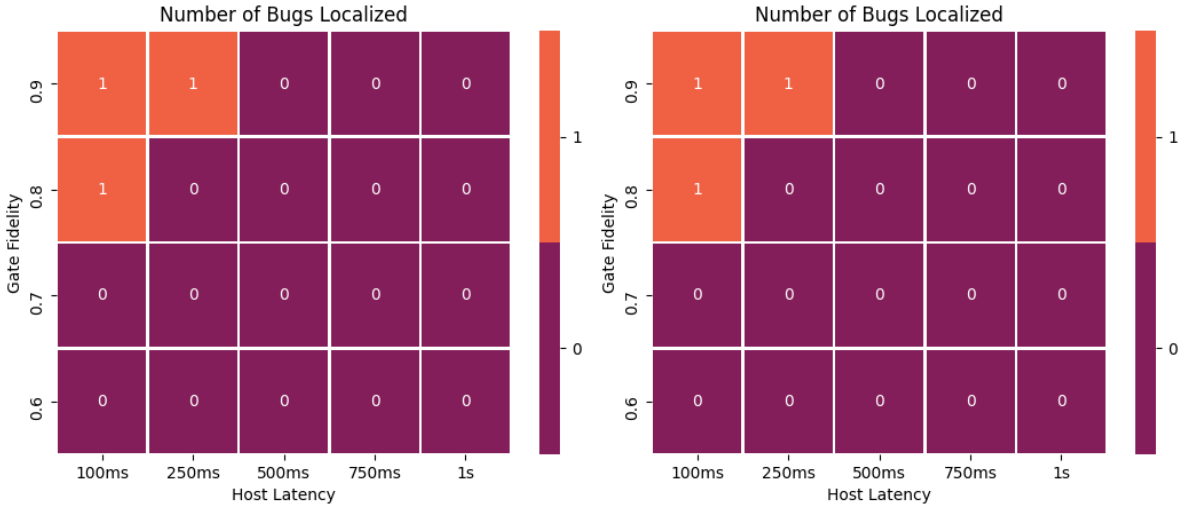
7.2.4. Mutant 4: Bugs in RSP and Delta1 Calculation

All the mutants considered so far had only 1 component that had a bug. In the following tests, we will have bugs in more that one component. For the mutant we will consider in this section, a bug is present in both C2 and C4 Components. The bug in C2 component is the same bug as present in Mutant 2, which is discussed in 7.2.2. The bug present in C4 component is that the calculation used to produce Delta1 value is wrong. The correct code and incorrect code of C4 component is listed in 7.11, 7.12, with the seeded defect present in Line 2.



(a) Entanglement Fidelity = 0.9

(b) Entanglement Fidelity = 0.8



(c) Entanglement Fidelity = 0.7

(d) Entanglement Fidelity = 0.6

Figure 7.10: Cross Sectional View of Mutant 3, BQC Application

```
def calcDelta1(alpha, theta1, p1, r1):
  return alpha-theta1+(p1+r1)*math.pi
```

Listing 7.11: Correct implementation of C4 Component

```
def calcDelta1(alpha, theta1, p1, r1):
  return alpha+theta1+(p1+r1)*math.pi/2
```

Listing 7.12: Incorrect implementation of C4 Component

The number of bugs localized for different values of (e_i, g_i, hl_i) is presented in Figure 7.11, with cross sectional view for each entanglement value presented in 7.12.

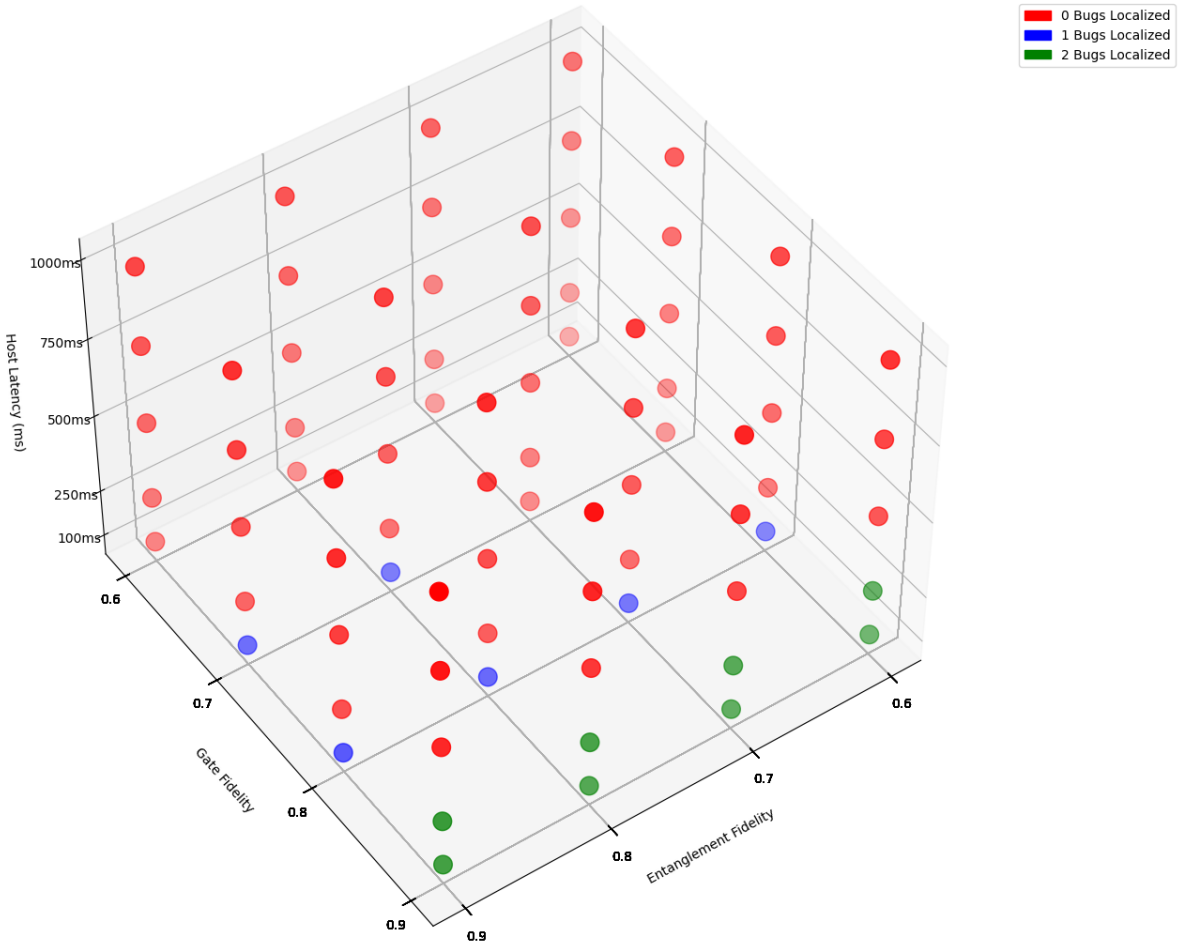
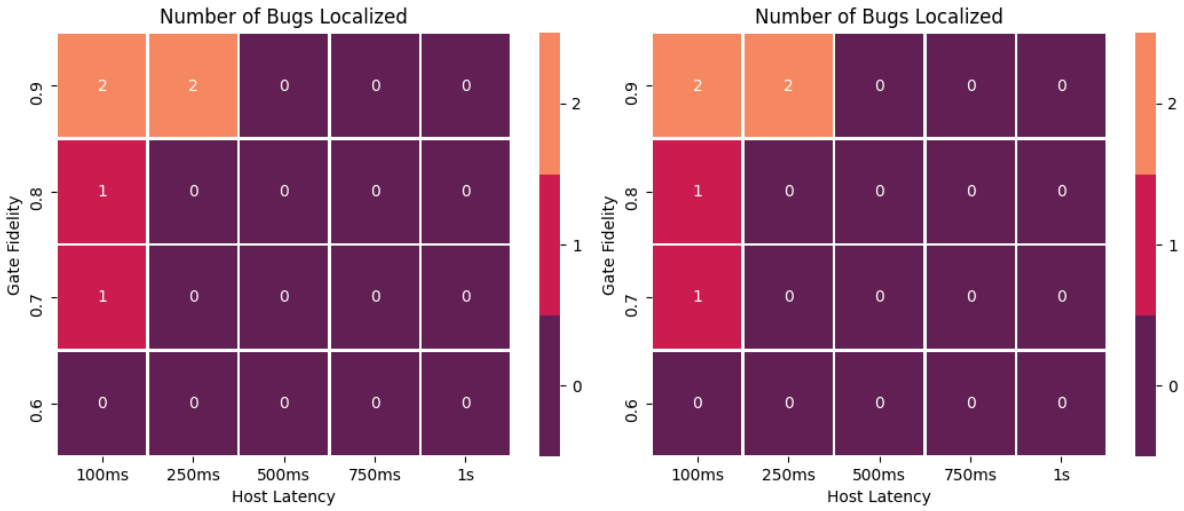


Figure 7.11: Number of Bugs Localized in Mutant 4 for BQC Application

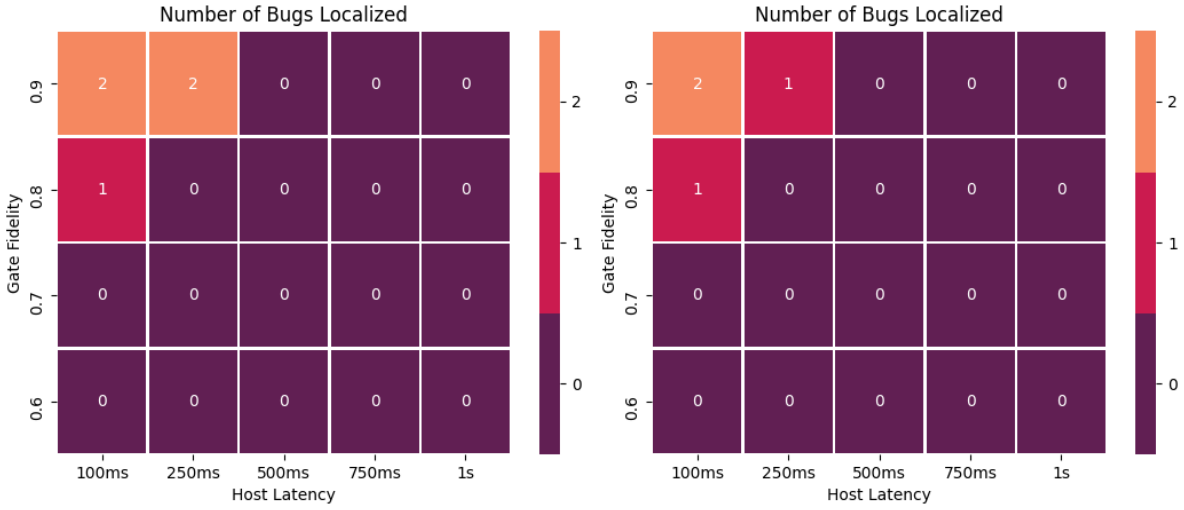
When host latency is 100ms and gate fidelity is 0.9, we find that transactions 1,2, and 6 fail. The sole candidate generated is $c2$ and hence we are able to localize the bug in C2 component. After correcting the bug in C2, we run the SFL technique again and find that transaction T1 has failed. The candidate generated from that is $(c1, c4)$, thus localizing the remaining bug in Component C4. For lower value of g_i, only transaction T6 fails, and thus the sole candidate (c1,c2) only localizes bug in Component C2. A possible reason why none of T1 or T2 transaction fail for the given input value is because the noise present in the hardware makes it harder to distinguish between the result when both the bugs are present.

For the case when host latency is 250ms and $g_i = 0.9, e_i \geq 0.7$, we find that transactions 1,2, and 6 fail. Thus similarly as before, we localize the bug in C2 and then the bug in C4 after running SFL again. For higher value of host latency, even a single



(a) Entanglement Fidelity = 0.9

(b) Entanglement Fidelity = 0.8



(c) Entanglement Fidelity = 0.7

(d) Entanglement Fidelity = 0.6

Figure 7.12: Cross Sectional View of Mutant 4, BQC Application

RSP call decays the qubit to a state in which we are not able to distinguish between faulty code and mutant code.

7.2.5. Mutant 5: Bugs in RSP and Delta2 Calculation

In this mutant, we introduce a fault in components C2 and C5. The fault in C2 is the same as in Mutant 2, and the fault in calculating delta 2 (C5 component) is presented below:

```
1 def calcDelta2(beta,theta2,p2,r2,m1,r1):
2   return math.pow(-1,(m1+r1))*beta
3         -theta2+(p2+r2)*math.pi
```

Listing 7.13: Correct implementation of C5 Component

```
1 def calcDelta2(beta,theta2,p2,r2,m1,r1):
2   return math.pow(-1,(m1+r1))*beta
3         +theta2+(p2+r2)*math.pi/2
```

Listing 7.14: Incorrect implementation of C5 Component

The number of bugs localized for different values of (e_i, g_i, hl_i) is presented in Figure 7.13, with cross sectional view for each entanglement value presented in 7.14.

The results coincide with that of Mutant 4, and the same reasoning applies for this case too.

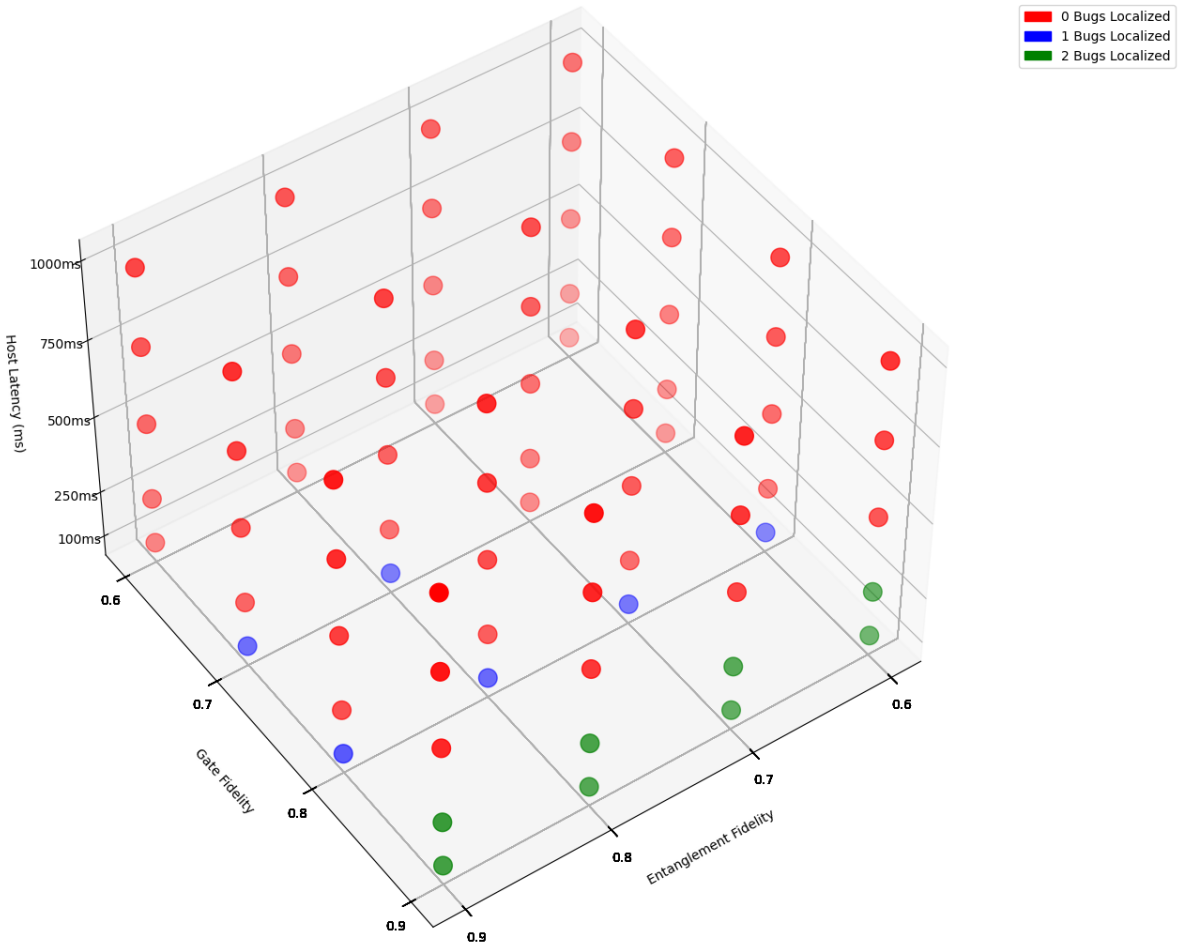
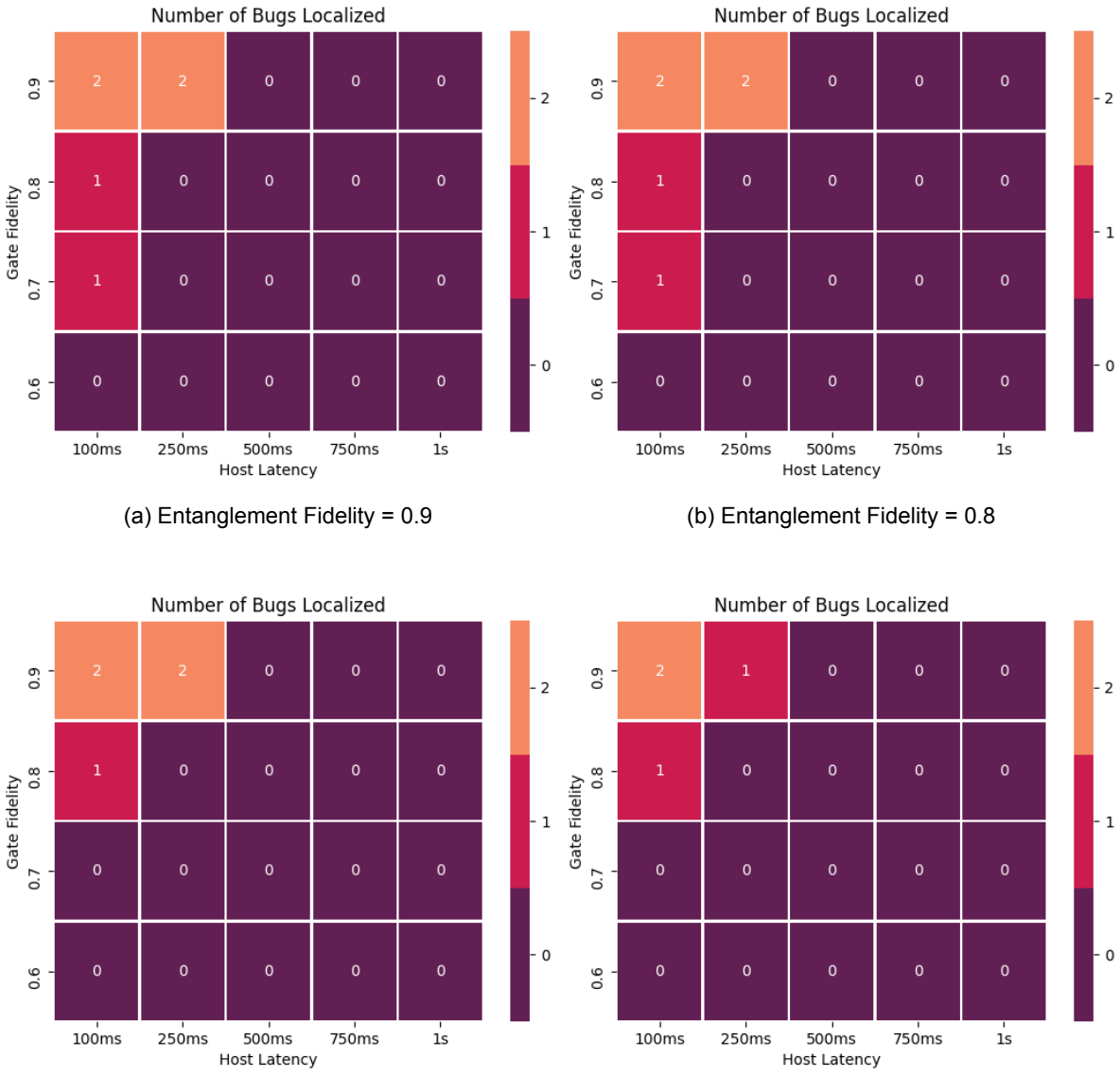


Figure 7.13: Number of Bugs Localized in Mutant 5 for BQC Application



(a) Entanglement Fidelity = 0.9

(b) Entanglement Fidelity = 0.8

(c) Entanglement Fidelity = 0.7

(d) Entanglement Fidelity = 0.6

Figure 7.14: Cross Sectional View of Mutant 5, BQC Application

7.2.6. Mutant 6: Multiple Faults

In the final mutant, we inject faults in 4 different components - C1, C2, C4 and C5. This is equivalent to the combination of mutants M3, M4 and M5. The number of bugs localized for different values of (e_i, g_i, hl_i) is presented in Figure 7.15, with cross sectional view for each entanglement value presented in 7.16.

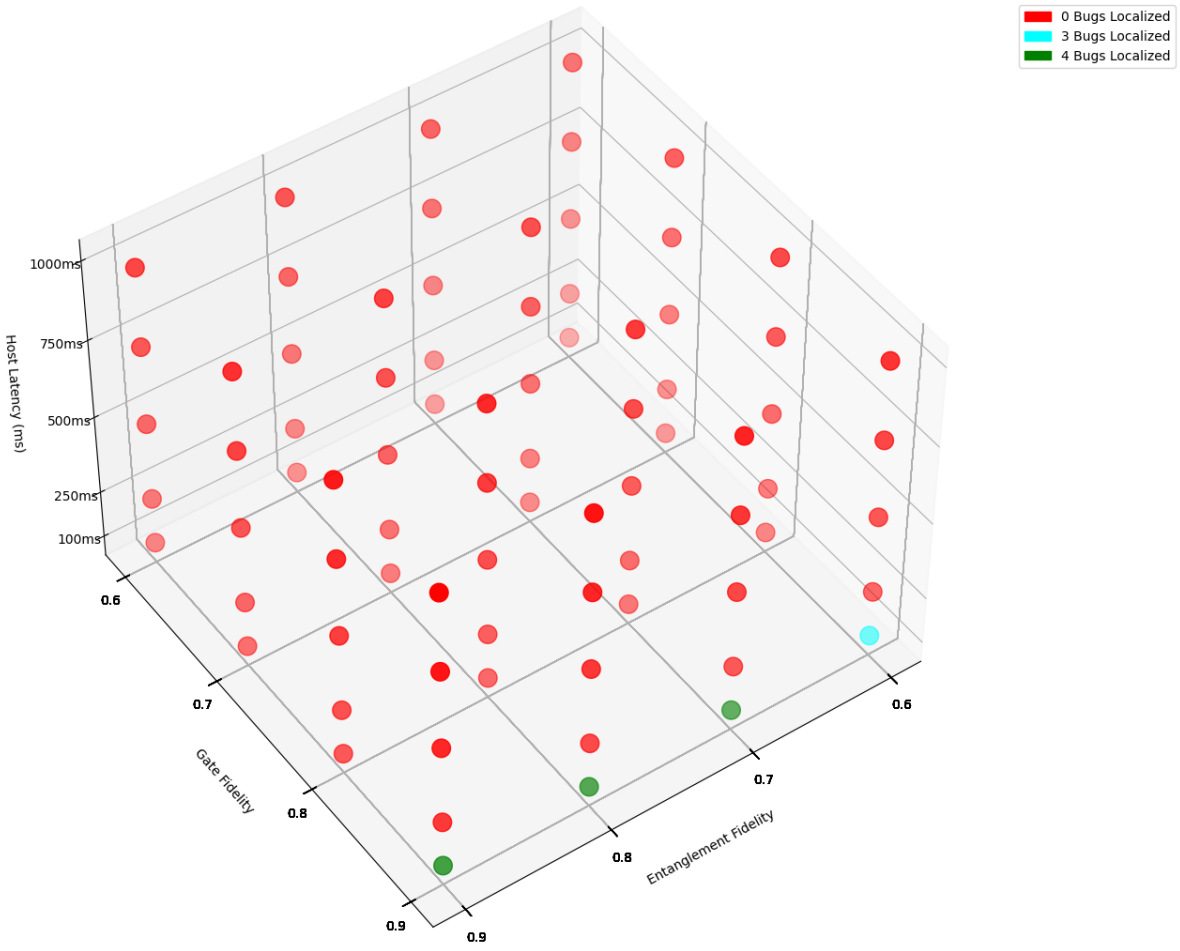
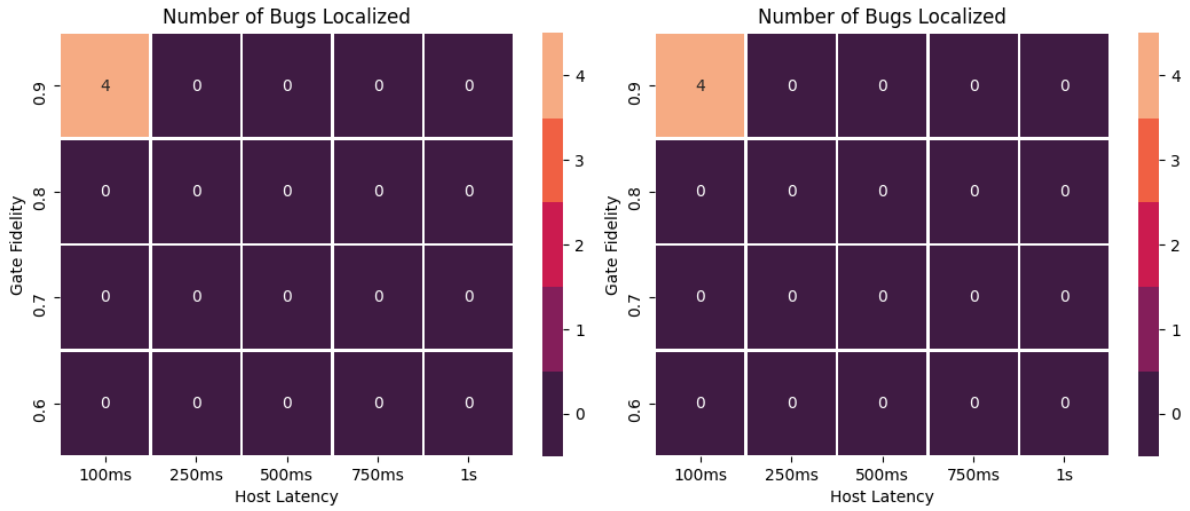


Figure 7.15: Number of Bugs Localized in Mutant 6 for BQC Application

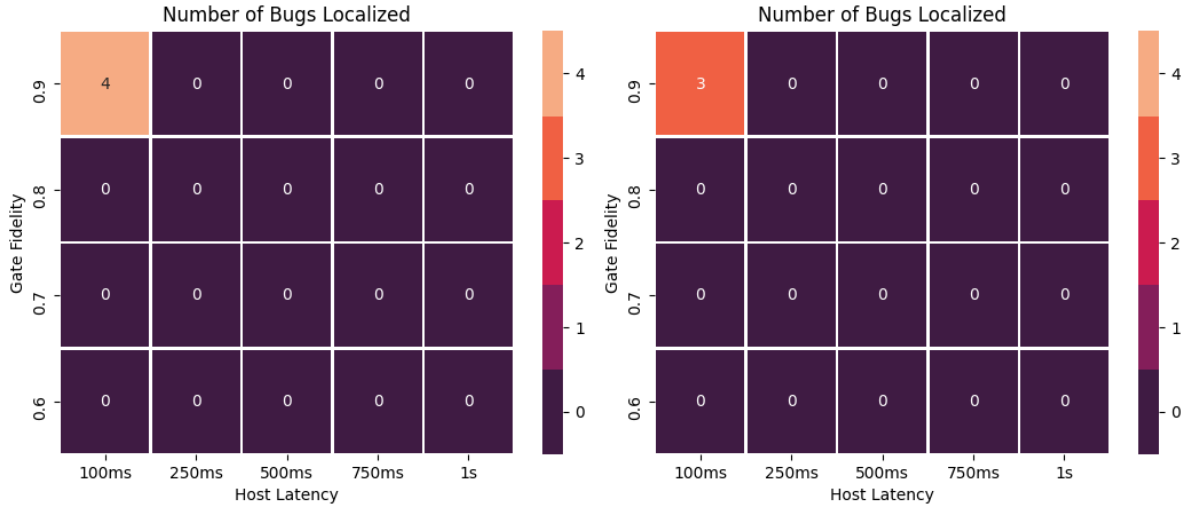
We find that we are able to localize bugs to all four components when the host latency is 100ms and $e_i > 0.7$. This is because for this configuration, transaction T4 fails. Since the candidate generated is (c1,c2,c4,c5), we are able to localize each bug. But when $e_i = 0.6$, only transactions T1 and T2 fail, thereby localizing the bugs only in components C2,C4 and C5. Thus a fault in component C1 (wrong EPR pair generation) is not detected, or localized.

One particular side note from this exercise is that transaction T4 failed for this particular mutant (when $e_i \geq 0.7$), but it did not for any of the 5 mutants discussed before. A possible reason for this is that for the given input values, the end result was distinguishable only when all four bugs were presented. If only one or two faults were present, T4 produce result that are indistinguishable from correct code, as seen in previous mutants. Moreover, noise also plays a role - given that transactions T4 and T5 have many noisy devices, it becomes harder to distinguish between faulty and cor-



(a) Entanglement Fidelity = 0.9

(b) Entanglement Fidelity = 0.8



(c) Entanglement Fidelity = 0.7

(d) Entanglement Fidelity = 0.6

Figure 7.16: Cross Sectional View of Mutant 6, BQC Application

rect results for the given set of input values. Other transactions do not have to deal with this many number of noisy devices and hence are able to distinguish between the two results.

7.3. Comparing SFL Testing Framework with Standard Testing Technique

In this section we compare the performance of the Q-SFL testing framework with that of the ad-hoc standard testing (SLT) Framework described in Chapter 4 (Algorithm 2). In the SLT technique, we run the whole application and compare its output with the correct output.

Thus, we will only be able to detect whether or not there is a bug which is shown by a dissimilarity between the correct output and ideal output of each test case.

7.3.1. State Teleportation

For ad-hoc testing state teleportation, we test it with 6 different input cases that corresponds to the state of the qubit that should be teleported. Then, we run the mutants and see whether or not the receiver gets the same state that the sender had. If they don't match, we mark the run as an error and say that we have detected discrepancies, and that bugs exist. Each experiment is run 1000 times.

First, we run the test with no noise - everything in hardware, qubit and entanglement is perfect. When we run it, we find the following test result in Table 7.4. A "FAIL" implies that the the receiver got a different state that the sender intended to send and hence we have detected a bug - which is good for us testers.

	M1	M2	M3	M4
$\|0\rangle$	PASS	FAIL	FAIL	PASS
$\|1\rangle$	FAIL	FAIL	FAIL	PASS
$\|+\rangle$	FAIL	FAIL	PASS	PASS
$\|-\rangle$	FAIL	FAIL	PASS	PASS
$\|i\rangle$	FAIL	PASS	FAIL	PASS
$\|-i\rangle$	FAIL	PASS	FAIL	PASS

Table 7.4: Ad-Hoc State Teleportation No Noise

Ideally, everything in this table should denote FAIL. However, from the table we find that the System Level Testing (SLT) testing doesn't even detect the presence of bugs for some of the test cases given. This already tells us that SFL test framework is better than ad-hoc for the given test case for it not only detects all these bugs, it also localizes them.

It is also interesting to check the SLT's already limited power when we have noise in our hardware. For the case when we have entanglement fidelity as 0.9 and if we were to change the gate fidelity (g_i) values from 0.99 to 0.90, we find that SLT framework's power to even detect bug falls of rapidly for the given input values 7.5, 7.6, 7.7.

The similar table for Mutant 4 is excluded because it already does not detect in the

	Mutant 1									
Input \ g_i	0.90	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99
$\|0\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
$\|1\rangle$	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
$\|+\rangle$	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
$\|-\rangle$	PASS	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL	FAIL
$\|i\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL
$\|-i\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL

Table 7.5: Ad-Hoc State Teleportation With Noise Mutant 1 for varying Gate Fidelity

	Mutant 2									
Input \ g_i	0.90	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99
$\|0\rangle$	PASS	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL	FAIL
$\|1\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL
$\|+\rangle$	PASS	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL	FAIL
$\|-\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL
$\|i\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
$\|-i\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS

Table 7.6: Ad-Hoc State Teleportation With Noise Mutant 2 for varying Gate Fidelity

	Mutant 3									
Input \ g_i	0.90	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99
$\|0\rangle$	PASS	PASS	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
$\|1\rangle$	PASS	PASS	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
$\|+\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
$\|-\rangle$	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
$\|i\rangle$	PASS	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL	FAIL
$\|-i\rangle$	PASS	PASS	PASS	PASS	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL

Table 7.7: Ad-Hoc State Teleportation With Noise Mutant 3 for varying Gate Fidelity

case when there are no noise and so we know for sure it won't detect even in case of noise. An interesting fact is that even when the noise in gate fidelity is 0.9, the testing framework is not able to detect whether or not there is a bug. Thus SFL is very versatile in that aspect.

7.3.2. Blind Quantum Computing Application

For system level testing BQC application, we test it with just two different input cases. Then, we run the mutants and see whether or not the end result is the same state that we want. If they don't match, we mark the run as an error and say that we have detected discrepancies, and that bugs exist.

It is interesting to see if SLT detects error for the two test cases we have. We have the following table 7.8 as the result, for noiseless case.

	M1	M2	M3	M4	M5	M6
$(\pi/2, \pi/2)$	FAIL	PASS	PASS	FAIL	FAIL	FAIL
$(\pi/2, -\pi/2)$	FAIL	PASS	PASS	FAIL	FAIL	FAIL

Table 7.8: Ad-Hoc BQC Application With No Noise

For the case when we have entanglement fidelity as 0.9 and if we were to change the gate fidelity values from 0.99 to 0.95, we find that the SLT framework's power to even detect bug falls of rapidly for the given input values.

	Input = $(\pi/2, \pi/2)$				
	0.95	0.96	0.97	0.98	0.99
Mutant 1	FAIL	FAIL	FAIL	FAIL	FAIL
Mutant 2	PASS	PASS	PASS	PASS	PASS
Mutant 3	PASS	PASS	PASS	PASS	PASS
Mutant 4	PASS	PASS	PASS	FAIL	FAIL
Mutant 5	PASS	PASS	PASS	PASS	FAIL
Mutant 6	PASS	PASS	FAIL	FAIL	FAIL

Table 7.9: Ad-Hoc BQC Application With Noise Input 1 for varying Gate Fidelity

Now we test with second test of input and gather the following result:

	Input = $(\pi/2, -\pi/2)$				
	0.95	0.96	0.97	0.98	0.99
Mutant 1	FAIL	FAIL	FAIL	FAIL	FAIL
Mutant 2	PASS	PASS	PASS	PASS	PASS
Mutant 3	PASS	PASS	PASS	PASS	PASS
Mutant 4	PASS	PASS	PASS	FAIL	FAIL
Mutant 5	PASS	PASS	PASS	PASS	FAIL
Mutant 6	PASS	PASS	FAIL	FAIL	FAIL

Table 7.10: Ad-Hoc BQC Application With Noise Input 2 for varying Gate Fidelity

Which has the similar result as that of the previous input value.

Hence, it is of prime importance to divide the application into very small components and then test them accordingly in order to find bugs. Bigger the application, larger the factor the noise plays and tougher it gets to detect discrepancies in results.

8

Conclusion and Outlook

In this chapter we summarize the key learnings gathered from the results presented in previous chapters. This is used to answer the research questions put forth in Chapter 1.

8.1. Reflection

The main purpose of this thesis was to introduce a method to test application layer network software and study its practicality in the current Noisy Intermediate Scale Quantum (NISQ [40]) Hardware setting. This thesis presented the first attempt to develop a testing framework (Chapter 5) whose underlying theory is application-independent, works irrespective of input setting, and can detect and localize bugs even in noisy settings (Chapter 7). We were able to come up with the Testing Framework after critically studying the shortcomings of current quantum testing methodologies (like Assertion Based Testing and Standard Testing Scheme as discussed in Chapter 4). The idea of Spectrum Based Fault Localization (SFL) was introduced into the framework because unlike classical software, we do not always have the luxury of breakpoints to test the value of qubits midway through the execution and try to find out where the bugs are located in our code. The Q-SFL technique, which uses component division and subset execution was able to detect and localize bugs correctly to the exact part of the code (Chapter 6). Evaluating the Q-SFL framework on buggy mutant state teleportation protocol and BQC Application on top of noisy hardware (Chapter 7) gave us an idea of how bad our hardware can get after which we cannot localize the bug for those applications. To complete the study, we also compared the performance of Q-SFL to that of the system level testing method. The result presented in Chapter 7 gives us a better picture of the power of our newly devised testing framework.

8.2. Answering Research Questions

In this section, we complete the answers of research questions posted in Chapter 1.

RQ1 : *To what extent are classical software testing techniques transferable to quantum programs, and how effectively can they be used?*

We have seen that breakpoints and qubit examination is neither direct nor input independent. Moreover, it is tough to use the assertions techniques in noisy environment (Chapter 4). Adopting a system level standard testing method that runs the whole application and checks the end result does not localize the bug - it merely detects them. However, its performance to detect bugs quickly falls off by increasing the noise in the hardware. SFL technique addresses these concerns and it is chosen as a candidate that can be transferable to quantum programs.

RQ2 : *What would an application independent quantum testing framework look like? What components does it have?*

The architecture of our testing framework was devised in Chapter 5. Currently, the onus is on the tester to divide the application into components and devise the trasaction set. The initial configuration (for example, qubit state initialization) and input values can be set in the framework itself. It is then tested transaction by transaction, and a test pass/fail verdict is carried out. Once every transaction is completed, the error vector is generated and using Bayesian reasoning, the probability associated for each component having a bug is calculated. This helps in correctly localizing the bug.

RQ3 : *How powerful is our testing frameworks? How many bugs can our testing framework detect, what kind of bugs, and how bad can our hardware get before we cannot properly detect and localize bugs?*

We have seen that for the state teleportation protocol and BQC application, we were able to correctly detect and localize all the bugs under noiseless condition. This gave us confidence that the theory behind SFL is strong and can be applied on quantum programs too. Once we set our base, we performed experiments on Q-SFL technique by varying the noise parameters in our hardware. This gave us an idea on how bad our hardware can get after which it becomes hard to properly detect and localize bugs. This result varies according to the application we are testing and we find that fewer the number of noisy hardware gates, the better we can detect and localize bugs.

RQ4 : *Given a buggy application, can our testing framework detect and localize bugs better than the standard testing technique?*

Finally, Q-SFL framework was put in to test by comparing its performance with the standard system level testing technique. As expected, the performance of system level testing technique was highly input dependent and does not do well under noisy conditions. It does not have the ability to localize the bug either. This highlights the need to properly divide the application and run it as a subset - since quantum programs do not have the luxury to stop the execution midway and examine the state of the qubits, it is important that we work around the breakpoint problem by dividing the components ourselves and test them accordingly.

A

State Teleportation Code

A.1. Full Library

The state teleportation library is shown in Listing A.1.

```
import math

def bellMeasurement(qubit, epr):
    qubit.cnot(epr)
    qubit.H()
    m1 = qubit.measure()
    m2 = epr.measure()
    return m1, m2

def correction(qubit, m1, m2):
    if m2 == 1:
        qubit.X()
    if m1 == 1:
        qubit.Z()

def createEPR(epr_socket):
    return epr_socket.create()[0]

def recvEPR(epr_socket):
    return epr_socket.recv()[0]

def initializeQubit(qubit, choice): #used only for testing purposes
    if choice == 2: #Prepare qubit in |1>
        q.X()
    elif choice == 3: #Prepare qubit in |+>
        q.H()
    elif choice == 4: #Prepare qubit in |->
        q.X()
        q.H()
    elif choice == 5: #Prepare qubit in |+i>
        q.rot_X(angle=-math.pi/2)
    elif choice == 6: #Prepare qubit in |-i>
        q.rot_X(angle=math.pi/2)
```

Listing A.1: Teleportation Protocol

A.2. Sender and Receiver Implementation

Using the library listed in A.1, the sender and receiver performs state teleportation by running the code listed in A.2 and A.3 respectively.

```
from teleportation import bellMeasurement,
                  createEPR
from netqasm.sdk import Qubit,EPRSocket
from netqasm.sdk.external import Socket,
         NetQASMConnection

socket = Socket("sender", "receiver")

epr_socket = EPRSocket("receiver")

kwargs = {
   "app_name": "sender",
   "epr_sockets": [epr_socket]
}

sender = NetQASMConnection(**kwargs)

with sender:
   epr = createEPR(epr_socket)
   q = Qubit(sender)
   initializeQubit(q, 1)

   m1, m2 = bellMeasurement(q, epr)
   sender.flush()
   socket.send(str(m1)+","+str(m2))
```

Listing A.2: Sender's Code

```
from teleportation import correction,
                  recvEPR
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import Socket,
            NetQASMConnection

socket = Socket("receiver", "sender")

epr_socket = EPRSocket("sender")

kwargs = {
   "app_name": "receiver",
   "epr_sockets": [epr_socket]
}

receiver = NetQASMConnection(**kwargs)

with receiver:
   epr = recvEPR(epr_socket)
   receiver.flush()
   incoming = socket.recv()
   m1 = int(incoming[0])
   m2 = int(incoming[2])
   correction(epr, m1, m2)

   m = epr.measure()
   receiver.flush()
   m = int(m)

   return {"Qubit" : m}
```

Listing A.3: Receiver's Code

A.3. Code for each transaction

Each of the following four transactions utilizes the main teleportation library listed in Appendix A.1. It mocks the execution of each spectra by importing only the components that are activated for the given transaction, and leaving out the rest. Thus in the SFL setting, we try to localize defects present in the teleportation program (Appendix A.1), with each transaction mocking the spectra defined by the activity matrix in Table 6.1.

A.3.1. Transaction T1

Transaction that uses components C1 (EPR generation) and C3 (Pauli correction).

```
from teleportation import createEPR
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import
         NetQASMConnection

epr_socket = EPRSocket("receiver")

kwargs = {
   "app_name": "sender",
   "epr_sockets": [epr_socket]
}

sender = NetQASMConnection(**kwargs)

with sender:
   # ACTIVATING COMPONENT C1
   epr = createEPR(epr_socket)
   m = epr.measure()
   return {"q1" : m}
```

Listing A.4: Code for qubit $|q_1\rangle$

```
from teleportation import correction,
                  recvEPR
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import
            NetQASMConnection

epr_socket = EPRSocket("sender")
kwargs = {
   "app_name": "receiver",
   "epr_sockets": [epr_socket]
}
receiver = NetQASMConnection(**kwargs)

with receiver:
   # ACTIVATING COMPONENT C1
   epr = recvEPR(epr_socket)

   # INITIALIZATION
   G = ['I', 'X', 'H', 'Rx']
   gate = 2
   epr.applyGate(G[gate])
   m1 = 1
   m2 = 0

   # ACTIVATING COMPONENT C3
   correction(epr, m1, m2)
   m = epr.measure()
   return {"q2" : m}
```

Listing A.5: Code for qubit $|q_2\rangle$

The above code is run for all 4 possible configuration of gate $\in \{0, 1, 2, 3\}$.

A.3.2. Transaction T2

Transaction that only uses component C2 (Bell state measurement).

```
from teleportation import bellMeasurement
from netqasm.sdk import Qubit
from netqasm.sdk.external import NetQASMConnection

kwargs = {"app_name": "tester"}
tester = NetQASMConnection(**kwargs)

with tester:
   # INITIALIZATION
   q1 = Qubit(tester)
   q2 = Qubit(tester)
   choice1 = 2
   choice2 = 3
   initializeQubit(q1, choice1)
   initializeQubit(q2, choice2)

   # ACTIVATING COMPONENT C2
   m1, m2 = bellMeasurement(q1, q2)
   return {"m1" : m1, "m2" : m2}
```

Listing A.6: Transaction T2, Qubits $|\psi\rangle, |q_1\rangle$

The above code is run for all 36 possible configuration of choice1, choice2 $\in \{1, 2, 3, 4, 5, 6\}$.

A.3.3. Transaction T3

Transaction that only uses component C3 (Pauli correction).

```
from teleportation import correction
from netqasm.sdk import Qubit
from netqasm.sdk.external import NetQASMConnection

kwargs = {"app_name": "tester"
tester = NetQASMConnection(**kwargs)
with tester:
    # INITIALIZATION
    q1 = Qubit(tester)
    choice = 4
    m1 = 0
    m2 = 1
    initializeQubit(q1, choice)

    # ACTIVATING COMPONENT C3
    correction(q1, m1, m2)
    m = q1.measure()
    return {"m" : m}
```

Listing A.7: Transaction T3, Qubits $|q_2\rangle$

The above code is run for all 6 values of choice $\in \{1, 2, 3, 4, 5, 6\}$.

A.3.4. Transaction T4

Transaction that uses component C2 (BSM) and C3 (Pauli correction).

```
from teleportation import
            bellStateMeasurement
from netqasm.sdk.external import Socket,
         NetQASMConnection

epr_socket = Socket("sender", "receiver")

kwargs = {"app_name": "sender"}

sender = NetQASMConnection(**kwargs)

with sender:
    # INITIALIZATION
    q1 = Qubit(sender)
    q2 = Qubit(sender)
    G = ['I', 'X', 'H', 'Rx']
    gate1 = 2
    gate2 = 4
    q1.applyGate(G[gate1])
    q2.applyGate(G[gate2])

    # ACTIVATING COMPONENT C2
    m1, m2 = bellStateMeasurement(q1, q2)
    socket.send(str(m1)+","+str(m2))
    return {"m1" : m1, "m2" : m2}
```

Listing A.8: Code for qubit $|\psi\rangle, |q_1\rangle$

```
from teleportation import correction
from netqasm.sdk.external import Socket,
            NetQASMConnection

socket = EPRSocket("receiver", "sender")

kwargs = {
    "app_name": "receiver",
}

receiver = NetQASMConnection(**kwargs)

with receiver:
    # INITIALIZATION
    q = Qubit(receiver)
    G = ['I', 'X', 'H', 'Rx']
    gate3 = 2
    q.applyGate(G[gate3])
    incoming = socket.recv()
    m1 = int(incoming[0])
    m2 = int(incoming[2])

    # ACTIVATING COMPONENT C3
    correction(q, m1, m2)
    m = q.measure()
    return {"m" : m}
```

Listing A.9: Code for qubit $|q_2\rangle$

The above code is run for all 64 possible configuration of gate1, gate2, gate3 $\in \{0, 1, 2, 3\}$.

B

BQC Code

B.1. Full Library

The BQC library is shown in Listing B.1.

```
import math

def createEPR(epr_socket):
    return epr_socket.create()[0]

def recvEPR(epr_socket):
    return epr_socket.recv()[0]

def rsp(qubit, angle):
    qubit.rot_Z(angle=angle)
    qubit.H()
    m = qubit.measure(store_array=False)
    return m

def cphaseGate(qubit1, qubit2):
    qubit2.cphase(qubit1)

def calcDelta1(alpha, theta1, p1, r1):
    return alpha - theta1 + (p1 + r1) * math.pi

def calcDelta2(beta, theta2, p2, r2, m1, r1):
    return math.pow(-1, (m1 + r1)) * beta - theta2 + (p2 + r2) * math.pi

def initializeQubit(qubit, choice): #used only for testing purposes
    if choice == 2: #Prepare qubit in |1>
        q.X()
    elif choice == 3: #Prepare qubit in |+>
        q.H()
    elif choice == 4: #Prepare qubit in |->
        q.X()
        q.H()
    elif choice == 5: #Prepare qubit in |+i>
        q.rot_X(angle=-math.pi/2)
    elif choice == 6: #Prepare qubit in |-i>
        q.rot_X(angle=math.pi/2)
```

Listing B.1: BQC Protocol

B.2. Client and Server Implementation

Using the library listed in B.1, the client and server performs BQC by running the code listed in B.2 and B.3 respectively.

```python
import math
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import Socket,
                  NetQASMConnection
from bqc import createEPR, rsp,
            calcDelta1, calcDelta2

alpha = math.pi/2
beta = math.pi/2
theta1 = math.pi/2
theta2 = math.pi/2
r1 = 1
r2 = 0

socket = Socket("client", "server")
epr_socket = EPRSocket("server")

kwargs = {
    "app_name": "client",
    "epr_sockets": [epr_socket]
}

client = NetQASMConnection(**kwargs)

with client:
    epr1 = createEPR(epr_socket)
    p2 = rsp(epr1, theta2)
    epr2 = createEPR(epr_socket)
    p1 = rsp(epr2, theta1)
    delta1 = calcDelta1(alpha, theta1,
                        p1, r1)
    socket.send(str(delta1))

    m1 = int(socket.recv())
    delta2 = calcDelta2(beta, theta2,
                p2, r2, m1, r1)
    socket.send(str(delta2))

    return {"p1": p1, "p2": p2}
```

Listing B.2: Client's Code

```python
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import Socket,
                NetQASMConnection

from bqc import recvEPR, cphaseGate, rsp

socket = Socket("server", "client")
epr_socket = EPRSocket("client")

kwargs = {
    "app_name": "server",
    "epr_sockets": [epr_socket]
}

server = NetQASMConnection(**kwargs)

with server:
    epr1 = recvEPR(epr_socket)
    epr2 = recvEPR(epr_socket)

    cphaseGate(epr1, epr2)
    delta1 = float(socket.recv())
    m1 = rsp(epr2, delta1)
    socket.send(str(m1))
    delta2 = float(socket.recv())
    m2 = rsp(epr1, delta2)
    return {"m1": m1, "m2": m2}
```

Listing B.3: Server's Code

B.3. Code for each transaction

Each of the following six transactions utilizes the main BQC library listed in Appendix B.1. It mocks the execution of each spectra by importing only the components that are activated for the given transaction, and leaving out the rest. Thus in the SFL setting, we try to localize defects present in the BQC program (Appendix B.1), with each transaction mocking the spectra defined by the activity matrix in Table 6.4.

B.3.1. Transaction T1

Transaction that uses components C2 (RSP) and C4 (Delta1 calculation).

```
import math
from netqasm.sdk.external import NetQASMConnection
from bqc import createEPR, rsp, calcDelta1

client = NetQASMConnection({"app_name": "client"})

with client:
    qubit = Qubit(client)

    # INITIALIZATION
    alpha = math.pi/2
    choice = 1
    initializeQubit(qubit, choice)
    theta1 = math.pi/2
    p1 = 1
    r1 = 0

    # ACTIVATING C4
    delta1 = calcDelta1(alpha, theta1, p1, r1)

    # ACTIVATING C2
    m = rsp(qubit, delta1)
    return {"m": m}
```

Listing B.4: Code for running transaction T1

The above code is run for all 12 possible configuration of choice $\in \{1, 2, 3, 4, 5, 6\}$ and alpha $\in \{\pi/2, -\pi/2\}$.

B.3.2. Transaction T2

Transaction that uses components C2 (RSP) and C5 (Delta2 calculation).

```
import math
from netqasm.sdk.external import NetQASMConnection
from bqc import createEPR, rsp, calcDelta2

client = NetQASMConnection({"app_name": "client"})
with client:
    qubit = Qubit(client)

    # INITIALIZATION
    beta = math.pi/2
    choice = 1
    initializeQubit(qubit, choice)
    theta2 = math.pi/2
    p2 = 1
    m1 = 0
    r2 = 1
```

```
    r1 = 0

    # ACTIVATING C5
    delta2 = calcDelta2(beta, theta2, p2, r2, m1, r1)

    # ACTIVATING C2
    m = rsp(qubit, delta2)
    return {"m": m}
```

Listing B.5: Code for running transaction T2

The above code is run for all 12 possible configuration of choice $\in \{1, 2, 3, 4, 5, 6\}$ and beta $\in \{\pi/2, -\pi/2\}$.

B.3.3. Transaction T3

Transaction that uses components C1 (EPR) and C3 (CPHASE).

```
from bqc import createEPR
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import
         NetQASMConnection

epr_socket = EPRSocket("server")

kwargs = {
    "app_name": "client",
    "epr_sockets": [epr_socket]
}

client = NetQASMConnection(**kwargs)

with client:
    # ACTIVATING COMPONENT C1
    epr = createEPR(epr_socket)

    # INITIALIZATION
    G = ['I', 'X', 'H', 'Rx']
    gate1 = 2
    epr.applyGate(G[gate1])
    m = epr.measure()
    return {"q1" : m}
```

Listing B.6: Client code for T3

```
from bqc import cphaseGate,recvEPR
from netqasm.sdk import EPRSocket,Qubit
from netqasm.sdk.external import
            NetQASMConnection

epr_socket = EPRSocket("client")
kwargs = {
    "app_name": "server",
    "epr_sockets": [epr_socket]
}
server = NetQASMConnection(**kwargs)

with server:
    # ACTIVATING COMPONENT C1
    epr = recvEPR(epr_socket)

    # INITIALIZATION
    qubit = Qubit(server)
    G = ['I', 'X', 'H', 'Rx']
    gate2 = 2
    epr.applyGate(G[gate2])

    # ACTIVATING COMPONENT C3
    cphaseGate(qubit, epr)

    m1 = qubit.measure()
    m2 = epr.measure()
    return {"m1" : m1, "m2" : m2}
```

Listing B.7: Server code for T3

The above code is run for all 16 possible configuration of gate1, gate2 $\in \{0, 1, 2, 3\}$.

B.3.4. Transaction T4

Transaction that uses components C1 (EPR), C2 (RSP), C4 (Delta1) and C5 (Delta2)

```
from bqc import createEPR, calcDelta1,rsp
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import
         NetQASMConnection

epr_socket = EPRSocket("server")

kwargs = {
   "app_name": "client",
   "epr_sockets": [epr_socket]
}

client = NetQASMConnection(**kwargs)

with client:
   # ACTIVATING COMPONENT C1
   epr = createEPR(epr_socket)

   # INITIALIZATION
   alpha = math.pi/2
   theta1 = math.pi/2
   p1 = 1
   r1 = 0

   # ACTIVATING C4
   delta1 = calcDelta1(alpha, theta1,
                       p1, r1)

   # ACTIVATING C2
   m = rsp(qubit, delta1)
   return {"m": m}
```

Listing B.8: Client code for T4

```
from bqc import calcDelta2,recvEPR,rsp
from netqasm.sdk import EPRSocket,Qubit
from netqasm.sdk.external import
            NetQASMConnection

epr_socket = EPRSocket("client")
kwargs = {
   "app_name": "server",
   "epr_sockets": [epr_socket]
}
server = NetQASMConnection(**kwargs)

with server:
   # ACTIVATING COMPONENT C1
   epr = recvEPR(epr_socket)

   # INITIALIZATION
   G = ['I', 'X', 'H', 'Rx']
   gate = 2
   epr.applyGate(G[gate])
   beta = math.pi/2
   theta2 = math.pi/2
   p2 = 1
   m1 = 0
   r2 = 1
   r1 = 0

   # ACTIVATING C5
   delta2 = calcDelta2(beta, theta2,
                  p2, r2, m1, r1)

   # ACTIVATING C2
   m = rsp(epr, delta2)
   return {"m" : m}
```

Listing B.9: Server code for T4

The above code is run for all 16 possible configuration of gate $\in \{0, 1, 2, 3\}$ and alpha, beta $\in \{\pi/2, -\pi/2\}$

B.3.5. Transaction T5

Transaction that uses components C2 (RSP), C3 (CPHASE), C4 (Delta1) and C5 (Delta2)

```
import math
from netqasm.sdk.external import NetQASMConnection
from bqc import rsp, calcDelta1, calcDelta2, cphaseGate

client = NetQASMConnection({"app_name": "client"})
with client:
    qubit1 = Qubit(client)
    qubit2 = Qubit(client)

    # INITIALIZATION
    alpha = math.pi/2
    beta = math.pi/2
    choice = 1
    initializeQubit(qubit1, choice)
    theta1 = math.pi/2
    theta2 = math.pi/2
    p1 = 1
    p2 = 1
    m1 = 0
    r2 = 1
    r1 = 0

    # ACTIVATING C3
    cphaseGate(qubit1, qubit2)

    # ACTIVATING C4
    delta1 = calcDelta1(alpha, theta1, p1, r1)

    # ACTIVATING C5
    delta2 = calcDelta2(beta, theta2, p2, r2, m1, r1)

    # ACTIVATING C2
    m1 = rsp(qubit1, delta1)
    m2 = rsp(qubit2, delta2)

    return {"m1": m1, "m2":m2}
```

Listing B.10: Code for running transaction T5

The above code is run for all 24 possible configuration of choice $\in \{1, 2, 3, 4, 5, 6\}$ and alpha, beta $\in \{\pi/2, -\pi/2\}$

B.3.6. Transaction T6

Transaction that uses components C1 (EPR) and C2 (RSP).

```
from bqc import createEPR,rsp
from netqasm.sdk import EPRSocket
from netqasm.sdk.external import
        NetQASMConnection

epr_socket = EPRSocket("server")

kwargs = {
    "app_name": "client",
    "epr_sockets": [epr_socket]
}

client = NetQASMConnection(**kwargs)

with client:
    # ACTIVATING COMPONENT C1
    epr = createEPR(epr_socket)

    # INITIALIZATION
    alpha = math.pi/2

    # ACTIVATING C2
    m = rsp(qubit, alpha)
    return {"m": m}
```

Listing B.11: Client code for T6

```
from bqc import recvEPR,rsp
from netqasm.sdk import EPRSocket,Qubit
from netqasm.sdk.external import
            NetQASMConnection

epr_socket = EPRSocket("client")
kwargs = {
    "app_name": "server",
    "epr_sockets": [epr_socket]
}
server = NetQASMConnection(**kwargs)

with server:
    # ACTIVATING COMPONENT C1
    epr = recvEPR(epr_socket)

    # INITIALIZATION
    G = ['I', 'X', 'H', 'Rx']
    gate = 2
    epr.applyGate(G[gate])
    beta = math.pi/2

    # ACTIVATING C2
    m = rsp(epr, beta)
    return {"m" : m}
```

Listing B.12: Server code for T6

The above code is run for all 16 possible configuration of gate $\in \{0, 1, 2, 3\}$ and alpha, beta $\in \{\pi/2, -\pi/2\}$

Bibliography

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. doi: 10.1109/IEEESTD.1990.101064.

[2] Cirq, a python framework for creating, editing, and invoking noisy intermediate scale quantum (nisq) circuits, 2019.

[3] Scott Aaronson and Alex Arkhipov. The computational complexity of linear optics. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 333–342, 2011.

[4] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[5] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.

[6] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[7] Stefanie Barz, Elham Kashefi, Anne Broadbent, Joseph F Fitzsimons, Anton Zeilinger, and Philip Walther. Demonstration of blind quantum computing. *science*, 335(6066):303–308, 2012.

[8] James O Berger. *Statistical decision theory and Bayesian analysis*. Springer Science & Business Media, 2013.

[9] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[10] Pierre Bourque, Robert Dupuis, Alain Abran, James W Moore, and Leonard Tripp. The guide to the software engineering body of knowledge. *IEEE software*, 16(6): 35–44, 1999.

[11] Anne Broadbent, Joseph Fitzsimons, and Elham Kashefi. Universal blind quantum computation. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 517–526. IEEE, 2009.

[12] Yu-Ao Chen, Qiang Zhang, Teng-Yun Chen, Wen-Qi Cai, Sheng-Kai Liao, Jun Zhang, Kai Chen, Juan Yin, Ji-Gang Ren, Zhu Chen, et al. An integrated space-to-ground quantum communication network over 4,600 kilometres. *Nature*, 589 (7841):214–219, 2021.

[13] Mike Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

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

[15] Axel Dahlberg, Matthew Skrzypczyk, Tim Coopmans, Leon Wubben, Filip Rozpędek, Matteo Pompili, Arian Stolk, Przemysław Pawełczak, Robert Knegjens, Julio de Oliveira Filho, et al. A link layer protocol for quantum networks. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 159–173. 2019.

[16] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[17] Christopher Ferrie. Self-guided quantum tomography. *Physical review letters*, 113(19):190404, 2014.

[18] Joseph F Fitzsimons. Private quantum computation: an introduction to blind quantum computing and related protocols. *npj Quantum Information*, 3(1):1–11, 2017.

[19] J Gambetta et al. Qiskit: An open-source framework for quantum computing, 2019.

[20] Aram W Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203–209, 2017.

[21] Bas Hensen, Hannes Bernien, Anaïs E Dréau, Andreas Reiserer, Norbert Kalb, Machiel S Blok, Just Ruitenberg, Raymond FL Vermeulen, Raymond N Schouten, Carlos Abellán, et al. Loophole-free bell inequality violation using electron spins separated by 1.3 kilometres. *Nature*, 526(7575):682–686, 2015.

[22] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426. Springer, 1994.

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

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

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

[26] Petar Jurcevic, Ali Javadi-Abhari, Lev S Bishop, Isaac Lauer, Daniela F Bogorin, Markus Brink, Lauren Capelluto, Oktay Günlük, Toshinari Itoko, Naoki Kanazawa, et al. Demonstration of quantum volume 64 on a superconducting quantum computing system. *Quantum Science and Technology*, 6(2):025020, 2021.

[27] Elham Kashefi, Dominik Leichtle, Luka Music, and Harold Ollivier. Securing quantum computations in the nisq era. *arXiv preprint arXiv:2011.10005*, 2020.

[28] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry fields: A software platform for photonic quantum computing. *Quantum*, 3:129, 2019.

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

[30] 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*, pages 1017–1030, 2020.

[31] Hoi-Kwong Lo and Hoi Fung Chau. Unconditional security of quantum key distribution over arbitrarily long distances. *science*, 283(5410):2050–2056, 1999.

[32] Enrique Moguel, Javier Berrocal, Jose García-Alonso, and Juan Manuel Murillo. A roadmap for quantum software engineering: applying the lessons learned from the classics. 2020.

[33] Leonie Mueck. Quantum software, 2017.

[34] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[35] Ryan O'Donnell and John Wright. Efficient quantum tomography. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 899–912, 2016.

[36] Alexandre Perez, Rui Abreu, and Arie Van Deursen. A theoretical and empirical analysis of program spectra diagnosability. *IEEE Transactions on Software Engineering*, 2019.

[37] Mario Piattini, Guido Peterssen, and Ricardo Pérez-Castillo. Quantum computing: A new software engineering golden age. *ACM SIGSOFT Software Engineering Notes*, 45(3):12–14, 2020.

[38] Mario Piattini, Guido Peterssen, Ricardo Pérez-Castillo, Jose Luis Hevia, Manuel A Serrano, Guillermo Hernández, Ignacio García Rodríguez de Guzmán, Claudio Andrés Paradela, Macario Polo, Ezequiel Murina, et al. The talavera manifesto for quantum software engineering and programming. In *QANSWER*, pages 1–5, 2020.

[39] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[40] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[41] Jaroslav Řeháček, Zdeněk Hradil, E Knill, and AI Lvovsky. Diluted maximum-likelihood algorithm for quantum tomography. *Physical Review A*, 75(4):042108, 2007.

[42] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[43] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[44] Peter W Shor and John Preskill. Simple proof of security of the bb84 quantum key distribution protocol. *Physical review letters*, 85(2):441, 2000.

[45] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[46] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q# enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, 2018.

[47] Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Rieffel, Rupak Biswas, and Salvatore Mandrà. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *npj Quantum Information*, 5(1):1–16, 2019.

[48] S Wehner et al. Qnodeos. unpublished, 2021.

[49] S Wehner et al. Squidasm. unpublished, 2021.

[50] K Wright, KM Beck, S Debnath, JM Amini, Y Nam, N Grzesiak, J-S Chen, NC Pisenti, M Chmielewski, C Collins, et al. Benchmarking an 11-qubit quantum computer. *Nature communications*, 10(1):1–6, 2019.

[51] Will Zeng, Blake Johnson, Robert Smith, Nick Rubin, Matt Reagor, Colm Ryan, and Chad Rigetti. First quantum computers need smart software. *Nature News*, 549(7671):149, 2017.

[52] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, et al. Quantum computational advantage using photons. *Science*, 370(6523):1460–1463, 2020.