RESEARCH-ARTICLE

Efficient Formal Verification of Quantum Error Correcting Programs

QIFAN HUANG, Chinese Academy of Sciences, Beijing, Beijing, China

LI ZHOU, Chinese Academy of Sciences, Beijing, Beijing, China

WANG FANG, The University of Edinburgh, Edinburgh, Scotland, U.K.

MENGYU ZHAO, Chinese Academy of Sciences, Beijing, Beijing, China

MINGSHENG YING, University of Technology Sydney, Sydney, NSW, Australia

Published: 10 June 2025
Accepted: 06 March 2025
Received: 15 November 2024

Efficient Formal Verification of Quantum Error Correcting Programs

QIFAN HUANG, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
LI ZHOU*, Institute of Software, Chinese Academy of Sciences, China
WANG FANG, School of Informatics, University of Edinburgh, United Kingdom
MENGYU ZHAO, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
MINGSHENG YING*, University of Technology Sydney, Australia

Quantum error correction (QEC) is fundamental for suppressing noise in quantum hardware and enabling fault-tolerant quantum computation. In this paper, we propose an efficient verification framework for QEC programs. We define an assertion logic and a program logic specifically crafted for QEC programs and establish a sound proof system. We then develop an efficient method for handling verification conditions (VCs) of QEC programs: for Pauli errors, the VCs are reduced to classical assertions that can be solved by SMT solvers, and for non-Pauli errors, we provide a heuristic algorithm. We formalize the proposed program logic in Coq proof assistant, making it a verified QEC verifier. Additionally, we implement an automated QEC verifier, Veri-QEC, for verifying various fault-tolerant scenarios. We demonstrate the efficiency and broad functionality of the framework by performing different verification tasks across various scenarios. Finally, we present a benchmark of 14 verified stabilizer codes.

CCS Concepts: • **Theory of computation → Logic and verification**; **Hoare logic**; • **Hardware → Quantum error correction and fault tolerance**.

Additional Key Words and Phrases: Formal verification, Quantum error correction, Quantum programming language, Hoare logic

ACM Reference Format:
Qifan Huang, Li Zhou, Wang Fang, Mengyu Zhao, and Mingsheng Ying. 2025. Efficient Formal Verification of Quantum Error Correcting Programs. *Proc. ACM Program. Lang.* 9, PLDI, Article 190 (June 2025), 26 pages. https://doi.org/10.1145/3729293

*Corresponding author: Li Zhou, Mingsheng Ying

Authors' Contact Information: Qifan Huang, huangqf@ios.ac.cn, Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China; Li Zhou, zhouli@ios.ac.cn, zhou31416@gmail.com, Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China; Wang Fang, fangw@ios.ac.cn, School of Informatics, University of Edinburgh, Edinburgh, United Kingdom; Mengyu Zhao, zhaomy@ios.ac.cn, Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China; Mingsheng Ying, mingsheng.ying@uts.edu.au, University of Technology Sydney, Sydney, Australia.

ACM 2475-1421/2025/6-ART190
https://doi.org/10.1145/3729293

1 Introduction

Beyond the current noisy intermediate scale quantum (NISQ) era [64], fault-tolerant quantum computation is an indispensable step towards scalable quantum computation. Quantum error correcting (QEC) codes serve as a foundation for suppressing noise and implementing fault-tolerant quantum computation in noisy quantum hardware. There have been more and more experiments illustrating the implementation of quantum error correcting codes in real quantum processors [3, 10, 16, 69, 88]. These experiments show the great potential of QEC codes to reduce noise. Nevertheless, the increasingly complex QEC protocols make it crucial to verify the correctness of these protocols before deploying them.

There have been several verification techniques developed for QEC programs. Numerical simulation, especially *stabilizer-based simulation* [1, 5, 36] is extensively used for testing QEC programs. While stabilizer-based simulations can efficiently handle QEC circuits with only Clifford operations [61] compared to general methods [84], showing the effectiveness and correctness of QEC circuits still requires millions or even trillions of test cases, which is the main bottleneck [36]. Recently, *symbolic execution* [32] has also been applied to verify QEC programs. It is an automated approach designed to handle a large number of test cases and is primarily intended for bug reporting. However, it has limited functionality, such as the inability to reason about non-Clifford gates or propagation errors, and it remains slow when verifying correct instances.

Program logic is another appealing verification technique. It naturally handles a class of instances simultaneously by expressing and reasoning about rich specifications in a mathematical way [39]. Two recent works pave the way for using Hoare-style program logic for reasoning about QEC programs. Both works leverage the concept of stabilizer, which is critical in current QEC codes to develop their programming models. Sundaram et al. [74] established a lightweight Hoare-like logic for quantum programs that treat stabilizers as predicates. Wu et al. [82, 83] studied the syntax and semantics of QEC programs by employing stabilizers as first-class objects. They proposed a program logic designed for verifying QEC programs with fixed operations and errors. Yet, at this moment, these approaches do not achieve usability for verifying large-scale QEC codes with complicated structures, in particular for real scenarios of errors that appear in fault-tolerant quantum computation.

Technical Challenge. There are still critical challenges to the efficient verification of large-scale QEC programs, as summarized below.

- *A suitable hybrid program logic supporting backward reasoning.* QEC codes are designed to correct possible errors, making error modeling crucial for verification. To this end, it is necessary to introduce classical variables to describe errors and measurement outcomes, as well as properties like the maximum number of correctable errors. Backward reasoning is then desired since it gives a simple but complete rule for classical assignment, while forward reasoning needs additional universal quantifiers to ensure completeness. As discussed in [80] and illustrated in Example 3.3, interpreting $\vee$ as classical disjunction suffers from the incompleteness problem even for QEC codes, making it necessary to choose quantum logic as base logic, where, $\vee$ is interpreted as the sum of subspaces.
- *Proving verification conditions generated by program logic.* Traditionally, after annotating the program, the program logic will generate verification conditions (entailment of assertion formulas). A complete and rigorous approach is to use formal proofs; however, this requires significant human effort. Another approach is to use efficient solvers to achieve automatic proofs. Unfortunately, quantum logic lacks efficient tools similar to SMT solvers: systematically handling quantum logic has been a longstanding challenge. On the one hand, the continuity of subspaces makes brute-force search ineffective, while on the other hand, the lack of distributive laws

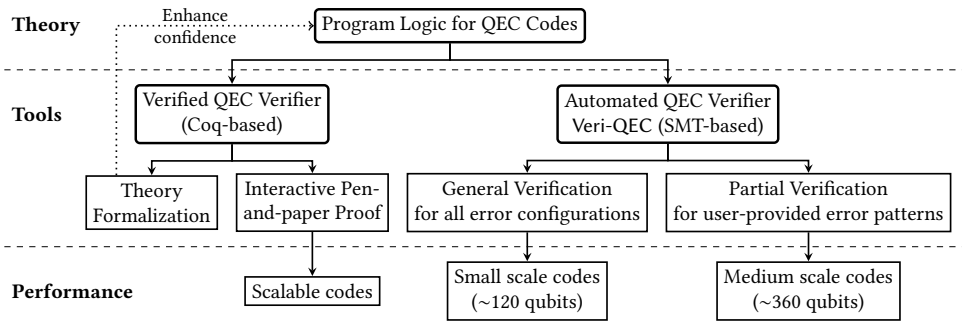

Fig. 1. Overall structure of our verification framework for QEC programs.

makes finding a (canonical) normal form particularly difficult. It remains unknown if assertion formulas for QEC codes can be efficiently processed.

Contributions. We propose a formal verification framework, summarized in Fig. 1, by proposing theoretical solutions to the above challenges, together with two implementations, (i) *the Coq-based verified QEC verifier* and (ii) *the SMT-based automatic QEC verifier Veri-QEC*, that ensure and illustrate the effectiveness of our theory. In detail, we contribute:

- *Assertion logic and program logic* (Section 3 and 4). Following [74, 83], we use Pauli expressions as atomic propositions and interpret them as the +1-eigenspace of the corresponding Pauli operator. We additionally introduce classical variables and interpret logical connectives based on quantum logic, e.g., interpreting $\vee$ as the sum of subspaces rather than as a union. Adopting the semantics for classical-quantum from [34], we establish a sound proof system for quantum programs.
- *Efficient handling of verification condition of QEC code* (Section 5). The verification condition generated by a QEC code is typically of the form

$$(P_1 \wedge \cdots \wedge P_n) \wedge \Phi_c \models \bigvee_{\mathbf{s}\in\{0,1\}^n} \left((-1)^{f_1(\mathbf{s})} P_1' \wedge \ldots \wedge (-1)^{f_n(\mathbf{s})} P_n'\right), \tag{1}$$

  where P_i, P_i' are Pauli expressions and Φ_c is a classical assertion. Progressing from simple to complex, we deal with the following cases: 1). $\{P_i'\} \subseteq \{P_j\}$. Then it is equivalent to compare phase, which can be efficiently solved by an SMT solver. 2). All P_i and P_j' commute. Then employ the fact that $P_i' = (-1)^{a_i} \prod_{k\in K_i} P_k$ since $\{P_i\}$ is a minimal generating set and $P \wedge Q = P \wedge QP$ [74] to reduce it to case 1). 3). A non-commuting pair P_i and P_j' exists. Then a heuristic algorithm is proposed to recursively eliminate P_j' from $\{P_i'\}$ based on the facts $(P \wedge Q) \vee (\neg P \wedge Q) = Q$ if P commute with Q, and finally reduce it to case 2).
- *A verified QEC verifier* (Section 6). We formalize our program logic in Coq proof assistant [77] based on CoqQ [90], i.e., proving the soundness of the proof system. This enhances confidence in the designed program logic. As a byproduct, this also allows us to manually formalize pen-and-paper proofs of scalable codes.
- *Automatic QEC verifier Veri-QEC* (Section 6 and 7). Veri-QEC is a practical tool developed in Python with the aid of Z3 and CVC5 SMT solvers [7, 29]. Veri-QEC supports verification in various scenarios, from standard errors to propagation errors or errors in correction steps, and from one cycle of QEC code to fault-tolerant implementation of small logical circuits. We examine Veri-QEC on 14 QEC codes selected from the stabilizer code family with $5 - 361$ qubits and perform different verification tasks based on the type of code and distance. Typical performance on surface codes includes: general verification for all error configurations up to 121 qubits within $\sim$ 200 minutes, and partial verification for user-provided error constraints up to 361 qubits within $\sim$ 100 minutes.

Comparison to Existing Works. Here we compare our work with works related to verifying QEC programs and leave the general discussion of related works in Section 8. Thanks to the efficiency of the stabilizer formalism in describing Clifford operations used in QEC programs, several works [67, 68, 74, 82, 83] utilize stabilizers as assertions in quantum programs. Among them, Rand et al. [67, 68] built stabilizer formalism by designing a type system of Gottesman types, upon which Sundaram et al. [74] further established a Hoare-like logic to characterize quantum programs consisting of Clifford gates, T gate and measurements. The proof system was built in forward reasoning; thus the disjoint union '$\uplus$' is employed to describe the post-measurement state. Wu et al. [83] focused more on QEC. They designed a programming language with a stabilizer constructor in the syntax, specifically for QEC programs. This programming language faithfully captures the implementation of QEC protocols. To verify the correctness of QEC programs more efficiently while ensuring the accurate characterization of their properties, they designed an assertion logic using sums of stabilizers as atomic propositions and *classical* logical connectives. Given fixed operations, errors, and exact results of the decoder, this framework can effectively prove the correctness of a given QEC program.

Compared with prior works, our verification framework stands out by incorporating classical variables into both programs and assertions. Our assertion language enables simultaneous reasoning about properties of subspaces and a family of quantum states, such as logical computational basis states, which previous QEC program logic could only handle individually. Together with the classical variables in the program, our framework can model and verify the conditions of errors that previous work cannot reason about, e.g. the maximum correctable number of errors. Our program logic provides strong flexibility and efficiency to insert errors anywhere in the QEC program, such as before and after logic operators and within correction steps, and then verify the correctness. This capability is crucial for the subsequent step of verifying the implementation of fault-tolerant quantum computing.

2 Motivating Example: The Steane Code

We introduce a motivating example, the Steane code, which is widely used in quantum computers [10, 11, 62, 70] to construct quantum circuits. A recent work [10] demonstrates the use of Steane code to implement fault-tolerant logical algorithms in reconfigurable neutral-atom arrays. We aim to demonstrate the basic concepts of our formal verification framework through the verification of Steane code.

2.1 Basic Notations and Concepts

Quantum state. Any state $|\psi\rangle$ of quantum bit (qubit) can be represented by a two-dimensional vector $\binom{\alpha}{\beta}$ with $\alpha, \beta \in \mathbb{C}$ satisfying $|\alpha|^2 + |\beta|^2 = 1$. Frequently used states include computational bases $|0\rangle \triangleq \binom{1}{0}$ and $|1\rangle \triangleq \binom{0}{1}$, and $|\pm\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle)$. The computational basis of an n-qubit system is $|\mathbf{s}\rangle \triangleq |s_1 s_2 \cdots s_n\rangle$ where $\mathbf{s}$ is a bit string, and any state $|\psi\rangle$ is a superposition $|\psi\rangle = \sum_{\mathbf{s}\in\{0,1\}^n} a_{\mathbf{s}}|\mathbf{s}\rangle$.

Unitary operator. The evolution of a (closed) quantum system is modeled as a unitary operator, aka quantum gate for qubit systems. Here we list some of the commonly used quantum gates:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix} \quad CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \quad iSWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -i & 0 \\ 0 & -i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The evolution is computed by matrix multiplication, for example, H gate transforms $|0\rangle$ to $H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \binom{1}{0} = \frac{1}{\sqrt{2}} \binom{1}{1} = |+\rangle$.

Projective measurement. We here consider the boolean-valued projective measurement $M = \{P_0, P_1\}$ with projections P_0 and P_1 such that $P_0 + P_1 = I$. Performing M on a given state $|\psi\rangle$, with probability $p_m = |P_m|\psi\rangle|^2$ we get m and post-measurement state $\frac{P_m|\psi\rangle}{\sqrt{p_m}}$ for $m = 0, 1$.

Pauli group and Clifford gate. The *Pauli group* on n qubits $\mathcal{P}_n$ consists of all Pauli strings g which are represented by the tensor product of n Pauli or identity matrices with multiplicative factor $\pm 1, \pm i$, i.e., $i^t p_1 \otimes \cdots \otimes p_n$, where $p_i \in \{I, X, Y, Z\}, t \in \{0, 1, 2, 3\}$. A state $|\psi\rangle$ is stabilized by $g \in \mathcal{P}_n$ (or a subset $S \subseteq \mathcal{P}_n$) , if $g|\psi\rangle = |\psi\rangle$ (or $\forall\, g \in S,\ g|\psi\rangle = |\psi\rangle$). The measurement outcome of the corresponding projective measurement M_g is always 0 iff $|\psi\rangle$ is a stabilizer state of g. A unitary V is a *Clifford gate*, if for any Pauli string g, $VgV^\dagger$ is still a Pauli string. All Clifford gates form the Clifford group, and can be generated by H, S, and *CNOT*.

Stabilizer code. An $[[n, k, d]]$ stabilizer code C is a subspace of the n-qubit state space, defined as the set (aka codespace) of states stabilized by an abelian subgroup S (aka stabilizer group) of Pauli group $\mathcal{P}_n$, with a minimal representation in terms of $n - k$ independent and commuting generators $\langle g_1, \ldots, g_{n-k}\rangle$ requiring $-I \notin S$. The codespace of C is of dimension 2^k and thus able to encode k logical qubits into n physical qubits. With additional k logical operators $\bar{Z}_1, \cdots, \bar{Z}_k$ that are independent and commuting with each other and S, we can define a k-qubit logical state $|z_1, \ldots, z_k\rangle_L$ as the state stabilized by $\langle g_1, \ldots, g_{n-k}, (-1)^{z_1}\bar{Z}_1, \ldots, (-1)^{z_k}\bar{Z}_k\rangle$ with $z_i \in \{0, 1\}$. We can further construct $\bar{X}_1, \ldots, \bar{X}_k$ such that $\bar{X}_i$ commute with $g \in S$ and $\bar{X}_i\bar{Z}_j = (-1)^{\delta_{ij}}\bar{Z}_j\bar{X}_i$ for all $i, j \in \{1, \cdots, k\}$, and regard $\bar{Z}_i$ (or $\bar{X}_i$) as logical Z (or X) gate acting on i-th logical qubit. d is the code distance, i.e., the minimum (Hamming) weight of errors that can go undetected by the code.

2.2 The [[7,1,3]] Steane Code

The Steane code encodes a logical qubit using 7 physical qubits. The code distance is 3, therefore it is the smallest CSS code [21] that can correct any single-qubit Pauli error. The generators $g_1, \ldots, g_6$, and logical operators $\bar{X}$ and $\bar{Z}$ of Steane code are as follows:

$$g_1 := X_1X_3X_5X_7 \qquad g_2 := X_2X_3X_6X_7 \qquad g_3 := X_4X_5X_6X_7 \qquad \bar{X} := X_1X_2X_3X_4X_5X_6X_7$$
$$g_4 := Z_1Z_3Z_5Z_7 \qquad g_5 := Z_2Z_3Z_6Z_7 \qquad g_6 := Z_4Z_5Z_6Z_7 \qquad \bar{Z} := Z_1Z_2Z_3Z_4Z_5Z_6Z_7.$$

In Table 1, we describe the implementations of logical Clifford operations and error correction procedures using the programming syntax introduced in Section 4.

As a running example, we analyze a one-round error correction process in the presence of single-qubit Pauli Y errors, as well as the Hadamard H error and T error serving as instances of non-Pauli errors. First, we inject propagation errors controlled by Boolean-valued indicators $\{e_{pi}\}$ at the beginning. A propagation error simulates the leftover error from the previous error correction process, which must be considered and analyzed to achieve large-scale fault-tolerant computing. Next, a logical operation H is applied followed by the standard error injection controlled by indicators $\{e_i\}$. Formally, $[e_i]q_i *= U$ means applying the error U on q_i if $e_i = 1$, and skipping otherwise. Afterwards, we measure the system according to generators of the stabilizer group, compute the decoding functions $f_{x,i}$ and $f_{z,i}$, and finally perform correction operations. The technical details of the program can be found in Section 5.2 and Appendix C.

The correctness formula for the program $\mathbf{Steane}(E) - \mathbf{H}$ can be stated as the Hoare triple[1]:

$$\left\{\left(\sum_{i=1}^{7}(e_i + e_{pi}) \le 1\right) \wedge \left((-1)^b\bar{X} \wedge g_1 \wedge \cdots \wedge g_6\right)\right\} \mathbf{Steane}(E) - \mathbf{H} \left\{(-1)^b\bar{Z} \wedge g_1 \wedge \cdots \wedge g_6\right\}. \quad (2)$$

[1]Following the adequacy theorem stated in [32], the correctness of the program is guaranteed as long as it holds true for only two predicates $(-1)^bZ \wedge \bigwedge_i g_i$ and $(-1)^bX \wedge \bigwedge_i g_i$. Furthermore, since Steane code is a self-dual CSS code, the logical X and Z operators share the same form. Therefore only logical Z is considered here.

Table 1. Program Implementations of logical operation and error correction using a 7-qubit Steane code.

Logical Operation		Error Correction	
	Command	Explanation	**Steane**$(E)-\mathbf{H}$ $E\in\{Y,H,T\}$
H	**for** $i\in 1\dots 7$ **do**	Propagation Error	**for** $i\in 1\dots 7$ **do** $[e_{pi}]q_i \mathrel{*}= E$ **end**
	$q_i \mathrel{*}= H$ **end**	Logical operation H	**for** $i\in 1\dots 7$ **do** $q_i \mathrel{*}= H$ **end**
S	**for** $i\in 1\dots 7$ **do**	Error injection	**for** $i\in 1\dots 7$ **do** $[e_i]q_i \mathrel{*}= E$ **end**
	$q_i \mathrel{*}= Z \fatsemi q_i \mathrel{*}= S$	Syndrome meas	**for** $i\in 1\dots 6$ **do** $s_i \coloneqq \mathbf{meas}[g_i]$ **end**
	end	Call decoder for Z	$z_1,\dots,z_7 \coloneqq f_z(s_1,s_2,s_3)$
CNOT	**for** $i\in 1\dots 7$ **do**	Call decoder for X	$x_1,\dots,x_7 \coloneqq f_x(s_4,s_5,s_6)$
	$q_i, q_{i+7} \mathrel{*}= CNOT$	Correction for X	**for** $i\in 1\dots 7$ **do** $[x_i]q_i \mathrel{*}= X$ **end**
	end	Correction for Z	**for** $i\in 1\dots 7$ **do** $[z_i]q_i \mathrel{*}= Z$ **end**

Here, b is a parameter denoting the phase of the logical state, e.g., $b=0$ for initial state $|+\rangle_L$ (i.e., state stabilized by $\bar{X}\wedge g_1\wedge\cdots\wedge g_6$) and final state $|0\rangle_L$ (i.e., state stabilized by $\bar{Z}\wedge g_1\wedge\cdots\wedge g_6$). The correctness formula claims that if there is at most one U error ($\sum_{i=1}^{7}(e_i+e_{pi})\le 1$), then the program transforms $|+\rangle_L$ to $|0\rangle_L$ (and $|-\rangle_L$ to $|1\rangle_L$), exactly the same as the error-free program that execute logical Hadamard gate H.

It appears hard to verify Eqn. (2) in previous works. [82, 83] can only handle fixed Pauli errors while **Steane** involves non-Pauli errors T with flexible positions. [67, 74] do not introduce classical variables and thus cannot represent flexible errors nor reason about the constraints or properties of errors. Fang and Ying [32] cannot handle non-Clifford gates, since non-Clifford gates change the stabilizer generators (Pauli operators) into linear combinations of Pauli operators, which are beyond their scope.

In the following sections, we will verify Eqn. (2) by first deriving a precondition A' (see Eqn. (8) for Y error and Eqn. (11) for T error) by applying the inference rules from Fig. 3, and then proving the verification condition $A\models A'$ based on the techniques proposed in Section 5.1.

3 An Assertion Logic for QEC Programs

In this section, we introduce a hybrid classical-quantum assertion logic on which our verification framework is based.

3.1 Expressions

For simplicity, we do not explicitly provide the syntax of expressions of Boolean (denoted by $BExp$); see Appendix A.1 for an example. Their value is fully determined by the state of the classical memory $m\in\mathtt{CMem}$, which is a map from variable names to their values. Given a state m of the classical memory, we write $[\![\cdot]\!]_m$ for the semantics of basic expressions in state m.

A special class of expressions was introduced by [74, 82], namely Pauli expressions. In particular, for reasoning about QEC codes with T gates, Sundaram et al. [74] suggests extending basic Pauli groups with addition and scalar multiplication with factor from the ring $\mathbb{Z}[1/\sqrt{2}]\triangleq\{x+y/\sqrt{2}\mid x,y\in\mathbb{Z}\}=\{(x+y\sqrt{2})/2^t\mid t\in\mathbb{N},x,y\in\mathbb{Z}\}$. We adopt a similar syntax of expressions in the ring $\mathbb{Z}[\frac{1}{\sqrt{2}}]$ and Pauli expressions for describing generators of stabilizer groups:

$$SExp:\quad S ::= (-1)^b \mid \sqrt{2} \mid S/2^t \mid S_1+S_2 \mid -S \mid S_1S_2 \quad \text{syntax for ring } \mathbb{Z}[\tfrac{1}{\sqrt{2}}]. \tag{3}$$

$$PExp:\quad P ::= p_r \mid sP \mid P_1P_2 \mid P_1+P_2 \quad \text{syntax for Pauli group with } s\in SExp. \tag{4}$$

In $SExp$, b is a Boolean expression and t is an expression of natural numbers. In $PExp$, p_r is an elementary gate defined as $p\in\{X,Y,Z\}$ with r being a constant natural number indicating the

qubit that p acts on. $SExp$ and $PExp$ are interpreted inductively as follows:

$$\llbracket(-1)^b\rrbracket_m \triangleq (-1)^{\llbracket b\rrbracket_m},\quad \llbracket\sqrt{2}\rrbracket_m \triangleq \sqrt{2},\quad \llbracket s/2^t\rrbracket_m \triangleq \frac{\llbracket s\rrbracket_m}{2^{\llbracket t\rrbracket_m}},$$
$$\llbracket s_1+s_2\rrbracket_m \triangleq \llbracket s_1\rrbracket_m+\llbracket s_2\rrbracket_m,\quad \llbracket -s\rrbracket_m \triangleq -\llbracket s\rrbracket_m,\quad \llbracket s_1s_2\rrbracket_m \triangleq \llbracket s_1\rrbracket_m\llbracket s_2\rrbracket_m$$
$$\llbracket p_r\rrbracket_m \triangleq I_1\otimes\cdots\otimes I_{r-1}\otimes p_r\otimes I_{r+1}\otimes\cdots\otimes I_n$$
$$\llbracket sP\rrbracket_m = \llbracket s\rrbracket_m\llbracket P\rrbracket_m,\quad \llbracket P_1P_2\rrbracket_m \triangleq \llbracket P_1\rrbracket_m\llbracket P_2\rrbracket_m,\quad \llbracket P_1+P_2\rrbracket_m \triangleq \llbracket P_1\rrbracket_m+\llbracket P_2\rrbracket_m.$$

Here, p_r is interpreted as a global gate by lifting it to the whole system, with $\otimes$ being the tensor product of linear operators, i.e., the Kronecker product if operators are written in matrix form. Such lifting is also known as cylindrical extension, and we sometimes omit explicitly writing out it. Note that it is redundant to introduce the syntax of the tensor product $p_{r_1}\otimes p_{r_2}$ with different r_1, r_2, since $\llbracket p_{r_1}\otimes p_{r_2}\rrbracket_m = I_1\otimes\cdots\otimes I_{r_1-1}\otimes p_{r_1}\otimes I_{r_1+1}\otimes\cdots\otimes I_{r_2-1}\otimes p_{r_2}\otimes I_{r_2+1}\cdots\otimes I_n = \llbracket p_{r_1}p_{r_2}\rrbracket_m$ if $r_1 < r_2$.

One primary concern of Pauli expression syntax lies in its closedness under the unitary transformations Clifford + T as claimed below. In fact, the factor $SExp$ is introduced to ensure the closedness under the T gate.

Theorem 3.1 (Closedness of Pauli expression under Clifford + T, c.f. [74]). *For any Pauli expression P defined in Eqn. (4) and single-qubit gate $U_1 \in \{X, Y, Z, H, S, T\}$ acts on q_i or two-qubit gate $U_2 \in \{CNOT, CZ, iSWAP\}$ acts on q_iq_j, there exists another Pauli expression $Q \in PExp$, such that for all $m \in$ CMem, $\llbracket Q\rrbracket_m = U_{1i}^\dagger\llbracket P\rrbracket_m U_{1i}$ or $\llbracket Q\rrbracket_m = U_{2ij}^\dagger\llbracket P\rrbracket_m U_{2ij}$.*

3.2 Assertion Language

We further define the assertion language for QEC codes by adopting Boolean and Pauli expressions as atomic propositions. Pauli expressions characterize the stabilizer group and the subspaces stabilized by it, while Boolean expressions are employed to represent error properties.

Definition 3.2 (Syntax of assertion language).

$$AExp:\quad A ::= b\in BExp \mid P\in PExp \mid \neg A \mid A\wedge A \mid A\vee A \mid A\Rightarrow A. \tag{5}$$

We interpret the assertion $A \in AExp$ as a map $\llbracket A\rrbracket : \texttt{CMem} \to \mathcal{S}(\mathcal{H})$, where CMem is the set of classical states, $\mathcal{S}(\mathcal{H})$ is the set of subspaces in global Hilbert space $\mathcal{H}$. Formally, we define its semantics as:

$$\llbracket b\rrbracket_m \triangleq \begin{cases} I_{\mathcal{H}} & \llbracket b\rrbracket_m = \textbf{true} \\ 0_{\mathcal{H}} & \llbracket b\rrbracket_m = \textbf{false} \end{cases},\quad \llbracket P\rrbracket_m \triangleq \mathrm{span}\{|\psi\rangle : \llbracket P\rrbracket_m|\psi\rangle = |\psi\rangle\},\quad \llbracket\neg A\rrbracket_m \triangleq \llbracket A\rrbracket_m^\perp,$$
$$\llbracket A_1\wedge A_2\rrbracket_m \triangleq \llbracket A_1\rrbracket_m\wedge\llbracket A_2\rrbracket_m,\quad \llbracket A_1\vee A_2\rrbracket_m \triangleq \llbracket A_1\rrbracket_m\vee\llbracket A_2\rrbracket_m,\quad \llbracket A_1\Rightarrow A_2\rrbracket_m \triangleq \llbracket A_1\rrbracket_m \rightsquigarrow \llbracket A_2\rrbracket_m$$

Boolean expression is embedded as null space or full space depending on its boolean semantics. Pauli expression is interpreted as its +1-eigenspace (aka codespace), intuitively, this is the subspace of states that are stabilized by it. It is slightly ambiguous to use $\llbracket P\rrbracket$ for both semantics of $PExp$ and $AExp$, while it can be recognized from the context if $\llbracket P\rrbracket_m$ refers to operator ($PExp$) or subspace ($AExp$). For the rest of connectives, $\llbracket\cdot\rrbracket$ is a point-wise extension of quantum logic, i.e., $^\perp$ as orthocomplement, $\wedge$ as intersection, $\vee$ as span of union, $\rightsquigarrow$ as Sasaki implication of subspaces, i.e., $a \rightsquigarrow b \triangleq \neg a\vee(a\wedge b)$. Sasaki implication degenerates to classical implication whenever a and b commute, and thus it is consistent with boolean expression, e.g., $\llbracket b_1\to b_2\rrbracket = \llbracket b_1\Rightarrow b_2\rrbracket$ where $\to$ is the boolean implication. See Appendix A.3 for more details.

3.3 Why Birkhoff-von Neumann Quantum Logic as Base Logic?

In this section, we will discuss the advantages of choosing the projection-based (Birkhoff-von Neumann) quantum logic as the base logic to verify QEC programs.

Quantum logic vs. Classical logic. A key difference is the interpretation of $\vee$, which is particularly useful for backward reasoning about **if**-branches, as shown by rule (If) in Fig. 3 that aligns with its counterpart in classical Hoare logic. However, interpreting $\vee$ as the classical disjunction is barely applicable for backward reasoning about measurement-based **if**-branches, as illustrated below.

Example 3.3 (Failure of backward reasoning about **if***-branches with classical disjunction).* Consider a fragment of QEC program $S \equiv b := \mathbf{meas}[Z_2]; \mathbf{if}\ b\ \mathbf{then}\ q_2 *= X\ \mathbf{else\ skip\ end}$, which first detects possible errors by performing a computational measurement[2] on q_2 and then corrects the error by flipping q_2 if it is detected. It can be verified that the output state is stabilized by $X_1 \wedge Z_2$ (i.e., in state $|+0\rangle_{q_1q_2}$) after executing S, if the input state is stabilized by X_1 (i.e., in state $|+\rangle_{q_1}|\psi\rangle_{q_2}$ for arbitrary $|\psi\rangle$). This fact can be formalized by correctness formula

$$\{X_1\}\ b := \mathbf{meas}[Z_2]; \mathbf{if}\ b\ \mathbf{then}\ q_2 *= X\ \mathbf{else\ skip\ end}\ \{X_1 \wedge Z_2\}. \tag{6}$$

When deriving the precondition with rule (If) where $\vee$ is interpreted as classical disjunction, one can obtain the semantics of precondition as $[\![A_0 \vee A_1]\!]' = [\![A_0]\!] \cup [\![A_1]\!] = \{|+0\rangle_{q_1q_2}, |+1\rangle_{q_1q_2}\}$, where $A_0 \triangleq X_1 \wedge Z_2$ and $A_1 \triangleq X_1 \wedge -Z_2$. This semantics of precondition is valid but far from fully characterizing all valid inputs mentioned earlier, i.e., states of the form $|+\rangle_{q_1}|\psi\rangle_{q_2}$ for arbitrary $|\psi\rangle$.

Quantum logic naturally addresses this failure, since the semantics of precondition is exactly the set of all valid input states: $[\![A_0 \vee A_1]\!] = \mathrm{span}\{[\![A_0]\!] \vee [\![A_1]\!]\} = \{\alpha|+0\rangle_{q_1q_2} + |+1\rangle_{q_1q_2} : \alpha, \beta \in \mathbb{C}\} = [\![X_1]\!]$. As Theorem A.11 suggested, the rules (If) and (Meas) maintain the universality and completeness of reasoning about broader QEC codes.

Projection-based vs. satisfaction-based approach. Although quantum logic offers richer algebraic structures, it is limited in expressiveness compared to observable-based satisfaction approaches [31, 85] and effect algebras [35, 49]: it cannot express or reason about the probabilistic properties of programs. However, this limitation is tolerable for reasoning about QEC codes. On one hand, errors in QEC codes are discretized as Pauli errors and do not directly require modeling the probability. On the other hand, a QEC code can perfectly correct discrete errors with non-probabilistic constraints. Therefore, representing and reasoning about the probabilistic attributes of QEC codes is unnecessary.

3.4 Satisfaction Relation and Entailment

In this section, we first review the representation of program states and then define the satisfaction relation, which specifies when the program states meet the truth condition of the assertion under a given interpretation.

Quantum states as density operators. The quantum system after a measurement is generally an ensemble of pure state $\{p_i, |\psi_i\rangle\}$, i.e., the system is in $|\psi_i\rangle$ with probability p_i. It is more convenient to express quantum states as partial density operators instead of pure states [61]. Formally, we write $\rho \triangleq \sum_i p_i|\psi_i\rangle\langle\psi_i| \in \mathcal{D}(\mathcal{H})$, where $\langle\psi_i|$ is the dual state, i.e., the conjugate transpose of $|\psi_i\rangle$.

Classical-quantum states. We follow [34] to define the program state in our language as a classical-quantum state $\mu : \mathtt{CMem} \to \mathcal{D}(\mathcal{H})$, which is a map from classical states to partial density operators over the whole quantum system. In particular, the singleton state, i.e., the classical state m associated with quantum state ρ, is denoted by (m, ρ).

Satisfaction relation. A one-to-one correspondence exists between projective operators and subspace, i.e., $X = \{|\psi\rangle : \mathsf{P}_X|\psi\rangle = |\psi\rangle\}$. Therefore, there is a standard way to define the satisfaction relation in projection-based approach [80, 91], i.e., a quantum state ρ satisfies a subspace X, written

[2]Note that $\mathsf{P}_{[\![Z_2]\!]_m} = |0\rangle_{q_2}\langle 0|$ and $\mathsf{P}_{[\![Z_2]\!]_m^\perp} = |1\rangle_{q_2}\langle 1|$, so $b := \mathbf{meas}[Z_2]$ represents the computational measurement on q_2 and assign the output to b.

$\rho \models X$, if and only if $\mathrm{supp}(\rho) \subseteq X$, or equivalently, $\mathsf{P}_X \rho \mathsf{P}_X = \rho$ (or $\mathsf{P}_X \rho = \rho$) where P_X is the corresponding projective operation of X. The satisfaction relation of classical-quantum states is a point-wise lifting:

Definition 3.4 (Satisfaction relation). Given a classical-quantum state μ and an assertion $A \in AExp$, the satisfaction relation is defined as: $\mu \models A$ iff for all $m \in \mathtt{CMem}$, $\mu(m) \models [\![A]\!]_m$.

The satisfaction relation faithfully characterizes the relationship of stabilizer generators and their stabilizer states, i.e., for a Pauli expression P, $|\psi\rangle\langle\psi| \models P$ iff $|\psi\rangle$ is a stabilizer state of $[\![P]\!]_m$ for any $m \in \mathtt{CMem}$. We further define the entailment between two assertions:

Definition 3.5 (Entailment). For $A, B \in AExp$, the entailment and logical equivalence are:

(1) A entails B, denoted by $A \models B$, if for all classical-quantum states μ, $\mu \models A$ implies $\mu \models B$.
(2) A and B are logically equivalent, denoted by $A \models\!\!\models B$, if $A \models B$ and $B \models A$.

The entailment relation is also a point-wise lifting of the inclusion of subspaces, i.e., $A \models B$ iff for all m, $[\![A]\!]_m \subseteq [\![B]\!]_m$. As a consequence, the proof systems of quantum logic remain sound if its entailment is defined by inclusion, e.g., a Hilbert-style proof system for $AExp$ is presented in Appendix A.4. In the (consequence) rule (Fig. 3) , strengthening the precondition and weakening the postcondition are defined as entailment relations of assertions. Therefore, entailment serves as a basis for verification conditions, which are established according to the consequence rule.

To conclude this section, we point out that the introduction of our assertion language enables us to leverage the following observation in efficient QEC verification:

Observation 3.1. *Verifying the correctness of quantum programs requires verification for all states within the state space. By introducing phase factor $(-1)^b$ to Pauli expressions, we can circumvent the need to verify each state individually. Consider a QEC code in which a logical state $|b_1 \cdots b_k\rangle_L$ is stabilized by the set of generators and logical operators $\langle g_1, \cdots, g_{n-k}, (-1)^{b_1}\bar{Z}_1, \cdots, (-1)^{b_k}\bar{Z}_k\rangle$. We can simultaneously verify the correctness for all logical states from the set $\{|b_1 \cdots b_k\rangle_L | b_1, \cdots, b_k \in \{0,1\}\}$, without introducing exponentially many assertions.*

4 A Programming Language for QEC Codes and Its Logic

In this section, we introduce our programming language and the program logic specifically designed for QEC programs.

4.1 Syntax and Semantics

The set of program commands *Prog* is defined as follows:

$$
\begin{array}{lll}
\mathit{Prog}: & S ::= \textbf{skip} \mid q_i := |0\rangle \mid q_i *= U_1 \mid q_i q_j *= U_2 & \text{where:} \\
& \quad x := e \mid x := \textbf{meas}[P] \mid S \fatsemi S & U_1 \in \{X, Y, Z, H, S, T\} \\
& \quad \textbf{if } b \textbf{ then } S \textbf{ else } S \textbf{ end} \mid \textbf{while } b \textbf{ do } S \textbf{ end} & U_2 \in \{CNOT, CZ, iSWAP\}
\end{array}
$$

where **skip** denotes the empty program, and $q_i := |0\rangle$ resets the i-th qubit to ground state $|0\rangle$. A restrictive but universal gate set is considered for unitary transformation, with single qubit gates from $\{X, Y, Z, H, S, T\}$ and two-qubit gates from $\{CNOT, CZ, iSWAP\}$, where i and j, as the indexes of unitaries, are constants and $i \neq j$ for two-qubit gates. $x := e$ is the classical assignment. In quantum measurement $x := \textbf{meas}[P]$, $P \in PExp$ is a Pauli expression which defines a projective measurement $\{M_0 = \mathsf{P}_{[\![P]\!]_m}, M_1 = \mathsf{P}_{[\![P]\!]_m^\perp}\}$; after performing the measurement, the outcome is stored in classical variable x. $S \fatsemi S$ is the sequential composition of programs. In if/loop commands, guard $b \in BExp$ is a Boolean expression, and the execution branch is determined by its value $[\![b]\!]_m$.

$$\text{(Skip)}\ \langle \textbf{skip}, (m,\rho)\rangle \to \langle \downarrow, (m,\rho)\rangle \qquad \text{(Init)}\ \langle q_i := |0\rangle, (m,\rho)\rangle \to \langle \downarrow, (m, \textstyle\sum_{k=0,1} |0\rangle_{q_i}\langle k|\rho|k\rangle_{q_i}\langle 0|)\rangle$$

$$\text{(Unit1)}\ \langle q_i *= U, (m,\rho)\rangle \to \langle \downarrow, (m, U_{q_i}\rho U_{q_i}^\dagger)\rangle \qquad \text{(Unit2)}\ \langle q_i q_j *= U, (m,\rho)\rangle \to \langle \downarrow, (m, U_{q_{i,j}}\rho U_{q_{i,j}}^\dagger)\rangle$$

$$\text{(Assign)}\ \langle x := e, (m,\rho)\rangle \to \langle \downarrow, (m[[\![e]\!]_m/x], \rho)\rangle \qquad \text{(Meas)}\ \frac{M_0 = \mathsf{P}_{[\![P]\!]_m}, M_1 = \mathsf{P}_{[\![P]\!]_m^\perp}}{\langle x := \textbf{meas}[P], (m,\rho)\rangle \to \langle \downarrow, (m[j/x], M_j\rho M_j^\dagger)\rangle}$$

$$\text{(Seq)}\ \frac{\langle S_1, (m,\rho)\rangle \to \langle S_1', (m',\rho')\rangle}{\langle S_1 \fatsemi S_2, (m,\rho)\rangle \to \langle S_1' \fatsemi S_2, (m',\rho')\rangle} \qquad \text{(If-F)}\ \frac{[\![b]\!]_m = \textbf{false}}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_0 \textbf{ end}, (m,\rho)\rangle \to \langle S_0, (m,\rho)\rangle}$$

$$\text{(While-F)}\ \frac{[\![b]\!]_m = \textbf{false}}{\langle \textbf{while } b \textbf{ do } S \textbf{ end}, (m,\rho)\rangle \to \langle \downarrow, (m,\rho)\rangle} \qquad \text{(If-T)}\ \frac{[\![b]\!]_m = \textbf{true}}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_0 \textbf{ end}, (m,\rho)\rangle \to \langle S_1, (m,\rho)\rangle}$$

$$\text{(While-T)}\ \frac{[\![b]\!]_m = \textbf{true}}{\langle \textbf{while } b \textbf{ do } S \textbf{ end}, (m,\rho)\rangle \to \langle S \fatsemi \textbf{while } b \textbf{ do } S \textbf{ end}, (m,\rho)\rangle}$$

Fig. 2. Operational semantics for QEC programs.

Our language is a subset of languages considered in [34], and we follow the same theory of defining operational and denotational semantics. In detail, a classical-quantum configuration is a pair $\langle S, (m,\rho)\rangle$, where S is the program that remains to be executed with extra symbol $\downarrow$ for termination, and (m,ρ) the current singleton states of the classical memory and quantum system. The transition rules for each construct are presented in Fig. 2. We can further define the induced denotational semantics $[\![S]\!] : (\mathtt{CMem} \times \mathcal{D}(\mathcal{H})) \to (\mathtt{CMem} \to \mathcal{D}(\mathcal{H}))$, which is a mapping from singleton states to classical-quantum states [34]. We review the technical details in Appendix A.5.

Expressiveness of the programming language. Our programming language supports Clifford + T gate set and Pauli measurements. Therefore, it is capable of expressing all possible quantum operations, in an approximate manner. The claim of expressiveness can be proved by the following observations:

(1) Clifford + T is a universal gate set [61]. Thus, according to the Solovay-Kitaev theorem, any unitary U can be approximated within error ϵ using $\Theta(\log^c(1/\epsilon))$ gates from this set.
(2) Measurement in any computational basis $|m\rangle = |a_1 a_2 \cdots a_n\rangle$ is performed by the projector $\mathsf{P}_m = \frac{\Pi_{i=1}^n (I + (-1)^{a_i} Z_i)}{2^n}$, which can be expressed using our measurement statements $x := \textbf{meas}[(-1)^{a_i} Z_i]$. Further, projective measurements augmented by unitary operations are sufficient to implement a general POVM measurement.

4.2 Correctness Formula and Proof System

Definition 4.1 (Correctness formula). The correctness formula for QEC programs is defined by the Hoare triple $\{A\}S\{B\}$, where $S \in Prog$ is a QEC program, $A, B \in AExp$ are the pre- and post-conditions. A formula $\{A\}S\{B\}$ is valid in the sense of partial correctness, written as $\models \{A\}S\{B\}$, if for any singleton state (m,ρ): $(m,\rho) \models A$ implies $[\![S]\!](m,\rho) \models B$.

The proof system of QEC program is presented in Fig. 3. Most of the inference rules are directly inspired from [34, 85, 91]. We use $A[e/x]$ (or $A[e_1/x_1, e_2/x_2, \cdots]$) to denote the (simultaneous) substitution of variable x or constant constructor $x \in \{X_r, Y_r, Z_r\}$ with expression e in assertion A. Based on the syntax of our assertion language and program constructors, we specifically design the following rules:

- Rule (Init) for initialization. Previous works [34, 85] do not present syntax for assertion language and give the precondition based on the calculation of semantics, which, however, cannot be directly expressed in $AExp$. We derive the rule (Init) from the fact that initialization

can be implemented by a computational measurement followed by a conditional X gate. As shown in the next section, the precondition is indeed the weakest precondition and semantically equivalent to the one proposed in [91].

- Rules for unitary transformation. We provide the rules for Clifford + T gates, controlled-Z (CZ) gate, as well as $iSWAP$ gate, which are easily implemented in superconducting quantum computers. It is interesting to notice that, even for two-qubit unitary gates, the pre-conditions can still be written as the substitution of elementary Pauli expressions.

Reasoning about Pauli errors. To model the possible errors occurring in the QEC program, we further introduce a syntax sugar $[b]q_i *= U$ for '**if** b **then** $q_i *= U$ **else skip end**' command, which means if the guard b is true then apply Pauli error $U \in \{X, Y, Z\}$ on q, otherwise skip. The corresponding derived rules are:

$$\left\{A[(-1)^b Y_i/Y_i, (-1)^b Z_i/Z_i]\right\}\ [b]q_i *= X\ \{A\} \qquad \left\{A[(-1)^b X_i/X_i, (-1)^b Z_i/Z_i]\right\}\ [b]q_i *= Y\ \left\{A\right\}$$
$$\left\{A[(-1)^b X_i/X_i, (-1)^b Y_i/Y_i]\right\}\ [b]q_i *= Z\ \left\{A\right\}.$$

Example 4.2 (Derivation of the precondition using the proof system). Consider a fragment of QEC program which describes the error correction stage of 3-qubit repetition code: **for** $i \in 1 \ldots 3$ **do** $[x_i]q_i *= X$ **end**. This program corrects possible X errors indicated by x_i. Starting from the post-condition $Z_1Z_2 \wedge Z_2Z_3 \wedge (-1)^b Z_1$, we derive the weakest pre-condition for this program:

$$\left\{Z_1Z_2 \wedge (-1)^{x_3} Z_2Z_3 \wedge (-1)^b Z_1\right\}\ [x_3]q_3 *= X\ \left\{Z_1Z_2 \wedge Z_2Z_3 \wedge (-1)^b Z_1\right\}$$
$$\left\{(-1)^{x_2} Z_1Z_2 \wedge (-1)^{x_3+x_2} Z_2Z_3 \wedge (-1)^b Z_1\right\}\ [x_2]q_2 *= X\ \left\{Z_1Z_2 \wedge (-1)^{x_3} Z_2Z_3 \wedge (-1)^b Z_1\right\}$$
$$\left\{(-1)^{x_2+x_1} Z_1Z_2 \wedge (-1)^{x_3+x_2} Z_2Z_3 \wedge (-1)^{b+x_1} Z_1\right\}\ [x_1]q_1 *= X\ \left\{(-1)^{x_2} Z_1Z_2 \wedge (-1)^{x_3+x_2} Z_2Z_3 \wedge (-1)^b Z_1\right\}$$

We break down the syntax sugar as a sequence of subprograms and use the inference rules for Pauli errors to derive the weakest pre-condition.

4.3 Soundness Theorem

In this subsection, we present the soundness of our proof system and sketch the proofs.

Theorem 4.3 (Soundness). *The proof system presented in Fig. 3 is sound for partial correctness; that is, for any $A, B \in AExp$ and $S \in Prog$, $\vdash \{A\}S\{B\}$ implies $\models \{A\}S\{B\}$.*

The soundness theorem can be proved in two steps. First of all, we provide the rigorous definition of the weakest liberal precondition $wlp.S.f_B$ for any program $S \in Prog$ and mapping $f_B : \mathtt{CMem} \to \mathcal{S}(\mathcal{H})$ and prove the correctness of this definition. Subsequently, we use structural induction to prove that for any $A, B \in AExp$ and $S \in Prog$ such that $\vdash \{A\}S\{B\}$, $[\![A]\!] \models wlp.S.[\![B]\!]$. Proofs are discussed in detail in Appendix A.7.

5 Verification Framework and a Case Study

Now we are ready to assemble assertion logic and program logic presented in the previous two section into a framework of QEC verification.

5.1 Verification Conditions

As Theorem A.11 suggests, all rules except for (While) and (Con) give the weakest liberal precondition with respect to the given postconditions. Then the standard procedure like the weakest precondition calculus can be used to verify any correctness formula $\{A\}S\{B\}$, as discussed in [86]:

(1) Obtain the expected precondition A' in $\{A'\}S\{B\}$ by applying inference rules of the program logic backwards.

(Skip) $\vdash \{A\}\ \mathbf{skip}\ \{A\}$ (Init) $\vdash \{(Z_i \wedge A) \vee (-Z_i \wedge A[-Y_i/Y_i, -Z_i/Z_i])\}\ q_i := |0\rangle\ \{A\}$

(Assign) $\vdash \{A[e/x]\}x := e\ \{A\}$ (Meas) $\vdash \{(P \wedge A[0/x]) \vee (\neg P \wedge A[1/x])\}\ x := \mathbf{meas}[P]\ \{A\}$

(U-X) $\vdash \{A[-Y_i/Y_i, -Z_i/Z_i]\}\ q_i *= X\ \{A\}$ (U-Y) $\vdash \{A[-X_i/X_i, -Z_i/Z_i]\}\ q_i *= Y\ \{A\}$

(U-Z) $\vdash \{A[-X_i/X_i, -Y_i/Y_i]\}\ q_i *= Z\ \{A\}$ (U-H) $\vdash \{A[Z_i/X_i, -Y_i/Y_i, X_i/Z_i]\}\ q_i := H\ \{A\}$

(U-S) $\vdash \{A[-Y_i/X_i, X_i/Y_i]\}\ q_i *= S\ \{A\}$ (U-T) $\vdash \{A[\frac{1}{\sqrt{2}}(X_i - Y_i)/X_i, \frac{1}{\sqrt{2}}(X_i + Y_i)/Y_i]\ q_i *= T\ \{A\}$

(U-CNOT) $\vdash \{A[X_iX_j/X_i, Y_iX_j/Y_i, Z_iY_j/Y_j, Z_iZ_j/Z_j]\}\ q_iq_j *= CNOT\ \{A\}$

(U-CZ) $\vdash \{A[X_iZ_j/X_i, Y_iZ_j/Y_i, Z_iX_j/X_j, Z_iY_j/Y_j]\}\ q_iq_j *= CZ\ \{A\}$

(U-iSWAP) $\vdash \{A[Z_iY_j/X_i, -Z_iX_j/Y_i, Z_j/Z_i, Y_iZ_j/X_j, -X_iZ_j/Y_j, Z_i/Z_j]\}\ q_iq_j *= iSWAP\ \{A\}$

(Seq) $\dfrac{\vdash \{A\}S_1\{B\} \quad \vdash \{B\}S_2\{C\}}{\vdash \{A\}S_1 ; S_2\{C\}}$ (If) $\dfrac{\vdash \{A_0\}S_0\{B\} \quad \vdash \{A_1\}S_1\{B\}}{\vdash \{(\neg b \wedge A_0) \vee (b \wedge A_1)\}\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_0\ \mathbf{end}\ \{B\}}$

(While) $\dfrac{\vdash \{b \wedge A\}S\{A\}}{\vdash \{A\}\ \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{end}\ \{\neg b \wedge B\}}$ (Con) $\dfrac{A \models A' \quad \vdash \{A'\}S\{B'\} \quad B' \models B}{\vdash \{A\}S\{A\}}$

Fig. 3. Inference rules for reasoning about QEC programs. For simplicity, we write $-P$ for $(-1)^{\mathbf{true}}P \in PExp$, write $P_1 - P_2$ for $P_1 + (-1)^{\mathbf{true}}P_2 \in PExp$, where $P, P_1, P_2 \in PExp$, and write $\frac{1}{\sqrt{2}}$ for $\frac{\sqrt{2}}{2^1} \in SExp$.

(2) Generate and prove the *verification condition* (VC) $A \models A'$ using the assertion logic.

Dealing with VC requires additional efforts, particularly in the presence of non-commuting pairs of Pauli expressions. However for QEC programs, there exists a general form of verification condition, which can be derived from the correctness formula:

Definition 5.1 (Correctness formula for QEC programs). Consider a program $S = \mathbf{Corr}(E) - \bar{U}$, which is generalized from the QEC program in Table 1. It operates on a stabilizer code with a minimal generating set $\{g_1, \cdots, g_{n-k}, \bar{L}_{n-k+1}, \cdots, \bar{L}_n\}$ containing n independent and commuting Pauli expressions. The correctness formula of this program can be expressed as follows:

$$\left\{ \bigwedge_i g_i \wedge \bigwedge_j \bar{L}_j \right\} S \left\{ \bigwedge_i g_i \wedge \bigwedge_j \bar{U}\bar{L}_j\bar{U}^\dagger \right\} \tag{7}$$

The verification condition to be proven is derived from this correctness formula with the aid of inference rules, as demonstrated below[3]:

$$\left(\bigwedge_i g_i \wedge \bigwedge_j \bar{L}_j \right) \wedge P_c \models \bigvee_{\mathbf{s} \in \{0,1\}^{n-k}} \left(\bigwedge_i (-1)^{r_i(\mathbf{s})+h_i(\mathbf{e})} g_i' \wedge \bigwedge_j (-1)^{r_j(\mathbf{s})+h_j(\mathbf{e})} \bar{L}_j' \right). \tag{8}$$

In Eqn. (8), P_c represents a classical assertion for errors, i, j range over $\{1, \cdots, n-k\}, \{n-k+1, \cdots, n\}$ respectively, The vector $\mathbf{s}$ encapsulates all possible measurement outcomes (syndromes) and $\mathbf{e}$ represents the error configuration. The semantics of $g_i, g_i', \bar{L}_j, \bar{L}_j'$ are normal operators. The terms $r_i(\mathbf{s}), r_j(\mathbf{s})$ denote the sum of all corrections effective for the corresponding operators, while

[3]Here, we assume the error in the correction step is always Pauli errors; otherwise, two verification conditions of the form Eqn. (8) are generated that separately deal with error before measurement and error in correction step.

$h_i(\mathbf{e}), h_j(\mathbf{e})$ account for the total error effects on the operators caused by the injected errors. The details of derivation are provided in Appendix B.1.

Let us consider how to prove Eqn. (8) in the following three cases:

(1) $\{g_i'\} \subseteq \{g_i\}$ and $\{\bar{L}_j'\} \subseteq \{\bar{L}_j\}$. The entailment is then equivalent to check $P_c \models \bigvee_{\mathbf{s}} \big(\bigwedge_i (r_i(\mathbf{s}) + h_i(\mathbf{e}) = 0) \wedge \bigwedge_j (r_j(\mathbf{s}) + h_j(\mathbf{e}) = 0) \big)$, which can be proved directly by SMT solvers.

(2) All $g_i, g_i', \bar{L}_j, \bar{L}_j'$ commute with each other. Since $\{g_i, \bar{L}_j\}$ is a minimal generating set, any g_i' or $\bar{L}_j'$ can be written as the product of $\{g_i, \bar{L}_j\}$ up to a phase ± 1, e.g., $(-1)^{\alpha_i} g_i' = \prod_{i \in \mathcal{I}_{i'}} g_i \prod_{j \in \mathcal{J}_{j'}} \bar{L}_j$, $(-1)^{\alpha_j} \bar{L}_j' = \prod_{i \in \mathcal{I}_{i'}} g_i \prod_{j \in \mathcal{J}_{j'}} \bar{L}_j$, so the entailment is equivalent to check $P_c \models \bigvee_{\mathbf{s}} \big(\bigwedge_i (r_i(\mathbf{s}) + h_i(\mathbf{e}) = a_i) \wedge \bigwedge_j (r_j(\mathbf{s}) + h_j(\mathbf{e}) = a_j) \big)$.

(3) There exist non-commuting pairs[4]. We consider the case that the total errors are less than the code distance; furthermore, g_i' is ordered such that $g_i' = U g_i U^\dagger$ for some unitary U, which can be easily achieved by preserving the order of subterms during the annotation step (1). The key idea to address this issue involves eliminating all non-commuting terms on the right-hand side (RHS) and identifying a form that is logically equivalent to the RHS. We briefly discuss the steps of how to eliminate the non-commuting terms, as outlined below:

   (a) Find the set $\mathcal{G} \subseteq \{g_i'\}$ such that any element $g_i' \in \mathcal{G}$ differs from g_i up to a phase; Find the set $\mathcal{L} \subseteq \{\bar{L}_j'\}$ such that $\bar{L}_j'$ differs from $\bar{L}_j$ up to a phase.

   (b) Update $\mathcal{G}$ and $\mathcal{L}$ by multiplying some $g_i' \in \mathcal{G}$ onto those elements, until $\mathcal{L}$ is empty and any $g_i' \in \mathcal{G}$ differs from g_i in only one qubit.

   (c) Replace those g_i' with g_i, and check if the phases of the remaining items are the same for all 2^k terms. If so, this problem can be reduced to the commuting case, since we can successfully use $(P \wedge Q) \vee (\neg P \wedge Q) = Q$ (P and Q commute with each other) to eliminate all non-commuting elements.

   To illustrate how our ideas work, we provide an concrete example in Section 5.2.2, which illustrates how to correct a single T error in the Steane code.

Soundness of the Methods.

After proposing the methods to handle the verification condition (VC), we now discuss the soundness of our methods case by case:

• *Commuting case.* If all $g_i, g_i', \bar{L}_j, \bar{L}_j'$ commute with each other, then the equivalence of the VC proposed in case (2) and Eqn. (8) can be guaranteed by the following proposition:

Proposition 5.2. *Given a verification condition of the form:*

$$\Big((-1)^{b_1} P_1 \wedge \ldots \wedge (-1)^{b_n} P_n\Big) \wedge P_c \models \bigvee_{\mathbf{s}} \Big((-1)^{b_1'} P_1' \wedge \ldots \wedge (-1)^{b_n'} P_n'\Big) \tag{9}$$

where $\{(-1)^{b_1} P_1, \ldots, (-1)^{b_n} P_n\}$, $\{(-1)^{b_1'} P_1', \ldots, (-1)^{b_n'} P_n'\}$ *are independent and commuting generators of two stabilizer groups* $S, S' \subseteq \mathcal{G}_n$, $\mathcal{G}_n$ *is the n-qubit Pauli group.* S *and* S' *satisfy* $-I \notin S, S'$. *If* $\{P_1, \ldots, P_n, P_1', \ldots, P_n'\}$ *commute with each other, then:*

I. For all i, *there exist a unique* $\alpha_i \in \{0, 1\}$ *and* $\{i_j\} \in 2^{[n]}$, *s.t.*$(-1)^{\alpha_i} P_i' = \Pi_j P_{i_j}$.

II. $P_c \models \bigwedge_{i=1}^n (b_i' = \alpha_i + \sum_j b_{i_j})$ *implies* $A \models A'$, *where* A, A' *are left and right hand side of Expression (9).*

The proof leverages the observation that any P_i' which commutes with all elements in a stabilizer group S can be written as products of generators of S [61]. We further use $P \wedge Q = QP$ to reformulate the LHS of Expression (9) and generate terms that differs from the RHS only by phases. The detailed proof of this proposition is postponed to Appendix B.2.

[4]We assume no error happens in the correction step; otherwise, we deal them in two separate VCs.

Table 2. Symbols and values appear in Eqn. (10)

Symbols	Values	Symbols	Values	Symbols	Values
$r_7(\mathbf{s})$	$\sum_{i=1}^{7} f_{z,i}$	$h_7(\mathbf{e})$	$\sum_{i=1}^{7} e_i$		
$h_1(\mathbf{e})$, $h_4(\mathbf{e})$	$e_1+e_3+e_5+e_7$	$h_2(\mathbf{e})$, $h_5(\mathbf{e})$	$e_2+e_3+e_6+e_7$	$h_3(\mathbf{e})$, $h_6(\mathbf{e})$	$e_4+e_5+e_6+e_7$
$r_1(\mathbf{s})$	$f_{z,1}+f_{z,3}+f_{z,5}+f_{z,7}$	$r_2(\mathbf{s})$	$f_{z,2}+f_{z,3}+f_{z,6}+f_{z,7}$	$r_3(\mathbf{s})$	$f_{z,4}+f_{z,5}+f_{z,6}+f_{z,7}$
$r_4(\mathbf{s})$	$f_{x,1}+f_{x,3}+f_{x,5}+f_{x,7}$	$r_5(\mathbf{s})$	$f_{x,2}+f_{x,3}+f_{x,6}+f_{x,7}$	$r_6(\mathbf{s})$	$f_{x,4}+f_{x,5}+f_{x,6}+f_{x,7}$

• *Non-commuting case.* The soundness of this case can be demonstrated by separately proving the soundness of step (a), (b) and step (c).

(1) *Step (a) and (b)*: Consider the check matrix H. If step (b) fails for some error configuration $\mathbf{e}$ with weight $w_{\mathbf{e}} \leq d-1$, then there exists a submatrix H_{sub} of size $(n-k)\times w_{\mathbf{e}}$, with columns being the error locations. The rank of the submatrix is $r < w_{\mathbf{e}}$, leading to a contradiction with the definition of d being the minimal weight of an undetectable error. This is because there exists another $\mathbf{e}'$ whose support is within that of $\mathbf{e}$, and $H\mathbf{e}' = 0$.
(2) *Step (c)*: The soundness is straightforward since $(P \wedge Q) \vee (\neg P \wedge Q) = Q$ whenever P and Q commute, which is the only formula we use to eliminate non-commuting elements.

5.2 Case Study: Steane Code (Continued)

To illustrate the general procedure of our verification framework, let us consider the 7-qubit Steane code presented in Section 2.2 with Y and T errors (H errors is deferred to Appendix C.2.

5.2.1 Case I: Reasoning about Pauli Y errors. We first verify the correctness of Steane code with Pauli Y errors. We choose Y error because its impact on stabilizer codes is equivalent to the composite effect of X and Z errors on the same qubit. In this scenario, the verification condition (VC) to be proved is generated from the precondition:[5]

$$\left\{\left(\sum_{i=1}^{7} e_i \leq 1\right) \wedge \left((-1)^b \bar{Z} \wedge \bigwedge_{i=1}^{6} g_i\right)\right\} \models \left\{\bigvee_{\mathbf{s}\in\{0,1\}^6} \left((-1)^{b+r_7(\mathbf{s})+h_7(\mathbf{e})} \bar{Z} \wedge \bigwedge_{i=1}^{6} (-1)^{r_i(\mathbf{s})+h_i(\mathbf{e})} g_i\right)\right\}. \quad (10)$$

No changes occur in Pauli generators $\bar{Z}$ and g_i, therefore according to case (1) in the proof of Eqn. (8), the verification condition is equivalent with $P_c \sqsubseteq P_c'$, where $P_c = \sum_{i=1}^{7} e_i \leq 1$, $P_c' = \bigvee_{\mathbf{s}\in\{0,1\}^6} \bigwedge_{i=1}^{7} (r_i(\mathbf{s}) + h_i(\mathbf{e}) = 0)$. We can prove the VC if the minimum-weight decoder f satisfies P_f:

$$P_f \triangleq \left(\sum_{i=1}^{7} x_i \leq \sum_{i=1}^{7} e_i\right) \bigwedge \left(\sum_{i=1}^{7} z_i \leq \sum_{i=1}^{7} e_i\right) \bigwedge \left(\bigwedge_{i=1}^{6} (r_i(\mathbf{s}) = \mathbf{s}_i)\right).$$

This P_f we give describes the necessary condition of a decoder: the corrections $r_i(\mathbf{s})$ are applied to eliminate all non-zero syndromes on the stabilizers; and weight of corrections should be less than or equal to weight of errors. Alternatively, if we know that f satisfies P_f (e.g., the decoder is given), we can identify P_c by simplifying P_c' without prior knowledge of P_c. Instead, if we are aiming to design a correct decoder f, we may extract the condition P_f from the requirement $P_c \sqsubseteq P_c'$.

5.2.2 Case II: Non-Pauli T Errors. Here we only show the processing of specific error locations $\mathbf{e}_{\mathbf{p5}} = 1$, e.g., the propagated error before logical H, to illustrate the heuristic algorithm proposed in Section 5. The general situation only makes the formula encoding more complicated but does not introdce fundamental challenges.

[5]The notations in Eqn. (10) may be a bit confusing, therefore we provide Table 2 to help explain the relationships of those notations. For details of the derivation please refer to Appendix C.1.

We consider the logical $|+\rangle_L$ and $\vdash\rangle_L$ state stabilized by the stabilizer generators and logical $\bar{X}$. The verification condition generated by the program should become [6]:

$$\left(\bigwedge_{i=1\ldots6} g_i\right) \wedge (-1)^b \bar{X} \models \bigvee_{\mathbf{s}\in\{0,1\}^6} \left(\left(\bigwedge_{i=1\ldots6} (-1)^{s_i} g_i'\right) \wedge (-1)^{b+r(\mathbf{s})} \bar{X}'\right). \tag{11}$$

In which $r(s) = \sum_{i=1}^{7} cx_i$ is the sum of X corrections, regarding the decoder as an implicit function of $\mathbf{s}$. We denote the group stabilized by $g_1, \cdots, g_6, \bar{X}$ as $\mathcal{S}$. The injected non-Pauli error $\mathbf{T_5}$ changes all X_5 to $\frac{1}{\sqrt{2}}(Y_5 - X_5)$, therefore the elements in set $\{g_1', \cdots, g_6', \bar{X}'\}$ are: $g_1' = \frac{1}{\sqrt{2}} X_1X_3(X_5 - Y_5)X_7$, $g_2' = X_2X_3X_6X_7$, $g_3' = \frac{1}{\sqrt{2}} X_4(X_5 - Y_5)X_6X_7, \bar{X}' = \frac{1}{\sqrt{2}} X_1X_2X_3X_4(X_5 - Y_5)X_6X_7$, $g_4' = Z_1Z_3Z_5Z_7$, $g_5' = Z_2Z_3Z_6Z_7$, $g_6' = Z_4Z_5Z_6Z_7$.

• *Step I: Update $\mathcal{G}$ and $\mathcal{L}$.* We obtain a subset from $\{g_1', \cdots, g_6', \bar{X}'\}$ whose elements differ from the corresponding ones in $\{g_1, \cdots g_6, \bar{X}\}$, which is $\{g_1', g_3', \bar{X}'\}$. Now pick $j_x = 1$ from this set and update g_3' and $\bar{X}'$, we can obtain a generator set of $\mathcal{S}'$: $g_1' = \frac{1}{\sqrt{2}} X_1X_3(X_5 - Y_5)X_7$, $g_2' = X_2X_3X_6X_7$, $g_3'' = X_1X_3X_4X_6$, $\bar{X}'' = X_2X_4X_6$, $g_4' = Z_1Z_3Z_5Z_7$, $g_5 = Z_2Z_3Z_6Z_7$, $g_6' = Z_4Z_5Z_6Z_7$. We update $g_3, \bar{X}$ at the same time and obtain another set of generators for $\mathcal{S}$: $\mathcal{S} = \{X_1X_3X_5X_7, X_2X_3X_6X_7$ $, X_1X_3X_4X_6, X_2X_4X_6, Z_1Z_3Z_5Z_7, Z_2Z_3Z_6Z_7, Z_4Z_5Z_6Z_7\}$. The generator sets only differ by g_1 and g_1'.

• *Step II: Remove non-commuting terms, check the phases of remaining elements.* The weakest liberal precondition on the right-hand side is now transformed into another equivalent form:

$$\bigvee_{\mathbf{s}\in\{0,1\}^6} \left((-1)^{s_1} g_1' \wedge (-1)^{s_2} g_2' \wedge (-1)^{s_2+s_3} g_3'' \wedge \left(\bigwedge_{i=4,5,6} (-1)^{s_i} g_i'\right) \wedge (-1)^{b+r(\mathbf{s})+s_1} \bar{X}''\right). \tag{12}$$

For P', Q whose elements are commute with each other, we can leverage $(P' \wedge Q) \vee (\neg P' \wedge Q) = Q$ to reduce the verification condition Eqn. (11) to the commuting case. In this case we have $P = g_1$, $P' = g_1'$ and Q being other generators, which is guaranteed by Step I. To prove the entailment in Eqn. (11), it is necessary to find two terms in Eqn. (12) whose phases only differ in s_1. Now rephrase each phase to t_i and find that Eqn. (11) has an equivalent form:

$$\left(\bigwedge_{i=1\ldots6} g_i\right) \wedge (-1)^b \bar{X} \models \bigvee_{\mathbf{t}\in\{0,1\}^7} \left((-1)^{t_1} g_1' \wedge (-1)^{t_2} g_2' \wedge (-1)^{t_3} g_3'' \wedge \left(\bigwedge_{i=4,5,6} (-1)^{t_i} g_i'\right) \wedge (-1)^{b+t_7} \bar{X}''\right). \tag{13}$$

The map $\mathbf{t} = u(\mathbf{s})$ is $t_1 = s_1, t_2 = s_2, t_3 = s_2 + s_3, t_4 = s_4, t_5 = s_5, t_6 = s_6, t_7 = \sum_{i=1}^{7} c_i + s_1$, which comes from the multiplication in Step I. To prove the entailment in Eqn. (13), we pick $\mathbf{t}$ according to step (c) in Section 5.1 and use $\mathbf{t} = u(\mathbf{s})$ as constraints to check phases of the remaining items. In this case the values of $\mathbf{s}_0$ and $\mathbf{s}_1$ are straightforward: $\mathbf{s}_0 = (0, 0, 0, 0, 0, 0)$ and $\mathbf{s}_1 = (1, 0, 1, 0, 0, 0)$. Then what remains to check is whether $t_7 = \sum_{i=1}^{7} cx_i + s_1 = 0$, which can be verified through the following logical formula for decoder: $H_z(\mathbf{cx}) = \mathbf{s_z} \wedge (\sum_i cx_i \leq \sum_i ex_i \leq 1) \implies \sum_{i=1}^{7} cx_i + s_1 = 0$.[7]

6 Tool Implementation

As summarized in Fig. 1, we implement our QEC verifiers at two levels: a verified QEC code verifier in the Coq proof assistant [77] for mechanized proof of scalable codes, and an automatic QEC verifier Veri-QEC based on Python and SMT solver for small and medium-scale codes.

[6] Only logical $\bar{X}$ is considered, since logical $\bar{Z}$ is an invariant at the presence of T errors because $T^\dagger ZT = Z$.

[7] The stabilizer generator g_1 is transformed to a Z-check after the logical Hadamard gate, so parity-check of Z are encoded in the logical formula and the syndrome s_1 guides the X corrections.

Verified QEC verifier. Starting from first principles, we formalize the semantics of classical-quantum programs based on [34], and then build the verified prover, proving the soundness of its program logic. This rules out the possibility that the program logic itself is flawed, especially considering that it involves complex classical-quantum program semantics and counterintuitive quantum logic. This is achieved by ~4700 lines of code based on the CoqQ project [90], which offers rich theories of quantum computing and quantum logic, as well as a framework for quantum program verification. We further demonstrate its functionality in verifying scalable QEC codes using repetition code as an example, where the size of the code, i.e., the number of physical qubits, is handled by a meta-variable in Coq.

Automatic QEC verifier Veri-QEC. We propose Veri-QEC, an automatic QEC code verifier implemented as a Python package. It consists of three components:

(1) Correctness formula generator. This module processes the user-provided information of the given stabilizer code, such as the parity-check matrix and logical algorithms to be executed, and generates the correctness formula expressed in plain context as the verification target.
(2) Verification condition generator. This module consists of 1) a parser that converts the program, assertion, and formula context into corresponding abstract syntax trees (AST), 2) a precondition generator that annotates the program according to inference rules (as Theorem A.11 suggests, all rules except (While) and (Con) give the weakest liberal precondition), and 3) a VC simplifier that produces the condition to be verified with only classical variables, leveraging assertion logic and techniques proposed in Section 5.1.
(3) SMT checker. This component adopts Z3 [29] to encode classical verification conditions into formulae of SMT-LIBv2 format, and invokes appropriate solvers according to the type of problems. We further implement a parallel SMT checking framework in our tool for enhanced performance.

Readers can refer to Appendix D for specific details on the tool implementation.

7 Evaluation of Veri-QEC

We divide the functionalities of Veri-QEC into two modules: the first module focuses on verifying the general properties of certain QEC codes, while the second module aims to provide alternative solutions for large QEC codes whose scales of general properties have gone beyond the upper limit of verification capability. In this case, we allow users to impose extra constraints on the error patterns.

Next, we provide the experimental results aimed at evaluating the functionality of our tool. In particular, we are interested in the performance of our tool regarding the following functionalities:

(1) The effectiveness and scalability when verifying the general properties for program implementations of QEC codes.
(2) The performance improvement when extra constraints of errors are provided by users.
(3) The capability to verify the correctness of realistic QEC scenarios with regard to fault-tolerant quantum computation.
(4) Providing a benchmark of the implementation of selected QEC codes with verified properties.

The experiments in this section are carried out on a server with 256-core AMD(R) EPYC(TM) CPU @2.45GHz and 512G RAM, running Ubuntu 22.04 LTS. Unless otherwise specified, all verification tasks are executed using 250 cores. The maximum runtime is set to 24 hours.

7.1 Verify General properties

We begin by examining the effectiveness and scalability of our tool when verifying the general properties of QEC codes.

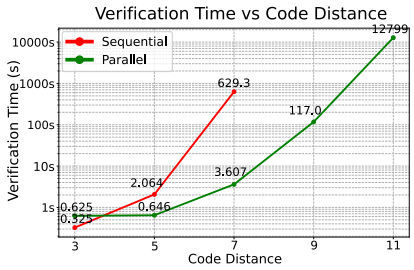

Fig. 4. Time consumed when verifying surface code in sequential/parallel.

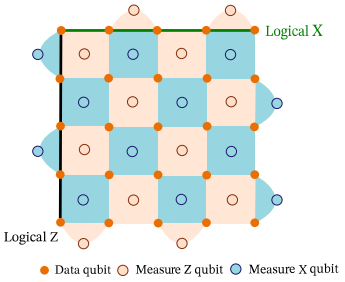

Fig. 5. Scheme of a rotated surface code with $d = 5$. Each coloured tile associated with the measure qubit in the center is a stabilizer (Flesh: Z check, Indigo: X check).

Methodology. We select the rotated surface code as the candidate for evaluation, which is a variant of Kitaev's surface code [30, 47] and has been repeatedly used as an example in Google's QEC experiments based on superconducting quantum chips [2, 3]. As depicted in Fig. 5, a $d = 5$ rotated surface code is a 5×5 lattice, with data qubits on the vertices and surfaces between the vertices representing stabilizer generators. The logical operators $\bar{X}_L$(green horizontal) and $\bar{Z}_L$ (black vertical) are also shown in the figure. Qubits are indexed from left to right and top to bottom.

For each code distance $d = 2t + 1$, we generate the corresponding Hoare triple and verify the error conditions necessary for accurate decoding and correction, as well as for the precise detection of errors. The encoded SMT formula for accurate decoding and correction is straightforward and can be referenced in Section 5.2:

$$\forall e_1, \ldots, e_n,\ \exists s_1, \ldots, s_{n-k},\ \sum_{i=1}^{n} e_i \leq \left\lfloor \frac{d-1}{2} \right\rfloor \Rightarrow \bigvee_{\mathbf{s} \in \{0,1\}^n} \left(\bigwedge_{i=1}^{n} (r_i(\mathbf{s}) + h_i(\mathbf{e}) = 0) \bigwedge P_f \right). \tag{14}$$

To verify the property of precise detection, the SMT formula can be simplified as the decoding condition is not an obligation:

$$\left(1 \leq \sum_{i=1}^{n} e_i \leq d_t - 1\right) \Rightarrow \left(\bigwedge_{i=k}^{n} (s_i = 0) \right) \wedge \left(\bigvee_{i=0}^{k-1} (h_i(\mathbf{e}) \neq 0) \right). \tag{15}$$

Eqn. (15) indicates that there exist certain error patterns with weight $\leq d_t$ such that all the syndromes are 0 but an uncorrectable logical error occurs. We expect an *unsat* result for the actual code distance d and all the trials $d_t \leq d$. If the SMT solver reports a *sat* result with a counterexample, it reveals a logical error that is undetectable by stabilizer generators but causes a flip on logical states. In our benchmark we verify this property on some codes with distance being 2, which are only capable of detecting errors. They are designed to realize some fault-tolerant non-Clifford gates, not to correct arbitrary single qubit errors.

Further, our implementation supports parallelization to tackle the exponential scaling of problem instances. We split the general task into subtasks by enumerating the possible values of e_i on selected qubits and delegating the remaining portion to SMT solvers. We denote $N(\text{bits})$ as the number of e_i whose values have been enumerated, and $N(\text{ones})$ as the count of e_i with value 1 among those already enumerated. We design a heuristic function $ET = 2d * N(\text{ones}) + N(\text{bits})$, which serves as the termination condition for enumeration.

Given its outstanding performance in solving formulas with quantifiers, we employ `CVC5` [7] as the SMT solver to check the satisfiability of the logical formulas in this paper.

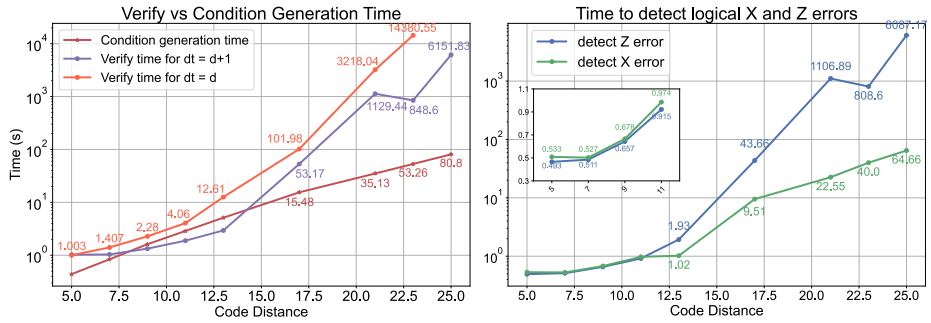

Fig. 6. Time consumed when verifying precise detection properties on surface code with distance d.

Results. *Accurate Decoding and Correction*: Fig. 4 illustrates the total runtime required to verify the error conditions for accurate decoding and correction, employing both sequential and parallel methods. The figure indicates that while both approaches produce correct results, our parallel strategy significantly improves the efficiency of the verification process. In contrast, the sequential method exceeded the maximum runtime of 24 hours at $d = 9$; we extended the threshold for solvable instances within the time limit to $d = 11$.

Precise Detection of Errors: For a rotated surface code with distance d, we first set $d_t = d$ to verify that all error patterns with Hamming weights $< d$ can be detected by the stabilizer generators. Afterward, we set $d_t = d+1$ to detect error patterns that are undetectable by the stabilizer generators but cause logical errors. The results show that all trials with $d_t = d$ report *unsat* for Eqn. (15), and trials with $d_t = d + 1$ report *sat* for Eqn. (15), providing evidence for the effectiveness of this functionality. The results indicate that, without prior knowledge of the minimum weight, this tool can identify and output the minimum weight undetectable error. Fig. 6 illustrates the relationship between the time required for verifying error conditions of precise detection of errors and the code distance.

7.2 Verify Correctness with User-provided Errors

Constrained by the exponential growth of problem size, verifying general properties limits the size of QEC codes that can be analyzed. Therefore, we allow users to autonomously impose constraints on errors and verify the correctness of the QEC code under the specified constraints. We aim for the enhanced tool, after the implementation of these constraints, to increase the size of verifiable codes. Users have the flexibility to choose the generated constraints or derive them from experimental data, as long as they can be encoded into logical formulas supported by SMT solvers. The additional constraints will also help prune the solution space by eliminating infeasible enumeration paths during parallel solving.

Results. We briefly analyze the experimental data [2, 3] and observe that the error detection probabilities of stabilizer generators tend to be uniformly distributed. Moreover, among the physical qubits in the code, there are always several qubits that exhibit higher intrinsic single-qubit gate error rates. Based on these observations, we primarily consider two types of constraints and evaluate their effects in our experiment. For a rotated surface code with distance d, the explicit constraints are as follows:

- Locality: Errors occur within a set containing $\frac{d^2-1}{2}$ randomly chosen qubits. The other qubits are set to be error-free.
- Discreteness: Uniformly divide the total d^2 qubits into d segments, within each segment of d qubits there exists no more than one error.

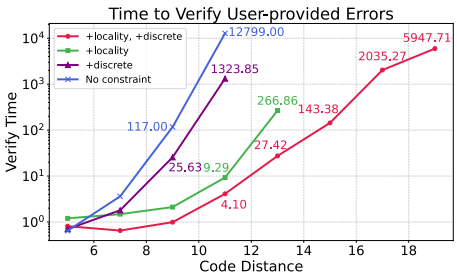

Fig. 7. Time consumed to verify the correctness of surface code with distances ranging from 5 to 19.

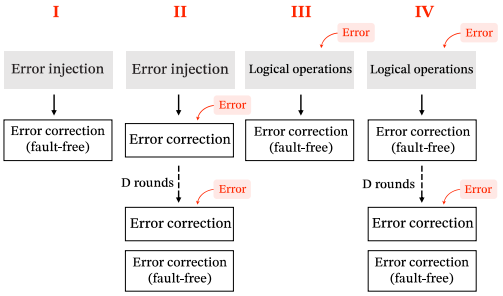

Fig. 8. Realistic fault-tolerant scenarios that are supported for verification.

The other experimental settings are the same as those in the first experiment.

Fig. 7 illustrates the experimental results of verification with user-provided constraints. We separately assessed the results and the time consumed for verification with the locality constraint, the discreteness constraint, and both combined. We will take the average time for five runs for locality constraints since the locations of errors are randomly chosen. Obviously both constraints contribute to the improvement of efficiency, yet yield limited improvements if only one of them is imposed; When the constraints are imposed simultaneously, we can verify the $d = 19$ surface code which has 361 qubits within $\sim$ 100 minutes.

Comparison with Stim [36]. Stim is currently the most widely used and state-of-the-art stabilizer circuit simulator that provides fast performance in sampling and testing large-scale QEC codes. However, simply using Stim in sampling or testing does not provide a complete check for QEC codes, as it will require a large number of samples. For example, we can verify a $d = 19$ surface code with 361 qubits in the presence of both constraints, which require testing on $\sum_{i=0}^{18} \binom{18}{i}(18)^i = 19^{18} \approx 2^{76}$ samples that are beyond the testing scope.

7.3 Towards Fault-tolerant Implementation of Operations in Quantum Hardware

We are interested in whether our tool has the capability to verify the correctness of fault-tolerant implementations for certain logical operations or measurements. In Fig. 8 we conclude the realistic fault-tolerant computation scenarios our tools support. In particular, we write down the programs of two examples encoded by Steane code and verify the correctness formulas in our tool. The examples are stated as follows:

(1) A fault-tolerant logical GHZ state preparation.
(2) An error from the previous cycle remains uncorrected and got propagated through a logical CNOT gate.

We provide the programs used in the experiment in Fig. 9 and Fig. 10. The program $\mathbf{Steane}(E)_i$ denotes an error correction process over i^{th} logical qubit.

7.4 A Benchmark for Qubit Stabilizer Codes

We further provide a benchmark of 14 qubit stabilizer codes selected from the broader quantum error correction code family, as illustrated in Table 3. We require the selected codes to be qubit-based and have a well-formed parity-check matrix. For codes that lack an explicit parity-check matrix, we construct the stabilizer generators and logical operators based on their mathematical construction and verify the correctness of the implementations. For codes with odd distances, we verify the correctness of their program implementations in the context of accurate decoding and correction.

for $i \in 8 \cdots 14$ **do** q_i *= H **end** ;
for $i \in 1 \cdots 3$ **do** **Steane**$(E)_i$ **end** ;
for $i \in 8 \cdots 14$ **do** q_i, q_{i-7} *= $CNOT$ **end** ;
for $i \in 1 \cdots 7$ **do** q_i, q_{i+7} *= $CNOT$ **end** ;
for $i \in 1 \cdots 3$ **do** **Steane**$(E)_i$ **end**

Fig. 9. QEC for logical GHZ state preparation.

for $i \in 1 \cdots 7$ **do** $[ep_{(i)}]q_i$ *= U **end** ;
for $i \in 1 \cdots 7$ **do** q_i, q_{i+7} *= $CNOT$ **end** ;
for $i \in 1 \cdots 2$ **do** **Steane**$(E)_i$ **end**

Fig. 10. QEC for logical CNOT gate with propagated errors.

Table 3. A benchmark of qubit stabilizer codes with logical-free scenario (*EMC*) considered in Table 4. We report their parameters $[[n, k, d]]$ and the properties we verified with the time spent. Parameters with variables indicate that this code has a scalable construction. If the exact d is unknown, we provide an estimation given by our tool in the bracket.

Target: Accurate Correction		
Code Name	Parameters	Verify time(s)
Steane code [72]	$[[7, 1, 3]]$	0.095
Surface code [30] ($d = 11$)	$[[d^2, 1, d]]$	12799
Six-qubit code [20]	$[[6, 1, 3]]$	0.252
Quantum dodecacode [20]	$[[11, 1, 5]]$	0.587
Reed-Muller code [73] ($r = 8$)	$[[2^r - 1, 1, 3]]$	1868.56
XZZX surface code [13] ($d_x = 9, d_z = 11$)	$[[d_x \times d_z, 1, \min(d_x, d_z)]]$	1067.16
Gottesman code [37] ($r = 8$)	$[[2^r, 2^r - r - 2, 3]]$	587.00
Honeycomb code [51] ($d = 5$)	$[[19, 1, 5]]$	1.55
Target: Detection		
Tanner Code I [55]	$[[343, 31, d \geq 4]]$	7086.36
Tanner Code II [55]	$[[125, 53, 4]]$	1667.81
Hypergraph Product [18, 48, 79]	$[[98, 18, 4]]$	289.37
Error-Detection codes		
3D basic color code [50] ($d_z = 2$)	$[[8, 3, 2]]$	2.88
Triorthogonal code [17] ($k = 64$)	$[[3k + 8, k, d_x = 6, d_z = 2]]$	144.94
Carbon code [38]	$[[12, 2, 4]]$	4.80
Campbell-Howard code [22] ($k = 2$)	$[[6k + 2, 3k, 2]]$	3.05

However, some codes have even code distances, including examples such as the 3D $[[8, 3, 2]]$ color code [50] and the Campbell-Howard code [22], which are designed to implement non-Clifford gates like the T-gate or Toffoli gate with low gate counts. These codes have a distance of 2, allowing error correction solely through post-selection rather than decoding. In such cases, the correctness of the program implementations is ensured by verifying that the code can successfully detect any single-qubit Pauli error. We list these error-detection codes at the end of Table 3.

8 Related Work

In addition to the works compared in Section 1, we briefly outline verification techniques for quantum programs and other works that may be used to check QEC programs.

Formal verification with program logic. Program logic, as a well-explored formal verification technique, plays a crucial role in the verification of quantum programs. Over the past decades, numerous studies have focused on developing Hoare-like logic frameworks for quantum programs [6, 19, 24, 33, 45]. *Assertion Logic.* [67, 68, 83] began utilizing stabilizers as atomic propositions.

[80] proposed a hybrid quantum logic in which classical variables are embedded as special quantum variables. Although slightly different, this approach is essentially isomorphic to our interpretation of logical connectives. *Program Logic.* Several works have established sound and relatively complete (hybrid) quantum Hoare logics, both satisfaction-based [34, 85] and projection-based [91]. However, these works did not introduce (countable) assertion syntax, meaning their completeness proofs do not account for the expressiveness of the weakest (liberal) preconditions. [74, 82, 83] focus on reasoning about stabilizers and QEC code, with our substitution rules for unitary statements drawing inspiration from their work. *Program logic in the verification of QEC codes and fault-tolerant computing.* Quantum relational logic [8, 58, 81] is designed for reasoning about relationships, making it well-suited for verifying functional correctness by reasoning equivalence between ideal programs and programs with errors. Quantum separation logic [40, 52, 57, 89], through the application of separating conjunctions, enables local and modular reasoning about large-scale programs, which is highly beneficial for verifying large-scale fault-tolerant computing. Abstract interpretation [87] uses a set of local projections to characterize properties of global states, thereby breaking through the exponential barrier. It is worth investigating whether local projections remain effective for QEC codes.

Symbolic techniques for quantum computation. General quantum program testing and debugging methods face the challenge of excessive test cases when dealing with QEC programs, which makes them inefficient. Symbolic techniques have been introduced into quantum computing to address this issue [9, 23, 27, 32, 44, 76]. Some of these works aim to speed up the simulation process without providing complete verification of quantum programs, while others are designed for quantum circuits and do not support the control flows required in QEC programs. The only approach capable of handling large-scale QEC programs is the recent work that proposed symbolic stabilizers [32]. However, this framework is primarily designed to detect bugs in the error correction process that do not involve logical operations and do not support non-Clifford gates.

Mechanized approach for quantum programming. The mechanized approach offers significant advantages in terms of reliability and automation, leading to the development of several quantum program verification tools in recent years (see recent reviews [26, 56]). Our focus is primarily on tools that are suitable for writing and reasoning about quantum error correction (QEC) code at the circuit level. *Matrix-based approaches.* QWIRE [63, 66] and SQIR [41] formalize circuit-like programming languages and low-level languages for intermediate representation, utilizing a density matrix representation of quantum states. These approaches have been extended to develop verified compilers [65] and optimizers [41]. *Graphical-based approaches.* [53, 54, 71], provide a certified formalization of the ZX-calculus [28, 46], which is effective for verifying quantum circuits through a flexible graphical structure. *Automated verification.* QBRICKS [25] offers a highly automated verification framework based on the Why3 [12] prover for circuit-building quantum programs, employing path-sum representations of quantum states [4]. *Theory formalization.* Ongoing libraries are dedicated to the formalization of quantum computation theories, such as QuantumLib [92], Isabelle Marries Dirac (IMD) [14, 15], and CoqQ [90]. QuantumLib is built upon the Coq proof assistant and utilizes the Coq standard library as its mathematical foundation. IMD is implemented in Isabelle/HOL, focusing on quantum information theory and quantum algorithms. CoqQ is written in Coq and provides comprehensive mathematical theories for quantum computing based on the Mathcomp library [59, 78]. Among these, CoqQ has already formalized extensive theories of subspaces, making it the most suitable choice for our formalization of program logic.

Functionalities of verification tools for QEC programs. Besides the comparison of theoretical work on program logic and other verification methods, we also compare the functionalities of our tool Veri-QEC with those of other verification tools for QEC programs. We summarize the functionalities

Table 4. Comparison of scenarios and functionalities between Veri-QEC and other tools. For scenarios, we denote $\bar{L}$ for logical gate implementation, E for error injection, M for measurement (error detection), C (C_E) for error correction (with error injection). We further identify three functionalities, **C** for general verification of correctness, **R** for reporting bugs, and **F** for fixed errors, that evaluated by ▲ if implemented, ○ if potentially supported but not yet implemented and – if cannot handle or beyond design. n/a indicates that **F** is unavailable in the error-free scenario.

Tools / Scenarios	Veri-QEC			VERITA [82, 83]			QuantumSE [32]			Stim [36]		
Functionality	**C**	**R**	**F**	**C**	**R**	**F**	**C**	**R**	**F**	**C**	**R**	**F**
error-free ($\bar{L}$)	▲	○	n/a	▲	○	n/a	○	○	n/a	○	○	n/a
logical-free (EMC)	▲	○	○	–	–	▲	▲	▲	○	–	–	▲
error in correction step ($\bar{L}MC_E$)	▲	○	○	–	–	○	▲	▲	○	–	–	▲
one cycle ($E\bar{L}EMC$)	▲	○	○	–	–	▲	▲	▲	○	–	–	▲
multi cycles ($E\bar{L}EMCE\bar{L}EMC\cdots$)	▲	○	○	–	–	▲	○	○	○	–	–	▲

of the tools in Table 4. VERITA [82, 83] adopts a logic-based approach to verify the implementation of logical operations with fixed errors. `QuantumSE` [32] is tailored for efficiently reporting bugs in QEC programs and shows potential in handling logical Clifford operations. Stim [36] employs a simulation-based approach, offering robust performance across diverse fault-tolerant scenarios but limited to fixed errors. Our tool Veri-QEC is designed for both general verification and partial verification under user-provided constraints, supporting all aforementioned scenarios.

9 Discussion and Future Works

In this paper, we propose an efficient verification framework for QEC programs, within which we define the assertion logic along with program logic and establish a sound proof system. We further develop an efficient method to handle verification conditions of QEC programs. We implement our QEC verifiers at two levels: a verified QEC verifier and a Python-based automated QEC verifier.

Our work still has some limitations. First of all, the gate set we adopt in the programming language is restricted, and the current projection-based logic is unable to reason about probabilities. Last but not least, while our proof system is sound, its completeness- especially for programs with loops- remains an open question.

Given the existing limitations, some potential directions for future advancements include:

(1) *Addressing the completeness issue of the proof system.* We are able to prove the (relative) completeness of our proof system for finite QEC programs without infinite loops. However, it is still open whether the proof system is complete for programs with while-loops. This issue is indeed related to the next one.
(2) *Extending the gate set to enhance the expressivity of program logic.* The Clifford + T gate set we use in the current program logic is universal but still restricted in practical applications. It is desirable to extend the syntax of factors and assertions for the gate sets beyond Clifford + T.
(3) *Generalizing the logic to satisfaction-based approach.* Since any Hermitian operator can be written as linear combinations of Pauli expressions, our logic has the potential to incorporate so-called satisfaction-based approach with Hermitian operators as quantum predicates, which helps to reason about the success probabilities of quantum QEC programs.
(4) *Exploring approaches to implementing an automatic verified verifier.* The last topic is to explore tools like F^* [60, 75], a proof-oriented programming language based on SMT, for incorporating the formally verified verifier and the automatic verifier described in this paper into a single unified solution.

Acknowledgement

We thank Bonan Su for kind discussions regarding on crafting the introduction section and Huiping Lin for for the revisions made to the introduction of stabilizer codes. In addition, we thank anonymous referees for helpful comments and suggestions. This research was supported by the National Key R&D Program of China under Grant No. 2023YFA1009403.

Data Availability Statement

The code for of this work (both the Coq formalization and the automatic verifier Veri-QEC) is available at https://github.com/Chesterhuang1999/Veri-qec, or at https://doi.org/10.5281/zenodo.15248774 (evaluated artifact [42]). The appendices are provided as the supplementary material, or see our extended version [43].

References

[1] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Phys. Rev. A* 70 (Nov 2004), 052328. Issue 5. doi:10.1103/PhysRevA.70.052328

[2] Rajeev Acharya, Dmitry A. Abanin, Laleh Aghababaie-Beni, Igor Aleiner, Google Quantum AI, et al. 2025. Quantum error correction below the surface code threshold. *Nature* 638, 8052 (01 Feb 2025), 920–926. doi:10.1038/s41586-024-08449-y

[3] Rajeev Acharya, Igor Aleiner, Richard Allen, Trond I. Andersen, Google Quantum AI, et al. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614, 7949 (01 Feb 2023), 676–681. doi:10.1038/s41586-022-05434-1

[4] Matthew Amy. 2018. Towards Large-scale Functional Verification of Universal Quantum Circuits. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018 (EPTCS, Vol. 287)*, Peter Selinger and Giulio Chiribella (Eds.). 1–21. doi:10.4204/EPTCS.287.1

[5] Simon Anders and Hans J. Briegel. 2006. Fast simulation of stabilizer circuits using a graph-state representation. *Phys. Rev. A* 73 (Feb 2006), 022334. Issue 2. doi:10.1103/PhysRevA.73.022334

[6] Alexandru Baltag and Sonja Smets. 2004. The logic of quantum programs. *Proc. QPL* (2004), 39–56. https://philsci-archive.pitt.edu/1799/

[7] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, et al. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[8] Gilles Barthe, Justin Hsu, Mingsheng Ying, Nengkun Yu, and Li Zhou. 2019. Relational Proofs for Quantum Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 21 (December 2019), 29 pages. doi:10.1145/3371089

[9] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. 2023. SymQV: Automated Symbolic Verification Of Quantum Programs. In *Formal Methods: 25th International Symposium, FM 2023*. Springer-Verlag, 181–198. doi:10.1007/978-3-031-27481-7_12

[10] Dolev Bluvstein, Simon J. Evered, Alexandra A. Geim, Sophie H. Li, Hengyun Zhou, et al. 2024. Logical Quantum Processor Based on Reconfigurable Atom Arrays. *Nature* 626, 7997 (Feb. 2024), 58–65. doi:10.1038/s41586-023-06927-3

[11] Dolev Bluvstein, Harry Levine, Giulia Semeghini, Tout T. Wang, Sepehr Ebadi, et al. 2022. A quantum processor based on coherent transport of entangled atom arrays. *Nature* 604, 7906 (01 Apr 2022), 451–456. doi:10.1038/s41586-022-04592-6

[12] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wroclaw, Poland, 53–64. https://inria.hal.science/hal-00790310

[13] J. Pablo Bonilla Ataides, David K. Tuckett, Stephen D. Bartlett, Steven T. Flammia, and Benjamin J. Brown. 2021. The XZZX surface code. *Nature Communications* 12, 1 (12 Apr 2021), 2172. doi:10.1038/s41467-021-22274-1

[14] Anthony Bordg, Hanna Lachnitt, and Yijun He. 2020. Isabelle marries dirac: A library for quantum computation and quantum information. *Archive of Formal Proofs* (2020).

[15] Anthony Bordg, Hanna Lachnitt, and Yijun He. 2021. Certified Quantum Computation in Isabelle/HOL. *Journal of Automated Reasoning* 65, 5 (01 June 2021), 691–709. doi:10.1007/s10817-020-09584-7

[16] Sergey Bravyi, Andrew W. Cross, Jay M. Gambetta, Dmitri Maslov, Patrick Rall, et al. 2024. High-threshold and low-overhead fault-tolerant quantum memory. *Nature* 627, 8005 (01 Mar 2024), 778–782. doi:10.1038/s41586-024-07107-7

[17] Sergey Bravyi and Jeongwan Haah. 2012. Magic-state distillation with low overhead. *Phys. Rev. A* 86 (Nov 2012), 052329. Issue 5. doi:10.1103/PhysRevA.86.052329

[18] Nikolas P. Breuckmann and Jens Niklas Eberhardt. 2021. Quantum Low-Density Parity-Check Codes. *PRX Quantum* 2 (Oct 2021), 040101. Issue 4. doi:10.1103/PRXQuantum.2.040101

[19] Olivier Brunet and Philippe Jorrand. 2004. Dynamic Quantum Logic For Quantum Programs. *International Journal of Quantum Information* 02, 01 (2004), 45–54. doi:10.1142/S0219749904000067

[20] A.R. Calderbank, E.M. Rains, P.M. Shor, and N.J.A. Sloane. 1998. Quantum error correction via codes over GF(4). *IEEE Transactions on Information Theory* 44, 4 (1998), 1369–1387. doi:10.1109/18.681315

[21] A. R. Calderbank and Peter W. Shor. 1996. Good quantum error-correcting codes exist. *Phys. Rev. A* 54 (Aug 1996), 1098–1105. Issue 2. doi:10.1103/PhysRevA.54.1098

[22] Earl T. Campbell and Mark Howard. 2017. Unified framework for magic state distillation and multiqubit gate synthesis with reduced resource cost. *Phys. Rev. A* 95 (Feb 2017), 022316. Issue 2. doi:10.1103/PhysRevA.95.022316

[23] Jacques Carette, Gerardo Ortiz, and Amr Sabry. 2023. Symbolic Execution of Hadamard-Toffoli Quantum Circuits. In *Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM 2023)*. Association for Computing Machinery, 14–26. doi:10.1145/3571786.3573018

[24] R. Chadha, P. Mateus, and A. Sernadas. 2006. Reasoning About Imperative Quantum Programs. *Electronic Notes in Theoretical Computer Science* 158 (2006), 19–39. doi:10.1016/j.entcs.2006.04.003 Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII).

[25] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-building Quantum Programs. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 148–177. doi:10.1007/978-3-030-72019-3_6

[26] Christophe Chareton, Sébastien Bardin, Dong Ho Lee, Benoît Valiron, Renaud Vilmart, and Zhaowei Xu. 2023. Formal Methods for Quantum Algorithms. In *Handbook of Formal Analysis and Verification in Cryptography*. CRC Press, 319–422. https://cea.hal.science/cea-04479879

[27] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023. An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits. *Proc. ACM Program. Lang.* 7, PLDI, Article 156 (jun 2023), 26 pages. doi:10.1145/3591270

[28] Bob Coecke and Ross Duncan. 2011. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics* 13, 4 (apr 2011), 043016. doi:10.1088/1367-2630/13/4/043016

[29] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[30] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. 2002. Topological quantum memory. *J. Math. Phys.* 43, 9 (2002), 4452–4505. doi:10.1063/1.1499754

[31] Ellie D'hondt and Prakash Panangaden. 2006. Quantum weakest preconditions. *Mathematical Structures in Computer Science* 16, 3 (2006), 429–451. doi:10.1017/S0960129506005251

[32] Wang Fang and Mingsheng Ying. 2024. Symbolic Execution for Quantum Error Correction Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 189 (June 2024), 26 pages. doi:10.1145/3656419

[33] Yuan Feng, Runyao Duan, Zhengfeng Ji, and Mingsheng Ying. 2007. Proof rules for the correctness of quantum programs. *Theoretical Computer Science* 386, 1 (2007), 151–166. doi:10.1016/j.tcs.2007.06.011

[34] Yuan Feng and Mingsheng Ying. 2021. Quantum Hoare Logic with Classical Variables. *ACM Transactions on Quantum Computing* 2, 4, Article 16 (Dec. 2021), 43 pages. doi:10.1145/3456877

[35] David J Foulis and Mary K Bennett. 1994. Effect algebras and unsharp quantum logics. *Foundations of physics* 24, 10 (1994), 1331–1352. doi:10.1007/BF02283036

[36] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (July 2021), 497. doi:10.22331/q-2021-07-06-497

[37] Daniel Gottesman. 1997. Stabilizer Codes and Quantum Error Correction. arXiv:quant-ph/9705052 [quant-ph]

[38] Markus Grassl and Martin Roetteler. 2013. Leveraging automorphisms of quantum codes for fault-tolerant quantum computation. In *2013 IEEE International Symposium on Information Theory*. 534–538. doi:10.1109/ISIT.2013.6620283

[39] Ian Grout. 2011. *Digital systems design with FPGAs and CPLDs*. Elsevier.

[40] Kesha Hietala, Sarah Marshall, Robert Rand, and Nikhil Swamy. 2022. Q*: Implementing Quantum Separation Logic in F*. *Programming Languages for Quantum Computing (PLanQC) 2022 Poster Abstract* (2022). https://khieta.github.io/files/drafts/qstar-planqc22.pdf

[41] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for Quantum circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan. 2021), 29 pages. doi:10.1145/3434318

[42] Qifan Huang, Li Zhou, Wang Fang, Mengyu Zhao, and Mingsheng Ying. 2025. *Artifact for 'Efficient Formal Verification of Quantum Error Correcting Programs'*. doi:10.5281/zenodo.15248774

[43] Qifan Huang, Li Zhou, Wang Fang, Mengyu Zhao, and Mingsheng Ying. 2025. Efficient Formal Verification of Quantum Error Correcting Programs. arXiv:2504.07732 [cs.PL]

[44] Yipeng Huang, Steven Holtzen, Todd Millstein, Guy Van den Broeck, and Margaret Martonosi. 2021. Logical Abstractions for Noisy Variational Quantum Algorithm Simulation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing

Machinery, 456–472. doi:10.1145/3445814.3446750
[45] Yoshihiko Kakutani. 2009. A Logic for Formal Verification of Quantum Programs. In *Advances in Computer Science - ASIAN 2009. Information Security and Privacy*, Anupam Datta (Ed.). Springer, Berlin, Heidelberg, 79–93. doi:10.1007/978-3-642-10622-4_7
[46] Aleks Kissinger and John van de Wetering. 2019. PyZX: Large Scale Automated Diagrammatic Reasoning. In *Proceedings 16th International Conference on Quantum Physics and Logic, QPL 2019, Chapman University, Orange, CA, USA, June 10-14, 2019 (EPTCS, Vol. 318)*, Bob Coecke and Matthew Leifer (Eds.). 229–241. doi:10.4204/EPTCS.318.14
[47] A Yu Kitaev. 1997. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys* 52, 6 (dec 1997), 1191. doi:10.1070/RM1997v052n06ABEH002155
[48] Alexey A. Kovalev and Leonid P. Pryadko. 2012. Improved quantum hypergraph-product LDPC codes. In *2012 IEEE International Symposium on Information Theory Proceedings*. IEEE, 348–352. doi:10.1109/isit.2012.6284206
[49] Karl Kraus, Arno Böhm, John D Dollard, and WH Wootters. 1983. *States, Effects, and Operations Fundamental Notions of Quantum Theory: Lectures in Mathematical Physics at the University of Texas at Austin*. Springer.
[50] Aleksander Kubica, Beni Yoshida, and Fernando Pastawski. 2015. Unfolding the color code. *New Journal of Physics* 17, 8 (aug 2015), 083026. doi:10.1088/1367-2630/17/8/083026
[51] Andrew J. Landahl, Jonas T. Anderson, and Patrick R. Rice. 2011. Fault-tolerant quantum computing with color codes. arXiv:1108.5738 [quant-ph]
[52] Xuan-Bach Le, Shang-Wei Lin, Jun Sun, and David Sanan. 2022. A Quantum Interpretation of Separating Conjunction for Local Reasoning of Quantum Programs Based on Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 36 (jan 2022), 27 pages. doi:10.1145/3498697
[53] Adrian Lehmann, Ben Caldwell, and Robert Rand. 2022. VyZX : A Vision for Verifying the ZX Calculus. arXiv:2205.05781 [quant-ph]
[54] Adrian Lehmann, Ben Caldwell, Bhakti Shah, and Robert Rand. 2023. VyZX: Formal Verification of a Graphical Quantum Language. arXiv:2311.11571 [cs.PL]
[55] Anthony Leverrier and Gilles Zémor. 2022. Quantum Tanner codes. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 872–883. doi:10.1109/FOCS54457.2022.00117
[56] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. 2023. Formal Verification of Quantum Programs: Theory, Tools, and Challenges. 5, 1, Article 1 (dec 2023), 35 pages. doi:10.1145/3624483
[57] Liyi Li, Mingwei Zhu, Rance Cleaveland, Alexander Nicolellis, Yi Lee, Le Chang, and Xiaodi Wu. 2024. Qafny: A Quantum-Program Verifier. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 24:1–24:31. doi:10.4230/LIPIcs.ECOOP.2024.24
[58] Yangjia Li and Dominique Unruh. 2021. Quantum Relational Hoare Logic with Expectations. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 198)*, Nikhil Bansal, Emanuela Merelli, and James Worrell (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 136:1–136:20. doi:10.4230/LIPIcs.ICALP.2021.136
[59] Assia Mahboubi and Enrico Tassi. 2022. *Mathematical Components*. Zenodo. doi:10.5281/zenodo.7118596
[60] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, et al. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 30–59. doi:10.1007/978-3-030-17184-1_2
[61] M.A. Nielsen and I.L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press.
[62] D. Nigg, M. Müller, E. A. Martinez, P. Schindler, M. Hennrich, T. Monz, M. A. Martin-Delgado, and R. Blatt. 2014. Quantum computations on a topologically encoded qubit. *Science* 345, 6194 (2014), 302–305. doi:10.1126/science.1253742
[63] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 846–858. doi:10.1145/3009837.3009894
[64] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. doi:10.22331/q-2018-08-06-79
[65] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2018. ReQWIRE: Reasoning about Reversible Quantum Circuits. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018 (EPTCS, Vol. 287)*, Peter Selinger and Giulio Chiribella (Eds.). 299–312. doi:10.4204/EPTCS.287.17
[66] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2017. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017. (EPTCS, Vol. 266)*, Bob Coecke and Aleks Kissinger (Eds.). 119–132. doi:10.4204/EPTCS.266.8
[67] Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. 2021. Gottesman Types for Quantum Programs. *Electronic Proceedings in Theoretical Computer Science* 340 (Sept. 2021), 279–290. doi:10.4204/eptcs.340.14

[68] Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. 2021. Static Analysis of Quantum Programs via Gottesman Types. arXiv:2101.08939 [quant-ph]
[69] C. Ryan-Anderson, J. G. Bohnet, K. Lee, D. Gresh, A. Hankin, et al. 2021. Realization of Real-Time Fault-Tolerant Quantum Error Correction. *Phys. Rev. X* 11 (Dec 2021), 041058. Issue 4. doi:10.1103/PhysRevX.11.041058
[70] C. Ryan-Anderson, N. C. Brown, M. S. Allman, B. Arkin, et al. 2022. Implementing Fault-tolerant Entangling Gates on the Five-qubit Code and the Color Code. arXiv:2208.01863 [quant-ph]
[71] Bhakti Shah, William Spencer, Laura Zielinski, Ben Caldwell, Adrian Lehmann, and Robert Rand. 2024. ViCAR: Visualizing Categories with Automated Rewriting in Coq. arXiv:2404.08163 [cs.PL]
[72] Andrew Steane. 1996. Multiple-particle interference and quantum error correction. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 452, 1954 (1996), 2551–2577. doi:10.1098/rspa.1996.0136
[73] A.M. Steane. 1999. Quantum Reed-Muller codes. *IEEE Transactions on Information Theory* 45, 5 (1999), 1701–1703. doi:10.1109/18.771249
[74] Aarthi Sundaram, Robert Rand, Kartik Singhal, and Brad Lackey. 2022. Hoare meets Heisenberg: A Lightweight Logic for Quantum Programs. http://rand.cs.uchicago.edu/files/heisenberg_logic_2023.pdf
[75] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, et al. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 256–270. doi:10.1145/2837614.2837655
[76] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, et al. 2022. Giallar: Push-Button Verification for the Qiskit Quantum Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, 641–656. doi:10.1145/3519939.3523431
[77] The Coq Development Team. 2022. The Coq Proof Assistant. doi:10.5281/zenodo.5846982
[78] The MathComp Analysis Development Team. 2024. MathComp-Analysis: Mathematical Components compliant Analysis Library. https://github.com/math-comp/analysis. Since 2017. Version 1.0.0.
[79] Jean-Pierre Tillich and Gilles Zémor. 2014. Quantum LDPC Codes With Positive Rate and Minimum Distance Proportional to the Square Root of the Blocklength. *IEEE Transactions on Information Theory* 60, 2 (2014), 1193–1202. doi:10.1109/TIT.2013.2292061
[80] Dominique Unruh. 2019. Quantum Hoare Logic with Ghost Variables. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. doi:10.1109/LICS.2019.8785779
[81] Dominique Unruh. 2019. Quantum relational Hoare logic. *Proc. ACM Program. Lang.* 3, POPL, Article 33 (Jan. 2019), 31 pages. doi:10.1145/3290346
[82] Anbang Wu. 2024. *Towards Large-Scale Quantum Computing*. Ph. D. Dissertation. UC Santa Barbara. https://www.proquest.com/dissertations-theses/towards-large-scale-quantum-computing/docview/3050756793/se-2
[83] Anbang Wu, Gushu Li, Hezi Zhang, Gian Giacomo Guerreschi, Yuan Xie, and Yufei Ding. 2021. QECV: Quantum Error Correction Verification. arXiv:2111.13728 [quant-ph]
[84] Xiaosi Xu, Simon Benjamin, Jinzhao Sun, Xiao Yuan, and Pan Zhang. 2023. A Herculean task: Classical simulation of quantum computers. arXiv:2302.08880 [quant-ph]
[85] Mingsheng Ying. 2012. Floyd–hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.* 33, 6, Article 19 (Jan. 2012), 49 pages. doi:10.1145/2049706.2049708
[86] Mingsheng Ying. 2024. *Foundations of Quantum Programming* (second edition ed.). Morgan Kaufmann.
[87] Nengkun Yu and Jens Palsberg. 2021. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 542–558. doi:10.1145/3453483.3454061
[88] Youwei Zhao, Yangsen Ye, He-Liang Huang, Yiming Zhang, Dachao Wu, et al. 2022. Realization of an Error-Correcting Surface Code with Superconducting Qubits. *Phys. Rev. Lett.* 129 (Jul 2022), 030501. Issue 3. doi:10.1103/PhysRevLett.129.030501
[89] Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. 2021. A Quantum Interpretation of Bunched Logic amp; Quantum Separation Logic. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–14. doi:10.1109/LICS52264.2021.9470673
[90] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. 2023. CoqQ: Foundational Verification of Quantum Programs. *Proc. ACM Program. Lang.* 7, POPL, Article 29 (jan 2023), 33 pages. doi:10.1145/3571222
[91] Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. An applied quantum Hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1149–1162. doi:10.1145/3314221.3314584
[92] Jacob Zweifler, Kesha Hietala, and Robert Rand. 2022. *QuantumLib: A Library for Quantum Computing in Coq.*

Received 2024-11-15; accepted 2025-03-06