Improving the Formal Verification of Quantum Stabilizer Codes in COQ-QECC

by

Tuan Khanh Bui

Student Number: 1580686

Supervised by Dr. Christine Rizkallah.

A research thesis submitted in fulfilment for the degree of

Bachelor of Science

in the

Department of Computing and Information Systems

THE UNIVERSITY OF MELBOURNE

2025

Abstract

Quantum computing is prone to noise, which can lead to errors in quantum computations. To address this issue, quantum error-correcting codes (or *quantum codes*) are designed to protect quantum information. However, verifying the detection and correction capabilities of quantum codes is a challenging task due to the exponential computing power required to reason about quantum circuits in Hilbert space. Attempts have been made to use the stabilizer formalism to reason about quantum codes algebraically, and these attempts have been formalized in the COQ-QECC framework using Rocq. However, the formalization has some limitations and redundancies that can be improved to enhance its usability and scalability. In this thesis, we analyze the COQ-QECC framework and propose several improvements to enhance its application and usability. These improvements have been applied to verify the detection capabilities of multiple quantum error-correction codes for double-qubit errors, including Shor's code and Surface code. Compared to the original COQ-QECC framework, our proposed changes have reduced the amount of work required to formalize and verify quantum error-correcting codes, and expanded the framework to verify the detection capabilities of quantum codes for double-qubit errors.

Declaration

I certify that

- this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- this thesis did not require clearance from the University's Ethics Committee.
- the thesis is approximately 11588 words in length (excluding text in images, table, bibliographies and appendices).

Signed:

Date: 01 Jan 1980

Acknowledgement

I would like to thank my supervisor, Dr. Christine Rizkallah, for guiding and supporting me throughout the course of this project. Your support and advice have been invaluable to the successful completion of this project.

I would also like to express my gratitude to the authors and maintainers of the Software Foundations series, which provided me with the necessary knowledge and skills to complete this project.

Finally, I am thankful to my family and friends for their support throughout this journey.

Contents

1 Introduction 5
1.1 Motivation and Research Question 5
1.2 Contributions 6
1.3 Thesis Structure 6

2. Background 7
2.1. Introduction to Quantum Computing 7
2.1.1. Hilbert Space and Quantum States 7
2.1.2. Quantum Measurement 11
2.1.3. Quantum Programs and Quantum Circuits 13
2.2. Quantum Error-Correcting Codes 14
2.2.1. Quantum Errors 14
2.2.2. Example: The Three-Qubit Flip Code 15
2.3. Stabilizer Formalism and Quantum Stabilizer Codes 16
2.3.1. Pauli Groups 16
2.3.2. Pauli Operators and Quantum Stabilizers 17
2.3.3. Stabilizer Code 18
2.4. Formal Methods and Verification 18
2.5. The Rocq Theorem Prover 19
2.5.1. The Gallina Language 19
2.5.2. Rocq Tactics 20
2.6. Dependencies in Rocq 21
2.6.1. Mathematical Components 21
2.6.2. COQ-QECC 22
2.7. Key Insights 25

3. Related Work 27
3.1. Verification of Circuit-based Programs 27
3.1.1. SQIR: Formal Language for Quantum Programs 27
3.1.2. QuantumLib: Mathematical Library for Quantum Computing 27
3.2. Verification of Quantum Error-Correcting Codes 27
3.2.1. COQ-QECC: Formal Verification of Quantum Stabilizer Codes 27
3.2.2. SQIR-QECC: Quantum Error-Correcting Codes in SQIR 28
3.3. Summary 28

4. Contribution to COQ-QECC 29
4.1. Analysis of the Formalization 29
4.2. Changes to the Formalization 33
4.2.1. Unification of Code Space and Stabilizers 33
4.2.2. Condition for Stabilizing Errors 34
4.3. Verification of Two-Qubit Detection Capability 35

5 Discussion and Conclusion 40
5.1 Summary 40
5.2 Limitations and Future Work 40

Bibliography 41

List of Tables

Table 1 Single-Qubit Pauli Matrices and Their Actions on Qubits 10

Table 2 Summary of the verification of different stabilizer codes using COQ-QECC 25

Table 3 COQ-QECC verification of error detection capabilities before and after the improvements 38

List of Figures

Figure 1 Quantum Circuit that constructs the Bell State $|\beta_{11}\rangle$ from the initial state $|00\rangle$ 13

Figure 2 Quantum Circuit that encodes a single-qubit $|\psi\rangle$ into the three-qubit flip code 15

Chapter 1

Introduction

Quantum computing is theoretically capable of solving certain problems that are intractable for classical computers, such as integer factorization and simulating quantum systems [12]. However, quantum computers are also susceptible to noise, which can lead to errors in computations. To address this issue, quantum error-correcting codes have been developed to protect quantum information.

The quantum error-correcting code (or *quantum codes*) is a component that protects quantum information against noise. Similar to classical error-correcting codes, quantum codes add redundant information to the original data, necessarily *encode* the quantum information into *codewords*. After the encoded data undergo the effects of noise, they can be decoded to detect and correct errors [12]. Quantum codes are essential for the development of fault-tolerant quantum computation, making the verification of their correctness a critical task.

1.1 Motivation and Research Question

One way of proving the detection and correction capabilities of quantum codes is to view the codes as linear transformations in Hilbert space. For each type of correctable error, one needs to show that the encoder and decoder circuits can detect and correct the errors. However, due to the nature of quantum computing, the dimension in which one calculates grows exponentially with the number of qubits, making the verification of large-scale quantum codes infeasible [4].

An alternative approach is to use the *stabilizer formalism*, in which certain quantum codes called *stabilizer codes* can be described by a set of operators called *stabilizers*. Physicists often prefer this approach, as high-dimensional matrix multiplications are replaced with group operations. Using specific properties of group theory, one can also avoid enumerating every possible combination of errors, which is a more efficient approach to verifying the detection and correction capabilities of quantum codes.

The stabilizer formalism has been a good pen-and-paper line of reasoning for physicists to prove the detection and correction capabilities of quantum codes. However, it is not a formal proof and not machine-checkable, which means it is prone to human errors. Inspired by the approach, Feng et al. [2025] proposed a formal verification framework called COQ-QECC for quantum codes using stabilizer formalism, which successfully verified the detection and correction capabilities of the 3-qubit bit-flip code and certain important properties of Shor's code.

COQ-QECC has been successfully applied to verify the detection and correction capabilities of multiple quantum codes, including Shor's code. However, the framework has some limitations and redundancies that can be improved to enhance its usability and scalability. In this thesis, we aim to

address these limitations and propose improvements to the COQ-QECC framework, with the goal of expanding its application and improving its reasoning. Specifically, we ask *"How can we improve the COQ-QECC framework to better mirror the stabilizer formalism and enhance its usability for verifying the detection capabilities of quantum stabilizer codes?"*

1.2 Contributions

In this thesis, we will present our analysis of the COQ-QECC framework, identifying its limitations and redundancies. After that, we will propose several improvements to the framework, including changes to the formalization and a new method to automatically verify the detection capabilities of stabilizer codes. We will also implement our proposed changes and demonstrate their effectiveness by applying them to verify the detection capabilities of Shor's code and other stabilizer codes.

1.3 Thesis Structure

We will start by providing the background of quantum computing, quantum error-correcting codes, and the stabilizer formalism in Chapter 2. In this chapter, we will also establish the necessary mathematical foundations and notations that will be used throughout the thesis. This chapter also includes a brief introduction to the COQ proof assistant and the COQ-QECC framework, which will be used for formal verification of quantum codes.

With the necessary background established, we will then review related work in Chapter 3, including existing formal verification frameworks for quantum programs, such as SQIR [8] and COQ-QECC [4]. Then, we will analyze and propose changes to the COQ-QECC framework in Chapter 4, as well as implementing a method to verify the detection capabilities of stabilizer codes for two-qubit errors.

Finally, we will conclude the thesis in Chapter 5, summarizing our contributions and discussing the limitations of our work and potential future directions for research.

Chapter 2.

Background

This chapter provides the necessary background on quantum computing, quantum error-correcting codes and formal verification. It also introduces the current state of COQ-QECC [4] - a formal verification framework for stabilizer codes - that we plan to improve in this thesis.

We first focus on the theory of quantum computing and quantum error-correcting codes. Section 2.1 is a brief introduction to quantum computing with necessary concepts and notations that will be used throughout the thesis. Then, Section 2.2 introduces the basic concepts of quantum error-correcting codes, which is followed by a discussion of the stabilizer formalism and stabilizer codes in Section 2.3.

After that, we will discuss the theory of formal verification and the tools used in this thesis. We will cover the basics of formal methods and formal verification in Section 2.4, followed by a brief introduction to the Rocq Theorem Prover in Section 2.5. Finally, we will discuss the current state of COQ-QECC in Section 2.6, together with its dependent libraries.

We will assume only a general background in linear algebra and computer science, with no prior knowledge of quantum computing or formal verification required. Readers who are already familiar with these topics can skip the corresponding sections and proceed to the summary (Section 2.7).

2.1. Introduction to Quantum Computing

As our primary focus is to improve the formal verification of stabilizer codes, it is important to understand the basics of quantum computing and quantum codes. In this section, we will introduce the fundamental concepts of quantum computing. We will also establish the notations and terminologies that will be used throughout the thesis, most of which are based on the book *Quantum Computation and Quantum Information* by Nielsen and Chuang [12].

2.1.1. Hilbert Space and Quantum States

In the discussion of quantum computing, quantum states are often described as vectors in the Hilbert space. As we only discuss a finite-dimensional complex vector space, the Hilbert space is exactly the inner product space [12].[1] As such, we can define the Hilbert space the same way an inner product space can be defined.

Let the Hilbert space of dimension n be denoted as $\mathcal{H}_n$, and complex numbers be denoted as $\mathbb{C}$. The inner product between two vectors u and v can be denoted as $\langle u, v\rangle$. This is a function that takes two vectors in the Hilbert space and returns a complex number.

[1] A Hilbert space is an inner product space that is complete with respect to the norm defined by the inner product. In the discussion of quantum computation, we only care about finite-dimensional Hilbert spaces, which are always complete.

$$\langle u, v\rangle : \mathcal{H}_n \times \mathcal{H}_n \to \mathbb{C} \tag{1}$$

The inner product can be viewed as matrix multiplication, where the first vector is transposed and conjugated.

$$\langle u, v\rangle = u^\dagger v \tag{2}$$

In this definition, $u^\dagger$ is the conjugate transpose of u. Defining the inner product allows us to define other concepts of linear algebra:

- Orthogonality: Two vectors u and v are orthogonal if and only if $\langle u, v\rangle = 0$.
- Norm: The norm is a function that takes in a vector and returns a non-negative real number:

$$\begin{aligned} \|u\| &: \mathcal{H}_n \to \mathbb{R}^+ \\ \|u\| &= \sqrt{\langle u, u\rangle} \end{aligned} \tag{3}$$

- Unit vector: A vector u is a unit vector if and only if $\|u\| = 1$.

In quantum mechanics, a quantum state is represented as a unit vector in Hilbert space. However, physicists often use a more compact notation called Dirac notation. A basic vector in Dirac notation is written as a ket, denoted as $|\psi\rangle$. From that, we can intuitively define the related notations:

- A bra is the conjugate transpose of a ket, denoted as $\langle\psi|$.
- The inner product between two states $|\psi\rangle$ and $|\phi\rangle$ is denoted as $\langle\psi|\phi\rangle$ (read as "bra-ket").
- The norm of a state $|\psi\rangle$ is denoted as $\||\psi\rangle\| = \sqrt{\langle\psi|\psi\rangle}$.

Another feature of the Hilbert space that is commonly used in quantum computing is the tensor product. The tensor product is a way to combine two Hilbert spaces into a larger Hilbert space. Let V and W be two Hilbert spaces of dimension m and n respectively. The tensor product of V and W (denoted as $V \otimes W$) is a Hilbert space of dimension mn. The tensor product of two vectors $|\psi\rangle$ in V and $|\phi\rangle$ in W is also denoted as $|\psi\rangle \otimes |\phi\rangle$, which is a vector in $V \otimes W$. In terms of matrix representation, the tensor product of two vectors can be represented using the Kronecker product.

Definition 2.1. The Kronecker product of two matrices A and B is a block matrix defined as:

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{pmatrix} \tag{4}$$

For example, the tensor product of two quantum states $|\psi\rangle = \binom{0}{1}$ and $|\phi\rangle = \binom{1}{0}$ is:

$$|\psi\rangle \otimes |\phi\rangle = \binom{0}{1} \otimes \binom{1}{0} = \begin{pmatrix} 0\binom{1}{0} \\ 1\binom{1}{0} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{5}$$

Now that we have defined how to represent quantum states in Hilbert space, we can define the basic unit of quantum information.

Single-Qubit System In classical computing, the basic unit of information is a bit, which can be in either states, 0 or 1. The qubit serves as an analogue to the classical bit, but is not limited to two discrete states. A qubit can be in a linear combination of states, which is called *superposition*:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \text{with } \alpha, \beta \in \mathbb{C} \tag{6}$$

The two states $|0\rangle$ and $|1\rangle$ are called the computational basis states. Upon measurement, the probabilities of observing $|0\rangle$ and $|1\rangle$ are $|\alpha|^2$ and $|\beta|^2$ respectively, which must sum to 1.

Formally, we can define a quantum state as follows:

Definition 2.2. A quantum state $|\psi\rangle$ is a unit vector in Hilbert space such that $\||\psi\rangle\| = 1$.

The requirement that the norm of a quantum state must be 1 is called the *normalization condition*. Because a quantum state is a vector in Hilbert space, it can be represented as a column vector. Using the convention that $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$, an arbitrary quantum state can be written as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\end{pmatrix} \tag{7}$$

Unitary Evolution Similar to how classical bits can be manipulated using logic gates, qubits can be manipulated using quantum gates. The postulate below gives us an idea of how quantum states evolve over time:

Postulate 2.3. The evolution of a closed quantum system is described by a unitary transformation. That is, the state $|\psi\rangle$ of the system at time t_1 is related to the state $|\phi\rangle$ of the system at time t_2 by a unitary operator U:

$$|\phi\rangle = U|\psi\rangle \tag{8}$$

Recall the definition of unitary operators:

Definition 2.4. A unitary operator U of dimension $n \times n$ is a linear operator such that $U^\dagger U = UU^\dagger = I$, where I is the identity operator of dimension $n \times n$.

The reason physicists use unitary operators to describe time evolution is that they preserve the norm of quantum states. This is important because the norm of a quantum state must always be 1, as it represents the total probability of observing all possible states.

Theorem 2.5. (Unitary Norm Preservation) Let $|\psi\rangle$ be a quantum state and U be a unitary operator. Then $\|U|\psi\rangle\| = 1$.

The proof of Theorem 2.5 can be found in Robert's Thesis [15] on page 44.

Pauli Matrices As an example of unitary operators, we introduce the Pauli matrices, which are a set of quantum operators that act on a single-qubit.

Operator	Matrix	Action on Qubit
Identity (I)	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$I\|0\rangle = \|0\rangle$ $I\|1\rangle = \|1\rangle$
Pauli-X (X)	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$X\|0\rangle = \|1\rangle$ $X\|1\rangle = \|0\rangle$
Pauli-Y (Y)	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$Y\|0\rangle = i\|1\rangle$ $Y\|1\rangle = -i\|0\rangle$
Pauli-Z (Z)	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$Z\|0\rangle = \|0\rangle$ $Z\|1\rangle = -\|1\rangle$

Table 1: Single-Qubit Pauli Matrices and Their Actions on Qubits

Pauli matrices are fundamental in quantum computing for several reasons. First, they represent rotation around the axes of the Bloch sphere, which is a geometric representation of a qubit's state. Second, their action on a qubit have clear interpretation. For example, the Pauli-X matrix flips the state of a qubit, while the Pauli-Z matrix flips the phase of a qubit.

Multi-Qubit System As the name suggests, a multi-qubit system is a quantum system that consists of more than one qubit. The combination is done using the tensor product, which allows us to represent the state of a multi-qubit system as a vector in a larger Hilbert space. For example, a two-qubit system with qubits $|\psi\rangle$ and $|\phi\rangle$ can be represented as:

$$|\psi\rangle \otimes |\phi\rangle = |\psi\rangle|\phi\rangle \tag{9}$$

The tensor product symbol can be omitted when there is no ambiguity. If the value of $|\psi\rangle$ is $\alpha|0\rangle + \beta|1\rangle$ and the value of $|\phi\rangle$ is $\gamma|0\rangle + \delta|1\rangle$, then the two-qubit system can be represented as:

$$|\psi\rangle|\phi\rangle = (\alpha|0\rangle + \beta|1\rangle)(\gamma|0\rangle + \delta|1\rangle) = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle \tag{10}$$

Note that we can write $|1\rangle|0\rangle$ as $|10\rangle$, which is a more compact representation. The dimension of a multi-qubit system is 2^n, where n is the number of qubits in the system [5]. This exponential growth in dimension is one of the reasons why quantum computing is so powerful, and why classical computers struggle to simulate quantum systems.

We will look at one of the most important properties of multi-qubit systems, which is entanglement. Entanglement is a phenomenon where the state of one qubit is dependent on the state of another, even when they are effectively separated. Let's consider the Bell state, which is a two-qubit entangled state:

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{11}$$

When we measure the first qubit, we will observe either $|0\rangle$ or $|1\rangle$ with equal probability. However, if we observe $|0\rangle$, then the second qubit will also be in the state $|0\rangle$ and vice versa. This is a unique property of quantum systems that has no classical analogue.

Finally, the general state of an n-qubit system can be written as a linear combination of all possible basis states:

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle \tag{12}$$

Note that the normalization condition from Definition 2.2 still applies, which means that the sum of the squares of the coefficients must be equal to 1:

$$\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1 \tag{13}$$

This is to ensure that the total probability of observing all possible states is 1.

2.1.2. Quantum Measurement

After the discussion about quantum states and unitary evolution, we will now discuss another important aspect of quantum computing, which is quantum measurement. Measurement is when we observe the state of a closed quantum system, which causes the system to be no longer closed.

Projective Measurement For many applications of quantum computation and quantum information, we are mostly interested in projective measurement [12]. A projective measurement is described as an observable:

Definition 2.6. (Quantum Observable) A quantum observable M is described as a square matrix that satisfies

$$M = M^\dagger \tag{14}$$

where $M^\dagger$ is the conjugate transpose of M. Matrices satisfying this property are also called Hermitian matrices.

An important property of Hermitian matrices is that they have real eigenvalues and orthonormal eigenvectors. This results in the following theorem:

Theorem 2.7. (Real Decomposition of Hermitian Matrix): For any Hermitian matrix M, M has a spectral decomposition with real eigenvalues.

$$M = \sum_m m P_m \tag{15}$$

with P_m being the projector onto the eigenspace of M corresponding to the eigenvalue m. P_m can be expressed as:

$$P_m = \sum_i |v_i\rangle\langle v_i| \tag{16}$$

where $|v_i\rangle$ are the eigenvectors corresponding to the eigenvalue m.

Note that $|v_i\rangle\langle v_i|$ is called an outer product, which is a matrix constructed by the multiplication of a column vector and a row vector. This product is often referred to as a projector onto the eigenspace corresponding to m_i. For example, the Pauli-X matrix has two eigenvalues, 1 and -1, with corresponding eigenvectors $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $|-\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$ respectively:

$$X = (1) \cdot |+\rangle\langle+| + (-1) \cdot |-\rangle\langle-| \tag{17}$$

A measurement applies the spectral decomposition of an observable to yield an eigenvalue and collapse the quantum state to the corresponding eigenvector. This is formalised as a postulate of quantum mechanics:

Postulate 2.8. A projective measurement is described by an observable M, which has a spectral decomposition:

$$M = \sum_m m P_m \tag{18}$$

Upon measurement, the probability of observing the eigenvalue m is given by:

$$p(m) = \langle\psi|P_m|\psi\rangle \tag{19}$$

And the state after a measurement that yield m is:

$$\frac{P_m|\psi\rangle}{\sqrt{p(m)}} \tag{20}$$

For example, if we use X (Equation 17) as the observable to measure an arbitrary quantum state $|\psi\rangle$

$$|\psi\rangle = \alpha|+\rangle + \beta|-\rangle \tag{21}$$

The probability of observing the eigenvalue 1 is:

$$\begin{aligned} \langle\psi|+\rangle\langle+|\psi\rangle &= (\alpha^*\langle+| + \beta^*\langle-|)\ |+\rangle\langle+|\ (\alpha|+\rangle + \beta|-\rangle) \\ &= \alpha^*\langle+|+\rangle\ \alpha\langle+|+\rangle \\ &= \alpha^*\alpha = \alpha^2 \end{aligned} \tag{22}$$

And the state after measuring the eigenvalue 1 is:

$$\frac{|+\rangle\langle+|\psi\rangle}{\sqrt{p(1)}} = \frac{|+\rangle\langle+|(\alpha|+\rangle + \beta|-\rangle)}{\alpha} = \frac{\alpha|+\rangle}{\alpha} = |+\rangle \tag{23}$$

For the purpose of this thesis, we are particularly interested in measurements that yield the results m with probability 1 and leave the state unchanged.

Corollary 2.9. (Measurement on an Eigenstate) If a state vector $|\psi\rangle$ is an eigenvector of an observable M corresponding to the eigenvalue m, measuring M on $|\psi\rangle$ will yield the result m with probability 1 and leave the state unchanged.

The proof for Corollary 2.9 can be found in Hall's book *Quantum Theory for Mathematicians* [7] in Section 3.6, where it appears as Proposition 3.11 (Eigenvector).

Limitation of Measurement and Implication Unlike classical computing, measurement in quantum computing destroys the quantum state. This necessarily prevents one from testing and debugging a quantum program, as both processes require the ability to exactly inspect the state of the system. Moreover, measurement does not provide us with all information about the quantum state, which is a limitation that is not present in classical computing. However, this does not mean that we cannot test quantum programs. Currently, there are statistic-based testing methods, where a quantum program is executed multiple times and the results are collected to estimate the probability distribution of the quantum state [1]. However, this is generally more complicated and costly to implement.

2.1.3. Quantum Programs and Quantum Circuits

Quantum programs are a series of quantum gates that manipulate the quantum states. These gates include single-qubit gates such as Pauli or Hadamard, and multi-qubit gates such as CNOT or Toffoli. A quantum program can be represented as a quantum circuit, similar to classical circuits. Figure 1 shows a simple quantum circuit as an example.

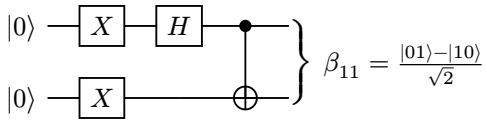


Figure 1: Quantum Circuit that constructs the Bell State $|\beta_{11}\rangle$ from the initial state $|00\rangle$

The horizontal lines represent the qubits, and the gates are applied from left to right. The first two gates are Pauli-X gates (X), which flip the state from $|0\rangle$ to $|1\rangle$. The Hadamard gate (H) is a single-qubit gate that creates an equal superposition of $|0\rangle$ and $|1\rangle$. Specifically, when input is $|1\rangle$, the output will be $|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$. Finally, the Controlled-Not (CNOT) gate $\bullet\!\!-\!\!\oplus$ is a two-qubit gate that flips the state of the target qubit (the second qubit) if the control qubit (the first qubit) is in the state $|1\rangle$.

No-cloning theorem Unlike classical computing, quantum computing does not allow us to perfectly copy an arbitrary quantum state.

Theorem 2.10. (No-cloning) There exists no unitary transformation U such that:

$$U(|\psi\rangle \otimes |0\rangle) = |\psi\rangle \otimes |\psi\rangle \tag{24}$$

The proof for this theorem can be found in Nielsen and Chuang's book *Quantum Computation and Quantum Information* [12] in Section 12.1.1, where it appears in box 12.1 (The no-cloning theorem). The no-cloning theorem has important implications for quantum computing, as it prevents us from using classical error-correcting codes to protect quantum information, which will be discussed in the next section.

2.2. Quantum Error-Correcting Codes

We now move to the main topic of this thesis, which is *quantum error-correcting codes*, or *quantum codes* for short. Note that although we use the term "quantum error-correcting codes" in this thesis, we will not discuss the error-correcting capabilities of quantum codes. Instead, we will focus on the detection capabilities of quantum codes, which are a subset of the error-correcting capabilities.

Similar to their classical analogue, quantum codes add redundant information to the original data to make it resilient against noise. From the article by [17], we can give a definition for quantum codes:

Definition 2.11. A quantum error-correcting code (or quantum code) is a scheme to encode quantum information into a larger Hilbert space, such that errors on the encoded information can be detected and corrected.

This section introduces the basic concepts of quantum error-correction to support the discussion of stabilizer formalism and stabilizer codes. We will first discuss the types of errors that can occur in quantum systems (Section 2.2.1), then introduce the properties of quantum error-correcting codes through a specific example - the three-qubit flip code (Section 2.2.2).

2.2.1. Quantum Errors

Quantum information is sensitive to noise, which can be caused by various factors such as uncontrolled interaction with the environment, or imperfections in the quantum hardware. These noises can cause errors to the quantum states, which are referred to as *quantum errors*.

Quantum errors are often viewed as unitary operators that act on the quantum states. Formally, a quantum error can be defined as a unitary operator E that acts on a quantum state $|\psi\rangle$:

$$|\psi\rangle \xrightarrow{E} E|\psi\rangle \tag{25}$$

As any arbitrary operator can be expressed as a linear combination of Pauli matrices, a quantum code that can correct all Pauli errors can also correct any arbitrary errors [12]. As such, physicists often focus on Pauli errors when designing quantum codes. For example, the effect of an X error on a single-qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is flipping the bit:

$$X|\psi\rangle = X(\alpha|0\rangle + \beta|1\rangle) = \alpha X|0\rangle + \beta X|1\rangle = \alpha|1\rangle + \beta|0\rangle \tag{26}$$

Compared to classical errors, quantum errors are more complicated. One reason is that errors such as Z has the effect of flipping the phase of a qubit, which has no classical analogue. Another reason is that the principle of quantum information is fundamentally different from classical computing due to several factors:

- *No cloning*: A simple error-correcting code in classical computing is the repeating code, where the original bit is repeated multiple times. However, this is not possible in quantum computing due to the no-cloning theorem, which states that there is no unitary operator that can clone an arbitrary quantum state [12].
- *Wave function collapse*: As discussed in Postulate 2.8, measurement in quantum computing collapse the quantum state to an eigenstate of the observable. Therefore, we cannot directly

measure the quantum state to detect errors, as this will destroy the information that we want to protect.

Therefore, quantum codes must be designed to detect and correct errors without violating the no-cloning theorem or destroying the quantum information. We will discuss how this can be implemented in the next section.

2.2.2. Example: The Three-Qubit Flip Code

The three-qubit flip code is a simple quantum code that can detect any single or double bit-flip error. This is similar to the classical three-bit repetition code, where the original bit is repeated three times. This is done by encoding a single-qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ into three qubits as follows:

$$|\psi\rangle_L = \alpha|0\rangle_L + \beta|1\rangle_L = \alpha|000\rangle + \beta|111\rangle \tag{27}$$

Where the subscript L indicates the encoded states, which are referred to as *logical states*. The set of all possible encoded states is called the *code space*, which is a subspace of the Hilbert space of the three qubits. This gives us a definition for codes:

Definition 2.12. A code space $\mathcal{C} \in \mathcal{H}$ of a quantum code C is a subspace of the Hilbert space that consists of all possible encoded states of C.

We also define some terminology that will be used in the rest of the thesis:

- A *codeword* is a quantum state in the code space. This is more general than some definitions that define codewords as the basis that span the code space (e.g. $|000\rangle$ and $|111\rangle$ in the three-qubit flip code).
- An *error state* is a quantum state that is not in the code space, which is caused by an error acting on a codeword.
- An *error space* is a subspace of the Hilbert space that consists of all possible error states.

One way to encode an arbitrary qubit into the three-qubit flip code is to use the quantum circuit below:

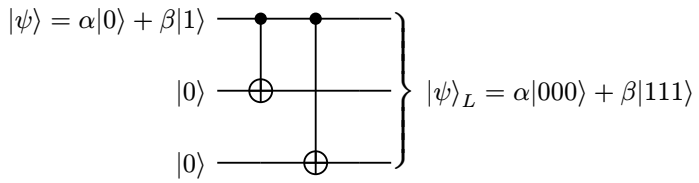


Figure 2: Quantum Circuit that encodes a single-qubit $|\psi\rangle$ into the three-qubit flip code

We can start considering the error detection capabilities of the three-qubit flip code by looking at the effect of a single bit-flip error on the encoded state. Suppose a bit-flip error X occurs on the first qubit:

$$|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle \overset{X_1}{\mapsto} \alpha|100\rangle + \beta|011\rangle \tag{28}$$

This evolution is labelled as X_1 to indicate that the error is equivalent to applying the Pauli-X operator on the first qubit. Specifically, X_1 is defined as the tensor product of the Pauli-X operator and the identity operator on the other two qubits:

$$X_1 = X \otimes I \otimes I \tag{29}$$

In general, we can define X_i as the operator that applies the single-qubit X gate on the i-th qubit and the identity operator on the other qubits. The same can be done for the other Pauli operators Y and Z, which we denote as Y_i and Z_i respectively.

The error can then be detected by measurement. As discussed, measurement should be done without destroying the quantum information. We give a definition for a type of measurement that can be used to detect errors without destroying the quantum information:

Definition 2.13. A measurement represented by an observable M is a *syndrome measurement* of a code $\mathcal{C}$ if measuring M preserves the state of any codeword in $\mathcal{C}$, i.e, $\forall|\psi\rangle \in \mathcal{C}, M|\psi\rangle = |\psi\rangle$.

There are multiple possible syndrome measurements for a quantum code, and the choice of syndrome measurement can affect the error detection capabilities of the code. For the three-qubit flip code, we can use the following syndrome measurement: Z_1Z_2 and Z_2Z_3.

Claim 2.14. The syndrome measurement Z_1Z_2 and Z_2Z_3 can detect any single and double bit-flip errors in the three-qubit flip code.

Proof. First, we can look at the spectral decomposition of the syndrome measurement Z_1Z_2.

$$Z_1Z_2 = Z \otimes Z \otimes I = (+1) \cdot (|00\rangle\langle 00| + |11\rangle\langle 11|) \otimes I + (-1) \cdot (|01\rangle\langle 01| + |10\rangle\langle 10|) \otimes I \tag{30}$$

By Corollary 2.9, we can see that measuring Z_1Z_2 on $|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle$ yield 1 with certainty. On the other hand, measuring Z_1Z_2 on $X_1|\psi\rangle_L = \alpha|100\rangle + \beta|011\rangle$ yield -1 with certainty. In general, the syndrome measurement Z_1Z_2 can be viewed as a parity check on the first two qubits, which can detect any single bit-flip error on the first two qubits.

Similarly, measuring Z_2Z_3 is a parity check on the last two qubits, which can detect any single bit-flip error on the last two qubits. If we consider any less than or equal to two bit-flip errors, at least one of the two syndrome measurements will yield -1, which indicates that an error has occurred.□

2.3. Stabilizer Formalism and Quantum Stabilizer Codes

The stabilizer formalism was originally introduced by Gottesman [1997] as a framework to reason about quantum error-correcting codes. It simplifies reasoning about a specific class of quantum codes called *stabilizer codes*, which can be described by a set of operators called *stabilizers*. In this section, we provide a brief overview of the stabilizer formalism and stabilizer codes, which is necessary to understand the formalization of stabilizer codes in Rocq.

2.3.1. Pauli Groups

We have defined Pauli matrices in Table 1, which are unitary operators that act on a single qubit. As they are Hermitian matrices, they can also be used as observables to perform syndrome

measurements. We first define the Pauli group on a single qubit, which is the set of all possible products of Pauli matrices and the identity matrix.

Definition 2.15. The single-qubit Pauli group $\mathcal{P}_1$ is defined as

$$\mathcal{P}_1 = \{\pm I, \pm iI, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\} \tag{31}$$

with the group operation being matrix multiplication.

From that definition, we can define the Pauli group on n qubits as the set of all possible tensor products of n single-qubit Pauli operators with an overall phase of ± 1 or $\pm i$.

Definition 2.16. The n-qubit Pauli group $\mathcal{P}_n$ is defined as

$$\mathcal{P}_n = \{P_1 \otimes P_2 \otimes ... \otimes P_n \mid P_i \in \{I, X, Y, Z\}\} \tag{32}$$

with the group operation being matrix multiplication.

remark. This definition does not include the overall phase value, as it is already accounted for in the single-qubit Pauli group. The overall phase value of the n-qubit Pauli group is also ± 1 or $\pm i$, which are all possible products of the overall phase value of the single-qubit Pauli group.

2.3.2. Pauli Operators and Quantum Stabilizers

First, we define a Pauli operator as a Hermitian operator of $\mathcal{P}_n$, which can be used as an observable to perform syndrome measurements.

Definition 2.17. An n-qubit Pauli operator P is a Hermitian element of P_n.

remark. Recall from Postulate 2.8 that any Hermitian operator can be used as an observable. This means that any Pauli operator can be used to perform measurements on a quantum state.

We can then define the *stabilizer* of a code space $\mathcal{C}$ to be:

Definition 2.18. A Pauli operator $S \in \mathcal{P}_n$ is a stabilizer of an n-qubit state $|\psi\rangle$ if $|\psi\rangle$ is the $+1$ eigenvector of S:

$$S|\psi\rangle = |\psi\rangle \tag{33}$$

We also call $|\psi\rangle$ a *stabilizer state* of S.

Let's look at the example of the three-qubit flip code. Recall the code space of the three-qubit flip code is spanned by two codewords:

$$|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle = \alpha|0\rangle_L + \beta|1\rangle_L \tag{34}$$

It is easy to verify that both codewords are stabilized by the Pauli operators Z_1Z_2 and Z_2Z_3. An observation that can be made is that the stabilizers of a code space are also the syndrome measurements of the code. This is not a coincidence, as the syndrome measurements of a code are defined to preserve the code space, which is the same as the definition of stabilizers.

2.3.3. Stabilizer Code

From Definition 2.18, we can define a stabilizer code:

Definition 2.19. A *stabilizer code* is a quantum code whose code space is precisely described by a set of stabilizers.

As seen in the example of the three-qubit flip code, the code space can be described by the stabilizers Z_1Z_2 and Z_2Z_3. This is equivalent to saying that from the stabilizers, we can construct the code space as the set of all stabilized states. It is trivial to see this in the case of the three-qubit flip code. The stabilizer Z_1Z_2 stabilizes any state that has the same value on the first two qubits, and the stabilizer Z_2Z_3 stabilizes any state that has the same value on the last two qubits. The only states that satisfy both conditions are $|000\rangle$ and $|111\rangle$, which span the code space of the three-qubit flip code.

From the definition of the stabilizer code, we can now define the quantum code as a set of stabilizers instead of a code space. The error-detection capabilities can also be described in terms of the stabilizers.

Theorem 2.20. (Error Detection Condition) A stabilizer S can detect an error E if

$$SE = -ES \tag{35}$$

where $-$ is the negation of global phase. In other words, S and E must anti-commute.

Proof. We already know that a syndrome measurement can detect an error if the measurement yields -1 with certainty. We need to show that the condition $SE = -ES$ is equivalent to the condition that measuring S on the state $E|\psi\rangle$ yields -1.

Let $|\psi\rangle$ be the codeword stabilized by S. We know that:

$$SE|\psi\rangle = -ES|\psi\rangle = -E|\psi\rangle \tag{36}$$

This shows that $E|\psi\rangle$ is an eigenvector of S corresponding to eigenvalue -1. By Corollary 2.9, measuring S on $E|\psi\rangle$ will yield -1 with certainty, which means that the error can be detected. □

2.4. Formal Methods and Verification

As mentioned, the main goal of this thesis is to extend the formal verification of stabilizer codes. As such, it is important to understand formal methods, formal verification, and the tools that are used for formal verification. Formal methods are mathematically based languages, tools, and techniques that are used to specify and verify information systems. Formal verification is a part of formal methods that uses those techniques and tools to prove the correctness of a system from its specification [3]. Formal verification has been proven useful especially in the field of computer science, where it was used to verify an operating system kernel [9] or a C compiler [10].

There are many tools that can be used for formal verification, including theorem provers and model checkers. In this study, we will primarily use the Rocq Theorem Prover (formerly known as Coq), as this is the tool used to build the COQ-QECC framework.

Theorem Provers Theorem Provers are tools that can be used to interactively construct formal proofs of mathematical theorems. The process is also called *interactive theorem proving*. Interactive theorem provers like Rocq are independent of the problem domain, and can be used to verify the correctness of different systems as long as the system can be expressed in a formal language. Some examples of theorem provers include Rocq, Isabelle, or Lean [2].

Unlike model checkers, which provide a push-button approach that automate the verification process, theorem provers trade automation for expressiveness and flexibility [5]. Theorem provers allow one to expressively formalize a system and prove its properties, but at the cost of requiring substantial human effort to construct the proofs [2]. This makes it suitable to verify small-scale systems that are too complex to be verified by model checkers, which is the characteristic of quantum programs [5].

Another comparison that is often made is between theorem provers and pen-and-paper proofs. Although pen-and-paper proofs are often more intuitive and easier to understand, they are much more prone to human errors, which can be hard to detect and correct. Theorem provers can also help to reason when large-scale case analysis is required, which pen-and-paper proofs are not suitable for.

2.5. The Rocq Theorem Prover

As mentioned, Rocq is a tool used for interactive theorem proving. It is shipped with a specification language called Gallina, which is a functional programming language that makes it possible to represent the usual types and programs of programming languages. We will introduce the basic concepts of Gallina and Rocq in this section, which will be useful for understanding the formalization of stabilizer codes. Readers who are already familiar with Rocq can skip this section.

2.5.1. The Gallina Language

Gallina is a functional programming language, which means it follows the functional programming paradigm. In functional programming, programs are constructed by applying and composing functions, and the output of a function is determined solely by its input values. Gallina features a flexible type system, which allows one to define new types and perform pattern matching on them. For example, one can define a boolean type with two constructors as follows:

```
Inductive bool : Type :=
| true : bool
| false : bool.
```

or implicitly:

```
Inductive bool : Type := true | false.
```

These definitions define a new type called `bool`, which has two constructors `true` and `false`. Then functions can be defined to operate on the boolean type using pattern matching:

```
Definition andb (b1 b2 : bool) : bool :=
  match b1, b2 with
  | true, true => true
  | _, _ => false
  end.
```

The function andb takes two booleans and matches their values simultaneously. If both are true, it returns true, otherwise it returns false.

In Gallina, types are first-class citizens, which means they can be passed as arguments to functions. As such, we can define a polymorphic type of lists as follows:

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

This list is created with two constructors, nil for the empty list and cons that combines an element of type A with another list to create a new list. This shows why the keyword Inductive is used to define types, as it allows one to define a type by referencing itself. To define a function that operates on lists, we naturally use recursion:

```
Fixpoint length (A : Type) (l : list A) : ℕ :=
  match l with
  | nil => 0
  | cons _ l' => 1 + length A l'
  end.
```

The function length takes a list of type A and matches it with the two constructors. If it is an empty list, it returns 0; otherwise it returns 1 plus the length of the list without the first element.

Finally, Gallina can be used to define theorems. For example, the commutativity of the andb function can be mathematically written as:

$$\forall a \forall b, a \wedge b = b \wedge a \tag{37}$$

which translates to Gallina as:

```
Theorem andb_comm : ∀ b1 b2 : bool, andb b1 b2  =  andb b2 b1.
```

Both the mathematical and Gallina definitions can be read as "for all boolean values b1 and b2, b1 and b2 is equal to b2 and b1".

2.5.2. Rocq Tactics

The main feature of Rocq is the ability to interactively construct formal proofs for theorems. This is done by using tactics, which are commands that are used to manipulate the proof state. There are a wide variety of tactics that can be used to simplify goals, apply lemmas, or perform case analysis. Continuing with the previous example, we can prove the commutativity of andb as follows:

```
Theorem andb_comm : ∀ b1 b2 : bool, andb b1 b2 = andb b2 b1.
 Proof. (* start the interactive proof construction *)
  intros b1 b2. (* move the variables into context *)
  destruct b1, b2; (* perform case analysis on the two booleans*)
  simpl; (* simplify the goal *)
  reflexivity. (* prove the goal of form x = x *)
 Qed. (* finish the proof and save it in the Rocq environment *)
```

The Qed. command checks the proof for any errors and remaining goals. If the proof is correct, it will be saved in the Rocq environment and can be used in other proofs.

Theorem proving in Rocq is interactive, meaning that the user can evaluate the proof state after each tactic is applied. It contains information about the current goal, variables, and hypotheses. In Rocq, the proof state can be viewed at any full stop in the proof.

Additional Materials As this section is only a brief introduction to Rocq, readers who are interested in learning more about Rocq can refer to one or more of the following resources:

- The *Software Foundations* series, which offers a comprehensive guide to Rocq's features and capabilities [14]
- The Rocq Reference Manual, which serves as a reference for Rocq's syntax, tactics, and libraries [16].

2.6. Dependencies in Rocq

This section introduces the Coq libraries that this work depends on. These include Feng's COQ-QECC [4], which is used to formalize stabilizer codes, and its dependent libraries.

2.6.1. Mathematical Components

Mathematical Components (MathComp) are a set of Coq libraries that provide formalization of mathematical structures and theories, including lists, prime numbers, and finite groups [11]. In particular, COQ-QECC takes advantage of the finite group theory formalization in MathComp to formalize Pauli Groups as finite groups. Recall the definition of groups:

Definition 2.21. A group is a set G with a binary operation '$\cdot$' that satisfies the following properties:

1. Closure: For all g_1, g_2 in G, $g_1 \cdot g_2$ is also in G.
2. Associativity: For all g_1, g_2, g_3, $(g_1 \cdot g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3)$.
3. Identity: There exists an element e in G such that for all g in G, $e \cdot g = g \cdot e = g$.
4. Inverse: For all g in G, there exists an element g^{-1} in G such that $g \cdot g^{-1} = g^{-1} \cdot g = e$.

From that, we can define finite groups and finite multiplicative groups:

Definition 2.22. A finite group is a group that has a finite number of elements.

Definition 2.23. A finite multiplicative group is a finite group that has multiplication as its binary operation.

The definition of finite multiplicative groups in MathComp is:

```
Record isMulGroup G of Finite G := {
  mulg : G -> G -> G;
  oneg : G;
  invg : G -> G;
  mulgA : associative mulg;
  mul1g : left_id oneg mulg;
  mulVg : left_inverse oneg invg mulg;
}.
```

Listing 1: Definition of Finite Multiplicative Groups in MathComp

In Listing 1, isMulGroup is a constructor for a finite multiplicative group, which takes a set of related components and properties on the finite type G. The three components are the binary operation mulg, the identity element oneg, and the inverse function invg. From those components, the constructor requires the three properties of associativity, identity, and inverse to be satisfied. As we can see, this definition is consistent with Definition 2.23, with the exception that right identity and right inverse are not required as they can be derived from the other properties.

2.6.2. COQ-QECC

Inspired by the pen-and-paper approach, Feng [2025] proposed a formal verification framework called COQ-QECC for stabilizer codes. The formalization is based on the Stabilizer Formalism, which allows one to reason about the detection and correction capabilities of stabilizer codes using group theory. It consists of three main components:

1. The formalization of Pauli groups, which are the building blocks of stabilizer codes.
2. The formalization of stabilizer codes, together with their detection and correction capabilities.
3. The application of the formalization to verify the detection and correction capabilities of different stabilizer codes, including the 3-qubit bit-flip code and Shor's code.

Note that the formalization provided here also includes subsequent works done by the creators of COQ-QECC. As such, some of the formalization may not be identical to the original COQ-QECC paper by Feng [2025].

Formalization of Pauli Groups The normal approach to formalize Pauli groups is to define them as a set of matrices, and then prove the group properties (Definition 2.23). However, this approach cannot be scaled to large number of qubits, as the size of the Hilbert space grows exponentially with the number of qubits. Instead, Feng [2025] defines Pauli groups as an inductive data type:

```
Inductive PauliBase : Type :=
| I : PauliBase
| X : PauliBase
| Y : PauliBase
| Z : PauliBase.
```

This definition defines a new type called PauliBase, which has four constructors I, X, Y, and Z. This allows one to represent and reason about Pauli bases without having to reason in the Hilbert space. Using that, the Quotient N-Qubit Pauli Group is defined as a tuple of n, as follows:

```
Definition PauliString n := { tuple n of PauliBase }.
```

This defines a new type called PauliString, which is a tuple of n Pauli bases. Note that "Quotient" indicates that the phase factor is not included. This is denoted as $\mathcal{P}_n/Z_4$, with Z_4 being the phase factor and $\mathcal{P}_n$ being the N-Qubit Pauli Group. The phase factor is then defined as a separate type:

```
Inductive phase : Type :=
| One : phase  (* 1 *)  | NOne : phase  (* -1 *)
| Img : phase  (* i *)  | NImg : phase  (* -i *).
```

The phase type has four constructors, One, NOne, Img, and NImg, which represent the four possible phase factors 1, -1, i, and $-i$ respectively. Combining the two definitions, the N-Qubit Pauli Group is defined as a tuple of n Pauli bases together with a phase factor.

```
Definition PauliGroup n  :=  prod  phase  (PauliString n).
```

The PauliGroup type is defined as a product of the phase type and the PauliString type, which represents an element of the N-Qubit Pauli Group.

Each data type mentioned above is equipped with operations and lemmas that are used to prove that they are indeed finite multiplicative groups. Using this, one can effectively reason about the properties of Pauli groups without having to reason with matrices in the Hilbert Space. This is especially useful in the case of Shor's code, as the size of the Hilbert space grows exponentially.

Formalization of Stabilizer Codes First, a Notation was introduced to represent the error operators and stabilizers.

```
Notation PauliOperator := PauliString
```

A Pauli Operator is an element of the Quotient N-Qubit Pauli Group, which is a tuple of n Pauli bases. This notation is used to represent the error operators and stabilizers. Our assumption about the error operators and stabilizers is that they all have global phase of 1, which is why we use the Quotient N-Qubit Pauli Group to represent them. The reason global phase of stabilizers can be assumed to be 1 is that stabilizers of quantum codes are defined to have $+1$ global phase. For error operators, we can assume that they have $+1$ global phase without loss of generality, as the global phase of an error operator does not affect the measurement results, which are equivalent to the detection capabilities of the code.

Then, the following global parameters are defined, which are used to represent a generic stabilizer code:

```
Variable (dim: ℕ).
Variable (SyndromeMeas: {set (PauliOperator dim)}).
Variable (DetectableErrors: {set (PauliOperator dim)}).
Variable (psi: Vector (2^dim)).
```

- dim is the dimension of the code

- SyndromeMeas is the set of stabilizers of the code, which are used to perform syndrome measurements
- DetectableErrors is the set of error operators that can be detected by the code
- psi is the code space, denoted by $|\psi\rangle$

In the stabilizer formalism, the stabilizers need to stabilize the code space, which is expressed as a predicate:

```
Definition obs_be_stabiliser := ∀ (M: PauliObservable dim), M ∈ SyndromeMeas -> stab M psi.
```

This predicate says that for all stabilizers M in the set of stabilizers SyndromeMeas, M must stabilize the code space $|\psi\rangle$. Note that the function stab is a predicate that checks if a Pauli operator stabilizes a quantum state. The details of the definition of stab are ignored in this thesis, as it is not relevant to the discussion.

Then, the detection capabilities of the code for a specific error were formalized as a predicate:

```
Definition anticommute {n} (A B: PauliOperator n) : bool :=
  mul_pn A B == negate_Pn (mul_pn B A).
Definition detectable_alg {n} (obs: {set PauliOperator n}) (E: PauliOperator n) : Prop :=
  ∃ M, (M ∈ obs) ∧ anticommute M E.
```

The anticommute function checks if two Pauli operators anti-commute using mul_pn and negate_Pn, which are functions that multiply two Pauli operators and negate a Pauli operator respectively. Then, the predicate detectable_alg says that an error operator E is detectable by the set of stabilizers obs if there exists a stabilizer M in obs that anti-commutes with E. This is consistent with the error detection condition in Theorem 2.20.

Using the definitions above, the detection capabilities of a stabilizer code were formalized as a predicate:

```
Definition errors_detectable_alg {n} (obs err: {set PauliOperator n}) : Prop :=
  ∀ E, E ∈ err -> (detectable_alg obs E)
```

The predicate errors_detectable_alg says that all error operators in the set of detectable errors err are detectable by the set of stabilizers obs. Finally, all the definitions above are combined to define a quantum error detecting code:

```
Record ErrorDetectingCode := BuildDetectingCode {
  dim: ℕ
  (* Codespace *)
  ; code: Vector (2^dim)
  (* Observables *)
  ; obs: {set PauliObservable dim}
  (* Detectable Errors *)
  ; err: {set PauliOperator dim}
  (* Obligation1: observables must be stabilisers of codespace *)
  ; ob1: obs_be_stabiliser obs code
  (* Obligation2: algebraic (anticommutation) detectability.`. *)
  ; ob2: errors_detectable_alg obs err
}.
```

Applications The formalization of stabilizer codes in COQ-QECC has been applied to verify the detection and correction capabilities of different stabilizer codes. The table below summarizes what has been verified by COQ-QECC for different stabilizer codes.

Stabilizer Code	Detection Capabilities	Correction Capabilities
Bit Flip Code	1-bit flip error	1-bit flip error
Phase Flip Code	1-bit phase error	1-bit phase error
Code [[4,4,2]]	1-bit error	N/A
Shor's Code	1-bit error	1-bit error
Repetition Code (5-qubit)	1-bit flip error	1-bit flip error
Steane's Code	1-bit error	1-bit error
Surface Code	1-bit error	1-bit error

Table 2: Summary of the verification of different stabilizer codes using COQ-QECC

As we can see, COQ-QECC has been used to verify the detection and correction capabilities of different stabilizer codes. However, the framework is missing an important feature, which is the ability to verify the detection capabilities of stabilizer codes for double-qubit errors. This is the main motivation for this thesis, which is to extend the COQ-QECC framework to verify the detection capabilities of stabilizer codes for double-qubit errors.

2.7. Key Insights

We will summarize the key insights from this chapter, which will be useful to understand the purpose of this thesis.

Quantum Programs are hard to Debug and Test As we have discussed in Section 2.1.2, measurements in quantum computing collapse the quantum state. This prevents one from directly measuring the quantum state to inspect the state of the program, which is a common debugging technique in classical computing. Moreover, it is not possible to extract full information about the quantum state, making it hard to look for errors in the program. Although there are statistic-based methods to estimate the state of a quantum program [1], it is hard to distinguish between logical errors and quantum noise.

Quantum Programs can be formally verified Unlike testing and debugging, formal verification does not require one to work with actual quantum programs. This makes formal verification a promising approach to verify the correctness of quantum programs. Formal methods also provide some advantages over testing. First, formal method help to reason about the correctness of a quantum program with generalized inputs, which is not possible with testing. Second, formal methods can enable the development of quantum programming languages that are correct by construction, which can help to reduce the number of errors in quantum programs [5].

Stabilizer Formalism is useful for verifying Stabilizer Codes Similar to how the stabilizer formalism simplifies reasoning about stabilizer codes in pen-and-paper proofs, it also simplifies reasoning about stabilizer codes in formal verification. This has been shown in Section 2.6.2, where the stabilizer formalism is used to reason about the detection and correction capabilities of stabilizer codes using group theory. This is especially useful when we want to reason about stabilizer codes with a large number of qubits, as the size of the Hilbert space grows exponentially with the number of qubits.

Chapter 3.

Related Work

Now that we have discussed the context required to understand the problem and the research question, we will review existing work in the area of quantum program verification.

We start by reviewing existing work in verifying quantum circuit-based programs, which is the foundation of our work. Then, we will review work in verifying quantum error-correcting codes, which is the main focus of this thesis. Finally, we will summarize the related work and discuss how our work fits into the existing literature.

3.1. Verification of Circuit-based Programs

3.1.1. SQIR: Formal Language for Quantum Programs

SQIR is a tool for verifying quantum programs in Rocq. The name SQIR stands for *Small Quantum Intermediate Representation*, as it provides a formal language to represent quantum programs. SQIR has been used to verify the correctness of many famous quantum algorithms, such as Grover's algorithm and Quantum Phase Estimation [8].

3.1.2. QuantumLib: Mathematical Library for Quantum Computing

QuantumLib is the foundational mathematical library that both COQ-QECC and SQIR depend on. It provides a set of mathematical definitions and theorems for reasoning about quantum computing in Coq [13]. Many functionalities in SQIR are based on QuantumLib, while COQ-QECC also depends on QuantumLib for certain mathematical definitions and theorems.

3.2. Verification of Quantum Error-Correcting Codes

3.2.1. COQ-QECC: Formal Verification of Quantum Stabilizer Codes

As mentioned in Chapter 2, COQ-QECC [4] is a formal verification framework for quantum codes using the stabilizer formalism. It provides a set of specifications and functions to formally verify the detection and correction capabilities of quantum codes. The framework necessarily answers the question of whether formal verification of quantum error-correcting codes can be efficiently achieved using the stabilizer formalism.

As seen before, COQ-QECC has been successfully applied to verify the detection and correction capabilities of multiple quantum codes, including Shor's code and the surface code. However, the framework has some limitations and redundancies that can be improved to enhance its usability and scalability, which is the main focus of this thesis. In Section 4.1, we will analyze the framework and identify its limitations and redundancies, which will be addressed in Section 4.2.

3.2.2. SQIR-QECC: Quantum Error-Correcting Codes in SQIR

In SQIR, there is also a formal verification for quantum error-correcting codes using quantum circuits. This part provides a formal verification of the encoder and decoder circuits for Three Qubit Bit-flip code and Nine Qubit Shor code, which are two important quantum codes. The verification is done by proving that the encoder and decoder circuits can correctly detect and correct certain types of errors.

However, this approach is limited to small-scale and key properties of large-scale quantum codes, as the verification of large-scale codes is infeasible due to the limitation of matrix multiplication approach. Another issue with the examples in SQIR-QECC is that the verification is done by manually proving the correctness of the encoder and decoder circuits, which can be time-consuming and non reusable. These problems are the main motivation for the development of COQ-QECC, which aims to provide a more efficient and scalable approach to formal verification of quantum error-correcting codes using the stabilizer formalism [4].

3.3. Summary

In this chapter, we have reviewed existing work in the area that is closely related to our research question. We have discussed the SQIR framework for verifying quantum programs, as well as the baseline COQ-QECC framework for verifying quantum stabilizer codes. As we have observed, the work before COQ-QECC has been limited by the scalability of matrix multiplication approach, while COQ-QECC still has some limitations and redundancies that can be improved to enhance its usability and scalability. In the next chapter, we will analyze the COQ-QECC framework and identify its limitations and redundancies.

Chapter 4.

Contribution to COQ-QECC

We now shift our focus to formalization of quantum stabilizer codes in COQ-QECC. While the formalization has been successful in verifying the detection and correction capabilities of quantum codes for single-qubit errors, certain assumptions have been made that limit its ability to generalize the verification to two-qubit errors. In this chapter, we will discuss the details of the formalization and the changes made to the formalization to allow it to verify the detection capabilities of quantum codes for two-qubit errors.

We start by going through the details of the formalization and how we improve it in Section 4.1. We will also apply the newly improved formalization to the Shor code to give an example of how it can be used to verify quantum codes. Then, we will discuss the changes made to the formalization to address the limitations and redundancy in Section 4.2. Finally, we will discuss how the new formalization can be used to verify the detection capabilities of quantum codes for two-qubit errors in Section 4.3.

This chapter is based on the theory of quantum error-correcting codes in Section 2.2, together with the formalization of quantum stabilizer codes in COQ-QECC (Section 2.6.2).

4.1. Analysis of the Formalization

In this section, we will analyze the formalization of quantum stabilizer codes in COQ-QECC. We will discuss how the formalization formalizes the detection capabilities of the quantum codes, how it verifies the detection capabilities of quantum codes for single-qubit errors, and the limitations of the formalization that prevent it from verifying the detection capabilities of quantum codes for two-qubit errors. We will also identify the redundancy in the formalization that can be removed to improve the formalization.

First, recall the definition of an error detecting code from Section 2.6.2:

```
Record ErrorDetectingCode := BuildDetectingCode {
  dim: ℕ
  (* Codespace *)
  ; code: Vector (2^dim)
  (* Observables *)
  ; obs: {set PauliObservable dim}
  (* Detectable Errors *)
  ; err: {set PauliOperator dim}
  (* Obligation1: observables must be stabilisers of codespace *)
  ; ob1: obs_be_stabiliser obs code
  (* Obligation2: algebraic (anticommutation) detectability. *)
  ; ob2: errors_detectable_alg obs err
}.
```

In order to verify the detection capabilities of a quantum code, we need to build the ErrorDetectingCode record. This requires us to prove two obligations:

1. The observables must be stabilizers of the codespace.
2. The errors must be detectable by the observables.

Redundancy in the Formalization Although the formalization has been successful in verifying the detection capabilities of quantum codes, it has some redundancy that can be removed. According to Definition 2.19, a stabilizer code and its code space can be precisely described by a set of stabilizers. Therefore, we only need to provide either the code space or the stabilizers, not both. In the current formalization, both the code space and the stabilizers (observables) were provided, which create an obligation to prove that the stabilizers are indeed stabilizers of the code space.

Moreover, the obligation obs_be_stabilizer only checks that the observables stabilize the code space, but it does not check that the group generated by the set of observables (or *stabilizer group*) spans the code space. Hence, the set of observables defined might not be sufficient to describe the code space, and therefore a stabilizer that stabilizes the code space might not be in the generated group of the observables.

Verification of Detection Capabilities for Single-Qubit Errors In order to verify the detection capabilities of a quantum code for two-qubit errors, we first look at how the formalization verifies those capabilities for single-qubit errors. Note that this work was done by authors of COQ-QECC [4], but is provided here for the completeness of our analysis.

As mentioned in the previous section, the formalization verifies the detection capabilities through the obligation errors_detectable_alg obs err, which checks that for every error E in the set of errors, there exists an observable in the set of observables such that E anti-commutes with that observable. The definition of errors_detectable_alg is provided below:

```
Definition detectable_alg {n} (obs: {set PauliOperator n}) (E: PauliOperator n) : Prop :=
  ∃ M, (M ∈ obs) ∧ anticommute M E.

Definition errors_detectable_alg {n} (obs err: {set PauliOperator n}) : Prop :=
∀ E, E ∈ err -> (detectable_alg obs E).
```

Let's consider the case of the Shor code. This would require proving that for every single-qubit error E in the set of errors, E anti-commutes with at least one of the stabilizers. Note that the list of stabilizers for Shor code is provided below:

```
(* The 3-qubit block operators *)
Definition BF1 := [p Z, Z, I].
Definition BF2 := [p I, Z, Z].
Definition III := [p I, I, I].
Definition XXX := [p X, X, X].

Definition gens : list (PauliOperator dim) :=
  [tuple of BF1 ++ III ++ III] :: (*Z Z I I I I I I I *)
  [tuple of BF2 ++ III ++ III] :: (*I Z Z I I I I I *)
  [tuple of III ++ BF1 ++ III] :: (*I I I Z Z I I I *)
  [tuple of III ++ BF2 ++ III] :: (*I I I I Z Z I I *)
  [tuple of III ++ III ++ BF1] :: (*I I I I I I Z Z I *)
  [tuple of III ++ III ++ BF2] :: (*Z Z I I I I I Z Z *)
  [tuple of XXX ++ XXX ++ III] :: (*X X X X X X I I I *)
  [tuple of III ++ XXX ++ XXX] :: (*I I I X X X X X X *)
  [::].
```

Consider the single-qubit error $E = [pX, I, I, I, I, I, I, I, I]$. We can see that E anti-commutes with the stabilizer $[pZ, Z, I, I, I, I, I, I, I]$, which is the first stabilizer in the list of stabilizers. Therefore, we can conclude that E is detectable by the set of stabilizers.

The formalization does not manually prove that every single-qubit error anti-commutes with at least one of the stabilizers. Instead, it uses a Rocq tactic to automatically prove this. The tactic is vm_compute, which has the ability to convert proof goals into a decidable form that can be evaluated to true or false. The decidable form of the proof goal is written as:

```
Definition covers {n} (gens : list (PauliOperator n)) : bool :=
  all
    (fun k => all
      (fun p => has
        (fun M => anticommute1 (nth I (val M) k) p)
        gens
      )
      (P1BaseGroup.X :: P1BaseGroup.Y :: P1BaseGroup.Z :: [::])
    )
    (iota 0 n).
```

The covers function checks that for every single-qubit Pauli operator p (i.e., X, Y, or Z) at a position k (X at position 0 would be $XIIIIIIII$), there exists a stabilizer in the list of stabilizers gen so that p anti-commutes with the operator at position k in that stabilizer. To use this function, we need to prove that evaluating covers gens returns true will imply actual detectability of the single-qubit errors. This is the content of the following theorem:

```
Definition getE {n} (p : PauliBase) (i : 'I_n) : PauliString n :=
  [tuple (if i == j then p else I) | j < n].

Definition single_errors {n} : {set ErrorOperator n} :=
  [set getE p i | p in [set: PauliBase] :\ I, i in [set: 'I_n]].

Theorem detect_all_single_check {n} (gens : list (PauliOperator n)) :
  covers gens -> errors_detectable_alg [set x in gens] single_errors.
```

The function getE takes a position i and a single-qubit Pauli operator p and returns the corresponding single-qubit error. The set single_errors is the set of all single-qubit errors. After applying detect_all_single_check, the formalization can then use vm_compute to evaluate covers gens to true, which implies that all single-qubit errors are detectable by the set of stabilizers. However, the proof of this theorem is not trivial, we need two lemmas to prove it.

```
Lemma coversP {n} (gens : list (PauliOperator n)) :
  covers gens ->
  ∀ (i : 'I_n) (p : PauliBase), p != I ->
    has (fun M => anticommute1 (tnth M i) p) gens.

Lemma detect_all_single {n} (gens : list (PauliOperator n)) :
(∀ (i : 'I_n) (p : PauliBase), p != I ->
   has (fun M => anticommute1 (tnth M i) p) gens) ->
    errors_detectable_alg [set x in gens] single_errors.
```

The first lemma converts the outer all functions of decidable form of covers into a forall, which can be used by Rocq to prove the goal. The second lemma then uses the resulting goal with forall to prove that the anticommute1 condition can lead to the errors_detectable_alg condition.

Then, the Theorem detect_all_single_check can be applied to any set of stabilizers to prove that all single-qubit errors are detectable by that set. For example, Shor code detection capabilities for single-qubit errors is verified in the following code block:

```
Theorem Shor_errors_detectable:
  errors_detectable_alg ShorSyndromeMeasure single_errors.
Proof. by apply: detect_all_single_check; vm_compute. Qed.

Definition ShorDetectingCodeAlg_double : ErrorDetectingCodeAlg :=
BuildDetectingCodeAlg dim
  ShorSyndromeMeasure
  double_errors
  Shor_errors_detectable_double.
```

As we can see, the prove can be done automatically by applying the theorem detect_all_single_check and using vm_compute to evaluate covers gens to true. This is how the formalization verifies the detection capabilities of all quantum codes for single-qubit errors.

Limitation of the verification There are certain limitations of the formalization that prevent it from verifying the detection capabilities of quantum codes for two-qubit errors. First, let's look at one of the Lemmas used to prove the theorem detect_all_single_check:

```
Lemma detect_all_single {n} (gens : list (PauliOperator n)) :
(∀ (i : 'I_n) (p : PauliBase), p != I ->
  has (fun M => anticommute1 (tnth M i) p) gens) ->
   errors_detectable_alg [set x in gens] single_errors.
```

We can see in the lemma that there is an extra condition that requires p to be different from the identity operator I. This is because the identity operator does not anti-commute with any operator, and therefore it is not detectable by any stabilizer. However, this condition is not sufficient if we want to generalize to two-qubit errors. In fact, the problem originates from the definition of errors_detectable_alg:

```
Definition detectable_alg {n} (obs: {set PauliOperator n}) (E: PauliOperator n) : Prop :=
 ∃ M, (M ∈ obs) ∧ anticommute M E.

Definition errors_detectable_alg {n} (obs err: {set PauliOperator n}) : Prop :=
 ∀ E, E ∈ err -> (detectable_alg obs E).
```

In this definition, we are missing the condition that E should not stabilize the code space. In fact, if E stabilizes the code space, then it will commute with all stabilizers, and therefore will not be detectable by any stabilizer. This is the reason why the formalization cannot verify the detection capabilities of some quantum codes for two-qubit errors, such as Shor code.

Let's look at a specific example of a two-qubit error that stabilizes the code space of Shor code. Consider the two-qubit error $E = ZZIIIIIII$. We can see that E stabilizes the code space (i.e., E is in the list of stabilizers). This implies that E will not be detectable by any stabilizer, and therefore the formalization will not be able to verify the detection capabilities of Shor code for this two-qubit error.

4.2. Changes to the Formalization

In this section, we will discuss the changes made to the framework to address the aforementioned limitations and redundancy. We will also discuss how these changes improve the framework and allow it to verify the detection capabilities of quantum codes for two-qubit errors.

4.2.1. Unification of Code Space and Stabilizers

As discussed, stabilizer codes can be precisely described by either the code space or the set of stabilizers. However, the current formalization requires both the code space and the set of stabilizers, together with an obligation to prove that the stabilizers indeed stabilize the code space. This creates redundancy in the formalization, and lacks the condition that the group generated by the set of stabilizers should span the code space.

In order to address this, the most straightforward solution is to remove the code space from the formalization

```
Record ErrorDetectingCodeAlg := BuildDetectingCodeAlg {
  dim_alg: ℕ
(* Observables *)
; obs_alg: {set PauliObservable dim_alg}
(* Detectable Errors *)
; err_alg: {set PauliOperator dim_alg}
(* Obligation2: algebraic (anticommutation) detectability.  Entails the
   semantic `errors_detectable` via `edc_errors_detectable`. *)
; ob_alg: errors_detectable_alg obs_alg err_alg
}.
```

Although the changes made to the formalization are minor, they have a significant impact on the time it takes to verify a quantum code. In fact, in ~ 350 lines of code written explicitly to verify the detection and correction capabilities of the Shor code (not including shared definitions and lemmas), most of them are dedicated to proving the stabilizers indeed stabilize the code space. By removing the code space from the formalization, we can significantly reduce the time it takes to verify a quantum code.

Additionally, the existence of the code space in the framework makes it differs from the stabilizer formalism, which does not require the code space to be explicitly defined. By removing the code space, we can make it more similar to the pen-and-paper stabilizer formalism, which is more intuitive and easier to reason about.

4.2.2. Condition for Stabilizing Errors

For the formalization to be able to verify the detection capabilities of quantum codes for two-qubit errors, we need to add a condition to check for errors that stabilize the code space. This can be done by adding a new definition trivial_alg that checks if an error is generated by the set of stabilizers. The definition of trivial_alg is provided below:

```
Definition stab_group_op {n} (obs: {set PauliOperator n}) : {set PauliOperator n} :=
  generated [set M | M in obs].

Definition trivial_alg {n} (obs: {set PauliOperator n}) (E: PauliOperator n) : Prop :=
  E ∈ stab_group_op obs.
```

The definition stab_group_op takes a set of stabilizers and returns the group generated by that set. Then, the definition trivial_alg checks if the error E is in the group generated by the set of stabilizers. If E is in the group generated by the set of stabilizers, then it will stabilize the code space, and therefore it will not be detectable by any stabilizer. The new definition of errors_detectable_alg will be:

```
Definition errors_detectable_alg {n} (obs err: {set PauliOperator n}) : Prop :=
  ∀ E, E ∈ err -> (detectable_alg obs E) ∨ (trivial_alg obs E).
```

This definition requires that for every error E in the set of errors, either E is detectable by the set of stabilizers, or E is in the group generated by the set of stabilizers. This will allow the formalization to verify the detection capabilities of quantum codes for two-qubit errors. Let's look at the previous example of Shor code where the two-qubit error $E = ZZIIIIIII$ stabilizes the code space. We can

see that E is in the group generated by the set of stabilizers, and therefore will satisfy the new definition of errors_detectable_alg. Note that the reason we want E to be in the generated group is that some errors, such as $E_1 = ZIZIIIII$, are not in the list of stabilizers, but they are in the group generated by it. Therefore, we need to check for errors that are in the group generated by the set of stabilizers, not just the list of stabilizers.

The definition is equivalent to checking if an error stabilizes the code space, as the group generated by the set of stabilizers is the same as the set of operators that stabilize the code space. However, this allows us to reason about the errors without having to perform matrix multiplication. This is important because COQ-QECC aims to avoid matrix multiplication, as the dimension of the Hilbert space grows exponentially with the number of qubits. The change also makes the formalization more similar to the stabilizer formalism, where a detectable error is required to not stabilize the code space [12].

4.3. Verification of Two-Qubit Detection Capability

After the formalization improvements, we can now verify the two-qubit detection capability of stabilizer codes. Our main goal for this section is to build a framework that can verify the two-qubit detection capability automatically using vm_compute, similar to how COQ-QECC verifies the one-qubit detection capability in Section 4.1.

First, we will define a two-qubit error as the product of two single-qubit errors.

```
Definition getE2 {n} (p1 : PauliBase) (i1 : 'I_n) (p2 : PauliBase) (i2 : 'I_n) : PauliOperator n :=
  mul_pnb (getE p1 i1) (getE p2 i2).
```

The function mul_pnb is the multiplication of two Pauli operators. We now define the set of two-qubit errors as the set of all two-qubit errors that can be generated by the multiplication of two single-qubit errors.

```
Definition double_errors {n} : {set ErrorOperator n} :=
  \bigcup_(i1 in [set: 'I_n])
  \bigcup_(i2 in [set: 'I_n] :\ i1)
    [set getE2 p1 i1 p2 i2 | p1 in [set: PauliBase] :\ I, p2 in [set: PauliBase] :\ I].
```

There are two conditions that we add to the definition of two-qubit errors to make sure it does not include single-qubit errors or the identity operator.

1. The two positions i_1 and i_2 must be different, which is ensured by the condition i2 in [set: 'I_n] :\ i1.
2. The two single-qubit Pauli operators p_1 and p_2 must be different from the identity operator, which is ensured by the condition p1 in [set: PauliBase] :\ I, p2 in [set: PauliBase] :\ I.

Then, similar to the single-qubit errors case, we can define a decidable form of the two-qubit errors detection capability, which we will call cover_double. The definition of cover_double is provided below:

```
Definition covers_double {n} (gens: list (PauliOperator n)) : bool :=
 let S := map (fun seq => @prodE_op n seq) (subseqs gens) in
 all (
  fun k1 => all (
   fun p1 => all (
    fun k2 => all (
     fun p2 =>
      (k1 != k2) ==> has (
       fun M => anticommute1 (nth I (val M) k1) p1 != anticommute1 (nth I (val M) k2) p2
      ) gens
      || has (
       fun T =>
        all (fun k => eq_op (nth I (val T) k) (wt2 p1 k1 p2 k2 k)) (iota 0 n)
      ) S
    ) (P1BaseGroup.X :: P1BaseGroup.Y :: P1BaseGroup.Z :: [::])
   ) (iota 0 n)
  ) (P1BaseGroup.X :: P1BaseGroup.Y :: P1BaseGroup.Z :: [::])
 ) (iota 0 n).
```

This is a more complex definition that covers, as it iterates through all possible two-qubit errors and has some additional conditions to check. Let's break down the definition of covers_double:

- S: Defined as the list of all possible products of the stabilizers in gens. This is used to check if a two-qubit error is in the group generated by the stabilizers.
- The outer all functions iterate through all possible two-qubit errors, which are defined by the positions k1 and k2, and the single-qubit Pauli operators p1 and p2.
- The first condition is equivalent to the detectable_alg condition, which check if there exists a stabilizer in gens that anti-commutes with the two-qubit error. This is done by checking if the anti commutation of the two single-qubit errors with the stabilizers are different.
- The second condition is equivalent to the trivial_alg condition, which check if the two-qubit error is equal to an element in S. This is done by checking all qubits in the two-qubit error with all qubits of the element in S to see if they are equal.

Similar to the single-qubit errors case, we then define a theorem detect_all_double_check that states that if covers_double gens returns true, then all two-qubit errors are detectable by the set of stabilizers gens.

```
Definition detect_all_double_check {n} (gens : list (PauliOperator n)) :
 covers_double gens -> errors_detectable_alg [set x in gens] double_errors.
```

To prove this, we also use the same approach as the single-qubit errors case, which is to prove two lemmas that convert the outer all functions of covers_double into forall:

```
Definition covers_doubleP {n} (gens: list (PauliOperator n)) :
 covers_double gens ->
 ∀ (i j : 'I_n) (p q : PauliBase), p != I -> q != I -> i != j ->
  has (fun M => anticommute1 (tnth M i) p != anticommute1 (tnth M j) q) gens
  ∨ let S := map (fun seq => @prodE_op n seq) (subseqs gens) in
     has (fun T => all (fun k => eq_op (nth I (val T) k) (wt2 p i q j k)) (iota 0 n)) S.
```

Then, we provide two lemmas that convert from the anticommute1 and eq_op conditions to the detectable_alg and trivial_alg conditions, respectively:

```
Lemma weight2_detectable {n} (gens : list (PauliOperator n)) (p q : PauliBase) (i j : 'I_n) :
  i != j ->
  has (fun M => anticommute1 (tnth M i) p != anticommute1 (tnth M j) q) gens ->
  detectable_alg [set x in gens] (getE2 p i q j).

Lemma weight2_trivial {n} (gens : list (PauliOperator n)) (p q : PauliBase) (i j : 'I_n) :
  i != j ->
  has (fun T => all (fun k => eq_op (nth I (val T) k) (wt2 p i q j k)) (iota 0 n))
    (map (fun seq => @prodE_op n seq) (subseqs gens)) ->
  trivial_alg [set x in gens] (getE2 p i q j).
```

Using these lemmas, we can then prove a lemma that converts the result of covers_doubleP into the errors_detectable_alg condition:

```
Theorem detect_all_double {n} (gens: list (PauliOperator n)) :
  (∀ (i j : 'I_n) (p q : PauliBase), p != I -> q != I -> i != j ->
    has (fun M => anticommute1 (tnth M i) p != anticommute1 (tnth M j) q) gens
    ∨ let S := map (fun seq => @prodE_op n seq) (subseqs gens) in
        has (fun T => all (fun k => eq_op (nth I (val T) k) (wt2 p i q j k)) (iota 0 n)) S)
    -> errors_detectable_alg [set x in gens] double_errors.
```

Using detect_all_double and covers_doubleP, we can then prove the theorem detect_all_double_check:

```
Definition detect_all_double_check {n} (gens : list (PauliOperator n)) :
  covers_double gens -> errors_detectable_alg [set x in gens] double_errors.
Proof.
  by move => H; apply detect_all_double; apply covers_doubleP; exact: H.
Qed.
```

This is a significant result, as it allows us to verify the detection capabilities of quantum codes for two-qubit errors automatically using vm_compute, similar to how COQ-QECC verifies the one-qubit detection capability in Section 4.1.

Application to Shor Code and Other Stabilizer Codes We can now apply the results of the previous section to verify the detection capabilities of quantum codes for two-qubit errors. For example, we can verify the detection capabilities of the Shor code for two-qubit errors using the following code block:

```
Theorem Shor_errors_detectable_double:
  errors_detectable_alg ShorSyndromeMeasure double_errors.
Proof.
  by apply: detect_all_double_check; vm_compute.
Qed.

Definition ShorDetectingCodeAlg_double : ErrorDetectingCodeAlg :=
  BuildDetectingCodeAlg dim
    ShorSyndromeMeasure
    double_errors
    Shor_errors_detectable_double.
```

As we can see, the proof can be done automatically by applying the theorem detect_all_double_check to convert the goal into a decidable form, and then using vm_compute to evaluate covers_double gens to true. Similarly, we can apply the same approach to verify the detection capabilities of other stabilizer codes for two-qubit errors. The table below shows the results of verifying the detection capabilities of some stabilizer codes for two-qubit errors using the improved formalization.

Stabilizer Code	**Detection Capability**	**Additional Detection Capability**
Bit Flip Code	1-bit flip error	2-bit error
Phase Flip Code	1-bit phase error	2-bit error
Code [[4,4,2]]	1-bit error	N/A
Shor's Code	1-bit error	2-bit error
Repetition Code (5-qubit)	1-bit flip error	2-bit error
Steane's Code	1-bit error	2-bit error
Surface Code	1-bit error	2-bit error

Table 3: COQ-QECC verification of error detection capabilities before and after the improvements

The results for Shor's Code, Steane's Code and Surface Code are expected, as they are all designed to correct single-qubit errors and detect two-qubit errors. Code [[4,4,2]] is a code that can detect single-qubit errors, but cannot detect two-qubit errors, so it is also expected that the improved formalization cannot verify the detection capabilities of this code for two-qubit errors.

However, for the Repetition code categories (Bit Flip Code, Phase Flip Code, and Repetition Code (5-qubit)), the results are rather unexpected, as they can all detect arbitrary two-qubit errors instead of just two-bit flip or two-bit phase errors. However, this does not mean that the improved formalization is wrong, but rather it is a result of the way these codes are designed. Note that the 2-qubit error detection capability does not mean that the code can detect one-qubit errors, as we only check for strictly two-qubit errors.

Let's look at the Bit Flip Code as an example to see why it can detect arbitrary two-qubit errors. Recall that the Bit Flip Code is a 3-qubit code that has two stabilizers ZZI and IZZ. Consider an arbitrary two-qubit error E. If there is at least one qubit that is affected by X or Y error, then E will anti-commute with at least one of the stabilizers, and therefore it will be detectable. If all two

qubits are affected by Z error, then E will be in the group generated by the stabilizers, and therefore it will also be detectable. Therefore, the Bit Flip Code can detect arbitrary two-qubit errors. The same reasoning can be applied to the Phase Flip Code and Repetition Code (5-qubit) to show that they can also detect arbitrary two-qubit errors. However, they cannot detect a single-qubit phase flip error like ZII, which is why they cannot detect arbitrary single-qubit errors.

Chapter 5

Discussion and Conclusion

In this thesis, we have analyzed COQ-QECC [4], a formal verification framework for quantum stabilizer codes, and proposed several improvements to enhance its application and usability. This chapter summarizes the contributions of this thesis and discusses the limitations of our work, as well as potential future directions for research.

5.1 Summary

We have analyzed redundancies and limitations in Section 4.1, proposed changes to address these issues in Section 4.2, and implemented a method to automatically verify the detection capability for two-qubit errors in Section 4.3. Compared to the original COQ-QECC framework, our contributions have significantly reduced the amount of work required to formalize and verify a quantum stabilizer code. We also expanded the framework to verify the double-qubit error detection capability of quantum codes, while maintaining the correctness and the main goal of the framework, which is to reason about quantum error-correcting codes using stabilizer formalisms instead of relying on matrix multiplication.

5.2 Limitations and Future Work

Our work has successfully improved the COQ-QECC framework. However, there is still work to be done to further enhance the framework's capabilities and usability.

Stabilizer Formalisms Correctness In the original COQ-QECC framework, the stabilizer formalisms' correctness is proven by showing that the stabilizer formalism is equivalent to reasoning in Hilbert space using matrix multiplication. However, our modifications have not yet fully proven this equivalence. This is an important aspect of the framework, as it ensures that the algebraic reasoning in the stabilizer formalism is consistent with the underlying quantum mechanics. Future work could focus on establishing this equivalence for the proposed changes, which would further strengthen the framework's correctness and reliability.

Undetectable and Uncorrectable Errors We have focused on verifying the detection capability for two-qubit errors in our work. However, we have not provided a method to verify that a quantum stabilizer code cannot detect or correct certain errors. The original COQ-QECC framework has manually proven some of these properties for specific codes, but a more general method to verify undetectable and uncorrectable errors would be beneficial. Future work could explore ways to automate the verification of these properties, which would further enhance the framework's capabilities.

Bibliography

[1] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. 2023. Quantum software testing: State of the art. *Journal of Software: Evolution and Process* 35, 4 (2023), e2419. https://doi.org/https://doi.org/10.1002/smr.2419

[2] Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. 2024. Formal Methods in Industry. *Form. Asp. Comput.* 37, 1 (December 2024). https://doi.org/10.1145/3689374

[3] Edmund M. Clarke and Jeannette M. Wing. 1996. Formal methods: state of the art and future directions. *ACM Comput. Surv.* 28, 4 (December 1996), 626–643. https://doi.org/10.1145/242223.242257

[4] Qiuyi Feng, Udaya Parampalli, and Christine Rizkallah. 2025. Formal Verification of Quantum Stabilizer Code. In *Proceedings of the 11th International Workshop on Coq for Programming Languages (CoqPL 2025)*, January 2025. Retrieved from https://popl25.sigplan.org/details/CoqPL-2025-papers/7/Formal-Verification-of-Quantum-Stabilizer-Code

[5] Qiuyi Feng. 2025. Formal Verification of Quantum Stabiliser Codes By Stabiliser Formalism. Master's thesis. The University of Melbourne. Retrieved from https://people.eng.unimelb.edu.au/rizkallahc/theses/chew-yi-feng-masters-thesis.pdf

[6] Daniel Gottesman. 1997. Stabilizer codes and quantum error correction. Doctoral dissertation.

[7] Brian C. Hall. 2013. *Quantum Theory for Mathematicians*. Springer New York. https://doi.org/10.1007/978-1-4614-7116-5

[8] Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021. Proving Quantum Programs Correct. (July 2021). https://doi.org/10.4230/LIPIcs.ITP.2021.21

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, 2009. Association for Computing Machinery, Big Sky, Montana, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[10] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[11] Assia Mahboubi and Enrico Tassi. 2022. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.7118596

[12] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press.

[13] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*, 2017. Association for Computing Machinery, Paris, France, 846–858. https://doi.org/10.1145/3009837.3009894

[14] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2026. *Logical Foundations*. Electronic textbook.

[15] Robert Rand. 2018. Formally Verified Quantum Programming. Dissertations. Retrieved from https://rand.cs.uchicago.edu/publication/rand-2018-formally/

[16] The Rocq Development Team. 2026. The Rocq Prover Reference Manual. Retrieved from https://rocq-prover.org/doc

[17] Joschka Roffe. 2019. Quantum error correction: an introductory guide. *Contemporary Physics* 60, 3 (2019), 226–245. https://doi.org/10.1080/00107514.2019.1667078