Abstract

Quantum Error Correction (QEC) is essential for the functioning of large-scale fault-tolerant quantum computers, and its implementation is a very sophisticated process involving both quantum and classical hardware. Formulating and verifying the decomposition of logical operations into physical ones is a challenge in itself. In this paper, we propose QECV, a verification framework that can efficiently verify the formal correctness of stabilizer codes, arguably the most important class of QEC codes. QECV first comes with a concise language, QECV-Lang, where stabilizers are treated as a first-class object, to represent QEC programs. Stabilizers are also used as predicates in our new assertion language, QECV-Assn, as logical and arithmetic operations of stabilizers can be naturally defined. We derive a sound quantum Hoare logic proof system with a set of inference rules for QECV to efficiently reason about the correctness of QEC programs. We demonstrate the effectiveness of QECV with both theoretical complexity analysis and in-depth case studies of two well-known stabilizer QEC codes, the repetition code and the surface code.
1 Introduction

Quantum error correcting (QEC) codes [1,2,3] are vital for implementing fault-tolerant quantum computation and overcoming the noise present in quantum hardware [4,5]. Quantum device vendors are exploring various quantum error correction codes to boost the error tolerance of quantum computation. For example, Google exploits the repetition code [6] to suppress errors in their Sycamore device [7]. IBM extends the surface code [1] to their low-degree superconducting quantum computers [8], and Amazon utilizes the concatenated cat code [2] to build a fault-tolerant qubit.

A central concept in QEC code design is that of a “stabilizer” [9]. The term refers to a quantum operator which expresses the correlations present among the physical qubits forming the logical qubit. Operations to encode logical states or detect and correct errors can be derived once the stabilizers of the QEC code are provided. In a stabilizer code, these primitive operations over logical qubits consist of quantum programs, one for each primitive. As an example, Fowler et al. [1] developed a series of programs on the surface code to implement the primitive operations (e.g., a logical X gate, H gate, and CNOT gate) necessary for universal fault-tolerant quantum computation. While executing these primitives, any stabilizer code implementation requires frequent measurements of the physical qubits to detect possible hardware errors and, thus, apply the appropriate correction operation.

When analyzing the correctness of a stabilizer code, there are two key aspects that need to be considered: 1) the correctness of the logical operation: The stabilizer code must implement the desired logical operation over the logical qubits by applying several physical operations over the constituent physical qubits. 2) the capability of error correction: When hardware errors happen, there exist protocols for error decoding and correction which are based on the information extracted by measurements in QEC codes.

To the best of our knowledge, there is no formal verification framework for QEC codes yet. Previous works on quantum error correction [1,3,10,11,12] demonstrate the correctness of the proposed QEC protocol by numerical simulation on QEC programs. However, this approach does not provide formal proof for the correctness of QEC codes. We asked ourselves the question:

Can one formally verify QEC codes using existing verification frameworks for general quantum programs since QEC codes are effectively a kind of quantum program?

In this vein, one well-developed method for quantum program verification [13,14,15] is to use dynamic techniques such as quantum simulation. This category of methods can accurately characterize the quantum state evolution of small quantum programs but can not scale up to large quantum programs with more than 50 qubits due to the exponential computation overhead [13]. This poor scalability of dynamic methods makes it inefficient for the verification of QEC codes since a reasonably fault-tolerant logical qubit would inevitably involve many physical qubits [1].

Another type of verification works [16,17,18,19,20,21,22], exploits static analysis techniques to reason about quantum programs. These works all naturally incur exponential computation overhead since they need to track the evolution of some Hermitian matrices, which are of dimension $O(4^n)$ for a $n$-qubit system. Yu and Palsberg [23] recently proposed a computationally efficient quantum abstract interpretation technique to reason about the correctness for certain kinds of assertions. Yet, trading in accuracy is not suitable for the verification of QEC codes which requires exact correctness.

Thus, our answer to the question above is:

No, adopting general verification frameworks sacrifices either scalability or accuracy.

To this end, we build a formal verification framework crafted for quantum stabilizer codes to squeeze out the best verification efficiency without compromising accuracy. Our approach rests on a central idea: while realizing scalable verification for a general quantum program is hard, it might be possible to efficiently verify the QEC codes with a deliberate separation between the hard and easy parts in the verification process. We observe that most parts in verifying QEC codes turn out to fall into the easy region because they can be efficiently processed by preserving the high-level stabilizer information. In particular, stabilizers provide a compact description for QEC codes [9]. Major components in QEC codes, e.g., error channels, Clifford gates, and parity measurements can all be described within the stabilizer formalism. Besides, it only takes $O(n^2)$ complexity to emulate Clifford operations on stabilizers [9]. Using stabilizers as predicates, we can potentially avoid unnecessary exponential computation overhead in general quantum program reasoning.

We first propose a concise QEC programming language, QECV-Lang, where stabilizers are treated as first-class objects. This allows QECV-Lang to represent in an intuitive and compact form different operations in the QEC implementation, ranging from encoding to decoding and to error correction. We develop operational semantics and denotational semantics for QECV-Lang, which lays the foundation for building up the syntax-directed verification system. One key enabler for our semantics design is the separation between quantum states of the physical qubits and the information captured by the stabilizers. It describes the former with partial density matrices and treats the latter as a classic program state. It significantly simplifies the computations associated with the stabilizers by avoiding the direct description of how the stabilizers are measured and instead focusing on how the high-level information is used in the decoding stage.

We further develop a new assertion language, named QECV-Assn, in which the predicates are defined by stabilizers. To enhance the logical expressive power, we introduce not only the standalone stabilizers, but also the arithmetic and logical expressions of them for expressing assertions. The key insight behind such a design is to form a universal state space for verification, as a standalone stabilizer could not represent the whole state space. We also remark that, despite our QECV-Lang and QECV-Assn are crafted for QEC codes, their design allows for broader applicability. Any quantum programs and quantum predicates that could be expressed in the quantum while-language [16] and Hermitian-based predicates [19] can also be expressed in our languages. With such an elegant property, our languages could potentially serve as the common foundation of both general quantum programs and QEC designs. It will therefore avoid the dilemma of choosing between a general
but less effective language or a domain-specific but more effective language.

Together with QECV-Lang and QECV-Assn, we further establish a sound quantum Hoare logic for QEC programs. This proof system can demonstrate exponential time and space saving for most QEC operations (e.g., state preparation, Pauli gates, and error detection) when, in a real QEC program, the predicate formulae is commutable with the stabilizer variable in our quantum Hoare logic. Even for the most challenging verification of the logical T gate implementation, our proof system may still have this strong advantage, depending on the actual T gate implementation of the target QEC code.

We give both theoretical analysis and detailed case studies for evaluating the proposed framework. We first compare QECV with the vanilla quantum Hoare logic [16] and quantum while-language [16] in terms of the complexity when describing and verifying the quantum stabilizer codes. We then give very detailed, step-by-step case studies of two well-known QEC codes. This allows the reader to familiarize the concepts behind our framework and its usage for verifying the correctness of quantum stabilizer codes.

To summarize, our major contributions are as follows:

- We propose QECV-Lang, a concise programming language for QEC codes and give a full specification of its syntax and operational/denotational semantics.
- We formulate a new assertion language QECV-Assn. It is the first effort that exploits stabilizers for building universal assertions on quantum states.
- We develop a sound quantum Hoare logic framework based on QECV-Assn and QECV-Lang with a set of inference rules to verify the correctness of QEC programs.
- We demonstrate the effectiveness of our framework with both theoretical complexity analysis and in-depth case studies of two well-known stabilizer QEC codes.

2 Background

In this section, we introduce the background for our work. We summarize our key notations in Table 1. We do not cover the basics of quantum computing (e.g., density operator, unitary transformation) and recommend [6] for reference.

| $\vert \psi \rangle$, $\vert \phi \rangle$, $\vert 0 \rangle$, $\vert 1 \rangle$, $\vert + \rangle$, $\vert - \rangle$ | pure quantum states; |
| $\rho$, $\vert \psi \rangle \langle \psi \vert$ | density operators; |
| $U$ | unitary transformations; |
| $O, M$ | Observable; Use $\mathcal{M}$ to stress measurement; |
| $H$ | the Hilbert space of quantum states; |
| $D(H)$ | the set of partial density operators on $H$; |

2.1 Quantum Error Correction and Stabilizer

Most QEC codes consist of three stages: encoding, decoding, and error correction. The encoding protocol projects an unprotected state of the data qubits into the subspace generated by the logical states. The decoding protocol detects potential errors by performing parity measurements on data qubits. Finally, the correction protocol removes the errors by driving the quantum state back to the logical subspace.

Stabilizer. The stabilizer formalism proposed by Gottesman [9] provides a unifying description of many QEC codes. Given a $n$-qubit state $\vert \psi \rangle$ and a Pauli string $s \in \otimes^n \{I, X, Y, Z\}$, we say that $s$ is a stabilizer of $\vert \psi \rangle$, or $\vert \psi \rangle$ is stabilized by $s$, if $s \vert \psi \rangle = \vert \psi \rangle$. When the stabilized state $\vert \psi \rangle$ is clear in the context, we will simply say that $s$ is a stabilizer, without referring to the stabilized state. We then can use multiple stabilizers to naturally identify a subspace, which is the intersection of the stabilizers’ projection subspaces, to represent the logical states. Moreover, stabilizers can also be observables and they can be measured to ascertain whether the state of the data qubits is in the correct subspace. These measurements are called stabilizer measurements. Another advantage of the stabilizer formalism is that it can describe standard quantum error channels on stabilized states. Once the stabilizers are determined, the QEC encoding, decoding, and error correction can be easily derived.

Due to the centrality of stabilizers in QEC codes, including them as fundamental concepts of the verification framework is a promising direction. Benefiting from the expressiveness of the stabilizer formalism, we believe that our verification framework for quantum stabilizer codes will be applicable to many existing QEC codes and will help designing novel implementations of fault-tolerant operations.

In the writing time of this paper, we also notice that Rand, Sundaram, Singhal and Lackey [24] [25] [26] develop an elegant type-checking system for general quantum programs based on the stabilizer formalism. Despite that we share some high-level insights in utilizing stabilizer formalism [9], we differ significantly in the overall optimization goal and the entire design framework. In addition, their work only consider quantum circuits and cannot deal with branch statements (e.g., if and while). These statements are indispensable for QEC programs. In contrast, we can handle branch statements by incorporating stabilizer variables in the design of quantum predicate logic. Last but not least, this paper also develops a compact language for QEC programs while their work follows the vanilla quantum circuit language [6].

2.2 Quantum Program Language

The quantum while-language proposed by Ying [16] provides a universal description of purely-quantum programs without classical variables. It focuses on characterizing basic quantum program structures and its syntax is defined in Backus-Naur form as follows:
Here Prog plays the role of a statement of the quantum while-language, but can also indicates the full program when seen as a sequence of statements. In the expressions above, q denotes a quantum variable and \( \overline{q} \) represents a quantum register associated with a finite number of quantum variables. The language constructs above are explained as follows: (1) \( \text{skip} \) does nothing; (2) \( q := |0\rangle \) prepare quantum variable \( q \) in state \( |0\rangle \); (3) \( \overline{q} := U[\overline{q}] \) perform unitary operation \( U \) on the quantum registers \( \overline{q} \); (4) \( \text{Prog}_1; \text{Prog}_2 \) is the sequencing of statements; (5) \( \text{case} \ M[\overline{q}] = m \rightarrow \text{Prog}_m \text{ end} \) measures the quantum variables in \( \overline{q} \) with semi-positive Hermitian operators \( M = \{ M_0, M_1, \ldots, M_m \} \) and executes program \( \text{Prog}_m \) if the measurement outcome is \( m \); (6) \( \text{while} \ M[\overline{q}] = 1 \ do \ \text{Prog}_1 \ \text{done} \) measures qubits \( \overline{q} \) and executes \( \text{Prog}_1 \) if the measurement outcome is 1. If the measurement outcome is 0, the while loop terminates. Here \( \text{Prog} \) is assumed to have only two possible outcomes, \( m = 0, 1 \).

The semantics of quantum while-language is developed by assuming partial density operators as quantum program states. The details can be found in [16].

### 2.3 Quantum Hoare Logic and Quantum Predicates

**Quantum Hoare Logic** [16] provides a syntax-directed proof system for reasoning about quantum program correctness. The basic computing unit of quantum Hoare logic is the Hoare tripe, which is in the form of \( \{ A \} \text{Prog} \{ B \} \). Here \( A \) and \( B \) are quantum predicates, \( \text{Prog} \) is the quantum program, \( A \) is often called precondition while \( B \) is called post-condition. The general meaning of the quantum Hoare tripe is that, if the input state satisfies \( A \), then the output state of \( \text{Prog} \) satisfies \( B \). The exact mathematical interpretation of the quantum Hoare tripe depends on the type of the predicates used.

A general quantum predicate is a Hermitian operator \( O \) [16] where \( 0 \leq Tr(O\rho) \leq 1, \forall \rho \in D(\mathcal{H}) \). A quantum state \( \rho \) satisfies the quantum predicate \( O \) depending on the value of \( Tr(O\rho) \), which represents the expectation value of \( O \) on state \( \rho \). In practice, measuring \( Tr(O\rho) \) can be very time-consuming. This can be avoided by restricting the predicate from a general Hermitian operator to a projection operator \( P \) with the property that \( P^2 = I \). We say that a quantum state \( \rho \) satisfies predicate \( P \) (denoted by \( \rho \models P \)) if \( P\rho = \rho \). A projection operator \( P \) can also be described by its subspace with eigenvalue +1, namely \( S_P = \{ |\psi\rangle \in \mathcal{H} : \langle \psi | P | \psi \rangle = 1 \} \) in the qubits’ Hilbert space \( \mathcal{H} \). Birkhoff and Neumann [2] define a quantum logic on the set of subspaces in \( \mathcal{H} \) in which, for example, the logical \( \text{and} \) corresponds to the intersection of subspaces. This construction induces logic operations on projection operators by considering the equivalent operation on the associated subspaces.

### 3 QECV-Lang

In this section, we introduce QECV-Lang, a concise language for QEC Programs. We define its syntax, operational semantics, and denotational semantics.

#### 3.1 Syntax

We restate the notation for quantum variables as follows: Define qVar as the set of quantum variables, \( q \) as a metavariable ranging over quantum variables, and \( \overline{q} \) to be a quantum register associated with a finite set of distinct quantum variables. We denote the state space of \( q \) by \( \mathcal{H}_q \) which is a two-dimensional Hilbert space spanned by the computational basis states \( \{ |0\rangle, |1\rangle \} \). The state space of \( \overline{q} \) is the tensor product of Hilbert spaces \( \mathcal{H}_{\overline{q}} = \otimes_{q \in \overline{q}} \mathcal{H}_q \).

Logical operations in QEC codes are often associated with changes in the set of stabilizer measurements. For example, the surface code [11] frequently turns on and turns off specific stabilizer measurement circuits to implement logical gates. Besides, the outcomes of stabilizer measurements act as signals for error correction. By introducing a stabilizer variable, which represents a stabilizer measurement circuit without the need of specifying its actual implementation, we can greatly simplify the description of QEC programs. We define the notations for the stabilizer variable as follows:

Define \( S \) as the set of stabilizers on qVar, \( s \) as an individual stabilizer in \( S \), sVar as the set of stabilizer variables, and \( f \) as a metavariable ranging over \( S \). To avoid \( S \) being uncountable, we assume that every \( s \in S \) only involves a finite number of qubits. The range of values for the stabilizer variable \( f \) is \( S \cup -S \cup iS \cup -iS \), where \( i \) is the imaginary unit.

We define the syntax of QECV-Lang as follows:

\[
\text{Prog} ::= \text{skip} \mid q := |0\rangle \mid \overline{q} := U[\overline{q}] \mid \text{Prog}_1; \text{Prog}_2 \\
\mid \text{case} \ M[\overline{q}] = m \rightarrow \text{Prog}_m \text{ end} \\
\mid \text{while} \ M[\overline{q}] = 1 \ do \ \text{Prog}_1 \ \text{done} \\
\mid \text{if} \ M[f, \overline{q}] \ \text{then} \ \text{Prog}_1 \ \text{else} \ \text{Prog}_0 \ \text{end} \\
\mid s^u \ ::= \pm s \mid \pm i s \mid \pm f
\]

The proposed language constructs consisting of instructions as follows: (1) \( \text{skip} \) does nothing; (2) \( q := |0\rangle \) resets quantum variable \( q \) to ground state \( |0\rangle \); (3) \( \overline{q} := U[\overline{q}] \) perform unitary operation \( U \) on quantum register \( \overline{q} \); (4) \( f := s^u \) assigns a unary stabilizer expression \( s^u \) to the stabilizer variable \( f \); (5) \( \text{Prog}_1; \text{Prog}_2 \) is the sequencing of programs; (6) if \( M[f, \overline{q}] \) then \( \text{Prog}_1 \) else \( \text{Prog}_0 \) end perform the stabilizer measurement represented by \( f \) on qubits \( \overline{q} \) (or in short, \( M[f, \overline{q}] \) measures qubits \( \overline{q} \) with the stabilizer \( f \) and executes program \( \text{Prog}_1 \) if the measurement outcome is 1. If the measurement outcome is 0, then \( \text{Prog}_0 \)). We can see (6) as a short-term for if \( M[f, \overline{q}] = 1 \) then \( \text{Prog}_1 \) else \( f = -f \); \( \text{Prog}_0 \) end. (7) \( \text{while} \ M[f, \overline{q}] \ do \ \text{Prog}_1 \ \text{done} \) measures \( f \) on qubits \( \overline{q} \), and perform \( \text{Prog}_1 \) if the measurement outcome is 1. If the measurement out-
come is -1, the sign of \( f \) will be flipped automatically, and then the while loop terminates.

The language constructs above are similar to those of the quantum while-language, except the part associated with stabilizer variables. Stabilizer variables can be used to describe operations on stabilizers. For example, to turn off one stabilizer measurement circuit in a QEC program, we can simply set \( f = \mathbb{I} \). The stabilizer variable can also serve to inform the error correction procedure. Every time we detect one or more stabilizer variables with a negative sign (this may happen after a stabilizer measurement), the decoder knows that at least one error affected the physical circuit. It proceeds to identifying the specific error and applies the corresponding correction. We define the error correction protocol as a function over stabilizer variables as follows:

**Definition 3.1 (Error correction protocol).** Define the function \( \text{correct}(f_0, f_1, \cdots) \) as an error decoding and correction protocol by measuring \( f_0, f_1, \cdots \).

As an example, we present a snippet of QECV-Lang code that corresponds to one stabilizer measurement of the surface code, where the parity qubit \( s \) is connected to four data qubits \( \{q_0, q_1, q_2, q_3\} \) with one Z-type stabilizer. Figure 1(a) shows the target stabilizer measurement circuit, while panel (b) shows an error correction program based on the stabilizer measurement in (a). The basic idea of the program in Figure 1(b) is that, if the state of \( q_0q_1q_2q_3 \) is not stabilized by the stabilizer variable \( f_0 \), then some errors have happened and the QEC program should correct them.

Comparing to the quantum while-language, QECV-Lang avoids the explicit introduction of parity qubits and the implementation of stabilizer measurements. Instead, it simply provides the stabilizer measurement circuits. The latter would, for example, depend on architectural properties like the underlying hardware connectivity.

While being optimized for QEC codes, QECV-Lang contains all the program constructs necessary to describe general quantum programs. Actually, all programs in quantum while-language can be translated into QECV-Lang by setting the stabilizer variable to a Pauli Z operator when qubits need to be measured.

### 3.2 Operational Semantics

The operational semantics of the proposed QECV-Lang are presented in Figure 2. Different from the quantum while-language, QECV-Lang denotes the state in QEC programs by the tuple \( (\rho, \sigma) \), where \( \rho \) is a partial density matrix that describes the current state of \( q \), and \( \sigma \) represents the current state of stabilizer variables. The quantum state \( \rho \) can be regarded as a function over quantum variables \( q \), and \( \rho(q) \) represents the reduced partial density matrix where quantum variables except \( q \) are all traced out. Likewise, \( \sigma \) represents a function over stabilizer variables, and \( \sigma(f) \) is defined to be the current value of \( f \). The stabilizer state \( \sigma \) is functionally similar to the classical program state \( [q] \), and we can define the substitution rule for \( \sigma \) in a similar way, which is then used in Figure 2.

**Definition 3.2 (Substitution rule).** The substitution rules for stabilizer state \( \sigma \) are defined as follows:

\[
\sigma[s/f_i](f_j) = \begin{cases} 
  s, & \text{if } i = j; \\
  \sigma(f_j), & \text{otherwise}
\end{cases}
\]

Rules in Figure 2 are self-explained and represent reformulation of concepts familiar in quantum computing. The notation in these rules follows the convention in programming language research, for example, the expressions over the bar in the inference rules are premises while the expression under the bar is conclusion. For pure quantum state operations in Figure 2, the operational semantics follows the flow in quantum while-language. For stabilizer related operations, we introduce extra operational semantics for the unary stabilizer expression \( s^- \), as shown in the top right corner of Figure 2. We then process the assignment operation on stabilizer variables with the substitution rule. When measuring the stabilizer variable \( f \), its sign will get flipped if the current quantum state is not a +1 eigenstate of \( f \). Thus, we include one operation in the “If -1” and “While -1” rule to take care of the sign flipping on \( f \).

To illustrate the use of the operational semantics in Figure 2, we revisit the program in Figure 1(b). We use \( (0, \{\}) \) to represent the initial state of \( (\rho, \sigma) \) before the program.

**Example 3.3 (Error correction experiment).** Assume the initialization \( q_3q_2q_1q_0 = |0000\rangle \) on data qubits is distorted by noise and data qubits are assigned to be \( |0001\rangle \). This may happen when, after the initialization in the logical subspace, a Pauli X error affects one of the qubits. For illustration purposes we consider that such error was on qubit \( q_0 \) and therefore use \( q_0 := X_{q_0} \) to correct the error. Notice that, in general, the error correcting protocol \( \text{correct}(f_0, \cdots) \) depends on the outcome of multiple stabilizer measurements. The program in Figure 2(b) then becomes

\[
\text{Prog} \equiv q_3q_2q_1q_0 := |0001\rangle; \quad f_0 := Z_{0123}; \\
\text{if } M[f_0, q_3q_2q_1q_0] \text{ then skip else } q_0 := X_{q_0}; \quad f_0 := :f_0 \text{ end.}
\]

We write \( Z_0Z_1Z_2Z_3 \) as \( Z_{0123} \) for simplicity. Then the evaluation of \( \text{Prog} \) with the operational semantics proceeds as follows:

\[
\langle \text{Prog}, \rho \rangle = q_3q_2q_1q_0 = |1110\rangle; \quad f_0 := Z_{0123}; \quad \text{if } M[f_0, q_3q_2q_1q_0] \text{ then skip else } q_0 := X_{q_0}; \quad f_0 := :f_0 \text{ end.} (0, \{\}) \Rightarrow (f_0 := Z_{0123}; \quad \text{if } M[f_0, q_3q_2q_1q_0] \text{ then skip else } q_0 := X_{q_0}; \quad f_0 := :f_0 \text{ end.} (0001), (0001), \{\})
\]
| (Skip)        | \(\text{skip}, (\rho, \sigma) \rightarrow (E, (\rho, \sigma))\) | (Stabilizer exp)       | \(c \cdot s, \sigma \rightarrow c \cdot s\) \(\pm f, \sigma \rightarrow \pm \sigma(f)\), \(c \in \{1, -1, i, -i\}\) |
|--------------|-------------------------------------------------|------------------------|----------------------------------------------------------------|
| (Initialization) | \(q := |0\rangle, (\rho, \sigma) \rightarrow (E, (\rho_0, \sigma))\) | (Assignment)           | \(\{f := s, (\rho, \sigma) \rightarrow (E, (\rho, \sigma[s/f]))\}\) |
| (Unitary)    | \(q := U|q\rangle, (\rho, \sigma) \rightarrow (E, (U\rho U^1, \sigma))\) | (Sequence)             | \(\langle \text{Prog}_2 ; \text{Prog}_1 \rangle \rightarrow \langle \text{Prog}_2' ; \text{Prog}_1 \rangle\) |
| (Sequence E) | \(E; \text{Prog}_2, (\rho, \sigma) \rightarrow (\text{Prog}_2' ; \text{Prog}_1, (\rho, \sigma))\) |                       | \(\langle \text{Prog}_1 ; \text{Prog}_2 \rangle \rightarrow \langle \text{Prog}_1 ; \text{Prog}_2, (\rho', \sigma')\rangle\) |
| (If 1)       | \(\text{if \[M[f, q]\] then \text{Prog}_1 \text{ else \text{Prog}_0 \text{ end}, (\rho, \sigma)} \rightarrow \langle \text{Prog}_0, (M_0\rho M_1^1, \sigma[-\sigma(f)/f])\rangle, M_0 = \frac{I - f}{2}\) |                       |                                                                     |
| (While 1)    | \(\text{while \[M[f, q]\] do \text{Prog}_1 \text{ done}, (\rho, \sigma)} \rightarrow \langle E, (M_0\rho M_1^1, \sigma[-\sigma(f)/f])\rangle\) |                       |                                                                     |
| (While 1)    | \(\text{while \[M[f, q]\] do \text{Prog}_1 \text{ done}, (\rho, \sigma)} \rightarrow \langle \text{Prog}_1 ; \text{while \[M[f, q]\] do \text{Prog}_1 \text{ done}, (M_1\rho M_1^1, \sigma)}\rangle\) |                       |                                                                     |

Figure 2: Operational semantics of QECV-Lang. Notations: \(\rho_0 = |0\rangle q\langle 0| + |0\rangle q\langle 1|\rho|0\rangle q\langle 0|\). \(\sigma(f)\) means to evaluate \(f\) in state \(\sigma\), while \(\sigma[s/f]\) means to replace the evaluation of \(f\) with \(s\) in the context \(\sigma\). Recall that \(E\) indicates the empty program.

\[
\begin{align*}
\text{[skip]}(\rho, \sigma) &= (\rho, \sigma) \\
\text{[q := |0\rangle]}(\rho, \sigma) &= (\rho_0, \sigma) \\
\text{[\(q := U|q\rangle\)]} &= (U\rho U^1, \sigma) \\
\text{[if \[M[f, q]\] then \text{Prog}_1 \text{ else \text{Prog}_0 \text{ end}]}(\rho, \sigma) &= \text{[Prog}_1 \{M_1\rho M_1^1, \sigma\} + \text{[Prog}_0 \{M_0\rho M_1^1, \sigma[-\sigma(f)/f]\}\} \\
\text{[while \[M[q]\] do \text{Prog}_1 \text{ done}]}(\rho, \sigma) &= \bigcup_{k=0}^{\infty} \text{[while}^k\text{]}(\rho, \sigma)
\end{align*}
\]

Figure 3: Denotational semantics of QECV-Lang. Notations: \(\text{while}^k\) is the \(k\)-th unrolling of \(\text{while}\).
ially by structural induction. For the while loop, the consistency
-proof set.

\[ \sigma \]

be compared if they are both not identity. We can then define the
-state be

\( O \)

the stabilizer set

\( S \)

state be

corrects errors when the errors appear, we have

\[ \Box[\psi] \]

(Equivalence of the denotational semantics and
-Proposition 3.4

corrected in each loop, the resulted state of
-state be

the bottom element of the CPO

\( [28] \), which does not provide any

information from stabilizer measurement into predicates. Consider a trivial lattice on the stabilizer set

\( S \), where

\( s_i \subseteq s_j \) if

\( s_i = I \), and

\( s_i \) and

\( s_j \) cannot be compared if they are both not identity. We can then define the partial order on
-state be

\( \sigma \) as follows:

\[ \sigma_1 \sqsubseteq \sigma_2 \text{ if } \sigma_1(f) \sqsubseteq \sigma_2(f), \forall f \in s\text{Var}, \]

which immediately induces a partial order on
-state be

\( \rho, \sigma \):

\[ (\rho_1, \sigma_1) \sqsubseteq (\rho_2, \sigma_2) \text{ if } \rho_1 \sqsubseteq \rho_2 \text{ and } \sigma_1 \sqsubseteq \sigma_2, \]

where

\( \rho_1 \sqsubseteq \rho_2 \) means that
-state be

\( T \rho_1 (O \rho_1 \rho_2) \leq T \rho_2 (O \rho_2), \) for any semi-

positive observable
-state be

The reason to select the partial order above is that it strictly ensures the consistency of fault-tolerant computation. If any error happens in one while loop and does not get corrected in time, the stabilizer state
-state be

will get some variables flipped. Let the resulted stabilizer state
-state be

\( \sigma' \). Obviously, there does not exist any
-state be

\( \sigma_1 \) s.t. \( \sigma \sqsubseteq \sigma_1 \) and
-state be

\( \sigma' \sqsubseteq \sigma_1 \). In this case, we just set
-state be

\( \cup \infty_{k=0} [\text{while}^{(k)}] (\rho, \sigma) \) to be \( \bot \),
-state be

the bottom element of the CPO
-state be

\( [28] \), which does not provide any

information for the program. On the other hand, if all errors get corrected in each loop, the resulted state of
-state be

while
-state be

can be calculated coordinate-wisely:

\[ (\cup \infty_{k=0} [\text{while}^{(k)}]) \rho, \cup \infty_{k=0} [\text{while}^{(k)}] \sigma \]

We connect the denotational semantics to the operational semantics through the following proposition:

**Proposition 3.4** (Equivalence of the denotational semantics and
-state be

the operational semantics). For a strict QEC program
-state be

\( \text{Prog} \) that corrects errors when the errors appear, we have
-state be

\[ \text{Prog} \{ (\rho, \sigma) \} \equiv \sum \{ (\rho', \sigma') : \text{Prog}(\rho, \sigma) \xrightarrow{s} (E, (\rho', \sigma')) \}, \]

where \( \xrightarrow{s} \) denotes the
-state be

reflective, transitive closure of \( \xrightarrow{s} \), and \( [\cdot] \) represents a multi-
-set.

**Proof.** Except the while loop, other statements can be proved triv-
ially by structural induction. For the while loop, the consistency can be proved with the assumption of a just-in-time error correc-
-tion. □

4 QECV-Assn

In this section, we first introduce an expressive assertion language
-state be

QECV-Assn and then derive a Hoare logic to verify QEC programs.

4.1 Syntax of QECV-Assn

Stabilizer is a kind of Hermitian operator and can be used as pre-
-dicate for QEC programs. We observe that the exponential computa-
tional overhead on Hermitian-based predicates may be circum-
vented by using stabilizers as predicates. In fact arithmetic oper-
-ations (e.g. addition and multiplication) between stabilizers can be completed within a time polynomial in the number of qubits.

This observation is particularly important for QEC programs in
-state be

which the majority of logical operations can be described with a
-few stabilizers and the corresponding predicate transformation can be framed as the multiplication of stabilizers.

However, as predicates, stabilizers are not universal. There are
-few stabilizers and the corresponding predicate transformation can be framed as the multiplication of stabilizers.

This observation is particularly important for QEC programs in
-state be

which the majority of logical operations can be described with a
-few stabilizers and the corresponding predicate transformation can be framed as the multiplication of stabilizers.

The program
-state be

\( \text{Prog} \)

is denoted as a super-operator
-state be

\( [\text{Prog}] \)

of a
-state be

\( \rho, \sigma \)

is given in Figure 3.

The denotational semantics of the
-state be

QECV-Lang is given in Figure 3. The program
-state be

\( \text{Prog} \)

is denoted as a super-operator
-state be

\( [\text{Prog}] \)

which acts on
-state be

\( (\rho, \sigma) \). While most rules in Figure 3 are self-explained, the
-state be

while
-state be

rule relies a partial order on
-state be

\( (\rho, \sigma) \) to compute the fixed point, i.e.,
-state be

the lowest upper bound
-state be

\( (\sqcap) \) for the complete partial ordering
-state be

(CPO)
-state be

of
-state be

\( \text{while}^{(k)} \).

To define the partial order on state
-state be

\( (\rho, \sigma) \), we first need to define a
-state be

partial order on the state of stabilizers. Consider a trivial lattice on the stabilizer set
-state be

\( S \), where
-state be

\( s_i \subseteq s_j \) if
-state be

\( s_i = I \), and
-state be

\( s_i \) and
-state be

\( s_j \) cannot be compared if they are both not identity. We can then define the partial order on
-state be

\( \sigma \) as follows:
-state be

\[ \sigma_1 \sqsubseteq \sigma_2 \text{ if } \sigma_1(f) \sqsubseteq \sigma_2(f), \forall f \in s\text{Var}, \]

which immediately induces a partial order on
-state be

\( \rho, \sigma \):
-state be

\[ (\rho_1, \sigma_1) \sqsubseteq (\rho_2, \sigma_2) \text{ if } \rho_1 \sqsubseteq \rho_2 \text{ and } \sigma_1 \sqsubseteq \sigma_2, \]

where
-state be

\( \rho_1 \sqsubseteq \rho_2 \) means that
-state be

\( T \rho_1 (O \rho_1 \rho_2) \leq T \rho_2 (O \rho_2), \) for any semi-
-positive observable.

The reason to select the partial order above is that it strictly ensures the consistency of fault-tolerant computation. If any error happens in one while loop and does not get corrected in time, the stabilizer state
-state be

will get some variables flipped. Let the resulted stabilizer state
-state be

\( \sigma' \). Obviously, there does not exist any
-state be

\( \sigma_1 \) s.t. \( \sigma \sqsubseteq \sigma_1 \) and
-state be

\( \sigma' \sqsubseteq \sigma_1 \). In this case, we just set
-state be

\( \cup \infty_{k=0} [\text{while}^{(k)}] (\rho, \sigma) \) to be \( \bot \),
-state be

the bottom element of the CPO
-state be

\( [28] \), which does not provide any

information for the program. On the other hand, if all errors get corrected in each loop, the resulted state of
-state be

while
-state be

can be calculated coordinate-wisely:

\[ (\cup \infty_{k=0} [\text{while}^{(k)}]) \rho, (\cup \infty_{k=0} [\text{while}^{(k)}]) \sigma \]

We connect the denotational semantics to the operational semantics through the following proposition:

**Proposition 3.4** (Equivalence of the denotational semantics and
-state be

the operational semantics). For a strict QEC program
-state be

\( \text{Prog} \) that corrects errors when the errors appear, we have
-state be

\[ \text{Prog} \{ (\rho, \sigma) \} \equiv \sum \{ (\rho', \sigma') : \text{Prog}(\rho, \sigma) \xrightarrow{s} (E, (\rho', \sigma')) \}, \]

where \( \xrightarrow{s} \) denotes the
-state be

reflective, transitive closure of \( \xrightarrow{s} \), and \( [\cdot] \) represents a multi-
-set.

**Proof.** Except the while loop, other statements can be proved triv-
ially by structural induction. For the while loop, the consistency can be proved with the assumption of a just-in-time error correc-
tion. □
\{A\}skip\{A\}
\{A[0]/q]\}q := \{0\}\{A\}
\{A\}q := Uq\{UA\U\}

U is a unitary, but written in the sum of stabilizers.
\{A\}f := \pm f\{A\}
\{A\}f := \langle f\{A\}\rangle

where \(s\) is commutable with \(A\), otherwise \(\{A\}f := s\{I\}\).

\{A\}Prog_1\{C\} \{C\}Prog_2\{B\}
\{A_1 \land \neg f\}Prog_1\{B\} \{A_0 \land -f\}Prog_0\{B\}

\{\sum_{i=0}^{1} A_i M_i\} if M[j, \bar{q}] then Prog_1 else Prog_0 end \{B\}

\{\sum_{i=0}^{1} A_i M_i\} while M[j, \bar{q}] do Prog_1 end \{A_0 \land -f\}

\{A \Rightarrow A'\} \{A'\}Prog\{B'\} = (B' \Rightarrow B)

\{A\}Prog\{B\}

Figure 4: Hoare rules for partial correctness assertions when \(A := s_e\).

stabilizer variables in \(A\) is essential for developing the quantum Hoare logic in section 4.2.

The semantics of \(A_0 \land A_1\) and other Boolean expressions can then be derived by structural induction:

- \((\rho, \sigma) \triangleright A_1 \land A_2 \iff (\rho, \sigma) \triangleright A_1\) and \((\rho, \sigma) \triangleright A_2\);
- \((\rho, \sigma) \triangleright A_1 \lor A_2 \iff (\rho, \sigma) \triangleright A_1\) or \((\rho, \sigma) \triangleright A_2\);
- \((\rho, \sigma) \triangleright (A_1 \Rightarrow A_2) \iff ((\rho, \sigma) \triangleright A_1) \Rightarrow ((\rho, \sigma) \triangleright A_2)\).

If an assertion \(A\) is satisfied by any program states \((\rho, \sigma)\), we simply denote such property as \(\triangleright A\).

The following lemma presents an important result for QECV-Assn that will be frequently utilized in later sections.

**Lemma 4.3** (Implication rule). For stabilizer expressions,

1. If \((\rho, \sigma) \triangleright s_{e_0}\) and \((\rho, \sigma) \triangleright s_{e_1}\), we have \((\rho, \sigma) \triangleright s_{e_0} \land s_{e_1}\) and \((\rho, \sigma) \triangleright \lambda_0 s_{e_0} + \lambda_1 s_{e_1}\), where \(\lambda_0 + \lambda_1 = 1\).
2. Assume \(s_{e_0}\) is not singular. If \((\rho, \sigma) \triangleright s_{e_0}\) and \((\rho, \sigma) \triangleright s_{e_1} s_{e_0}\), we have \((\rho, \sigma) \triangleright s_{e_1}\).
3. Assume \(as_{e_0} + bs_{e_1}\) is a stabilizer in \(A\) is commutable with \(s_{e_0}\) and \(s_{e_1}\), and \((\rho, \sigma) \triangleright s_{e_2}\), then \((\rho, \sigma) \triangleright as_{e_0} + bs_{e_1} s_{e_2}\).

**Proof.** Details are postponed to Appendix A.1

Rules from classical Boolean predicates can also be used for QECV-Assn, such as the rules for disjunction and conjunction. We will use these rules directly without extra description. Especially, the identity operator \(I\) represents \textbf{True} and the empty operator \(0\) represents \textbf{False} in QECV-Assn.

### 4.2 Partial Correctness

A partial correctness assertion in QECV-Assn has the form: \{A\}c\{B\}, where \(A, B \in \text{QECV-Assn}\), and \(c \in \text{QECV-Lang}\). We first present the Hoare logic for partial correctness assertions in which the precondition \(A\) is a stabilizer expression \(s_e\), as shown in Figure 4. We will extend the Hoare logic to Boolean expressions like \(A_1 \land A_2\) in Proposition 4.7.

The proof rules in Figure 4 are syntax-directed and reduce proving a partial correctness assertion of a compound statement to proving partial correctness assertions of its sub-statements. We only explain some rules below, since most rules are self-explained.

In the initialization rule, \(A_0 \land M_0 = 0\) means that \(A_0^\rho_0 = \rho_0^\rho\) and \(A_0\) commutes with all stabilizer variables in \(\sigma\). This can be seen as the quantum version substitution rule. A more useful case of the initialization rule is when all qubits are reset to \(0\), and for the \(n\)-qubit system, we have

\[
\{I\}q_{n-1} \cdots q_0 := 0^n \{Z_0 \land Z_1 \land \cdots \land Z_n\}. \tag{3}
\]

In the unitary rule, we represent unitary matrices as the sum of stabilizers in order to utilize the cheap computational cost of stabilizer multiplication.

The rules for condition and while loop resemble their classical counterparts except the state may be changed by the branching condition. A direct derivative of the Condition rule is to make \(A_1 = A_0 = A\) as follows,

**Lemma 4.4.** \(\{A \land \neg f\} \text{Prog}_1\{B\} \{A \land \neg f\} \text{Prog}_0\{B\}\)

**Proof.** Note that \(M_0 + M_1 = I\).

Likewise, by letting \(A_1 = A_0 = A\), we have

**Lemma 4.5.** \(\{A \land \neg f\} \text{Prog}_1\{A\}\)

The consequence rule is a powerful tool for the verification of QEC programs since it can encode facts of QEC codes into partial correctness assertions. The following example demonstrates the usage of the proposed Hoare rules, including the consequence rule:

**Example 4.6.** Assume \(\text{Prog} := f := Z_1; if M[j, q_1] then skipelse q_1 := X q_1; q_0 := X q_0 end\).

We prove \(\{Z_0 Z_1\} \text{Prog}\{Z_0\}\) as follows:

- \(\{Z_0 Z_1\} f := Z_1; \{Z_0 Z_1\}\)
- \(\{Z_0 Z_1\} Z_0 Z_1 = (Z_0 Z_1) \frac{I + Z_0}{2} = Z_0 + \frac{Z_1}{2}, (Z_0 Z_1) M_0 = -Z_0 \frac{I - Z_0}{2}\)
- \(\{Z_0 Z_1 \land \neg Z_0\} q_1 := X q_1; \{Z_0 Z_1 \land \neg Z_0\}\)
- \(\{Z_0 Z_1 \land \neg Z_0\} q_0 := X q_0; \{Z_0 Z_1 \land Z_0\}\)
- \(\{Z_0 Z_1 \land Z_0\} \text{skip} \{Z_0 Z_1 \land Z_0\}\)
- \(\{Z_0 Z_1 \land Z_0\} f := Z_1; \{Z_0 Z_1 \land Z_0\}\)
- \(\{Z_0 Z_1 \land Z_0\} q_1 := X q_1; q_0 := X q_0; \{Z_0 Z_1 \land Z_0\}\)
- \(\{Z_0 Z_1 \land Z_0\} \text{seq} \{Z_0 Z_1 \land Z_0\}\)

With the consequence rule, we have \(\{Z_0 Z_1\} \text{Prog}\{Z_0\}\). Now we extend the Hoare rules in Figure 4 to other Boolean assertions in QECV-Assn.
**Proposition 4.7.** We restate the Hoare rules for classical Boolean assertions as follows.

If \( A_0 \) \( \text{Prog} \{ B_0 \} \land \{ A_1 \} \\{ \text{Prog} \{ B_1 \} \}, \{ A_0 \land A_1 \} \\{ \text{Prog} \{ B_0 \land B_1 \} \}; \)

If \( A_0 \) \( \text{Prog} \{ B_0 \} \lor \{ A_1 \} \\{ \text{Prog} \{ B_1 \} \}, \{ A_0 \lor A_1 \} \\{ \text{Prog} \{ B_0 \lor B_1 \} \}; \)

\( \{ \text{I} \}\{ \text{Prog} \{ I \} \}, \{ 0 \}\{ \text{Prog} \{ B \} \} \), where \( B \) is any assertion, and \( 0 \) represents an empty set of program states. For example, if \( s_{e_1} \) and \( s_{e_2} \) anti-commutes, \( s_{e_1} \land s_{e_2} = 0 \).

**Proof.** Details are postponed to Appendix [A.1] □

We prove a lemma for error correction which is frequently used in verification sections later.

**Proposition 4.8** (Decoding correctness). Assume an invalid error decoding and correction protocol for \text{correct} function. Let \( S \) be the set of all active stabilizer measurements in error correction, and define \( A_S = \land_{s_i \in S} a_i \), then

\[ \{ I \}\text{correct}(f_0, f_1, \cdots) \{ A_S \}, \]

\[ \{ A \land A_S \}\text{correct}(f_0, f_1, \cdots) \{ A \land A_S \} \],

where \( f_0, f_1, \cdots \) enumerate all elements of \( S \).

**Proof.** Details are postponed to Appendix [A.1] □

Finally, we prove the soundness of Hoare rules in Figure 4.

**Theorem 4.9** (Soundness). The proof system in Figure 4 is sound for the partial correctness assertions.

**Proof.** Details are postponed to Appendix [A.1] □

## 5 Theoretical Analysis

In this section, we give a theoretical analysis of both the program size and the computational complexity of our framework for implementing and verifying surface codes [1, 30], respectively.

### 5.1 Program Size

We first compare the program size (i.e., the number of statements) when implementing the surface code [1, 30], in the qWhile-Lang (i.e., the quantum while-language) and the QECV-Lang (see row 2-3 of Table 2). In surface code, we consider two approaches to encode a logical qubit, the planar code and the double defect code (detailed implementation of these codes can be found in [1, 30]). For the distance-\( d \) surface code, the planar version requires \( O(d^2) \) data qubits, \( O(d^3) \) parity qubits as well as \( O(d^2) \) stabilizers. The double-defect version introduces an overhead of a factor 10 in all the three quantities.

As a code size estimation of QECV-Lang, we only need one statement per stabilizer measurement, whereas the qWhile-Lang requires at least eight gate operations to describe the circuit measuring a stabilizer [1]. Thus, QECV-Lang provides \( 8 \times \) program size compression for the surface code implementations.

### 5.2 Verification Complexity of Clifford Gates

The defining property of Clifford operations is that, given a Clifford gate \( G \) and a stabilizer \( s \), \( GsG^\dagger \) must also be a stabilizer, i.e., Clifford operations do not increase the number of stabilizers in the assertion.

By framing both assertions and unitary operations in the language of stabilizers, QECV can efficiently processes the verification of Clifford operations. The efficiency stems from the low cost of multiplying stabilizers, which is \( O(d) \) because the length of the stabilizers for logical states is at most \( d \) for a distance-\( d \) surface code. In this way, we avoid representing stabilizers as exponentially-large matrices. Therefore, QECV only incurs \( O(d^3) \) computational overhead for the planar surface code and \( O(10d^3) \) computational overhead for the double-defect surface code.

However, the vanilla quantum Hoare logic in qWhile-Lang cannot exploit the property of Clifford operations and the low computational complexity of stabilizer multiplication. The Clifford operations are treated like any other unitary operations and the predicate in qWhile-Lang is a Hermitian matrix of size \( O(2d^2d^2) \), whereas \( n_d \) is the number of data qubits and \( n_p \) is the number of parity qubits. Hoare rules with such predicates incurs at least \( O(2d^2d^2 \times 2d^2d^2) \) computational overhead. Thus, the verification with qWhile-Lang requires \( O(8d^2d^2d^2) \) time for the planar surface code, and \( O(8d^2d^2d^2) \) for the double-defect surface code.

In summary, the proposed language design, assertion design, and the logic proof system can significantly simplify the verification of all Clifford operations of stabilizer codes.

| Metric         | Method   | Planar surface code | Double-defect surface code |
|----------------|----------|---------------------|----------------------------|
| Statements #   | qWhile-Lang | \( O(8d^2d^2) \)  | \( O(8d^2d^2) \)          |
| Verification   | QECV     | \( O(d^2) \)       | \( O(10d^2) \)            |
| Complexity     | QECV     | \( O(8d^2d^2) \)   | \( O(8d^2d^2d^2) \)       |

Table 2: Comparison of QECV and qWhile-Lang on implementing and verifying surface codes.

### 5.3 Verification Complexity of T Gate

The logical T gate is usually the most challenging problem in quantum program verification in general. The T gate, loosely speaking, represents a fundamental boundary between classical and quantum computing. A quantum program with T gates cannot be efficiently and precisely simulated or verified on a classical computer. In QECV, the number of stabilizers in our predicate will rapidly increase when the program to be verified has some T gates.

With this being said, we argue that verifying a QEC implementation of one logical T gate could be easier in many cases. The exact verification efficiency would be determined by the amount of the non-Clifford operations involved in the implementation of a logical T gate. In the surface code, for example, there is only one non-Clifford single-qubit physical gate [31] for a logical T gate. The verification complexity will remain \( O(d^2) \) because the number of stabilizer terms in a predicate is still \( O(1) \). As such, QECV can still hold the exponential advantage for surface code. We remark again that such advantages come from our stabilizer-centric design in developing the verification framework.
6 Case Study I: Repetition Code

In this section and the next section, we give step-by-step case study on two well-known QEC codes to guide through the usage of our framework. For each QEC code, we first express its implementation in our QECV-Lang. Then we verify the correctness of the logical operation with our proof system and show that the implemented QEC code can correct local errors on the physical qubits.

We start from the repetition code. For the initialization operation, the expected behavior is that, for arbitrary input state, the output state should be in the logical state $|0_L\rangle$, which is the simultaneous eigenstate of the logical Z operator $Z_L$ and the stabilizers $Z_0Z_1$ and $Z_1Z_2$. Thus, we set the pre-condition to $\{I\}$ and the post-condition to $\{Z_L \land Z_0Z_1 \land Z_1Z_2\}$, as formulated below.

**Proposition 6.1** (Initialize to $|0\rangle$). **For the program Prog in Figure 5(a), we have $\{I\} \text{Prog} \{Z_L \land Z_0Z_1 \land Z_1Z_2\}$.

**Proof.** For the initialization, $\{I\} q_0q_1q_2 := |000\rangle \{Z_0 \land Z_1 \land Z_2\}$ and $\{Z_L \land Z_0Z_1 \land Z_1Z_2\} f_1 := Z_0Z_1Z_2 \{Z_0 \land Z_1 \land Z_2\}$ and $\{Z_0 \land Z_1 \land Z_2\} f_1 := Z_0Z_1Z_2 \{Z_0 \land Z_1 \land Z_2\}$. Since $Z_0 \land Z_1 \land Z_2 = Z_0Z_1Z_2$ and $Z_0 \land Z_1 \land Z_2 = Z_0Z_1Z_2$, we have $Z_0 \land Z_1 \land Z_2 \land Z_1Z_2$. Then by Proposition 4.8 $\{Z_L \land Z_0Z_1 \land Z_1Z_2\} \text{correct}(f_0, f_1) \{Z_0 \land Z_1 \land Z_2 \land (Z_0Z_1) \land (Z_1Z_2)\}$. By the consequence rule, we get $\{I\} \text{Prog} \{Z_L \land Z_0Z_1 \land Z_1Z_2\}$ since $Z_0 \land Z_1 \land Z_2 = Z_0Z_1Z_2$.

We then verify the logical X operation. It is sufficient to verify two cases, the output state $|1_L\rangle$ under the input state $|0_L\rangle$, and vice versa. Arbitrary logical states can be processed as the linear combination of these two cases by taking advantage of the linearity of the logical X operation. Since $|0_L\rangle$ corresponds to the predicate $Z_L \land Z_0Z_1 \land Z_1Z_2$, and $|1_L\rangle$ corresponds to the predicate $-Z_L \land Z_0Z_1 \land Z_1Z_2$, we have the following proposition:

**Proposition 6.2** (Logical X gate). **For the program Prog in Figure 5(b), we have $\{Z_L \land Z_0Z_1 \land Z_1Z_2\} \text{Prog} \{Z_L \land Z_0Z_1 \land Z_1Z_2\}$ and $\{-Z_L \land Z_0Z_1 \land Z_1Z_2\} \text{Prog} \{Z_L \land Z_0Z_1 \land Z_1Z_2\}$.

**Proof.** Note that $X_0X_1X_2Z_0X_0X_1X_2 = -Z_1Z_2Z_3 = -Z_L$, $X_0X_1X_2Z_0X_1X_2Z_2X_0X_1X_2 = Z_0Z_1$, $X_0X_1X_2Z_0X_1X_2Z_2X_0X_1X_2 = Z_1Z_2$.

Likewise, for the logical Z gate, we only need to verify that, the pre-condition $\{X_L \land Z_0Z_1 \land Z_1Z_2\}$ relates to the post-condition $\{X_L \land Z_0Z_1 \land Z_1Z_2\}$, and vice versa.

**Proposition 6.3** (Logical Z gate). **For the program Prog in Figure 5(c), $\{X_L \land Z_0Z_1 \land Z_1Z_2\} \text{Prog} \{X_L \land Z_0Z_1 \land Z_1Z_2\}$ and $\{-X_L \land Z_0Z_1 \land Z_1Z_2\} \text{Prog} \{X_L \land Z_0Z_1 \land Z_1Z_2\}$.

**Proof.** Note that $Z_0Z_1Z_2X_0Z_0Z_1Z_2 = -X_0X_1X_2 = -X_L$.

The verification of the logical CNOT gate involves four pre-conditions: $Z_{L0}I_{L1}$, $X_{L0}X_{L1}$, $I_{L0}I_{L1}$, and $I_{L0}Z_{L1}$, where $Z_0Z_1 \land Z_1Z_2 \land Z_2Z_4 \land Z_4Z_5$ are omitted for simplicity. These four Pauli strings are able to represent any input state by multiplication and addition. The post-conditions for these four pre-conditions are $Z_{L0}I_{L1}$, $X_{L0}X_{L1}$, $I_{L0}I_{L1}$, and $Z_{L0}Z_{L1}$. While the first three post-conditions are straightforward to understand, we elaborate on the fourth post-condition. The pre-condition $I_{L0}Z_{L1}$ specifies pure states of the form $(a|0\rangle + b|1\rangle) |L\rangle |0\rangle$.

**Proposition 6.4** (Logical CNOT). **For the program Prog in Figure 5(d), assume $A_S = Z_0Z_1 \land Z_2Z_3 \land Z_3Z_4 \land Z_4Z_5$, we have $\{Z_{L0}I_{L1} \land A_S\} \text{Prog} \{Z_{L0}I_{L1} \land A_S\}$, $\{X_{L0}I_{L1} \land A_S\} \text{Prog} \{X_{L0}X_{L1} \land A_S\}$, $\{I_{L0}I_{L1} \land A_S\} \text{Prog} \{I_{L0}X_{L1} \land A_S\}$, and $\{I_{L0}Z_{L1} \land A_S\} \text{Prog} \{I_{L0}Z_{L1} \land A_S\}$.

**Proof.** Note that for control qubit $a$ and target qubit $b$, $\text{CNOT}_{ab} = \frac{1}{2} (I + X_b + Z_a - X_bZ_a)$. Details are postponed to Appendix A.2.

6.3 Verification on Noise Injection

QECV can also reason about the correctness with hardware noise.

**Program 6.1** (Quantum repetition code error correction). For the quantum repetition code in Figure 5 define the error correction protocol as follows. For $f_0 = Z_0Z_1$, $f_1 = Z_1Z_2$.

**Correct**($f_0, f_1)$ :=

if $M[Z_0Z_1, q_0q_1]$ then
if $M[Z_1Z_2, q_1q_2]$ then skip
else $q_2 := X_q$; $f_1 := \neg f_1$ end
else if $M[Z_1Z_2, q_1q_2]$ then $q_0 := X_q$; $f_0 := \neg f_0$
else $q_1 := X_q$; $f_0 := \neg f_0$; $f_1 := \neg f_1$ end

end.
Proposition 6.6. For the program Prog in Figure 2(b), where a Z error happens on $q_1$, we have
\[ \{X_L \land Z_0 Z_1 \land Z_1 Z_2\} \text{Prog} \{\neg X_L \land Z_0 Z_1 \land Z_1 Z_2\} \text{ and } \{\neg X_L \land Z_0 Z_1 \land Z_1 Z_2\} \text{Prog} \{X_L \land Z_0 Z_1 \land Z_1 Z_2\}, \]
which is not the desired behaviour of the logical X gate.

Proof. Similar to the proof in Proposition 6.5.

7 Case Study II: Surface Code

In this section, we present the verification of the double-defect surface code [11]. There are two types of stabilizers in surface code, of which one is called ‘Z-type’ stabilizer as it only consists of Pauli Z operators and the other one is called ‘X-type’ stabilizer as it only contains Pauli X operators. These two types of stabilizers together enable high error tolerance of surface code as well as the ability to correct both bit-flip error and phase-flip error simultaneously. Implementing surface code has been pursued by several major quantum computing vendors including IBM [3] and Google [32].

7.1 Surface Code

Double-defect surface code includes a series of logical operations to support fault-tolerant quantum computation, such as initialization, measurement, defect enlarging, defect shrinking, logical single-qubit gates (X, Z, H), qubit moving, braiding and the logical CNOT gate. In this section, we only verify qubit initialization, qubit moving, logical Pauli gates and logical qubit braiding. The verification of remaining operations is similar to or can be built upon the verified operations. For example, qubit measurement is the reversing process of qubit initialization, and the logical CNOT gate is the concatenation of three braiding operations.

Without loss of generality, we only consider the distance-3 surface code. The logical operations shown in Figure 7 is implemented in QECV-Lang in Figure 8. We mainly focus on the operations on the X-cut qubit, which is a kind of logical qubit created by disabling X-type stabilizers. We present the initialization operation in Figure 7(a) which initializes a X-cut qubit to the logical state $|+_L\rangle$, i.e., the +1 eigenstate of the logical X operator $X_L$ in Figure 7(b).

Figure 7(b) shows the logical Z gate $Z_L$. We then implement the qubit moving operation and the logical H gate shown in Figure 7(c)(d).
The qubit moving operation will not change the logical state. For the logical H gate, we only presents a simplified version which is enough to demonstrate the core idea of the logical H gate in [Ref]. Finally, we include the verification of qubit initialization (to $|0_L\rangle$), the logical X gate and the braiding operation in Appendix [Ref.

All proofs in this section will be postponed to Appendix [Ref.]

7.2 Verification of Logic Operations

As specified by the surface code [Ref.], any valid logical state should always be in the +1 eigenspace of all active stabilizers on the surface code array, $S = \{s_0,s_1,\cdots\}$. For simulation, we omit the stabilizers that does not involve in proof. For example, $(|0_L\rangle,|0_L\rangle,|sigma|\rangle = Z_L \land s_0 \land s_1 \cdots$ will be denoted by $(|0_L\rangle,|0_L\rangle,|sigma|\rangle = Z_L$. In the cases where we need to stress other active stabilizers in the array, we will have $(|0_L\rangle,|0_L\rangle,|sigma|\rangle = Z_L \land A_S$, where $A_S = \land_{s \in SS}s$. We first verify the initialization to $|+L\rangle$. The expected functionality of the initialization operation is to prepare arbitrary state into a desired state, as shown in the following proposition. The precondition of the partial correctness is just $L$ which allows any state. As for the post-condition, notice that $|+L\rangle$ is stabilized by the stabilizer $X_0X_1X_2X_4$ (i.e., $L$) and other active stabilizers in $S$. The verification of initialization to $|0_L\rangle$ is similar and postponed to Appendix [Ref.]

Proposition 7.1 (Initialize $|+\rangle_L$). For the program $Prog$ in Figure 8(a) which initializes a X-cut logical qubit to $|+\rangle_L$, as shown in Figure 8(a), we have $\{I\}Prog\{X_0X_1X_2X_4 \land A_S\}$.

The verification of the logical Z gate is similar to that in quantum repetition code and the precondition and post-condition can be derived in a similar way. The verification of logical X gate is similar and postponed to Appendix [Ref.]

Proposition 7.2 (Logical Z gate). For program $Prog$ in Figure 8(b) which implements the logical Z gate in Figure 8(b), we have $\{X_L\}Prog\{-X_L\}$ and $\{-X_L\}Prog\{X_L\}$, where $X_L = X_0X_1X_2X_4$.

To reason about qubit moving, the key is to prove that the logical state is preserved. We first represent the current state of data qubits with the stabilizer language. The following lemma shows that there is a one-to-one mapping between the stabilizer expressions and the logical quantum states.

Lemma 7.3. For a X-cut qubit state $|\psi\rangle$, if $|\psi\rangle = a|0\rangle_L + b|1\rangle_L$ ($|a|^2 + |b|^2 = 1$), then there is a unique $(aZ_L + bX_L)|\psi\rangle$ such that $(aZ_L + bX_L)|\psi\rangle = |\psi\rangle$, and in this case $a = \frac{\sigma^2 - \beta^2}{\sigma^2 + \beta^2}$ and $b = \frac{2\alpha\beta}{\sigma^2 + \beta^2}$.

Conversely, for a X-cut qubit state $|\psi\rangle$, if $(\frac{\sigma^2 - \beta^2}{\sigma^2 + \beta^2}Z_L + \frac{2\alpha\beta}{\sigma^2 + \beta^2}X_L)|\psi\rangle$ is the space spanned by $\{0_L\}$, then $|\psi\rangle = a|0\rangle_L + b|1\rangle_L$, up to a local phase.

We then apply Lemma 7.2 to verify the vertical qubit moving operation in Figure 8(c). The verification of the horizontal qubit moving is similar. The following proposition confirms that the logical state is not changed since the precondition and the post-condition have equal coefficients w.r.t. the logical X and logical Z operators.

Proposition 7.4 (Vertical qubit moving). For program $Prog$ in Figure 8(c) which implements the qubit moving operation in Figure 7(c), we have $\{aZ_L + bX_L\}Prog\{aZ_L' + bX_L'\}$, where $Z_L = Z_0Z_1Z_2$, $X_L = X_2X_3X_4X_6$, $Z_L' = Z_0Z_1Z_2Z_6$, $X_L' = X_0X_2X_5X_10$, $a, b \in \mathbb{C}$.

While the implementation of the logical H gate requires isolating the defects of a logical qubit, the core of logical H operation is to perform local H gates for the area outlined in Figure 7(d), which alone is a distance-3 planar surface code. We will only verify that part of program for simplicity. Other parts of the logical H operation can be verified on top of verified operations above. Like in the case in quantum repetition code, we only need to verify that the logical X operator is transformed into the logical Z operator and the logical Z operator is transformed into the logical X operator.

Proposition 7.5 (Logical H gate). For the program $Prog$ in Figure 8(d) which implements the simplified logical H gate in Figure 8(d), we have $\{Z_L\}Prog\{X_L\}$, $\{X_L\}Prog\{Z_L\}$, where $Z_L = Z_1Z_6Z_11$, $X_L = X_5X_6X_7$, $Z_L' = Z_1Z_6Z_7$, $X_L' = X_1X_6X_11$ for the distance-3 planar surface code outlined in Figure 7(d).

7.3 Verification on Noise Injection

To reason the correctness when noise exists, we assume a minimal weight perfect matching (MWPM) decoder [Ref] and error correction for the surface code array. When the error only happens on one qubit, it can be easily detected by the decoder and be corrected, as indicated below:

Proposition 7.6. For the program $Prog$ in Figure 9(b) which implements a noisy version of the logical Z gate in Figure 7(b), $\{X_L \land A_S\}Prog\{-X_L \land A_S\}$, $\{-X_L \land A_S\}Prog\{X_L \land A_S\}$, where $X_L = X_0X_1X_2X_4$.

When we increase the Z error location by one, the error correction protocol may fail.

Proposition 7.7. For the program $Prog$ in Figure 9(b) $\{X_L \land A_S\}Prog\{X_L \land A_S\}$, $\{-X_L \land A_S\}Prog\{-X_L \land A_S\}$, which is not the desired behavior of the logical Z gate.

The proposition 7.7 is expected because a distance-3 surface code cannot correct errors on more than $\lfloor \frac{d}{2} \rfloor$ qubits. More complicated cases can be proved in a similar way with our verification framework.

8 Conclusion

Quantum error correction is the bedrock of fault-tolerant quantum computation, and its verification is of significant importance for the forthcoming large-scale quantum computing. In this paper, we propose QECV, an efficient verification framework for stabilizer codes. QECV first comes with a concise language, QECV-Lang, which incorporates stabilizers to represent QEC programs. Stabilizers together with stabilizer expressions are also used as predicates in our new assertion language QECV-Assn. We then derive a sound quantum Hoare logic to efficiently reason about the correctness of QEC programs. Finally, We evaluate QECV by a theoretical complexity analysis and case studies on two QEC codes. We believe this work will spark more interest in the verification of QEC programs which may become a prevalent programming paradigm in the near future.
QECV: A Verification Framework for Quantum Error Correction Codes

Figure 7: Primitive operations in the double-defect surface code. X stabilizers are yellow, and Z stabilizers are blue.

Figure 8: Implementation of primitive operations of the double-defect surface code in QECV-Lang.
\[ q_{49} q_{13} := Z_4 Z_9 Z_{13} q_4 q_{913} \]

// an Z error on qubit \( q_9 \):
\[ q_{49} q_{913} := Z_4 Z_9 Z_{13} q_4 q_{913} \]

// Z errors on qubit \( q_9, q_{13} \):
\[ q_9 := Z \bar{q}_9 \]
\[ q_{13} := Z_13 q_{13} \]

\( \text{correct}(f_0, f_1, \cdots) \)

(a) a Z error on \( q_9 \).

\( \text{correct}(f_0, f_1, \cdots) \)

(b) two Z errors on \( q_9, q_{13} \).

Figure 9: Noisy programs of the logical \( Z \) gate in Figure 7(b).

References

[1] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. Physical Review A, 86:032324, 2012.

[2] Christopher Chamberland, Kyungjoo Noh, Patricio Arrangoiz-Ariola, Earl T. Campbell, Connor T. Hann, Joseph Iverson, Harald Puttermann, Thomas C. Bohnadonwicz, Steven T. Flammia, Andrew J. Keller, Gil Refael, John Preskill, Liang Jiang, Amir H. Safavi-Naeini, Oskar J. Painter, and Fernando G. S. L. Brandão. Building a fault-tolerant quantum computer using concatenated cat codes. arXiv: Quantum Physics, 2020.

[3] Christopher Chamberland, Guanyu Zhu, Theodore J. Yoder, Jared B. Hertzberg, and Andrew W. Cross. Topological and subsystem codes on low-degree graphs with flag qubits. Physical Review X, 10, 2020.

[4] John Preskill. Quantum computing in the nisq era and beyond. Quantum, 2018.

[5] Adam Holmes, Mohammad Reza Jokar, Ghasem Pasandi, Yongshen Ding, Massoud Pedram, and Frederic T. Chong. Niq+: Boosting quantum computing power by approximating quantum error correction. 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pages 556–569, 2020.

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

[7] Adam Jozef Zalcman, Alan Derk, Alan Ho, Alex Opromcak, Alexander Korotkov, Alexandre Bourassa, Andre Gregory Petukhov, Andreas Bengtsson, Andrew Dunsforth, Anthony Meigrant, Austin Fowler, Bálint Pató, Benjamin Chiaro, Benjamin Villalonga, Brian Burket, Brooks Riley Foxen, Catherine Erickson, Charles Neill, Chris Quintana, Cody Jones, Craig Michael Gidney, Daniel Eppens, Daniel Sank, Dave Landhuis, David A Buell, Doug Strain, Dvir Kafri, Edward Farhi, Eric Ostby, Erik Lucero, Evan Jeffrey, Fedor Kostroitsa, Frank Carlton Arute, Hartmut Neven, Igor Aleiner, Jamie Yao, Jarrod Ryan McClean, Jeremy Patterson Hilton, Jimmy Chen, Jonathan Arthur Gross, Joseph Bardin, Josh Matus, Juan Atalaya, Julian Kelly, Kevin Miao, Kevin Satzinger, Kostyantyn Kechedzhi, Kunal Arya, Marco Szalay, Marissa Giustina, Massoud Mohseni, Matt McEwen, Matt Trevithick, Matthew Neeley, Matthew P Harrigan, Michael Broughton, Michael Newman, Murphy Yuezhen Niu, Nicholas Bushe-nell, Nicholas Redd, Nicholas Rubin, Ofer Naaman, Orion Martin, Paul Victor Klimov, Pavel Laptev, Pedram Roushan, Ping Yeh, Rami Barends, Roberto Collins, Ryan Babbush, Sabrina Hong, Sean Demura, Sean Harrington, Seon Kim, Sergei Isakov, Sergio Boixo, Ted White, Thomas E O’Brien, Trent Huang, Trevor Mccourt, Vadim Smelyanskiy, Vladimir Shvarts, William Courtney, Wojtek Mruczkiewicz, Xiao Mi, Yu Chen, and Zhang Jiang. Exponential suppression of bit or phase flip errors with repetitive quantum error correction. Nature, 2021.

[8] Christopher Chamberland, Guanyu Zhu, Theodore J. Yoder, Jared B. Hertzberg, and Andrew W. Cross. Topological and subsystem codes on low-degree graphs with flag qubits. Physical Review X, 10:011022, Jan 2020.

[9] Daniel Gottesman. Stabilizer codes and quantum error correction. arXiv: Quantum Physics, 1997.

[10] Lingling Lao and Carmen Garcia Almudever. Fault-tolerant quantum error correction on near-term quantum processors using flag and bridge qubits. Physical Review A, 101:032333, 2020.

[11] Rui Chao and Ben Reichardt. Flag fault-tolerant error correction for any stabilizer code. arXiv: Quantum Physics, 2019.

[12] Kyungjoo Noh and Christopher Chamberland. Fault-tolerant bosonic quantum error correction with the surface–gottesman-kitaev-preskill code. Physical Review A, 101:012316, 2020.

[13] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T. Chong. Full-state quantum circuit simulation by using data compression. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019.

[14] Li Zhou, Nengkun Yu, and Mingsheng Ying. An applied quantum hoare logic. Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019.

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

[16] Mingsheng Ying. Floyd–hoare logic for quantum programs. ACM Trans. Program. Lang. Syst., 33:19:1–19:49, 2012.

[17] Mingsheng Ying and Yangjia Li. Reasoning about parallel quantum programs. ArXiv, abs/1801.11334, 2018.

[18] Dominique Unruh. Quantum hoare logic with ghost variables. 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pages 1–13, 2019.

[19] Ellie D’Hondt and P. Panangaden. Quantum weakest preconditions. Mathematical Structures in Computer Science, 16:429 – 451, 2006.

[20] Peter Selinger. Towards a quantum programming language. Mathematical Structures in Computer Science, 14:527 – 586, 2004.

[21] Yuan Feng and Mingsheng Ying. Quantum hoare logic with classical variables. ArXiv, abs/2008.06812, 2020.
[22] Yuan Feng, Sanjiang Li, and Mingsheng Ying. Verification of distributed quantum programs. *ArXiv*, abs/2104.14796, 2021.

[23] Nengkun Yu and Jens Palsberg. Quantum abstract interpretation. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.

[24] Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. Static analysis of quantum programs via gottesman types. *ArXiv*, abs/2101.08939, 2021.

[25] Robert Rand, Kartik Singhal, Brad Lackey, and Microsoft. Extending gottesman types beyond the clifford group. In *arXiv*, 2021.

[26] Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. Gottesman types for quantum programs. *ArXiv*, abs/2109.02197, 2021.

[27] Garrett Birkhoff and John von Neumann. The logic of quantum mechanics. *Annals of Mathematics*, 37:1–26, 1936.

[28] Glynn Winskel. The formal semantics of programming languages - an introduction. In *Foundation of computing series*, 1993.

[29] Mark M. Wilde. Quantum information theory. In *Quantum Information Theory*, 2013.

[30] Clare Horsman, Austin G. Fowler, Simon J. Devitt, and Rodney Van Meter. Surface code quantum computing by lattice surgery. *New Journal of Physics*, 14:123011, 2012.

[31] Austin G. Fowler. Minimum weight perfect matching of fault-tolerant topological quantum error correction in average o(1) parallel time. *Quantum Inf. Comput.*, 15:145–158, 2015.

[32] Zijun Kevin J. Juan Alexander N. Andrew Daniel Chris Mat Chen Satzinger Atalaya Korotkov Dunsworth Sank Qui, Zijun Chen, Kevin J Satzinger, Juan Atalaya, Alexander N. Korotkov, Andrew Dunsworth, Daniel Thomas Sank, Chris Quintana, Matthew J. McEwen, Rami Barends, Paul Klimov, Sabrina Hong, Cody Jones, Andre Petukhov, Dvir Kafri, Sean Demura, Brian Burkett, Craig Gidney, Austin G. Fowler, Alexandru Paler, Harald Putterman, Igor L. Aleiner, Frank Arute, Kunal Arya, Ryan Babbush, Joseph C. Bardin, Andreas Bengtsson, Alexandre Bourassa, Mick Broughton, Bob B. Buckley, David A. Buell, Nicholas Bushnell, Benjamin Chiaro, Roberto Collins, William Courtney, Alan R. Derk, Daniel Eppens, Catherine Erickson, Edward Farhi, Brooks Foxen, Marissa Giustina, Ami Greene, Jonathan A. Gross, Matthew P. Harrigan, Sean D. Harrington, Jeremy P. Hilton, Alan Ho, Trent Huang, William J. Huggins, L B Ioffe, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Kostyantyn Kechedzhi, Seon Kim, Alexei Y. Kitaev, Fedor Kostritsa, David Landhuis, Pavel Laptev, Erik Lucero, Orion Martin, Jarrod R. McClean, Trevor McCourt, Xiao Mi, Kevin C. Miao, Masoud Mohseni, Shirin Montazeri, Wojciech Mruczkiewicz, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles J. Neill, Michael Newman, Murphy Yuezhen Niu, Thomas E. O’Brien, Alexander Opremcak, Eric P. Ostby, Bálint Pató, N Redd, Pedram Roushan, Nicholas C Rubin, Vladmir Shvarts, Doug Strain, Marco Szalay, Matthew D Trevithick, Benjamin Villalonga, Theodore White, Z. Jamie Yao, P Yeh, Juhwan Yoo, Adam Zalewski, Hartmut Neven, Sergio Boixo, Vadim N. Smelyanskiy, Yu Chen, Anthony Megrant, and Julian Kelly. Exponential suppression of bit or phase errors with cyclic error correction. *Nature*, 595:383 – 387, 2021.
A Appendix

A.1 Proof in QECV-Assn

Lemma 4.3 (Implication rule). For stabilizer expressions,

1. If $(\rho, \sigma) \models s_{e_0}$ and $(\rho, \sigma) \models s_{e_1}$, we have $(\rho, \sigma) \models s_{e_0}s_{e_1}$ and $(\rho, \sigma) \models \lambda_0 s_{e_0} + \lambda_1 s_{e_1}$, where $\lambda_0 + \lambda_1 = 1$.
2. Assume $s_{e_0}$ is not singular. If $(\rho, \sigma) \models s_{e_0}$ and $(\rho, \sigma) \models s_{e_0}s_{e_0}$, we have $(\rho, \sigma) \models s_{e_1}$.
3. Assume $(as_{e_0} + bs_{e_1})\rho = \rho$, every stabilizer in $s$ is commutative with $s_{e_0}$ and $s_{e_1}$, and $(\rho, \sigma) \models s_{e_2}$, then $(\rho, \sigma) \models as_{e_0} + bs_{e_1}s_{e_2}$.

Proof. For the first rule, note that $(s_{e_0}s_{e_1})\rho = s_{e_0}(s_{e_1}\rho) = s_{e_0}\rho = \rho$. Also, $\forall f \in \sigma$, $s_{e_0}s_{e_1}f = s_{e_0}f s_{e_1} = f s_{e_0} s_{e_1}$. Thus, $(\rho, \sigma) \models s_{e_0}s_{e_1}$.

For the second rule, note that $s_{e_1}\rho = s_{e_1}(s_{e_0}\rho) = (s_{e_1}s_{e_0})\rho = \rho$, and $\forall f \in \sigma$, $(f s_{e_1})s_{e_0} = s_{e_1} s_{e_0} f = (s_{e_1}f) s_{e_0}$. Since $s_{e_0}$ is not singular, we have $f s_{e_1} = s_{e_1}f$, thus $(\rho, \sigma) \models s_{e_1}$.

For the final rule, notice that $(as_{e_0} + bs_{e_1}s_{e_2})\rho = as_{e_0}\rho + bs_{e_1}s_{e_2}\rho = as_{e_0}\rho + bs_{e_1}\rho = \rho$.

Finally, it is easy to see in all these rules, the stabilizer in $s$ is commutative with the target stabilizer expressions.

Proposition 4.7. We restate the Hoare rules for classical Boolean assertions as follows,

if $\{ A_0 \} \textbf{Prog} \{ B_0 \} \land \{ A_1 \} \textbf{Prog} \{ B_1 \}$, $\{ A_0 \land A_1 \} \textbf{Prog} \{ B_0 \land B_1 \}$;
if $\{ A_0 \} \textbf{Prog} \{ B_0 \} \lor \{ A_1 \} \textbf{Prog} \{ B_1 \}$, $\{ A_0 \lor A_1 \} \textbf{Prog} \{ B_0 \lor B_1 \}$;

$\{ I \} \textbf{Prog} \{ I \}$, $\{ 0 \} \textbf{Prog} \{ B \}$, where $B$ is any assertion, and 0 represents an empty set of program states. For example, if $s_{e_1}$ and $s_{e_2}$ anti-commutes, $s_{e_1} \land s_{e_2} = 0$.

Proof. We first prove the conjunction rule. Since $A_0 \land A_1 \Rightarrow A_0$, $A_0 \land A_1 \Rightarrow A_1$, then by the consequence rule, we have $\{ A_0 \land A_1 \} \textbf{Prog} \{ B_0 \}$ and $\{ A_0 \land A_1 \} \textbf{Prog} \{ B_1 \}$, i.e., $\{ A_0 \land A_1 \} \textbf{Prog} \{ B_0 \land B_1 \}$. For the disjunction rule, notice that if $(\rho, \sigma) \models (A_0 \lor A_1)$, then either $(\rho, \sigma) \models A_0$ or $(\rho, \sigma) \models A_1$. Finally, $\{ I \} \textbf{Prog} \{ I \}$ always holds since any state $(\rho, \sigma)$ satisfies $I$. $\{ 0 \} \textbf{Prog} \{ B \}$ is true because $(\rho, \sigma) \models 0 \Rightarrow [P] (\rho, \sigma) \models B$.

Proposition 4.8 (Decoding correctness). Assume an valid error decoding and correction protocol for correct function. Let $S$ be the set of all active stabilizer measurements in error correction, and define $A_S \equiv \wedge_{s_i \in S} s_i$, then

$\{ I \} \textbf{correct} \{ f_0, f_1, \ldots \} \{ A_S \}$,
$\{ A \land A_S \} \textbf{correct} \{ f_0, f_1, \ldots \} \{ A \land A_S \}$,

where $f_0, f_1, \ldots$ enumerate all elements of $S$.

Proof. First, any valid correction function will project the state into one quiescent state of the QEC code. It’s the definition of QEC code error correction.

Second, note that the assertion $A \land A_S$ represents error-free states in the QEC code, thus any valid correct protocol will place a skip statement for correcting the error-free state. Assume the correct protocol is implemented based on the look-up table, since $A \land A_S \land \neg f_i = 0$, then by the condition rule and $\{ 0 \} \textbf{Prog} \{ A \land A_S \}$ (Lemma 4.7), we directly get $\{ A \land A_S \} \textbf{correct} \{ f_0, f_1, \ldots \} \{ A \land A_S \}$.

Theorem 4.9 (Soundness). The proof system in Figure 1 is sound for the partial correctness assertions.

Proof. (1) Skip. Note that the skip rule does not change the program state.

(2) Initialization. By the definition of the substitution rule, $(\rho, \sigma) \models A[0/\rho]$ is equivalent to $(\rho_0^0, \sigma) \models A$, then the state after initialization $(\rho', \sigma) = (\rho_0^0, \sigma)$ also satisfies $A$.

(3) Unitary. Note that $(UAU^\dagger)(U\rho U^\dagger) = UApU^\dagger$, so $(UAU^\dagger)(U\rho U^\dagger) \Leftrightarrow Ap = \rho$.

(4) Assignment. For the first rule, assume $(\rho, \sigma) \models A$, then $A$ is commutative with $f$. Then, $A$ is also commutative with $-f$. Thus, $(\rho, \sigma') = (\rho, \sigma[-f/f])$ also satisfies $A$.

The second rule is obviously correct, but it limits the selection of $A$.

(5) Sequencing. Assume $(\rho, \sigma) \models A$, then $[P_0; P_1](\rho, \sigma) \models C$ by the hypothesis $\{ A \} P_0 \{ C \}$. On the other hand $[P_0; P_1](\rho, \sigma) = [P_1][P_0](\rho, \sigma) \models B$ by the hypothesis $\{ C \} P_1 \{ B \}$.

(6) Condition. First, $\sum A_i M_i$ is a legal stabilizer expression because $M_i = \frac{I + f_i}{2}$ and $M_0 = \frac{I - f_i}{2}$ are legal stabilizer expressions. Assume $(\rho, \sigma) \models A$, then $\sigma(f)$ is commutative with $A$, so is $M_1$ and $M_0$. Thus, $AM_1M_1^\dagger = M_1AM_1M_1^\dagger = M_1\rho M_1^\dagger$. Likewise, we have $AM_0M_0^\dagger = M_0AM_0^\dagger = M_0 \rho M_0^\dagger$. Let $A = \sum A_i M_i$, then $AM_1M_1^\dagger = A_1M_1^\dagger(M_1\rho M_1^\dagger) = A_0M_0^\dagger(M_0\rho M_0^\dagger) = A_1(M_1\rho M_1^\dagger)$ since $M_1M_1 = M_1$, $M_1M_0 = 0$. Thus, we have $A_1M_1M_1^\dagger = M_1 \rho M_1^\dagger$. Since $f$ is commutative with both $A_1$ and $A_0$, we have $(M_1\rho M_1^\dagger, \sigma) \models A_1$.
and \((M_0 \rho M_0^\dagger, \sigma[-f/f]) \models A_0\). Also, \((M_1 \rho M_1^\dagger, \sigma) \models f\) and \((M_0 \rho M_0^\dagger, \sigma[-f/f]) \models -f\). Thus, if \((\rho, \sigma) \models \sum_i A_i M_i\), we have \((M_1 \rho M_1^\dagger, \sigma) \models A_1 \wedge f\) and \((M_0 \rho M_0^\dagger, \sigma) \models A_0 \wedge -f\). Since \(\{A_1 \wedge f\} P_1\{B\}\) and \(\{A_0 \wedge -f\} P_0\{B\}\), by the semantics of the condition statement, we have \(\{\sum_i A_i M_i\} \text{if } M[f], \bar{q} \text{ then } P_i \text{ else } P_2\{B\}\).

(7) While. The proof of the While rule is quite similar to that of the Condition rule. \(\sum_i A_i M_i\) is called the invariant of the loop. If the execution enters the loop body, then by \(\{A_1 \wedge f\} P_0(\sum_i A_i M_i)\), we still have \((\rho, \sigma) \models \sum_i A_i M_i\) for the next loop. So, when the while loop terminates, we always have \((\rho, \sigma) \models A_0 \wedge -f\).

To prove the While rule more formally, we only need to show the partial correctness holds for \(\text{while}(k)\), as \(\text{while}\) is the disjunction of \(\text{while}(k), k = 0, 1, 2, \ldots\).

(8) Consequence. Assume \((\rho, \sigma) \models A\), then \((\rho, \sigma) \models A'\) by \(A \Rightarrow A'\). Since \(\{A'\} \text{Prog}\{B'\}\), we have \([P](\rho, \sigma) \models B'\). Then \([P](\rho, \sigma) \models B\) by \(B' \Rightarrow B\). Thus, \(\{A\} \text{Prog}\{B\}\).

A.2 Verification of Quantum Repetition Code

**Proposition 6.4** (Logical CNOT). For the program \(\text{Prog} \) in Figure 6(a), assume \(A_S = Z_0 Z_1 \wedge Z_4 Z_5 \wedge Z_3 \wedge Z_4 Z_5\), we have \(Z_{L_0} I_{L_1} \wedge A_S\).

**Proof.** First, for control qubit \(a\) and target qubit \(b\), \(\text{CNOT}_{ab} = \frac{1}{2}(I + X_a + Z_a - X_a Z_a)\). Then

(1) \(\{Z_{L_0} I_{L_1}\} \text{Prog}\{Z_{L_0} I_{L_1}\}\). Note that both \(\text{CNOT}_{03}, \text{CNOT}_{14}\) and \(\text{CNOT}_{25}\) are commutative with \(Z_{L_0}\), so \(\text{CNOT}_{03} Z_{L_0} \text{CNOT}_{03} = Z_{L_0}\).

(2) \(\{X_{L_0} I_{L_1}\} \text{Prog}\{X_{L_0} X_{L_1}\}\). Note that \(\text{CNOT}_{03} Z_{L_0} \text{CNOT}_{03} = Z_{L_0} X_3\). Since \(X_3\) is commutative with \(\text{CNOT}_{14}, \text{CNOT}_{14} X_{L_0} X_{L_1} \text{CNOT}_{14} = (\text{CNOT}_{14} X_{L_0} \text{CNOT}_{14}) X_3 = X_{L_0} \text{CNOT}_{14} X_3\). Finally, \(\text{CNOT}_{25} X_{L_0} X_{L_1} \text{CNOT}_{25} = X_{L_0} \text{CNOT}_{25} X_{L_1}\).

(3) \(\{L_0 X_{L_1}\} \text{Prog}\{L_0 X_{L_1}\}\). Note that both \(\text{CNOT}_{03}, \text{CNOT}_{14}\) and \(\text{CNOT}_{25}\) are commutative with \(X_{L_1}\).

(4) \(\{I_{L_0} Z_{L_1}\} \text{Prog}\{I_{L_0} Z_{L_1}\}\). Note that \(\text{CNOT}_{03} Z_{L_1} \text{CNOT}_{03} = Z_0 Z_{L_1}\), \(\text{CNOT}_{14} Z_0 Z_{L_1} \text{CNOT}_{14} = Z_0 \text{CNOT}_{14} Z_{L_1}\), \(\text{CNOT}_{25} Z_0 Z_{L_1} \text{CNOT}_{25} = Z_0 \text{CNOT}_{25} Z_{L_1}\), \(\text{CNOT}_{25} Z_0 Z_{L_1} \text{CNOT}_{25} = Z_0 \text{CNOT}_{25} Z_{L_1}\), \(\text{CNOT}_{25} Z_0 Z_{L_1} \text{CNOT}_{25} = Z_0 Z_{L_1}\).

Finally, we can prove that \(\{Z_0 Z_1\} \text{Prog}\{Z_0 Z_1\}, \{Z_1 Z_2\} \text{Prog}\{Z_1 Z_2\}, \{Z_3 Z_4\} \text{Prog}\{Z_0 Z_1 Z_4 Z_5\}, \{Z_4 Z_5\} \text{Prog}\{Z_1 Z_2 Z_4 Z_5\}\) in a similar way. Combining all these facts, we can prove the desired partial correctness on the logical CNOT gate.

A.3 Verification of the Surface Code

**Program A.1** (Initialize \(0_L\)). For the initialization operation in the figure below, which initializes an X-cut logical qubit to \(0_L\).

We have \(\text{Prog} := q := |0\rangle; f_0 := X_0 X_1 X_2 X_3; f_1 := X_4 X_6 X_7 X_9; f_2 := X_0 X_11 X_12 X_14; f_3 := X_14 X_16 X_17 X_18; f_4 := Z_1 Z_3 Z_4 Z_6; f_5 := Z_2 Z_4 Z_5 Z_7; f_6 := Z_0 Z_8 Z_{11}; f_7 := Z_{11} Z_{13} Z_{14} Z_{16}; f_8 := Z_{2} Z_{12} Z_{10} Z_{12}; f_9 := Z_0 Z_4 Z_{11} Z_{14}; f_{10} := \cdots \text{correct}(f_0, f_1, \ldots); f_0 := I; f_1 := I; f_2 := I; f_3 := I; f_4 := Z_1 Z_3 Z_6; f_5 := Z_2 Z_5 Z_7; f_6 := Z_0 Z_8 Z_{11}; f_7 := Z_{11} Z_{13} Z_{16}; f_8 := Z_7 Z_{10} Z_{12}; f_9 := Z_{12} Z_{15} Z_{17}; f_{10} := \cdots; f_{w+1} := Z_4; f_{w+2} := Z_9; f_{w+3} := Z_{14}; // set q_1, q_9, q_{14} to |0\rangle; \text{if } f_{w+1}, q_9 \text{ then skip else } q := X_3 X_6 X_7 X_9 \bar{q}; f_{w+1} := Z_4 \text{ end } \text{if } f_{w+2}, q_9 \text{ then skip else } \bar{q} := X_3 X_{11} X_{12} X_{14}; f_{w+2} := Z_9 \text{ end} \)
if \( M[f_{w+3}, q_{14}] \) then skip else \( q := X_{14}X_{16}X_{17}X_{18}q; f_{w+1} := Z_{14} \) end;
\( f_1 := X_{4}X_{6}X_{7}X_{9}; f_2 := X_{9}X_{11}X_{12}X_{14}; \text{correct}(f_0, f_1, \ldots) \).

**Proposition A.1** (Initialize \(|0_L\)). For the program \( \text{Prog} \) in \( \text{Program A.1} \) which initializes a X-cut logical qubit to \(|0_L\), \( \{I\} \text{Prog}\{Z_4Z_9Z_{14}\} \). Here \( Z_4Z_9Z_{14} \) is the logical \( Z \) operator \( Z_L \).

**Proof.** By Proposition A.8 after \text{correct} function, \( (\rho, \sigma) \models (f_0 \land f_1 \land f_2 \cdots) \). The following stabilizer assignments which turn off \( X \)-stabilizers will just forward the precondition.

For simplicity, assume there are \( w \) stabilizers in the surface code array. Let \( A = \{0, \ldots, w - 1\} \), then \( (f_0 \land f_1 \land f_2 \cdots) = \bigwedge_{i \in A} f_i \).

Since \( f_0 \land f_1 \Rightarrow X_0X_1X_2X_6X_7X_9 \) and \( X_0X_1X_2X_6X_7X_9 \) is commutative with \( Z_4 \), \( \{\bigwedge_{i \in A} f_i\} f_{w+1} := Z_4 \{\bigwedge_{i \in A}\{0, 1\} f_i \land X_0X_1X_2X_6X_7X_9\} \). Likewise, we know that after \( f_{w+2} := Z_9 \), the precondition will become \( \{\bigwedge_{i \in A}\{0, 1, 2, 3\} f_i\} \land X_0X_1X_2X_6X_7X_{11}X_{12}X_{16}X_{17}X_{18} \).

Note that \( \{\bigwedge_{i \in A}\{0, 1, 2, 3\} f_i\} \land X_0X_1X_2X_6X_7X_{11}X_{12}X_{16}X_{17}X_{18} \Rightarrow X_0X_1X_2X_6X_7X_{11}X_{12}X_{16}X_{17}X_{18} \).

Let \( A = X_0X_1X_2X_6X_7X_{11}X_{12}X_{16}X_{17}X_{18} \).

\( c = f_{w+1}, q_{14} \) then skip else \( q := X_0X_6X_7X_9q; f_{w+1} := Z_4 \) end. It’s easy to see that \( \{A \land Z_4\} \text{skip}\{A \land Z_4\} \), and \( \{A \land Z_4\} q_4 := X_4q_4; f_{w+1} := Z_4 \{A \land Z_4\} \). Thus, \( \{A\} \text{c}\{A \land Z_4\} \). Then, after reset \( q_{14} \), the precondition will become: \( \{A \land Z_4 \land Z_9 \land Z_{14}\} \).

Again, the following stabilizer assignments will just forward the precondition. By the implication rule, we have that \( A \land Z_4 \land Z_9 \land Z_{14} \Rightarrow A \land Z_4Z_9Z_{14} \). Since \( Z_4Z_9Z_{14} \) and all assertions in \( A \) are commutable with stabilizers \( f_0, f_1, \ldots \), we have \( \{A \land Z_4Z_9Z_{14}\} \text{correct}(f_0, f_1, \ldots) \{\bigwedge_{i \in A} f_i \land Z_4Z_9Z_{14} \land X_0X_1X_2X_6X_7X_{11}X_{12}X_{16}X_{17}X_{18}\} \). Then by applying the consequence rule, we get \( \{I\} \text{Prog}\{Z_4Z_9Z_{14}\} \).

**Program A.2** (Logical X gate). For the logical X gate \( X_L \) in the Figure below:

![Logical X gate diagram](https://via.placeholder.com/150)

we have \( \text{Prog} := q_0q_1q_2q_4 := X_0X_1X_2X_4q_0q_1q_2q_4 \).

**Proposition A.2** (Logical X gate). For program \( \text{Prog} \) in Figure (d), we have \( \{Z_L\} \text{Prog}\{\neg Z_L\} \) and \( \{\neg Z_L\} \text{Prog}\{Z_L\} \), where \( Z_L = Z_4Z_9Z_{14} \).

**Proof.** Notice that \( (X_L)(Z_L)(X_L)^\dagger = -Z_L \).

**Lemma 7.3.** For a X-cut qubit state \(|\psi\rangle\), if \(|\psi\rangle = \alpha|0_L\rangle + \beta|1_L\rangle \) \(|\alpha|^2 + |\beta|^2 = 1\), then there is a unique \((aZ_L + bX_L)\) s.t. \((aZ_L + bX_L)|\psi\rangle = |\psi\rangle\), and in this case \( a = \frac{\alpha^2 - \beta^2}{\alpha^2 + \beta^2} \) and \( b = \frac{2\alpha\beta}{\alpha^2 + \beta^2} \).

Conversely, for a X-cut qubit state \(|\psi\rangle\), if \((aZ_L + bX_L)\) \(*\) \(|\psi\rangle = |\psi\rangle\), and \(|\psi\rangle\) is in the space spanned by \(|0_L\rangle, |1_L\rangle\), then \(|\psi\rangle = \alpha|0_L\rangle + \beta|1_L\rangle\), up to a global phase.

**Proof.** For the first part, we can get \( a = \frac{\alpha^2 - \beta^2}{\alpha^2 + \beta^2} \) and \( b = \frac{2\alpha\beta}{\alpha^2 + \beta^2} \) simply by solving the equation \((aZ_L + bX_L)|\psi\rangle = |\psi\rangle\).

For the second part, assume \(|\psi\rangle\) is a logical \( Z_L \) state of \(|\psi\rangle\), and \(|\psi\rangle = \alpha\lambda Z_L + \beta\lambda X_L\), if there is a \( a\lambda Z_L + b\lambda X_L \) s.t. \((a\lambda Z_L + b\lambda X_L)|\psi\rangle = |\psi\rangle\) and \((a\lambda Z_L + b\lambda X_L)|\psi\rangle = |\psi\rangle\). Then, we have \( a = \frac{\alpha^2 - \beta^2}{\alpha^2 + \beta^2} = \frac{2\alpha^2 - \beta^2}{\alpha^2 + \beta^2} = \frac{2\alpha^2 - \beta^2}{\alpha^2 + \beta^2} = \frac{2\alpha^2 - \beta^2}{\alpha^2 + \beta^2} \), which is equivalent to \( 1 \frac{2}{1 + \frac{\beta^2}{\alpha^2}} = 1 \frac{2}{1 + \frac{\beta^2}{\alpha^2}} \). Thus, \( a = \frac{\alpha^2}{\beta^2} \), i.e., \(|\psi\rangle = |\psi\rangle\) up to a global phase.
Proposition 7.4 (Vertical qubit moving). For program Prog in Figure 8(c) which implements the qubit moving operation in Figure 7(c), we have \( \{aZ_L + bX_L\} Prog \{aZ_L' + bX_L'\} \), where \( Z_L = Z_0Z_1Z_2, X_L = X_2X_3X_4X_6, Z'_L = Z_0Z_1Z_2Z_6, X'_L = X_6X_8X_9X_{10} \), \( a, b \in \mathbb{C} \).

Proof. After the first correction function, the precondition is transformed into: \( (aZ_L + bX_L) \land f_i \). The three following stabilizer assignments will forward the precondition. Then by the implication rule, \( (aZ_L + bX_L) \land f_i \Rightarrow (aZ_L + bX_2X_3X_4X_8X_9X_{10}) \land f_i \), so for the next stabilizer assignment \( f_{w+1} = Z_6 \), precondition \( (aZ_L + bX_2X_3X_4X_8X_9X_{10}) \land f_i \) will be forwarded. Note that \( (aZ_L + bX_2X_3X_4X_8X_9X_{10}) \land f_i \Rightarrow aZ_L + bX_2X_3X_4X_8X_9X_{10} \), let \( A = aZ_L + bX_2X_3X_4X_8X_9X_{10} \).

For the if statement, \( \{A \land f_{w+1}\} \) skip \( \{A \land f_{w+1}\} \) and \( \{A \land -f_{w+1}\} \). By implication rule, \( A \land f_{w+1} \Rightarrow A \land f_{w+1} \). The next three stabilizer assignment will forward \( aZ_LZ_6 + bX_2X_3X_4X_8X_9X_{10} \). Then with the correct function and the consequence rule, we get that \( \{aZ_L + bX_L\} Prog \{aZ_L' + bX_L'\} \), i.e., the logical state is not changed by the qubit moving operation.

A braiding operation involves many data qubits, and at least 51 data qubits will be referenced in the problem. To simplify the program, we will use the qubit moving as primitive. \( qmov(X_L, X'_L) \) means to move the defect that changes the logical X operation of a X-cut qubit from \( X_L \) to \( X'_L \), and \( qmov(Z_L, Z'_L) \) to move the defect that changes the logical Z operation of a Z-cut qubit from \( Z_L \) to \( Z'_L \).

Program A.3 (Braiding). In the figure below, we braid a Z-cut qubit with a X-cut qubit:

The associated program is Prog :=
\[
qmov(Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7, Z_8, Z_9, Z_{10}, Z_{11}, Z_{12}, Z_{13}, Z_{14}, Z_{15}, Z_{16}, Z_{17}, Z_{18}, Z_{19}, Z_{20}, Z_{21}, Z_{22}, Z_{23}, Z_{24}, Z_{25}, Z_{26}, Z_{27}, Z_{28}, Z_{29}, Z_{30}, Z_{31}, Z_{32}, Z_{33}, Z_{34}, Z_{35}, Z_{36}, Z_{37}, Z_{38}, Z_{39}, Z_{40}, Z_{41}, Z_{42}, Z_{43}, Z_{44}, Z_{45}, Z_{46})
\]

\[
qmov(Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7, Z_8, Z_9, Z_{10}, Z_{11}, Z_{12}, Z_{13}, Z_{14}, Z_{15}, Z_{16}, Z_{17}, Z_{18}, Z_{19}, Z_{20}, Z_{21}, Z_{22}, Z_{23}, Z_{24}, Z_{25}, Z_{26}, Z_{27}, Z_{28}, Z_{29}, Z_{30}, Z_{31}, Z_{32}, Z_{33}, Z_{34}, Z_{35}, Z_{36}, Z_{37}, Z_{38}, Z_{39}, Z_{40}, Z_{41}, Z_{42}, Z_{43}, Z_{44}, Z_{45}, Z_{46})
\]

\[
qmov(Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7, Z_8, Z_9, Z_{10}, Z_{11}, Z_{12}, Z_{13}, Z_{14}, Z_{15}, Z_{16}, Z_{17}, Z_{18}, Z_{19}, Z_{20}, Z_{21}, Z_{22}, Z_{23}, Z_{24}, Z_{25}, Z_{26}, Z_{27}, Z_{28}, Z_{29}, Z_{30}, Z_{31}, Z_{32}, Z_{33}, Z_{34}, Z_{35}, Z_{36}, Z_{37}, Z_{38}, Z_{39}, Z_{40}, Z_{41}, Z_{42}, Z_{43}, Z_{44}, Z_{45}, Z_{46})
\]

\[
qmov(Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7, Z_8, Z_9, Z_{10}, Z_{11}, Z_{12}, Z_{13}, Z_{14}, Z_{15}, Z_{16}, Z_{17}, Z_{18}, Z_{19}, Z_{20}, Z_{21}, Z_{22}, Z_{23}, Z_{24}, Z_{25}, Z_{26}, Z_{27}, Z_{28}, Z_{29}, Z_{30}, Z_{31}, Z_{32}, Z_{33}, Z_{34}, Z_{35}, Z_{36}, Z_{37}, Z_{38}, Z_{39}, Z_{40}, Z_{41}, Z_{42}, Z_{43}, Z_{44}, Z_{45}, Z_{46})
\]
According to Fowler, the verification of the braiding operation only need to focus on four configurations of logical states on a pair of logical qubits: $X_{L1} \otimes I_{L2}, I_{L1} \otimes X_{L2}, X_{L1} \otimes Z_{L2}$ and $Z_{L1} \otimes I_{L2}$.

**Proposition A.3 (Braiding).** For the program $\text{Prog}$ in Program 3, 

\[\{X_{L1} I_{L2}\} \text{Prog} \{X_{L1} X_{L2}\}, \{I_{L1} Z_{L2}\} \text{Prog} \{Z_{L1} Z_{L2}\}, \{I_{L1} X_{L2}\} \text{Prog} \{I_{L1} X_{L2}\} \text{ and } \{Z_{L1} I_{L2}\} \text{Prog} \{Z_{L1} I_{L2}\} \]

**Proof.** To simplify the proof, we let $A_S = \land_i f_i$, i.e., the assertion generated by current active stabilizers in the surface code array. Note that $A_S$ may change at different time-step. The proof of the braiding operation involves tedious computation and we only give a sketch of the proof here.

1. **Prove** \(\{X_{L1} I_{L2}\} \text{Prog} \{X_{L1} X_{L2}\}\). Since $X_{L1} I_{L2} = X_{L1}$, we only need to focus on the reasoning on $X_{L1}$ only. From the verification of the qubit moving, \(\{X_{L1} I_{L2}\} \text{qmov}(Z_{38} Z_{43} Z_{44} Z_{48}, Z_{39} Z_{44} Z_{45} Z_{49})\) \(\{X_{L1} X_{L2}\}\) (after correct function, \(\{X_{L1} I_{L2}\} \otimes X_{L1} \otimes A_S\) becomes \(\{X_{L1} X_{L2}\} \otimes A_S\)). Then, after all these qubit moving operations, we will get

\(\{X_{L1} I_{L2}\} \text{Prog} \{X_{L1} X_{L2}\}\).

Apply implication rule on $A_S$, we get

\(A_S \Rightarrow (X_{10} X_{15} X_{16} X_{21})(X_{21} X_{26} X_{27} X_{32})(X_{32} X_{37} X_{38} X_{43})(X_{33} X_{38} X_{39} X_{44})(X_{43} X_{45} X_{46} X_{49})(X_{25} X_{28} X_{33})(X_{12} X_{17} X_{18} X_{23})(X_{11} X_{16} X_{17} X_{22}) = (X_{10} X_{15} X_{16} X_{21})(X_{21} X_{26} X_{27} X_{32})(X_{32} X_{37} X_{38} X_{43})(X_{33} X_{38} X_{39} X_{44})(X_{43} X_{45} X_{46} X_{49})(X_{25} X_{28} X_{33})(X_{12} X_{17} X_{18} X_{23})(X_{11} X_{16} X_{17} X_{22})\).

Then, by the consequence rule, we have \(\{X_{L1} I_{L2}\} \text{Prog} \{X_{L1} X_{L2}\}\).

2. **Prove** \(\{I_{L1} Z_{L2}\} \text{Prog} \{Z_{L1} Z_{L2}\}\). Before the qubit moving operation involves qubits in $Z_{L2}$, the precondition \(\{I_{L1} Z_{L2}\}\) will be forwarded by the qubit moving operation. So, we only need to elaborate on $\text{qmov}(Z_{38} Z_{43} Z_{44} Z_{48}, Z_{39} Z_{44} Z_{45} Z_{49})$.

Before measuring $q_{44}$ in X basis, the assignment statement about $X_{44}$ will turn the precondition \(\{I_{L1} Z_{L2}\}\) into

\(Z_{L2}(Z_{39} Z_{44} Z_{45} Z_{49})\), following the previous verification steps of qubit moving. The if statement on $X_{44}$ and $q_{44}$ will then transform the precondition into \(Z_{L2}(Z_{39} Z_{44} Z_{45} Z_{49}) \otimes X_{44}\). The following assignment statement about $Z_{38} Z_{43} Z_{44} Z_{48}$ will turn the precondition \(Z_{L2}(Z_{39} Z_{44} Z_{45} Z_{49}) \otimes X_{44}\) into \(Z_{L2}(Z_{39} Z_{44} Z_{45} Z_{49}) \otimes X_{44}\). Likewise, the remaining qubit moving operations will change the precondition \(Z_{L2}(Z_{39} Z_{44} Z_{45} Z_{49})\) to \(Z_{L2}(Z_{40} Z_{45} Z_{46} Z_{50})\), \ldots, until \(Z_{L2}(Z_{59} Z_{10} Z_{15})\), which is just $Z_{L1} Z_{L2}$. Thus, \(\{I_{L1} Z_{L2}\} \text{Prog} \{Z_{L1} Z_{L2}\}\).

3. **Prove** \(\{I_{L1} X_{L2}\} \text{Prog} \{I_{L1} X_{L2}\}\). Recall the verification of the qubit moving operation. It is easy to see that \(\{I_{L1}\} \text{qmov} \{I_{L1}\}\) for any qubit moving operation in $P$. On the other hand, the qubit moving operations in $P$ does not involve any qubits in $X_{L2}$, so precondition $I_{L1} X_{L2}$ will be forwarded by all qubit moving operations, i.e., \(\{I_{L1} X_{L2}\} \text{Prog} \{I_{L1} X_{L2}\}\).

4. **Prove** \(\{Z_{L1} I_{L2}\} \text{Prog} \{Z_{L1} I_{L2}\}\). Since $Z_{L1} I_{L2} = Z_{L1}$, we only focus on the reasoning of $Z_{L1}$ here. It is obvious that starting from $Z_{59} Z_{10} Z_{15}$, the logical Z operator finally returns to $Z_{59} Z_{10} Z_{15}$ by a series of qubit moving operations. Thus, \(\{Z_{L1} I_{L2}\} \text{Prog} \{Z_{L1} I_{L2}\}\).