A Survey on Testing and Analysis of Quantum Software

MATTEO PALTENGHI, Department of Computer Science, University of Stuttgart, Germany
MICHAEL PRADEL, Department of Computer Science, University of Stuttgart, Germany

Quantum computing is getting increasing interest from both academia and industry, and the quantum software landscape has been growing rapidly. The quantum software stack comprises quantum programs, implementing algorithms, and platforms like IBM Qiskit, Google Cirq, and Microsoft Q#, enabling their development. To ensure the reliability and performance of quantum software, various techniques for testing and analyzing it have been proposed, such as test generation, bug pattern detection, and circuit optimization. However, the large amount of work and the fact that work on quantum software is performed by several research communities, make it difficult to get a comprehensive overview of the existing techniques. In this work, we provide an extensive survey of the state of the art in testing and analysis of quantum software. We discuss literature from several research communities, including quantum computing, software engineering, programming languages, and formal methods. Our survey covers a wide range of topics, including expected and unexpected behavior of quantum programs, testing techniques, program analysis approaches, optimizations, and benchmarks for testing and analyzing quantum software. We create novel connections between the discussed topics and present them in an accessible way. Finally, we discuss key challenges and open problems to inspire future research. This sentence was added to verify the local `latexdiff` workflow.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computer systems organization → Quantum computing**; • **Software and its engineering → Software testing and debugging**.

1 Introduction

Quantum computing is a new paradigm of computation that promises to find the solution to problems that are intractable for classical computers, or provide a speed up over the classical counterpart, such as factoring large numbers [98] or searching unsorted databases [30]. The field is attracting increasing attention from researchers, governments, and industry. While many efforts aim at improving the hardware for quantum computation, there also is a large stream of work on the equally important problem of creating quantum software. The reliability of this software is crucial, because unreliable or inefficient software can hinder or even nullify progress on the hardware side.

Developing reliable and efficient quantum computing software poses several unique challenges for the developers. First, unlike classical computers that process bits, which can be either 0 or 1, quantum computers process qubits, which can be in a superposition of 0 and 1. Besides offering potential to solve new types of problems more efficiently, this leads to a much larger search space when testing possible inputs for quantum programs, stressing the need for efficient testing techniques. Second, quantum programs use a probabilistic model of computation, where the output of a quantum program is non-deterministic, leading to possibly different outputs for the same input. This is in contrast to the vast majority of classical programs and makes it harder to check the correctness of a quantum program. Third, quantum programs exhibit properties typical of quantum mechanics, which are often unintuitive, such as entanglement,

Authors' Contact Information: Matteo Paltenghi, mattepalte@live.it, Department of Computer Science, University of Stuttgart, Germany; Michael Pradel, michael@binaervarianz.de, Department of Computer Science, University of Stuttgart, Germany.

Manuscript submitted to ACM

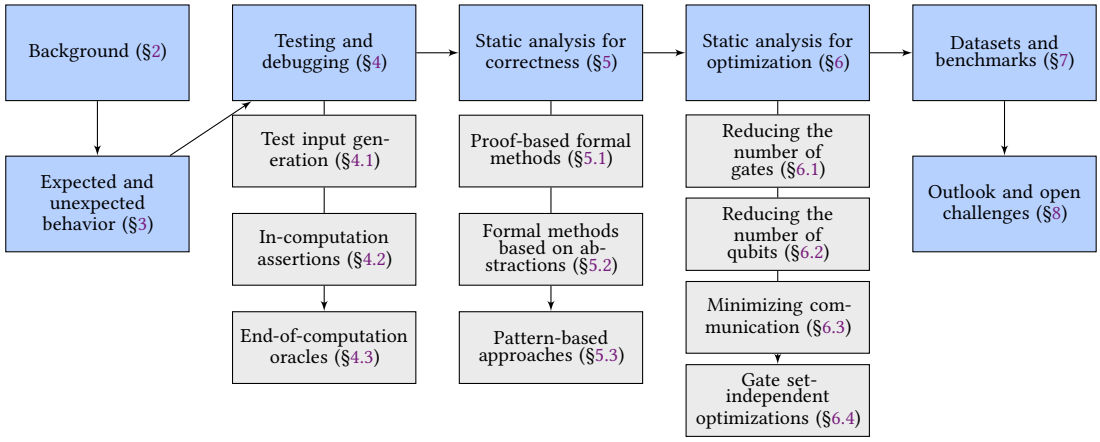


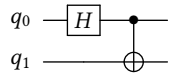
Fig. 1. Overview of the topics covered in the survey.

superposition, and interference, which can lead to bugs that are hard to find and fix. Indeed, recent work [62, 77] has documented that the percentage of *quantum-specific bugs* in quantum software constitutes a significant portion ranging from 40% to 80% of the total bugs in the software. Fourth, due to the *no-cloning theorem* [114], we cannot copy a qubit's state, making it impossible to observe the state without destroying it. This makes step-by-step debugging of quantum programs extremely challenging, unlike classical programs.

To increase the reliability of the quantum software stack, many testing and analysis techniques have been proposed, including testing approaches to detect bugs in quantum software and program analyses to check its correctness or optimize its performance. Many optimization techniques lie at the frontier between software and hardware, as they aim to optimize the quantum program for the underlying hardware. This survey aims to provide a comprehensive overview of the current state of the art in testing and analysis of quantum software, highlighting advancements in the field and open challenges that need to be addressed in the future.

In this survey, we focus on testing and analysis of quantum software, including work that aims at improving either the reliability or the performance of quantum software. Given the wide range of quantum computing research topics, we focus on software manipulating qubits based on the circuit model, where quantum programs are represented as sequences of quantum gates. While we acknowledge that simulators can be used for testing quantum programs, we exclude extensive discussion of simulation approaches because their primary goal is to provide an alternative execution environment, not to analyze or test the programs themselves. In total, our survey covers 102 pieces of work published or publicly shared as pre-print between 2007 and 2023.

To contextualize this paper within existing research, we compare it with existing surveys addressing various facets of the field. Some existing surveys discuss software engineering practices within the quantum computing field [127], the various software components within the quantum computing ecosystem [96], and existing quantum programming languages [34]. Other work covers work on benchmarking the hardware components of the quantum computing stack [93], applications of quantum computing in combinatorial optimization [28], and work on verifying whether a quantum state or operation possesses certain properties [66]. A recent article discusses 16 testing techniques aimed at quantum software [27], i.e., a fraction of those we discuss here. Overall, despite the large amount of work done on



(a) Quantum circuit.

state	prob	bit string
$\frac{1}{\sqrt{2}}$	50%	00
0	0	01
0	0	10
$\frac{1}{\sqrt{2}}$	50%	11

(b) Final state of the circuit in (a).

$$\begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle$$

(c) Quantum state as linear combination of basis states.

Fig. 2. Examples of a quantum circuit (a) that creates an entangled state (b) for which we have 50% probability to observe the bitstring 00 and 50% to observe 11. In (c), we show how the output can be represented as a linear combination of the basis states.

testing and analyzing quantum software, there currently is no comprehensive survey that summarizes this field in a way accessible to interested outsiders.

As shown in Figure 1, our survey is structured into several sections. We start by introducing the basic concepts of quantum computing and its software stack (Section 2). Next is a section on expected and unexpected behavior (Section 3), where we present ways to specify what a quantum program should do. Following this, Section 4 covers different testing techniques and debugging tools. Subsequently, Section 5 and 6 discuss work on static analysis. Specifically, Section 5 focused on correctness, i.e., methods that aim to detect bugs in quantum software or prove a program's correctness with respect to some specification. In contrast, Section 6 focuses on optimization, i.e., methods that aim to optimize quantum software, e.g., by reducing the number of qubits, the number of gates or improving the mapping to the underlying hardware. We also discuss datasets and benchmarks that are available for evaluating approaches on quantum software testing and analysis (Section 7). Finally, we conclude with a discussion of open research challenges and possible directions for future work (Section 8).

2 Background

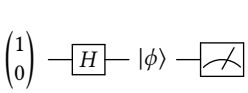
We introduce essential notation and concepts for this survey. First, we cover fundamental quantum computing concepts, including the quantum model of computation and state representation. Next, we discuss the quantum computing software stack, including platform and application code.

2.1 Quantum Computing Fundamentals

Quantum Model of Computation. Quantum programs are often represented as circuits, akin to digital logic circuits. A quantum circuit is a sequence of gates applied to registers containing *qubits*, the fundamental unit of quantum information, similar to classical bits. Unlike classical bits that are either 0 or 1, qubits can be in both states 0 and 1 simultaneously, known as *superposition*. Figure 2a shows a quantum circuit, where qubits are lines and gates are boxes. The computation starts from the left with qubits initialized in a specific state, usually all-zero, and proceeds to the right as qubits are processed by a sequence of gates.

Quantum State. To allow for superposition, a quantum system's state, comprising one or more qubits, is represented by a column vector of complex numbers. Each entry, called an *amplitude*, links to the probability of observing a specific classical state upon measurement. This *state vector*, denoted in Dirac notation $|\psi\rangle$, has length 2^n, where n is the number of qubits. The set of all possible states, called *Hilbert space* $\mathcal{H}$, is defined as $\mathcal{H} = \mathbb{C}^{2^n}$, where $\mathbb{C}$ denotes complex numbers.

For a single qubit, its state is a vector of two complex numbers. In its general form, this state can be written as a superposition of two basis states, $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$. Here, $|0\rangle$ and $|1\rangle$ form an orthonormal basis of the Hilbert space $\mathcal{H}$,



(a) Application of an H gate to the state $(1\ 0)^T = |0\rangle$, followed by a measurement.

$$|\phi\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

(b) State vector after applying the H gate (matrix multiplication).

$$p(0) = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{2}$$

(c) Probability of obtaining the state 0 after the measurement.

Fig. 3. Circuit with a Hadamard gate and measurement (a), the corresponding output state after the application of the gate operation (b), and the result of the final measurement (c).

known as the *computational basis*. The coefficients α and β are complex numbers whose squared magnitudes give the probability of observing the corresponding basis state, satisfying $\|\psi\| = 1$.

For multiple qubits, Figure 2b shows the output state vector of the two-qubit circuit in Figure 2a. Given the two-qubit circuit, its state vector has length $2^2 = 4$, i.e., $\psi_0 = (\frac{1}{\sqrt{2}}\ 0\ 0\ \frac{1}{\sqrt{2}})^T$.[1] The first qubit q_0 is the least significant bit, and the state vector is ordered such that the first complex number corresponds to $|00\rangle$, the second to $|01\rangle$, the third to $|10\rangle$, and the fourth to $|11\rangle$. Figure 2c shows the state vector as a linear combination of classical states, where the coefficients are the amplitudes of the bit strings. A classical state is a quantum state where one basis state's amplitude is 1, and the rest are 0. When two qubits interact, they may become *entangled*, meaning their state cannot be decomposed into individual qubit state vectors. For instance, in Figure 2a, the final state (Figure 2b) cannot be decomposed, thus it is entangled.

Gates. Operations applied to qubits are *quantum gates* or *unitary operators*. A gate transforms the qubits' state vector $G : \mathcal{H} \rightarrow \mathcal{H}$ and is represented by a *unitary* matrix. A unitary matrix U satisfies $U^\dagger U = I$, where $U^\dagger$ is the conjugate transpose of U and I is the identity matrix. This ensures that any gate can be inverted by applying its conjugate transpose, making quantum computation reversible. Quantum computation is a sequence of gates applied to the state vector via matrix multiplication. For instance, applying a unitary operator U to an initial state $|\psi\rangle_0$ yields $|\psi\rangle_1 = U\,|\psi\rangle_0$ via matrix-vector multiplication. The Hadamard gate H is fundamental for creating superposition (Figure 3a), with its matrix shown in Figure 3b. Other notable gates are the single-qubit Pauli gates and the two-qubit CNOT gate. The Pauli gates (X, Y, Z, I) can represent any single-qubit operation when applied sequentially with the right coefficients. The CNOT gate flips the target qubit if the control qubit is in state $|1\rangle$, and it often entangles qubits, as in Figure 2b.

Measurement. Unlike gates, measurement operations are not reversible. A *measurement* is described by m square matrices $\{M_0, ..., M_i, .., M_m\}$ satisfying $\sum_{i=0}^{m} M_i^\dagger M_i = I$, where i indexes possible measurement outcomes. Applying the operator M_i to the state $|\psi\rangle$ yields outcome i with probability $p(i) = \langle\psi|\, M_i^\dagger M_i\, |\psi\rangle$, and the post-measurement state is $\frac{M_i|\psi\rangle}{\sqrt{p(i)}}$. For example, a single-qubit measurement in the computational basis, i.e., $\{|0\rangle, |1\rangle\}$, is described by the operators $M_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ and $M_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$. Figure 3c shows how to compute the probability of obtaining state $|0\rangle$ after applying the Hadamard gate and measuring the qubit. Measurement is a probabilistic operation that collapses the quantum state into a classical state. Once measured, the original quantum state cannot be recovered, but repeating the same measurement on the collapsed state will yield the same result. The most common measurement uses the Z-basis, related to the Pauli-Z gate, returning either $|0\rangle$ or $|1\rangle$. Other measurements, like the X-basis, return $|+\rangle$ or $|-\rangle$, where $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

[1] For space reasons, we show the column vectors as transposed row vectors.

```
1 from qiskit import QuantumCircuit, Aer, execute
2 qc = QuantumCircuit(2, 2)
3 qc.h(0)
4 qc.cx(0, 1)
5 qc.measure([0, 1], [0, 1])
6 simulator = Aer.get_backend('qasm_simulator')
7 result = execute(qc, simulator, shots=1000).result()
8 print(result.get_counts(qc))
9 # output: {'00': 502, '11': 498}
```

Fig. 4. Qiskit program generating the circuit in Figure 2 and performing measurements.

```
1 q_after:=H(q_before);
```

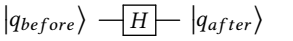


Fig. 5. Application of Hadamard gate in Silq (top) and its circuit representation (bottom)

2.2 Quantum Computing Software Stack

Quantum computing software is divided into platform code and program code. Platform code provides a mean to define, compile, and execute program code, which in turns solves specific algorithmic problems.

Platform Code. Referring to Paltenghi and Pradel [77], quantum computing platforms usually provide three main components: (a) a quantum programming language to express programs, (b) a compiler to transform programs before execution, and (c) an execution environment to run programs on simulators or quantum hardware. Examples include IBM's Qiskit, Google's Cirq, Xanadu's PennyLane, Quantinuum's TKET, and Microsoft's Q#. Platforms often integrate analysis and optimization techniques to ensure efficient and correct execution on hardware. Simulators, which precisely track quantum program states, are also useful for debugging aspects unobservable on real hardware.

Program Code. Program code is written by users to solve specific problems using the platform's infrastructure. Figure 4 shows a Qiskit program with a quantum register of two qubits and a classical register of two bits. Quantum circuits in Qiskit apply a sequence of gates (lines 3-4) to an initial all-zero state, resulting in a final state that is measured (line 5). Since measurement is probabilistic and destructive, the program runs multiple times (line 7) to derive a probability distribution reflecting the computation's outcome. The comment at line 9 underscores the probabilistic nature of quantum computing, showing roughly equal likelihood for bit strings `00` and `11`.

The representation of quantum gate computation in a program is crucial for analysis. There are two approaches: *memory-based* and *value-based* semantics. Memory-based semantics apply qubit operations by referencing the quantum register index, e.g., `qc.h(0)` (line 3), which applies a Hadamard gate to the first qubit. Developers must track qubit memory locations, similar to pointers in classical programming. This representation is used by frameworks like Qiskit, Cirq, and the OpenQASM language [22, 23]. Instead, value-based semantics represents operations as function calls that take an input quantum state variable and return an output state. For example, Figure 5 shows a Hadamard gate in the Silq language [9]. Besides Silq, this representation is also used by the intermediate representations QSSA [85] and QIRO [41], which are dialects of the Multi-Level Intermediate Representation (MLIR) [48].

3 Expected and Unexpected Behavior in Quantum Software

The goal of testing and many analyses is to check that the program behaves as expected. The following discusses how to specify what is the expected behavior of a quantum program.

3.1 Specifying Expected Behavior of Quantum Programs

Specifying the expected behavior of a quantum program is challenging due to its probabilistic nature. We present three methods: distribution-based, quantum state-based, and unitary-based specifications.

3.1.1 Distribution-Based Program Specification. Since the output of a quantum program is probabilistic, the expected behavior of a quantum program is often represented by a probability distribution [5, 110, 111].

Definition 3.1 (Distribution-based program specification). Given a quantum program P with n qubits, a distribution-based program specification S_{distr} states that, for a valid input x, the output follows a probability distribution over the possible outputs of P: $S_{distr}(P, x) = \{(y, p) \mid y \in \{0,1\}^n, p \in [0,1]\}$ where y is a possible output of P and p is the probability of y being the output of P. Note that $\sum_{(y,p)\in S_{distr}(P,x)} p = 1$, and $S_{distr}(P, x)$ is a set of size 2^n because there are 2^n possible classical outputs that can be observed with a measurement.

3.1.2 Quantum State-Based Program Specification. Another way to describe the behavior of a quantum program is via the Dirac notation of quantum mechanics [132]. The output quantum state can be described as a superposition of the basis states of the qubits.

Definition 3.2 (Quantum state-based program specification). Given a quantum program P with n qubits, a quantum state-based program specification S_{state} states that, for a valid input x, the output of P can be represented using either Dirac notation or a vector in Hilbert space: $S_{state}(P, x) = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_{2^n-1} |2^n - 1\rangle = \begin{pmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_{2^n-1} \end{pmatrix}^T$ where $\alpha_k \in \mathbb{C}$ are amplitudes and $|k\rangle$ are the basis states of the n qubits. Note that $\sum_{k=0}^{2^n-1} |\alpha_k|^2 = 1$ because the quantum state is normalized. As is common practice in quantum mechanics, we represent a possible output in decimal instead of binary, e.g., $|0\rangle = |00\cdots0\rangle$, $|1\rangle = |00\cdots1\rangle$, and $|2\rangle = |00\cdots10\rangle$.

Using Dirac notation allows applying linear algebra rules to simplify specifications. For example, the quantum state of three qubits q_1, q_2, q_3, represented as $|\psi\rangle = \frac{1}{2}|000\rangle + \frac{1}{2}|010\rangle + \frac{1}{2}|100\rangle + \frac{1}{2}|110\rangle$, can be rewritten as a tensor product of simpler states: $|\psi\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \otimes |0\rangle$. Since the last qubit is always $|0\rangle$, it can be factored out of the tensor product. As a special case of a quantum state-based program specification, the coefficients can be functions of the input value x. For example, the expected output state for the Quantum Fourier Transform (QFT), analogous to the classical Fourier Transform, is: $S_{state}(QFT, x) = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi i k x/2^n} |k\rangle$ where x is the input, i is the imaginary unit, and k is the basis state index $|k\rangle$. Distribution-based and quantum state-based specifications are related since the former derives from the latter by squaring the amplitudes' absolute values. Generally, S_{state} holds more information than S_{distr}, as it can describe negative amplitudes. For instance, the superposition states $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ are equivalent in a distribution-based specification, both yielding $|0\rangle$ and $|1\rangle$ with probability $\frac{1}{2}$. However, they differ in a quantum state-based specification due to their distinct amplitudes.

3.1.3 Unitary-Based Program Specification. A third way to describe the expected output state of a program leverages the representation of quantum operations as matrices [118]. The idea is to specify the expected quantum state as a unitary matrix U_P that transforms an initial all-zero state into the expected output state. Typically, this matrix is derived by multiplying unitary matrices of simpler operations. This coincides with having a correct reference program Q, composed of n known elementary operations, used to compute the expected matrix U_P.

Definition 3.3 (Unitary-based program specification). Given a quantum program P with n gates, its unitary-based specification S_{unit} is the unitary matrix representing the program: $S_{unit}(P) = U_P = U_n \cdot U_{n-1} \cdot \ldots \cdot U_1$ where U_i is the unitary matrix of the i-th gate in the known program Q.

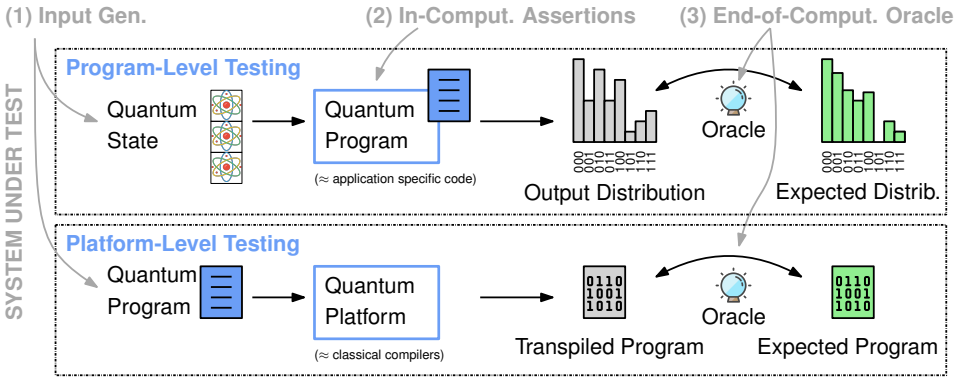


Fig. 6. Overview of testing approaches for quantum software.

While each program corresponds to a unique unitary matrix, the same matrix, and thus behavior, can result from different sequences of operations.

3.2 Specifying Expected Behavior of Quantum Platforms

Unlike quantum programs that operate on quantum states, quantum computing platforms process classical data, such as source code and intermediate representations. The expected behavior of a platform is to transform a program while preserving its semantics. Such transformations convert the program P_1, a sequence of n quantum operations, into an equivalent program P_2 that is more efficient or hardware-compatible, possibly with a different number of operations. A common formalism to assess transformation correctness is the *equivalence relation* between two quantum programs.

Definition 3.4 (Program equivalence). Two quantum programs P_1 and P_2, represented by two unitary matrices U_1 and U_2, are equivalent, denoted $U_1 \equiv U_2$, if there exists a real number $\phi \in \mathbb{R}$ such that $U_1 = e^{i\phi} U_2$.

Note that a special case of program equivalence is *program equality*, denoted $U_1 = U_2$, which is equivalent to $U_1 \equiv U_2$ with $\phi = 0$. The value ϕ is called the *global phase*, a common concept in quantum mechanics where the global phase of a quantum state is not observable. Thus, when measuring the state $|\psi'\rangle = e^{i\phi} |\psi\rangle$, the probability of observing a given outcome is the same as for the state $|\psi\rangle$. Proving two programs are equivalent means showing they have the same unitary matrix, up to a global phase.

4 Testing and Debugging

Following the literature on software testing, we call the software that gets tested the *system under test* (SUT). An input given to the SUT, together with a check of the expected behavior of the SUT, form a *test case*. The input is called the *test input* and the routine that checks if the SUT behaves as expected is called the *oracle* [7].

We group approaches for testing quantum computing software along two axes. The first axis, shown vertically in Figure 6, concerns the SUT targeted by an approach. We distinguish between testing quantum programs, akin to traditional application testing, and testing quantum computing platforms, akin to compiler testing [15]. The second axis, shown horizontally in Figure 6, addresses the testing subproblem tackled by an approach. We consider three subproblems: (1) test input generation, i.e., generating inputs for the SUT; (2) in-computation assertions, i.e., checking

Table 1. Overview of testing approaches.

Paper	Approach	Testing subproblem			Automation	
		Input gen.	Oracle		Manual	Auto
			In-comp.	End-of-comp.		
Testing of quantum programs:						
Wang et al. [110]	QuCat	✓		✓		✓
Wang et al. [111]	QuSBT	✓		✓		✓
Long and Zhao [59]	QSharpTester	✓		✓		✓
Huang and Martonosi [40]	Stat. Assert.		✓	✓	✓	
Liu et al. [56]	Runtime Assert.		✓	✓	✓	
Li et al. [51]	Proq		✓	✓	✓	
Liu and Zhou [58]	NDD/Swap-based		✓	✓	✓	
Testing of quantum commputing platforms:						
Paltenghi and Pradel [78]	MorphQ	✓		✓		✓
Wang et al. [109]	QDiff	✓		✓		✓
Xia et al. [115]	Fuzz4All	✓		✓		✓

program correctness at runtime by asserting properties on a program state, such as the value in a qubit or register; (3) end-of-computation oracles, i.e., verifying the program's output against an expected reference.

Testing quantum programs is uniquely challenging because observing a state destroys it, and copying is impossible due to the no-cloning theorem [114]. These limitations make debugging harder than for classical programs. This issue does not affect quantum computing platforms, which adapt classical compiler techniques to quantum theory without performing any quantum execution. Thus, our survey discusses in-computation assertions only for quantum programs.

Sections 4.1–4.3 detail subproblems in testing quantum software and approaches to address them. Table 1 summarizes the surveyed work. Finally, Section 4.4 briefly discusses debugging quantum software.

4.1 Test Input Generation

Knowing which input to feed to the software under test is crucial for finding bugs. The problem of generating meaningful test inputs is referred to as test input generation or fuzzing [134].

A quantum program with n qubits has 2^n possible classical inputs. Considering quantum states as inputs, the number of possibilities becomes virtually infinite due to arbitrary superpositions of those 2^n classical inputs. For example, a quantum circuit with $n = 2$ qubits has $2^2 = 4$ classical inputs (00, 01, 10, 11), but an infinite number of quantum inputs, such as the superposition state $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. This vast input space makes exhaustive testing impractical, so techniques to find likely bug-inducing inputs have been proposed. For instance, QuCat [110] uses combinatorial testing [72] to generate test inputs for quantum programs, similar to classical software testing. Similarly, QuSBT [111] employs genetic algorithms for search-based software testing [33] to generate test inputs that maximize failing test cases. While effective on 30 programs with injected bugs, QuSBT has yet to be applied to real-world programs. To reduce the burden of the large input space, Long and Zhao [59] propose QSharpTester, generating test inputs based on equivalence classes, including classical and superposition states.

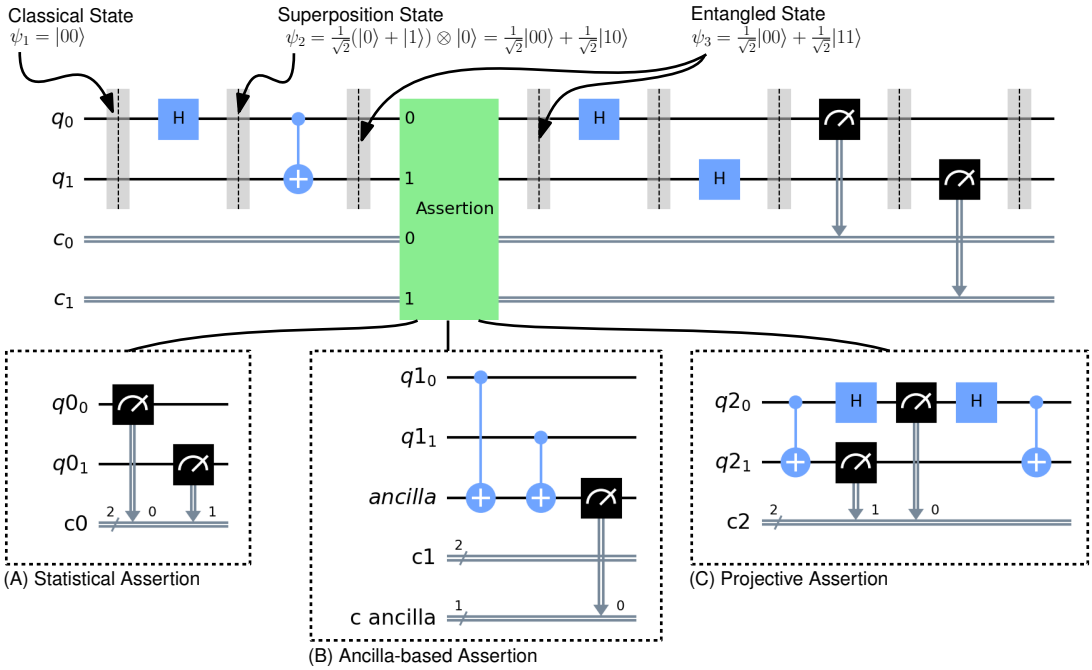


Fig. 7. Example circuit under test (top) with three kinds of approaches to check assertions during the computation (bottom).

In testing quantum platforms, various methods generate quantum programs as test inputs to trigger crashes or unexpected output distributions. Using differential testing [15], Wang et al. [109] introduce QDiff, which mutates programs by adding, removing, and replacing operations, or by executing the same operation on different qubits. Using metamorphic testing [17], Paltenghi and Pradel [78] propose MorphQ, which generates valid quantum programs via templates and a grammar-based generator, then applies quantum-specific metamorphic transformations. A metamorphic transformation consists of a program transformation and an expected effect on the program's semantics, e.g., applying a sub-circuit and its inverse, which is expected to preserve the original program semantics. These transformations enable MorphQ to explore complex platform features, e.g., advanced optimizations and OpenQASM machine code conversion. More recently, inspired by the success of large language models, Xia et al. [115] introduce Fuzz4All, a versatile fuzzer that generates realistic-looking quantum programs using large language models, effectively uncovering corner cases in the targeted quantum computing platform. Additionally, Fuzz4All can target specific platform features derived from the platform documentation in natural language, such as the most recent changelog.

4.2 In-Computation Assertions

Due to the characteristics of quantum mechanics, the state of a quantum program cannot be directly observed without destroying it. To nevertheless check the correctness of a quantum program at runtime, several techniques have been proposed. Figure 7 shows an example circuit (top) with three assertion-checking approaches (bottom): (a) statistical assertions, (b) ancilla-based assertions, and (c) projective assertions. The following presents the three approaches.

4.2.1 Statistical Assertions. Proposed by Huang and Martonosi [40], the idea of statistical assertions is to run the program up to a breakpoint and then measure all the qubits in the program. Because measuring destroys the quantum state, the remaining program execution is ignored. This process is repeated multiple times and the output distribution is compared against a specific reference distribution. This method supports assertions on three kinds of quantum states: classical states, superposition states, and entangled states. Examples of these states are shown in Figure 7 (top) as $|\psi_1\rangle$, $|\psi_2\rangle$, and $|\psi_3\rangle$, respectively. In $|\psi_1\rangle = |00\rangle$, the reference distribution is a classical distribution, i.e., all the probability mass is concentrated on a single outcome. In $|\psi_2\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$, the reference distribution is a uniform distribution across two possible outcomes, $|00\rangle$ and $|10\rangle$, while the probability of any other outcome is zero. In $|\psi_3\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$, the reference distribution is a uniform distribution across two possible outcomes, $|00\rangle$ and $|11\rangle$, and the probability of any other outcome is zero. Note that in $|\psi_3\rangle$ the two qubits are entangled because whenever the first qubit is $|0\rangle$ the second qubit is also $|0\rangle$, and likewise for $|1\rangle$.

4.2.2 Ancilla-based Assertions. Inspired by quantum error correction, usually implemented in hardware, Liu et al. [56] propose *assertion circuits*. This circuit uses ancilla qubits and indirect measurements to assert the same properties as Huang and Martonosi [40], but without stopping the program execution at the breakpoint. Figure 7b shows an example of an assertion circuit. Measuring the ancilla qubits does not disturb the quantum state of the SUT's qubits. Thus, the state $|\psi_3\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ is preserved as the output of the assertion circuit. The only effect of the assertion circuit is that the ancilla qubits are now in a specific state that depends on the outcome of the measurement. Specifically, the ancilla qubit, initialized as $|0\rangle$, remains $|0\rangle$ if the assertion holds; otherwise, it flips to $|1\rangle$. Unlike statistical assertions, assertion circuits can check a qubit's phase, which is not observable with a simple output distribution. However, they can only check assertions where all states in the entanglement have the same parity, i.e., the number of 1s in the bit string is either all even or all odd. For instance, the entangled state $|\psi\rangle = a|000\rangle + b|011\rangle + c|101\rangle + d|111\rangle$, where a, b, c, and d are complex numbers, cannot be asserted since $|111\rangle$ has different parity.

4.2.3 Projective Assertions. Li et al. [51] propose Proq and *projective measurements*, or short *projections*. Their method ensures that if the quantum state is $|phi_a\rangle$ and we apply the projective measurement P_A, which can be viewed as an assertion, then the state $|phi_a\rangle$ remains unchanged. Since quantum computers typically implement only projections in the computational basis, i.e., those that return either a state $|0\rangle$ or $|1\rangle$ for a single qubit, Proq introduces an approach called *implementation in the computational basis* to allow for more general assertions. The key idea is to add unitary transformations before and after the projective measurement to implement arbitrary projective measurements, as shown in Figure 7c. This approach can also assert new types of quantum states, such as general entangled states, not supported by assertion circuits. For highly entangled states with many qubits, where the unitary implementation is tricky to obtain, they introduce the approximation of *local projections*, checking only a subset of qubits at a time. This approach is close in spirit to the approach based on combinatorial testing proposed by Wang et al. [110], where only pairs or triplets of qubits are considered at the same time.

Finally, building on Liu et al. [56] and *non-destructive discrimination* (NDD) of quantum states [31, 42], Liu and Zhou [58] propose swap-based and NDD-based methods to check quantum program correctness via ancilla measurements. These methods also support new assertion types, such as mixed states and membership assertions. Indeed, besides the vector representation $|\psi\rangle$ of a quantum state (Section 2), there is also the *density matrix* ρ, obtained via the outer product of the state with itself: $\rho = |\psi\rangle\langle\psi|$. A state is *pure* if it can be described with a single vector, whereas a state is *mixed* if it is a weighted sum of pure states: $\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$ where p_i is the probability of the state $|\psi_i\rangle$. Note that a pure state is a special case of a mixed state where $p_i = 1$ for a single i. Mixed states often model noise in quantum

Table 2. Types of assertions supported by different approaches: fully (✓), partially (≈), or not supported (empty). Assertions can be on classical, superposition, entangled, or membership states. Limitations: (a) state destruction, (b) mid-circuit measurement, (c) extra qubits, (d) extra operations.

	Types of Assertions				Limitations			
Approach	**Class.**	**Superp.**	**Entang.**	**Memb.**	**State destr.**	**Mid-circ. Meas.**	**Extra qub.**	**Extra ops.**
Statistical assertions:								
Stat.Assert. [40]	✓	≈	≈		✓			
Ancilla-based assertions:								
Runtime Assertion [56]	✓	✓	≈				✓	✓
Swap-Based [58]	✓	✓	✓	≈			✓	✓
NDD-Based [58]	✓	✓	✓	≈			✓	✓
Projective assertions:.								
Proq [51]	✓	✓	✓			✓		✓

programs. The mixed state assertion checks if a quantum state is a mixture of pure states, whereas the membership assertion checks if a quantum state is a member of a set of quantum states.

The approaches discussed above support different types of assertions and have different limitations, as summarized in Table 2. In terms of scalability, *statistical assertions* require measurements proportional to the number of assertion points, as the program cannot proceed further. In contrast, methods based on *assertion circuits* and *projective measurements* do not increase the number of required measurements, since runtime assertions leave the state intact.

4.3 End-of-Computation Oracles

An end-of-computation oracle checks the correctness of the output of a program at the end of the computation. We discuss such oracles for testing both quantum programs and quantum computing platforms. For quantum programs, these oracles differ from in-computation assertions in two ways. First, the oracle can measure and destroy the quantum state, as no further computation follows. Second, end-of-computation oracles typically check the entire output distribution, not just part of the quantum state. For quantum platforms, an end-of-computation oracle usually checks the result of compiling or transforming a quantum program.

End-of-computation oracles relate to two terms from physics: quantum property testing and quantum state tomography. *Quantum property testing* [66] checks if a quantum state is close to an expected state, focusing on efficiency and theoretical contexts, like verifying a quantum algorithm properties. In contrast, end-of-computation oracles apply to concrete quantum algorithm implementations. *Quantum state tomography* [73] reconstructs a quantum state's representation through multiple measurements on identical system copies, requiring many program runs. While quantum state tomography can serve as an end-of-computation oracle, it is costly, as measurements grow exponentially with qubits.

The following discusses approaches to implement end-of-computation oracles in three groups. Section 4.3.1 presents *precise oracles*, which reliably identify specific failing cases, detecting bugs in the SUT. Section 4.3.2 presents *imprecise oracles*, which characterize the correctness of the entire output distribution. We recognize that this distinction can blur, as some methods serve both purposes. However, we find it useful to distinguish them due to their different goals and requirements. Finally, Section 4.3.3 discusses *equivalence oracles*, used to check if two quantum programs are equivalent. Figure 8 provides an overview of the discussed approaches.

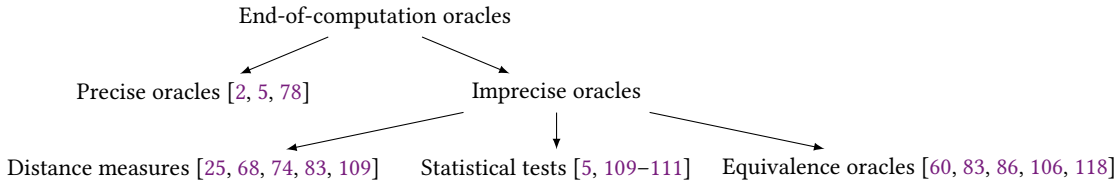


Fig. 8. Overview of approaches that use or propose end-of-computation oracles.

4.3.1 Precise Oracles. The precise oracles are designed to find specific bugs in the SUT by observing clear indicators of failure. For instance, Ali et al. [5] propose the wrong output oracle, which flags a bug every time the program produces an output that is impossible under the correct expected program execution, such as an output with zero probability in the program specification. This approach assumes a noise-free environment, as NISQ architectures may occasionally produce incorrect output bit strings. Abreu et al. [2] apply the idea of metamorphic testing [17] to quantum programs, where a metamorphic oracle is a function that takes the output of the SUT and the input and returns a boolean value indicating if the output is correct. Paltenghi and Pradel [78] focuses on a traditional crash oracle, where a bug in the quantum computing platform is signaled by a crash during the execution of a transpiled quantum program.

4.3.2 Imprecise Oracles. Imprecise oracles assess the overall correctness of the output distribution. They use two main strategies: (i) distance measures and (ii) statistical tests. Next we present examples and discuss shared challenges.

Distance Measures. To compare the output distribution of a quantum program with the expected distribution, several distance measures have been proposed. Formally, the task is to compare two probability distributions P and Q to determine their similarity. Each distribution is a vector of probabilities, where the i-th element represents the probability of the i-th outcome, i.e., the probability of observing the i-th bit string as output. For a program with n qubits, there are 2^n possible outcomes. Among the distance measures are the Hellinger distance, cross-entropy, Kullback-Leibler divergence, Jensen-Shannon divergence, and Total Variation distance. Notably, not all are symmetric, e.g., Kullback-Leibler is asymmetric, while Jensen-Shannon is symmetric, combining two Kullback-Leibler divergences with the average distribution $M = \frac{1}{2}(P + Q)$. Fidelity, instead of using probability distributions directly, works with the density matrices ρ and σ of the quantum states corresponding to P and Q. Thus, we first estimate the density matrices from the distributions using methods like quantum state tomography.

Statistical Test. Common statistical tests include the Kolmogorov-Smirnov test, Pearson's chi-squared test, and Wilcoxon signed-rank test. The Kolmogorov-Smirnov test is non-parametric for continuous distributions, using cumulative distribution functions $\mathcal{P}$ and $\mathcal{Q}$ of P and Q. Here, $\mathcal{P}_i$ is the fraction of samples in P less than or equal to bit string i, with bit strings sorted by their decimal values, e.g., $000 \rightarrow 0$, $001 \rightarrow 1$, $010 \rightarrow 2$. While converting bit strings to decimal values is necessary, this sorting may not be optimal, as bit strings are not independent. Alternative versions for discrete [24] and multivariate [44] cases exist. For quantum state verification, Yu et al. [124] discuss in detail which statistical methods can verify the correctness of a quantum program's output distribution.

Thresholds and Sample Sizes. Both statistical tests and distance measures require a threshold to determine if output distributions differ significantly, indicating a bug. Distance measures use a heuristic threshold to assess deviation from the expected output. Statistical tests yield a p-value, showing the likelihood of an observed result under the null hypothesis (i.e., the program is correct). A p-value below 0.05 typically indicates a significant difference in output

distribution. Estimating the right number of samples is still an open problem, but Wang et al. [109] propose a formula for the Kolmogorov-Smirnov test, based on the fact that Kolmogorov-Smirnov distance is bounded by L1 distance.

4.3.3 Equivalence Oracles. When the expected output can be described by a *reference program*, we can use *equivalence checking* oracles to verify if the SUT's computation matches the reference. Peham et al. [86] uses ZX-calculus to check the equivalence of two quantum programs for quantum compilation. Xu et al. [118] proposes an approach based on SAT solver to discover new equivalences between chains of gates. In their work, Long and Zhao [60] propose algorithms to check program equivalence, assess if a program is unitary, namely reversible, and to determine whether a program is the identity. Unruh [106] proposes a relational version of Hoare logic (rQHL) for quantum programs, which can be used to prove the equivalence of two programs. A technique called Feynman [6] allows to check the equivalence of two Clifford group quantum programs in polynomial time, by using a simulation approach based on the path sum representation. It also handles circuits of this kind with up to 100 qubits and thousands of gates. Quest [83] verifies approximate semantic equivalence of two programs using the Hilbert-Schmidt distance between their unitary matrices.

4.4 Debugging of Quantum Programs

Debugging, i.e., identifying and understanding bugs in a program, is crucial in software development. For quantum programs, it is particularly challenging due to the probabilistic nature of quantum mechanics and the no-cloning theorem, which prevents copying an unknown quantum state. Yet, there is relatively little work to support debugging of quantum programs. Metwalli and Meter [64] propose to automate the task of running only parts of a quantum program, letting users exclude qubits and select circuit segments to execute. Another method [100] profiles quantum programs similarly to traditional CPU profilers. Their method informs the user how often a subroutine is called, highlighting optimization opportunities like removing unused subroutines or replacing them with more efficient ones.

5 Static Analysis for Correctness

This section covers static analysis methods to detect bugs or prove correctness without executing the software on a quantum computer, unlike the previous section on testing techniques requiring execution on a quantum device. We also discuss approaches that run quantum programs on simulators, considered a form of static analysis since they do not use real quantum computers.

Table 3 summarizes the discussed approaches, categorized by analyzed software (quantum programs or platforms) and analysis technique. We identify three kinds of analysis techniques: First, *proof-based formal methods*, which aim at exact reasoning about computations, e.g., based on Hoare logic, separation logic, and incorrectness logic, and rely on theorem provers or deductive frameworks. Second, *formal methods based on abstraction*, such as abstract interpretation, that may rely on higher degrees of approximation of the quantum state than the previous group. Third, *pattern-based approaches* that look for recurring bug patterns, e.g., in the form of source code fragments that have a specific syntactic structure, specific data flow relations between code elements, or a specific sequence of quantum gates. These approaches trade off precision and scalability: proof-based methods are most precise but least scalable, while pattern-based methods are more scalable but less precise.

Several approaches adapt classical static analysis techniques to quantum, including quantum Hoare logic [121], abstract interpretation [123], symbolic execution [71], syntax-based analysis [128], and data flow-based analysis [45, 79]. Other analyses are specifically developed for quantum programs, such as *entanglement analysis* [87, 116, 125], which is

Table 3. Overview of analysis methodologies and target System Under Test (SUT).

	Kinds of analysis approaches		
Analyzed software	**Proof-based formal methods (Section 5.1)**	**Formal methods based on abstraction (Section 5.2)**	**Pattern-based approaches (Section 5.3)**
Program	Incorrectness logic [120], Quantum Hoare logic [121, 133], QHLProver [57], Interactive provers [11, 35], QWIRE [92], Qbricks [14], CoqQ [132], SymQV [8], Quantum separation logic [131]	Quantum abstract interpretation [123], Abstraqt [10], AutoQ [18, 19], Entanglement analysis [87, 116], Twist [125], ScaffCC [43]	Qchecker [128], LintQ [79], Quantum-CPG [45], QSmell [16]
Platform	VOQC [36, 37], Giallar [104], ReQwire [91]	-	-

used to track which qubits are entangled with each other, and *uncomputation analysis* [25, 80], which tracks qubits that are used as ancilla and are still entangled with the rest of the program state, and thus cannot be used for other purposes.

Sections 5.1, 5.2, and 5.3 discuss these three families of approaches. We cover their representation of quantum computation semantics, the properties they check, and their automation levels.

5.1 Proof-Based Formal Methods

The goal of the approaches discussed in the following is to formally specify properties of quantum programs and prove these properties using formal methods, such as logic, theorem provers, or SMT solvers.

5.1.1 Representations of Computations. Different approaches use various representations for quantum states and operations, including state vectors, path sums, and (reduced) density matrices. Figure 9 shows common representations with an example. We first explain these representations and then discuss their use in existing techniques.

The top of Figure 9 shows the quantum circuit, with different quantum state representations below. The upper gray box in Figure 9 shows the state vector representation at different points in the circuit. A state vector represents the quantum state as a vector $|\Psi\rangle$ of complex amplitudes, one for each basis state. The lower gray box shows the path sum representation, which expresses a quantum state as a sum of all possible paths leading to it. Each path is a sequence of possible quantum states from the initial to the final states. Finally, the two green boxes at the bottom right show the density matrix of a specific state, marked in green throughout the figure. The top one shows the full density matrix $\rho = |\Psi\rangle \langle\Psi|$, while the bottom one shows the reduced density matrix.

State vectors and density matrices are widely used due to their accuracy [35, 57, 120, 121, 133]. The downside of these representations is their computation and memory cost which is exponential in the number of qubits. Bordg et al. [11] use a state vector modeled as complex numbers in the Isabelle theorem prover, but it has limited power since it cannot represent mixed states. CoqQ [132] uses the more expressive Dirac notation, which is easier for reasoning when using the MathComp library. To avoid the exponential cost of state vectors and density matrices, other approaches, like the Feynman tool [6] and Qbricks [14], use path sums. Another method, SymQV [8], employs an SMT-compatible symbolic state representation. Here, the state is represented as a set of symbolic variables that represent the qubits of the programs and that are manipulated by the quantum gates. This representation reduces computational cost and memory usage by avoiding full state vector and matrix construction when possible.

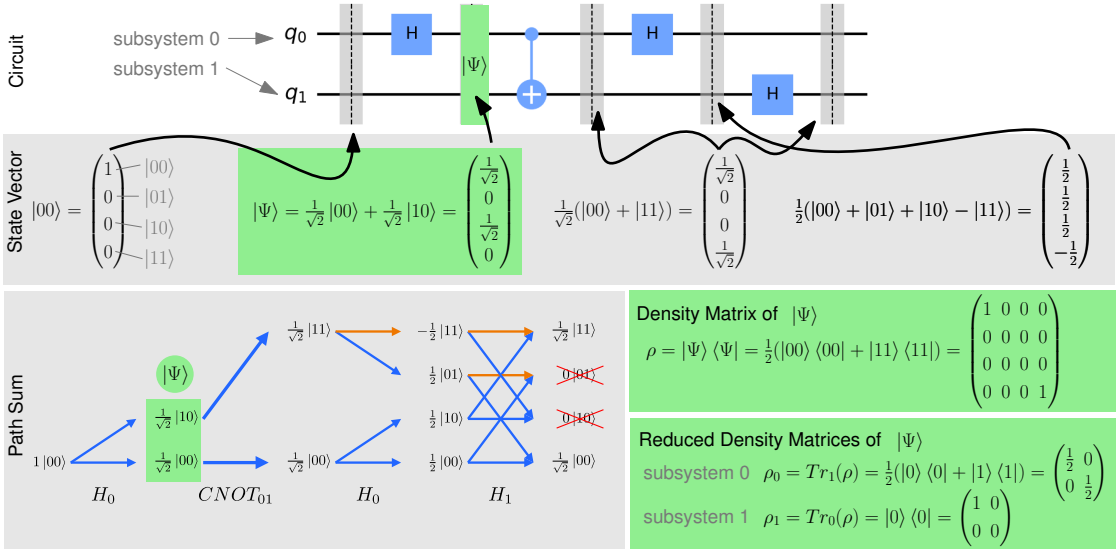


Fig. 9. Quantum circuit and various representations of quantum states during computation. The state vector and path sum represent the entire circuit, while the green boxes show a specific state $|\psi\rangle$ with a (reduced) density matrix.

5.1.2 Kinds of Checked Properties. The properties checked by different approaches vary significantly, from verifying the correctness of a specific QFT implementation [6] to fundamental quantum computing theorems like the no-cloning theorem, which states that an exact copy of an unknown quantum state cannot be made [11].

Inspired by classical Hoare logic [38], Ying [121] formalizes quantum Hoare logic (QHL). Like its classical counterpart, a quantum Hoare triple is $\{P\}C\{Q\}$, where P and Q are quantum predicates for pre- and post-conditions, and C is the quantum program. Several approaches build on QHL [57, 121, 133], differing in the supported predicates P and Q, e.g., arbitrary quantum predicates [57, 121] or projections [133]. Projections are specific quantum predicates, defined as $P = \sum_{i=1}^{n} |i\rangle\langle i|$, where n is the number of qubits and $|i\rangle$ is the computational basis. Projections enable simpler inference rules, and hence, smaller proof trees, but cannot capture all quantum properties. Zhou et al. [133], in their work on *applied* quantum Hoare logic (aQHL) use projections to simplify inference rules and ranking functions of QHL. They also introduce rules for reasoning about robustness, useful for inexact quantum programs like those in quantum machine learning. Most approaches allow expressing first-order logic properties of quantum programs, making them versatile for proving properties. Two examples of provable correctness properties are: (1) Grover's algorithm has a higher probability of measuring the correct solution than a threshold, given a fixed number of iterations [35]; (2) the HLL algorithm returns the exact solution for any input state, given certain assumptions about the solution's existence [133].

A dual view of Hoare logic is *incorrectness logic* [75], which aims to prove the existence of bugs rather than program correctness. It is based on the triple $\{presumption\}C\{result\}$, where the presumption is a predicate assumed true before execution, and the result is a predicate true on a subset of possible final states. Inspired by incorrectness logic, Yan et al. [120] propose *quantum incorrectness logic* (QIL), using projections for presumption and result. Another logic adapted to quantum is separation logic [94], which reasons about program memory modularly. Separation logic has been adapted to quantum to reason about entanglement and separability of subsystems [131].

The Feynman tool [6] allows proving the equivalence of two quantum programs or that two quantum programs implement different functions. Qbrick [14], extending Feynman's state representation, allows expressing first-order logic properties. Both Qbrick [14] and QWIRE [92] allow for specifying properties of parametric quantum programs, e.g., an single algorithm that can operate on a variable number of qubits, using two different theorem provers, namely Why3 and Coq. Finally, SymQV [8] can verify first-order logic properties of quantum programs. For instance, it can check that in the quantum teleportation protocol, the state of the first qubit is transferred to the last qubit, and no operations cross the boundary between the first two qubits and the last one. Namely, the first two qubits are the sender's, and the last one is the receiver's.

5.1.3 Level of Automation. The above techniques rely on axiomatic proof systems, with automation levels ranging from manual to fully automatic. Early methods are mostly manual or semi-automatic, requiring users to construct proofs and provide intermediate lemmas. This applies to Hoare logic [121], incorrectness logic [75], and SQIR [35], in which verifying Grover and QPE took one and two person-weeks, respectively. Qbricks [14] automates proofs using a domain-specific language (Qbricks-DSL) for quantum programs and another for specifications (Qbricks-SPEC). It automatically proves 95% of lemmas and proof obligations in their evaluation. Two fully automated approaches are Giallar [104] and SymQV [8]. Giallar uses SMT solvers to verify circuit transformations by an optimizer in a quantum platform. It extracts invariants from three loop templates and uses pre-verified rewriting rules to prove circuit transformation correctness. SymQV [8] is a symbolic execution framework that proves first-order logic properties of quantum programs using an SMT solver. A potential barrier to adoption is that many approaches require users to use a specific programming language, often custom or domain-specific, such as the "quantum while language" [121] or Qbricks-DSL [14], or intermediate representations embedded in theorem provers like SQIR [36, 37] and QWIRE [92]. Most languages are low-level and tailored to specific methods, except SQIR, which converts to and from OpenQASM.

5.2 Formal Methods based on Abstraction

The following discusses approaches that use formal methods but rely on abstractions to reduce analysis complexity. These methods represent the quantum state abstractly, simplifying reasoning but potentially losing some information.

5.2.1 Representations of Computations. A common analysis target here is entanglement, tracking which qubits are entangled. Due to the exponential memory needed for precise state tracking, the *stabilizer* formalism [1] is often used. This formalism represents the quantum state $|\psi\rangle$ using Pauli operators Q that stabilize it, satisfying $Q|\psi\rangle = |\psi\rangle$. While compact and efficient, it is limited to states produced by Clifford gates, which are not universal for quantum computation. To handle more programs, Perdrix [87] use diagonal bases to abstract the quantum state, though this is quite lossy. Inspired by stabilizers' precision, Honda [39] extend stabilizers to allow some uncertainty when non-Clifford gates like the T gate are used. The key idea is that non-Clifford gates' effects can be bounded locally and removed later. While Honda [39] is more precise than Perdrix [87], its use beyond entanglement analysis remains unclear. Twist [125] instead uses the concept of fractional permissions to model entanglement along the computations, and the usual density matrix to represent the quantum state whenever they need to resolve the quantum state to a specific value at runtime. ScaffCC [43] tracks qubits that might be entangled, assuming any two-qubit gate creates entanglement.

Other approaches abstract the quantum state more generally, beyond entanglement. Yu and Palsberg [123] use reduced density matrices (Figure 9). Note that with reduced density matrices, reconstructing the full quantum state is generally impossible, making the representation lossy. Yu and Palsberg [123] use reduced density matrices to represent entanglement between qubit pairs. This saves space, representing 50 qubits with $\binom{50}{2} = 1225$ matrices instead of 2^{50}

complex numbers. Their approach handles programs with up to 300 qubits, a significant improvement over naive simulation. Besides pairs of qubits, the approach can consider any number of qubits together, but this generalization creates multiple abstract domains, complicating their selection for a given program.

Abstraqt [10] uses a path sum representation, approximating non-Clifford gates with Pauli sums, then condensing them into a single summand for efficiency. This over-approximation enhances time and space efficiency compared to simulators. However, the approach loses precision quickly when reasoning about programs with Clifford and T gates. AutoQ [19] introduces a binary tree to represent the quantum state, where the height of the tree is the number of qubits, each branch corresponds to a computational basis, and each leaf corresponds to the complex amplitude of that state. A tree automaton is also used to represent a set of states and the operations via gates are operations directly on the automaton. AutoQ's tree representation [19] is space- and computation-efficient. For example, it can encode the output of the Bernstein-Vazirani algorithm with a linearly sized tree by merging the shared branches of the tree.

5.2.2 Kinds of Checked Properties. The properties checked by different approaches vary significantly, with the most efficient ones checking only simple properties like entanglement absence. For example, Perdrix [87] checks entanglement between qubit subsets, assuming all CNOT gates entangle. Honda [39] also checks entanglement between qubit subsets but has limited handling of non-Clifford gates. Twist [125] instead allows reasoning about entanglement by exploiting annotations of "pure types", and assertions that express when some qubits are separable from the rest of the program (called "cast" assertion) or when two sets of qubits are not entangled (called "split" assertion). ScaffCC [43] runs a disentanglement check to ensure all ancilla qubits are not entangled with result qubits. Yu and Palsberg [123] checks if a target state is in the span of, i.e., subspace generated by, certain vectors, like the GHZ state $span(|000\rangle, |111\rangle)$. This is limited as it cannot represent more complex superpositions. Abstraqt [10] checks if qubits are reset to zero at the program's end AutoQ [19] checks if the output state is a sub-language of a tree automaton representing the post-condition, allowing versatile properties to be checked as tree automata. AutoQ focuses on detecting bugs or functional non-equivalence between two quantum programs rather than proving their correctness.

5.2.3 Level of Automation. The first two works on abstract interpretation [39, 87] are fully manual, requiring users to apply abstract semantic rules by hand. Twist [125] needs type and assertion annotations for quantum expressions but is fully automatic, using static analysis and runtime checks via a simulator. Similarly, ScaffCC [43] requires users to annotate ancilla qubits in Scaffold programs. Adding new analyses in ScaffCC requires manual implementation in the compiler. Yu et al. [123] is fully automatic, needing users to specify the span or support of the final state or define loop invariants, as in Grover's algorithm. Abstraqt [10] and AutoQ [19] are fully automatic, with Abstraqt requiring users to specify correctness properties and AutoQ needing user-defined post-conditions for non-equivalence checks.

5.3 Pattern-Based Approaches

The third group of approaches identifies and matches specific syntactic or semantic constructs, known as *patterns*, in the target language. Table 4 summarizes these approaches, which recognize patterns at varying abstraction levels, from purely syntactic to more semantic, like data-flow analysis.

5.3.1 Representations of Computations. All approaches in this category target quantum programs written in Qiskit [16, 45, 79, 128], thus Python code. QChecker [128] analyzes the abstract syntax tree (AST) of quantum programs, representing variable assignments as *QP attributes* and function calls as *QP operations*. A QP attribute example is a variable assigned to a quantum register, like `('qreg', 'QuantumRegister(3)')`; a QP operation example is a function call applied

Table 4. Overview of pattern-based approaches.

Approach	LintQ [79]	Quantum-CPG [45]	QChecker [128]	Qsmell [16]
Syntax matching (AST)	✓	✓	✓	✓
Data flow	✓	✓		
Quantum data flow	✓	✓		✓
Control flow	✓	✓		
Circuit composition	✓			
Classical-quantum link		✓		

to a quantum circuit, like `('circuit.h(2)')`. LintQ [79] instead converts the quantum program into the CodeQL intermediate representation and then uses the CodeQL framework to reason about both syntax in the form of the AST, but also derive data-flow and control-flow analysis. The CodeQL representation consists of a database of facts that encode the program elements and the relationships between them. Beside classical data flow, LintQ also supports quantum data flow to track gate application order on qubits.'s Qsmell [16] represents the quantum program using an execution matrix derived from the compiler. In this matrix, rows correspond to qubits, cells to gates, and columns to execution timestamps. Alternatively, Qsmell supports AST-based analysis without data-flow or control-flow. Quantum-CPG [45] converts the quantum program into a code property graph (CPG) [119], encoding both data-flow and control-flow. Like LintQ, it captures the execution order of quantum gates on qubits using a graph representation of quantum data flow.

5.3.2 Kinds of Checked Properties. Each approach recognizes different patterns based on the abstraction level and expressiveness of the quantum program representation, as summarized in Table 5. Despite using different representations, there are overlaps in the patterns they recognize. Note that even if two approaches recognize the same pattern, they may do it differently, leading to varied bug detection abilities. QChecker [128] identifies syntax-level bug patterns using Python's AST, inspired by Bugs4Q dataset [130]. It focuses on local issues like non-existent APIs or incorrect function parameters. LintQ [79] recognizes the most patterns, including those from empirical studies and related work. LintQ is also the only approach that models composed circuits, i.e., complex circuits that are build from multiple sub-circuits. Modeling this and other concepts allows LintQ to recognize additional bug patterns, such as code that uses the `compose` API incorrectly, and data flow-related patterns, such as applying a gate to a qubit after the qubit has been measured. QSmell [16] identifies code smells validated by a developer survey. Examples include overly long quantum programs and qubits unused for extended periods, increasing decoherence risk. Finally, Quantum-CPG [45] detects programming errors and classical code smells. It uniquely models the qubit-classical register link, identifying unused measurement results. We refer to the original papers for the full list of patterns recognized by each approach.

5.3.3 Level of Automation. Compared to proof-based and abstraction-based methods, pattern-based approaches have the lowest entry barrier. They only need the quantum program's source code and run fully automatically. Pattern-based approaches can be extended to recognize new patterns with some effort. Adding patterns to QChecker or QSmell requires Python code to detect patterns in the AST or execution matrix. For LintQ, it involves writing CodeQL queries using LintQ's abstractions.

6 Static Analysis to Optimize Quantum Programs

Beyond validating and verifying quantum programs, static analysis can optimize them. Optimization is crucial for NISQ computers [89], which face four main challenges:

Table 5. Bug patterns recognized by different pattern-based approaches.

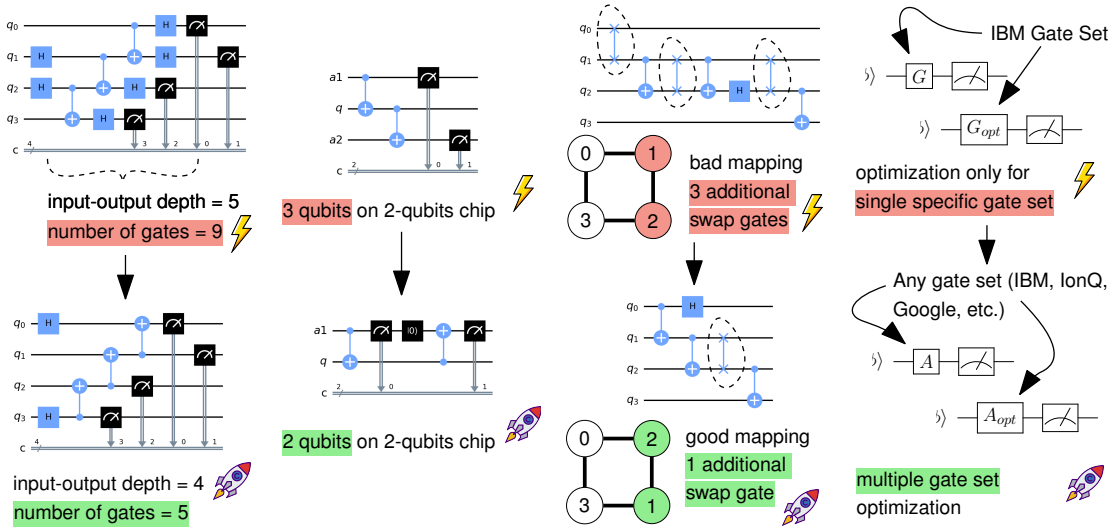
Approaches: LintQ (✳) Quantum-CPG (●) QChecker (✕) QSmell (⊙)			
Bug Pattern	**Support**	**Bug Pattern**	**Support**
Gate errors		**Circuit size**	
Custom / incorrect gates	✕⊙	Classical register too small	✳✕
Repeated gates	⊙	Long circuit	⊙
Superfluous operation	●	Oversized circuit	✳
Measurement issues		**Quantum-classical interface**	
Operation after measurement	✳✕⊙	Constant classical bit	✳●
Condition without measurement	✳●	Non-parametrized circuit	⊙
Double measurement	✳	Result bit not used	●
Measure all	✳	Constant result bit	●
Miscellaneous		**Qubit initialization and layout**	
Use non-existent API	✳✕	Idle qubits	⊙
Parameter error	✕	Initialization of qubits	⊙
Call error	✕	No physical layout	⊙
QASM error	✕		
Ghost compose	✳		
Operation after optimization	✳		
Discarded order	✕		

- (C1) *Noisy hardware*. Current gate implementations suffer from a non-negligible error rate. Moreover, values stored in a quantum register suffer from the decoherence effect, where the imperfect isolation of the qubits from their environment may destroy the quantum state, and hence, limit the maximum length of a computation.
- (C2) *Constrained resources*. The limited number of qubits demands for a careful allocation of registers.
- (C3) *Limited connectivity*. Due to quantum computer's physical design, not all qubits can interact with each other.
- (C4) *Gate-set variation*: Since each quantum computer has its own available gates, the same program may appear significantly different when executed on different machines, requiring distinct optimization strategies.

These challenges have inspired various optimization approaches that analyze and transform quantum programs. We group these approaches by their optimization goals. As shown in Figure 10, each challenge (C1–C4) corresponds to a specific optimization goal (G1–G4). To tackle noisy hardware (C1), optimizations aim to reduce gate count (G1), shortening circuit depth, as shown in Figure 10a. For constrained resources (C2), methods reduce the number of qubits used (G2), as shown in Figure 10b. Addressing limited connectivity (C3) involves optimizing qubit mapping (G3), e.g., by minimizing the use of extra swap gates, as shown in Figure 10c. Finally, to address gate-set variations (C4), optimizations aim for hardware-agnosticism by targeting programs written in any gate set (G4), as shown in Figure 10d.

In the following section, we discuss different approaches grouped by their optimization goals, noting that some approaches pursue multiple goals and are discussed in multiple sections. Table 6 summarizes the optimization goals pursued by different approaches. We exclude error correction and error mitigation work, as they usually do not rely on program analysis but either post-process an algorithm's output or provide an additional hardware mechanism.

Fig. 10. Challenges faced when dealing with NISQ computers and corresponding optimization goals pursued by analyses.



(a) Noisy hardware challenge (C1) and its mitigation by reducing the number of gates (G1).

(b) Constrained resources challenge (C2) and its mitigation by reducing the number of qubits (G2).

(c) Limited connectivity challenge (C3) and its mitigation by minimizing communication (G3).

(d) Gate set variation challenge (C4) and its mitigation by being gate set independent (G4).

Table 6. Approaches for quantum program optimization and their optimization goals

Reduce number of gates (G1) Reduce number of qubits (G2) Minimize communication (**G3**) Gate-set independence (G4)			
Approach	**Goals**	**Approach**	**Goals**
PyZX [46]	G1	Time-optimal A* [126]	**G3**
Hybrid ZX-calculus [12]	G1	Layer-by-layer A* [135]	**G3**
Relaxed peephole optimization [54]	G1	Genetic mapper [47]	**G3**
Rule-based [70]	G1	WPM [99]	**G3**
QContext [55]	G1	SABRE [50]	**G3**
QSSA [85]	G1	Learning-based mapping [3]	**G3**
PCOAST [84]	G1	OLSQ [101]	**G3**
Assertion-based optimization [32]	G1	SMT-based mapping [112]	**G3**
SQUARE [25]	G1 G2 **G3**	QUEST [83]	G1 **G3**
Unqomp [80]	G1 G2	VOQC [36, 37]	G1 **G3**
Reqomp [81]	G1 G2	Queso [117]	G1 G4
QIRO [41]	G1 G2	Parametrized Circuit Inst. [122]	G1 G4
Wire Recycling [76]	G2	Quartz [118]	G1 G4
Silq [9]	G2	Quarl [52]	G1 G4
RL-based Mapping [88]	**G3**		

6.1 Reducing the Number of Gates (G1)

Reducing the number of gates addresses the noisy hardware challenge (C1) by directly reducing the number of imperfect gates, and indirectly by shortening the time that a qubit is in a superposition state, thus mitigating the effect of decoherence. A common method to reduce gate count is to apply program transformations, also called rewriting rules, that replace sequences of gates with shorter, equivalent ones. Most approaches for this goal focus on reducing CNOT gates, as they are frequent and error-prone. The top circuit in Figure 10a is optimized into the bottom one, reducing gate count and depth. The example shows a complex optimization done by Quartz [118], which automatically discovers and applies rewriting rules to transform the circuit into an equivalent but shorter version. Regardless of the techniques, some obstacles in achieving gate reduction are the vast search space for rewrite rules and finding the optimal sequence to apply. In particular, when some rewrite rules are applied in a sequence some intermediate states might have a higher number of gates than the original state, thus the optimization pass needs to be able to backtrack and try a different sequence of rules. Moreover, some rewrite rules need a global view of the circuit, i.e., the optimization pass must reason about the entire circuit, not just a local window of gates. Table 6 lists approaches targeting gate reduction.

Local program transformations. These approaches apply context-independent transformations, focusing mainly on adjacent gates. Relaxed peephole optimization [54] uses local optimization methods similar to classical compilers. It inspects small windows of contiguous operations, replacing them with equivalent sequences having fewer CNOT gates. User annotations indicating known qubit states, such as basis or pure states, enable more aggressive optimizations. For instance, if the input state is annotated, it can replace sequences of two-qubit gates, which normally result in three CNOT gates in the Qiskit compiler, with a single CNOT gate and four one-qubit gates using state preparation circuits. PyZX [46] leverages the theory of the ZX-calculus [20] to convert a quantum circuit to a semantically equivalent ZX-diagram representation and then applies rewriting rules to reduce the gates in that intermediate representation. One limitation is the lack of guaranteed efficient translation of the final ZX-diagram back to a quantum circuit, known as the circuit extraction problem. Another approach [70] uses a set of five fixed rules that are applied according to heuristically defined schedules.

Context-aware program transformations. Another class of methods is context-aware, meaning that they also directly or indirectly reason about the context around the circuit segment to optimize. PCOAST [84] exploits the commutative properties of the Pauli strings to merge Pauli rotations gates even in the presence of non-unitary operations, such as measurements. It relies on a Pauli-based representation of the circuit, called *PCOAST graph*, where nodes are both quantum gates and measurements, and edges represent where two gates do not commute, namely their execution order is fixed. QContext [55] uses a library of possible gate decomposition among which to choose the best variant to decompose Toffoli and CNOT gates depending on the context, namely predecessors and successors of the current gate, and hardware topology. The gate variant library is created via least square optimization or exhaustive search. Borgna et al. [12] extends the ZX-calculus to a hybrid quantum-classical circuit model that includes both quantum and classical operations, and then applies rewriting rules to reduce the number of gates.

Automatically deriving transformations. Instead of relying on a fixed set of rules, another group of approaches looks for new rewrite rules automatically. The problem of discovering rules consists in finding a set of circuits, typically small ones, that are equivalent to each other. Thus, each pair of circuits in the set forms a rewriting rule. Quartz [118] automatically discovers new rewriting rules by using the *RepGen* algorithm that efficiently constructs via recursion all equivalent classes of circuits with n gates and q qubits, called (n, q)-complete equivalent circuit classes. To find

potentially equivalent circuits, it uses a fast fingerprinting mechanism, followed by a more rigorous and expensive check with an SMT solver. The new rewriting rules are then applied to the circuits using a cost-based backtracking search algorithm, where the cost is the number of gates in the circuit. QUESO [117] automatically discovers new rewriting rules by using a novel data structure called *polynomial identity filter* to efficiently create clusters of equivalent circuits. This mechanism converts each circuit into a polynomial and checks if two polynomials derived from different circuits are equal under some randomized evaluation of their variables. The rewrite rules are then applied with a beam-search algorithm that uses a cost function based on the number of gates in the circuit. Another work [65], which we call *SAT-based CNOT*, uses a SAT solver to find the optimal rewrite rules to reduce the number of CNOT gates in all subcircuits that use CNOT and T gates. They rely on a *phase polynomial representation* that represents each circuit using only CNOT and T gates with a linear reversible function g and a polynomial $p(x_1, ..., x_n)$ defining a diagonal phase transformation. Note that more than one circuit can share the same representation and the goal of the approach is to iteratively find the representation with the smallest number of CNOT gates by solving a SAT problem.

Classical approaches. Some approaches leverage classical optimization techniques to optimize quantum programs. For example, QIRO [41] and QSSA [85] both leverage the Multi-Level Intermediate Representation (MLIR) to explicitely represent the dataflow in quantum programs via a static single assignment (SSA) graph, but instead of representing flow of classical data, they represent the flow of quantum state from a gate to another. MLIR allows both to apply classical optimization techniques to quantum programs. QIRO applies classical optimizations like sub-expression elimination and function inlining, along with quantum-specific ones like gate cancellation and loop optimization. QSSA focuses on quantum-specific optimizations like redundancy and dead-code elimination.

Approximate synthesis. Approximate synthesis involves replacing a set of gates with a shorter sequence that is close to, but not exactly, the original circuit. Parametrized Circuit Instantiation [122] uses a concept called numerical instantiation to replace a usually long sequence of gates with a shorter parametrized circuit that is then optimized in its parameters to find an equivalent circuit via numerical optimization. Note that the optimization does not guarantee an exact solution. QUEST [83] splits a circuit into partitions and generates approximate versions of each partition via *approximate synthesis* techniques such that the number of CNOT gates is reduced. It uses dual annealing optimization to combine partitions and generate a set of diverse versions of the full circuit, such that the output of running the set is both close to the ideal distribution and with lower number of CNOT gates.

Learning-based strategies. Beside knowing the rules to apply, it is crucial to know when to apply them. For instance, Quarl [53] replaces the backtracking search of Quartz with a reinforcement learning agent. It represents a quantum circuit as a graph and then processes it with a graph neural network.

6.2 Reducing the Number of Qubits (G2)

Reducing the number of qubits is a crucial optimization goal to address the constrained resources challenge (C2). In particular, since most programming languages allow the user to manage memory allocation by allocating quantum memory, the user might sometime use qubits in a suboptimal way, and hence, an optimization can try to reduce the number of qubits used by the program. The example in Figure 10b shows a quantum program that uses three qubits that would not fit on a two-qubit quantum computer, whereas the optimized version uses only two qubits.

The example in Figure 10b is optimized by the approach of Paler et al. [76]. The approach uses the key insight that qubits are usually not needed during the entire computation, but only between their initialization and measurement.

The circuit is represented as a *causal graph*, where nodes are gates, inputs, or outputs. Edges represent the data-flow dependencies between gates, i.e., the fact that one gate takes the output quantum state of another gate as its input. Paler et al. [76] propose two heuristics to search for the best candidates for wire recycling, based on the pre-defined wire ordering of circuit synthesis methods or the time proximity between two gates in the causal graph.

Uncomputation. A significant cluster of approaches [9, 25, 80, 82] that reduce the number of qubits are based on the idea of *uncomputation*, where the value of the auxiliary qubits that are no longer needed are reverted back to their initial state of $|0\rangle$ by applying the inverse of the operations that were applied to them. These auxiliary qubits are called *ancilla qubits* and are used to store intermediate results. Uncomputation is needed in quantum programs because when using ancilla qubits, the quantum state of the ancilla qubits is entangled with the quantum state of the other qubits, and simply discarding them via reset or measurement would also have unwanted side effects on the other qubits. SQUARE [25] combines uncomputation with a cost-benefit analysis to decide which qubits to uncompute. Unqomp [80] avoids adding redundant gates by automatically synthesizing uncomputation blocks whenever there are nested subroutines. They represent the program as a circuit graph, which is a direct acyclic graph with gates as nodes and dependencies as edges. Reqomp [82] extends Unqomp by introducing the concept of recomputation, where instead of uncomputing an ancilla qubit after its last use, it is uncomputed earlier and then recomputed when needed. This leads to further reduction in the number of qubits used as ancilla. To achieve this, Reqomp extends the circuit graph representation of Unqomp by adding value indices that track the state of qubits. Silq [9] is a high-level quantum programming language that provides a type system and type checker to better identify uncomputation opportunities. In particular, Silq introduces the *QFree* and *MFree* function type to annotate functions whose semantics can be described classically and functions without measurements, respectively. This information is relevant for uncomputation because a *QFree* function introduces no entanglement, making uncomputation easier, whereas knowing that a function is *not MFree* means that the function has measurements, which are not reversible and thus the function cannot be automatically reverted by uncomputation.

6.3 Minimizing Communication (G3)

Current quantum computers typically allow each physical qubit to interact with only a few others (C3). We refer to the qubit indices used when writing a quantum program as *logical qubits* or *pseudo qubits*, whereas we denote the corresponding qubit indices used on hardware as *physical qubits*. The problem of finding a suitable correspondence between the logical qubits and the physical qubits is referred to by different names: *qubit mapping* [135], *placement* [97], and *allocation* [99]. When two qubits need to interact but are not directly connected in hardware, we need to introduce extra gates to move the quantum state from one physical qubit to another until the two logical qubits are physically connected, perform the operation, and then move the quantum state back. This is done by adding *swap gates* to connect the two states. This concept of moving quantum information along a specific path of gates is known as *routing* [21]. Unfortunately, because each swap gate introduces noise and errors, the optimization pass needs to minimize the number of swap gates inserted, or in other words, minimize the communication cost between the logical qubits. Figure 10c illustrates how a poor mapping can triple the number of swap gates compared to an optimal one.

Figure 11 provides a detailed example, showing both the *connectivity graph* (logical qubits and their interactions in the program) and the *coupling graph* (physical qubit connections). In the top-center, we show the logical program, i.e., the circuit before being mapped to hardware, where each qubit can potentially interact with any other qubits. However, the initial mapping does not allow the first gate to be performed due to a mismatch between the connectivity graph and the coupling graph. The first CNOT gate g_0 requires qubits q_0 and q_2 to interact, but in the initial mapping they map

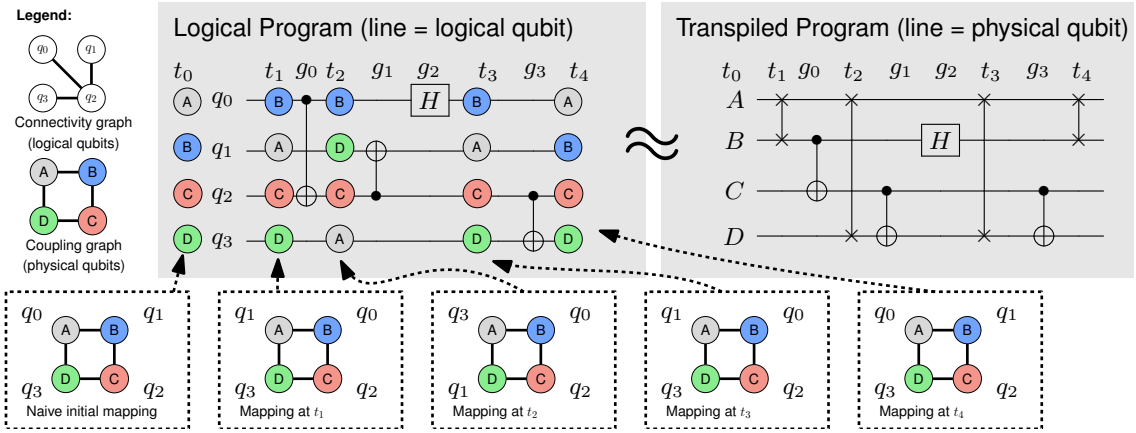


Fig. 11. Example of a quantum program's logical view (left) and physical view (right). Hardware constraints necessitate remapping logical qubits to physical ones during computation to ensure all two-qubit operations can be executed on the square coupling graph.

to physical qubits A and C, which are not connected. Thus, the compiler must adjust the mapping. This happens at various stages during the computation and each change of mapping corresponds to one or more additional swap gates, as shown in the transpiled program of Figure 11.

The difficulty in achieving this goal is that the qubit mapping problem is NP-complete [99], thus the optimization pass needs to rely on heuristics to find a good mapping. The number of possible mappings or permutations is large and grows factorially ($O(n!)$) with the number n of qubits. In particular, mappings that may initially seem efficient may become inefficient later in the circuit. As a result, the optimization pass needs to efficiently explore the search space.

To mitigate the impact of extra gates introduced by the mapping, various methods aim to minimize added gates or limit circuit depth. Zulehner et al. [135] and Zhang et al. [126] both use A* search to find optimal mappings efficiently. They rely on an *admissible heuristic* to guide the search, ensuring the cost (number of swap gates) is always underestimated, thus guaranteeing an optimal solution. The two approaches [126, 135] differ in search space exploration. The former [135] explores layer-by-layer, where a layer is a set of gates executable in parallel with a single mapping. The latter [126] explores globally, aiming to reduce circuit depth and execution time.

Other approaches use heuristics based on the configuration of the connectivity graph and on the number of pairs of logical qubits interacting via gates. For instance, Kole et al. [47] propose a genetic algorithm to minimize a cost function that includes the distance between two logical qubits in terms of how many swap gates are needed to connect them physically, but also on how many gates are executed between them. WPM [99] uses the connectivity graph for initial mapping, offering an exact dynamic programming solution for small circuits and a heuristic for larger ones.

The presence of a gate between two qubits can be seen as a constraint to be considered by a mapping. Approaches that follow this view apply either satisfiability modulo theories (SMT) solvers [101, 112] or mixed integer programming (MIP) [97]. The constraints in these mathematical representations encode the connections between physical qubits and between logical qubits. Once encoded as constraints, the SMT solver or MIP can find the optimal mapping by minimizing the number of swap gates introduced. However, since solvers are expensive to run, the innovation of each approach is on how they introduce extra constraints to reduce the constraint solving time. OLSQ [101] comes in two variants: The base OLSQ approach considers all constraints and sends them to the Z3 solver. The OLSQ-TB approach

is more efficient because it reduces redundant mapping variables between so-called transitions, i.e., sets of parallel SWAP gates that are executed between two gates of the original circuit. Wille et al. [112] use a similar approach, but approximates the full SMT problem formulation by noticing that usually only a subset of physical qubits is used by the logical qubits. Furthermore, their approach introduces swap gates only before certain, heuristically defined gates. Shafaei et al. [97] formulate the qubit mapping problem as a MIP problem, where the circuit is chunked into layers of m consecutive gates, and then the mapping is optimized for each layer, instead of considering the connectivity graph of the entire circuit. However, the solution by the MIP solver is not guaranteed to be correct because some logical qubits that should interact might no be close to each other in the final mapping. Thus, a further check is needed to ensure the correctness of the mapping, followed by occasional re-mapping.

Another way to address the limited connectivity challenge (C3) is via uncomputation, prioritizing uncomputation and reusing nearby qubits. For example, SQUARE [25] introduces a locality-aware allocation strategy to decide which qubit to uncompute based on its costs. The intuition is that recycling a nearby qubit via uncomputation may be better than moving a fresh qubit from a distant location. To balance and optimize this trade-off between communication and uncomputation, SQUARE uses a cost-benefit analysis to decide whether to uncompute a qubit or not.

All the previous approaches assume that all physical qubits are identical to each other, and hence, their connections with other qubits are equally reliable. However, the error rates of applying gates on different physical locations varies. Some approaches [69, 103] consider the gate fidelity of different qubits when addressing the qubit mapping problem. Tannu and Qureshi [103] propose two schemes called variation-aware qubit movement (VQM) and variation-aware qubit allocation (VQA) to optimize the movement and allocation of qubits to avoid weak qubits and links. To do so, their approach collects device characteristics, such as gate fidelities, and uses them to guide the qubit mapping. Then, when deciding along which path in the coupling graph to move a pair of qubits, VQM will consider the shortest paths and those that are at most $k = 4$ nodes longer, to pick the best path with the highest reliability of the links. Instead, VQA augments the layer-by-layer A* [135] baseline algorithm by computing the most reliable mapping at each layer and by then using the existing A* search to find an optimal way to connect the mapping across layers. Murali et al. [69] show that using an SMT solver that jointly minimizes the number of swap gates and the error rate of the final circuit improves program success rates. They rely on benchmark data to estimate the error rates of the gates in the hardware.

Another popular heuristic, which is now part of the Qiskit compiler, is SABRE [50]. It leverages the insight that, because quantum programs are reversible, finding an optimal mapping for the original program is equivalent to finding an optimal mapping for the reversed program. SABRE analyzes the circuit in the original order to generate a mapping, and then updates it by analyzing the circuit in the reversed order, thus considering global circuit information.

Learning-based approaches. Acampora and Schiattarella [3] propose a learning-based approach to the qubit mapping problem, where they use a deep neural network to predict the best mapping for a given quantum program by learning from a dataset of the best mapping found by the Qiskit transpiler [4]. The problem is cast as a classification problem, where the input is the quantum program and the output is the best mapping for each qubit. Since a purely learning-based approach can still generate infeasible solutions, they perform a post-processing step that ensures the generated mapping to be feasible. Pozzi et al. [88] use reinforcement learning to find the optimal mapping, where a reward is given each time that two-qubit states are brought close enough to each other in the hardware that a two-qubit gate can be executed. Analogously to reinforcement learning, Venturelli et al. [108] cast the qubit mapping problem as a temporal planning problem and use a temporal planner to find the optimal mapping.

6.4 Gate-Set-Independent Optimizations (G4)

As current quantum computers rely on different technologies, the operations they can perform on the qubits may vary. For example, the gate set used by IBM quantum computers includes the following basic gates: $U1(\lambda)$, $U2(\phi, \lambda)$, $U3(\theta, \phi, \lambda)$, and the two-qubit CX gate. Other vendors and approaches express their optimizations in different gate sets, such as the RzQ gate set used by Hietala et al. [37], which includes the H, X, $R_zQ(q)$, and CX gates. To address the challenge of varying gate sets, optimizations ideally should be gate set independent, i.e., able to optimize a quantum program regardless of the gate set being used. The example in Figure 10d shows a pictorial representation of the gate set variation challenge (C4) and its mitigation by being gate set independent and having a hardware-agnostic optimization pass. The difficulty lies in reasoning about generic unitaries applied to qubits, often involving matrix properties, rather than relying on specific gate properties. Given the great diversity of gate sets, optimizations that are highly effective for a specific gate set might not be effective for another gate set, thus automatic discovery of new rewrite rules is crucial.

Most of the approaches that are gate set independent have mechanisms to automatically discover new rewrite rules. Two examples mentioned above are Quartz [118] and Queso [117], which automatically discover new rewrite rules by using a fast mechanism to create clusters of equivalent circuits from which rewrite rules can be directly extracted for any gate set. Another approach that is gate set independent is parametrized circuit instantiation [122]. It uses a numerical instantiation technique to find equivalent circuits in the new target gate set. Unlike previous methods, it does not use rewrite rules but finds optimal parameters to make a circuit in the new gate set equivalent to the original.

7 Datasets and Benchmarks

This section reviews research efforts providing benchmarks and evaluation settings for testing and analyzing quantum programs. We categorize datasets into general-purpose and task-specific. Table 7 summarizes these datasets and their characteristics. We cover the following aspects for each dataset: (a) program size, typically in qubits or files; (b) dataset size: number of programs or files; (c) source: collected (e.g., from GitHub) or generated (e.g., from templates); (d) format: language or format used; (e) characteristics: unique features of the programs. We exclude datasets for quantum machine learning and error correction codes, as they are not directly related to quantum software testing and analysis. We also exclude datasets with only randomly generated circuits without specific applications.

7.1 General-Purpose Datasets

By general-purpose dataset, we mean a collection of diverse quantum programs without task-specific details or ground truth. SupermarQ [105] includes 52 circuits ranging from 3 to 1000+ qubits. Generated using templates from eight benchmark applications, these programs cover diverse features measured by metrics like entanglement ratio, critical depth, and liveness. Each program has a known solution or is efficiently simulatable on classical computers. QASMBench [49] includes 48 circuits of varying sizes, divided into small (2-5 qubits), medium (6-15 qubits), and large (15+ qubits). The latest version has 137 circuits, with the largest at 433 qubits. The dataset has been used to evaluate testing tools like Giallar [104] and analysis tools like QSSA [85]. MQT Bench [90] includes 70,000 circuits ranging from 2 to 130 qubits. It supports benchmarking at four abstraction levels: algorithmic, target-independent, target-dependent with native gates, and target-dependent with specific connectivity graphs. Programs are in OpenQASM, generated from templates.

The QED-C Benchmark [61] contains application-oriented benchmark programs divided in three categories: tutorial, subroutines, and functional. Each benchmark program is scalable, i.e., representing a family of circuits that can be

Table 7. Dataset of Quantum Programs. * means dataset not shared by the authors.

Name	Program size	Dataset size	Source	Format	Characteristics
General-purpose					
SupermarQ [105]	3-1000+ qubits	52	Generated	OpenQASM	8 applications
QasmBench [49]	433 qubits	137	Generated	OpenQASM	9+ domains
MQT-Bench [90]	2-130 qubits	70,000	Generated	OpenQASM	4 abstr. levels
QED-C Benchmark [61]	Configurable	14+	Generated	Python (Qiskit)	14 applications
Task-specific: Correctness					
Bugs4Q [130]	Single file	42	GitHub	Python (Qiskit)	Single platform
Paltenghi and Pradel [77]	Multi-file	223	GitHub	Multi-language	18 platforms
Zhao et al. [129]*	-	391	GitHub	Multi-language	22 frameworks
Muskit [63]	Configurable	-	Generated	Python (Qiskit)	-
QMutPy [26]	Configurable	-	Generated	Python (Qiskit)	-
Task-specific: Optimization					
RevLib [113]	Configurable	154+	Generated	Custom format	8 families
Queko [102]	Up to 900 qubits	900+	Generated	OpenQASM	Mapping stage
Acampora et al. [4]	2-15 qubits	47,111	Generated	Connectivity Graph	Calibration data
Mori et al. [67]	Configurable	-	Generated	Python (Qiskit)	-

automatically generated. At the time of writing, the official repository contains 14 benchmark circuits where the number of qubits is configurable up to what is supported by the simulator. The programs are written in Qiskit, Cirq, and Braket. However, only the Qiskit version is available for all the benchmark circuits. Although mainly designed to benchmark hardware, the benchmark could also be used with testing and analysis techniques.

7.2 Task-Specific Datasets

These datasets either contain known bugs to evaluate the correctness of testing tools, and bug detectors, or they are used to validate the effectiveness of analysis tools, such as mapping or optimization tools.

Correctness Benchmarks. Benchmarks targeted at the correctness of quantum programs typically are programs with known bugs, and optionally their corresponding fixes. Often, these programs are extracted from the version history of quantum computing repositories. Regarding the correctness of quantum circuits, Bugs4Q [130] contains 42 buggy programs written in Qiskit. They are sourced from developers' discussions on GitHub issues and StackOverflow, and each is accompanied by a corresponding bug fix. The programs are small circuits or excerpts of larger programs. For example, LintQ [79] uses this dataset to assess the recall of its static analysis in identifying programming issues. Regarding the software of the platforms, Paltenghi and Pradel [77] collected 223 buggy program from the version history of 18 quantum computing platforms. Each bug is accompanied by a corresponding minimal bug fix including multi-file fixes. Analogously Zhao et al. [129] collected 391 real-world bugs, but from 22 quantum machine learning frameworks, such as PennyLane and TorchQuantum. The dataset is not publicly available, however. Besides collecting bugs, other approaches generate mutants of quantum circuits to evaluate the effectiveness of testing tools in detecting bugs. Muskit [63] generates mutants of quantum circuits using mutation operators like gate replacement, insertion, and deletion. QMutPy [26] extends the same idea by ensuring replaced gates match the original gate's input qubits.

Both tools operate on Qiskit programs and can generate numerous mutants. Usandizaga et al. [107] use Muskit to study mutant characteristics, releasing a dataset of 700,000 mutants from 382 real-world circuits.

Optimization Benchmarks. Another popular task-oriented family of datasets is used to evaluate techniques that optimize quantum programs (Section 6). RevLib [113] contains reversible circuits to benchmark reversible logic synthesis tools. It contains 154 functions for generating reversible circuits, each serving as a template for a family of circuits with different sizes. The circuits are grouped into eight families, one of which is specifically for quantum gates and contains five functions. The programs include adders, multipliers, and other arithmetic circuits. The format is a custom format that describes the circuit as a list of gates. QUEKO [102] is a dataset of quantum circuits in the OpenQASM format, to be used in the mapping stage of quantum circuit compilation. The authors create the dataset by heuristically generating circuits where the optimal layout is known. The underlying circuits are designed to run on coupling graphs with up to 54 qubits. The benchmark is divided into two parts: one with circuits of low depth (up to 45) for NISQ devices, and one with high depth (up to 900) for scaling studies. Acampora et al. [4] introduce a dataset of random quantum circuits with their corresponding best layout found by the Qiskit transpiler when given a specific calibration data as input. The circuits have sizes ranging from 2 to 15 qubits, and the dataset contains 47,111 circuits. Note that only the connectivity map of each circuit is provided, not the circuit itself. This dataset is used to evaluate a learning-based approach to find the best layout for a given quantum circuit [3]. Similar to artificially generating bugs to evaluate testing tools for correctness, Mori et al. [67] propose an unoptimization approach to generate quantum circuits as a benchmark for optimization tools. The approach uses four operations: redundant gate insertion, commuting gate swapping, gate decomposition, and gate synthesis. Programs are generated in Qiskit and PyTket, and evaluated on respective platforms.

8 Outlook and Challenges for Future Research

Although initial progress has been made in the area of quantum software testing and analysis, the problems it deals with are far from being solved. This section discusses open research challenges for future work to address.

8.1 Scalability: Efficient Analysis and Optimization Techniques

Most existing analyses techniques are applied to relatively small programs, e.g., with few dozens of qubits. However, as the size of available quantum computers is increasing, we expect quantum software to follow a similar trend. In response, testing and analysis techniques will have to scale to larger programs. Some ideas to address this challenge include the use of machine learning techniques to improve the scalability of optimization methods, such as efficiently using reinforcement learning to schedule and optimize quantum operations [88, 108]. Alternatively, high performance computing methods can be incorporated in new analyses methods to capture the behavior of large programs in an efficient but precise way. In particular, results from high-performance computing for efficient quantum simulation approaches could inspire the software analysis community on how to scale their approaches. Another promising direction is to customize analyses and optimizations to specific program classes, such as optimizing for a particular circuit family to enhance efficiency.

Beyond the scalability of testing and analysis techniques, future research also will have to increase the scale of evaluations and benchmarks toward programs with hundreds or thousands of qubits. Some existing benchmarks, e.g., at the intersection with hardware benchmarking [61, 90], generate large-scale quantum programs from templates, which provides a first step toward larger-scale benchmarks. A possible bottleneck in scaling up benchmark size is the need of ground truth data to evaluate the effectiveness of the testing and analysis techniques. In particular to benchmark

mapping approaches (Section 6.3) an optimal mapping needs to be known to evaluate the quality of the mapping found by a novel approach. Similarly, for bug detection (Section 5.3) a set of known bugs is needed to evaluate the effectiveness of the tool. Some possible directions include the automatic generation of ground truth data, e.g., by using unoptimization [67] or large language models that inject realistic bugs into existing quantum programs.

8.2 Quantum-Specific Test Oracles

Test oracles for quantum software can either adapt ideas from classical computing, such as looking for program crashes, or leverage the probabilistic nature of quantum computing. However, early attempts of using statistical tests to achieve the latter [109] turn out to be ineffective when paired with an automatic test generation approach, such as fuzzing. The reason is that the probability that a statistical oracle reports a false positive increases when larger numbers of tests are generated and executed. Future work could further study the effectiveness of different statistical tests to understand which tests are suitable as a high-precision test oracle. Another direction to alleviate the oracle problem could be to leverage methods from the equivalence checking domain [86]. However, the scalability of these methods remains an open question. Simulators have proven beneficial for testing quantum platforms [78, 115]. However, extending their use to large programs is challenging due to the exponential space required for perfect simulation. Abstraction-based analysis approaches (Section 5.2.3) offer a first step to address this challenge by leveraging approximate state reasoning. More work is needed to obtain a reliable testing oracle out of those methods, which often still rely on developer specifications.

8.3 Balancing Standardization and Diversity

OpenQASM [23, 49] and Qiskit are becoming de facto standards in the quantum computing field, a trend that has proven beneficial for interoperability and tool adoption. Their widespread adoption by the software testing and analysis community is helping to solidify the real-world impact of new testing and analysis methodologies. However, as more and more techniques and datasets primarily utilize QASM or Qiskit, there is a growing concern about the loss of diversity and the ability to generalize findings. We envision future work to strike a balance between embracing these standards and exploring new languages and platforms. Some emerging trends are the potential reuse of classical compiler infrastructure and intermediate representations bridging to the LLVM and MLIR ecosystems [41, 85].

8.4 Developer Tools for Debugging

Understanding and debugging a quantum program is still significantly more difficult than, e.g., a classical Java or Python program. Important reasons are the low level at which quantum programs are written and the unintuitive quantum mechanical properties. Future work on debugging techniques could aim for intuitive program state visualizations and interactive debugging tools to support the developers. As seen in Section 4, providing a step-by-step execution of a quantum program largely remains an open challenge. Despite some initial progress [64], more work is needed, including work that seeks validation from developers regarding the usefulness of a debugging tool. A promising direction is exploiting results from the quantum error correction community for debugging, as illustrated by the work of Liu et al. [56]. The testing and debugging community could also include quantum developers in the design and evaluation of these tools, as their feedback will certainly help in understanding their most pressing pain-points.

8.5 Large Language Models for Quantum Software Testing

Large Language Models (LLMs) have shown significant potential in generating and reasoning about source code in classical computing. Fuzz4All [115] represents a successful application of LLMs in quantum platform testing, where an

LLM is used to generate valid quantum programs for testing the Qiskit platform. The abilities of LLMs in understanding recurrent patterns in quantum computing code could be used for other testing and analysis tasks. For instance, LLMs could automatically infer annotations and specifications for quantum programs, e.g., by annotating which qubit serves as ancilla, which can lower the barrier for adopting formal methods (Section 5.2). Another promising direction is using LLMs to find effective heuristics and code variants for optimizing quantum programs. Finally, LLMs could automatically fix bugs detected by static analysis tools (Section 5.3), thus automating program repair [29].

8.6 Raising the Abstraction Level

As quantum computing evolves toward more complex programs, we expect to see programming models with high-level abstractions. Currently, quantum computing programs are predominantly defined at the gate level, which can be cumbersome and limit higher-level conceptual thinking. Initiatives like OpenQASM 3 [49] and ScaffCC's High Level QASM [43] have begun addressing these challenges by introducing more abstract programming primitives. Such abstractions demand for future developments in quantum software testing and analysis to understand the higher-level structures and patterns emerging in quantum algorithms. As a concrete example, future mutation testing techniques could evolve beyond simple gate-level mutations [26] by offering more complex mutations, such as altering quantum oracle specifications, removing specific quantum calls, or recomposing circuits in novel ways. Similarly, static analysis methods (Section 5.3), such as LintQ [79], which currently operate largely at the gate level, must advance to understand and reason about more complex code constructs.

8.7 Correctness Specification for Dynamic Circuits

Most of the current quantum programs have a fixed number of operations known at compile time, followed by measurement. Recent advance in quantum hardware allow for a new class of programs where this is not always true. These programs, called *dynamic circuits*, allow for operations that depend on the measurement outcome of previous operations or on the result of some concurrent classical computation. Note that the classical computation can be done on a classical computer, but its result needs to be fed into the quantum computer at runtime so that the quantum program can use it and decide which operation to execute next. An example of dynamic circuit are the quantum neural networks [95] and the quantum variational algorithms [13], where the quantum program is a sequence of operations, which are interleaved by measurement and classical optimization routines, and those operations are iteratively executed until a certain condition is met. Program using the new hardware features open up new challenges for program analysis, e.g., when reasoning about the control flow of a program. Moreover, this new class of programs will likely need novel kinds of specification of the expected behavior, since a single output distribution is insufficient in describing the behavior across iterations. Some idea could involve the use of multiple admissible output distributions, one for each iteration of the algorithm, or a theoretical framework that reasons on the changes in output distributions between iterations.

9 Conclusion

In this survey, we give a unified overview of the critical challenges surrounding testing and analysis of quantum software. We discuss the state of the art in the field and provide an extensive overview of the existing literature, spanning several research communities, including quantum computing, software engineering, programming languages, and formal methods. We discuss how the testing problem is formulated by considering both expected and unexpected behaviors of quantum programs. We report on static analysis methods for both correctness and optimization of quantum programs.

We also provide an overview of existing datasets and benchmarks that can support future research. Finally, we highlight open challenges in the field with the hope of inspiring new work and advancements.

References

[1] Scott Aaronson and Daniel Gottesman. 2004. Improved Simulation of Stabilizer Circuits. *Physical Review A* 70, 5 (Nov. 2004), 052328. https://doi.org/10.1103/PhysRevA.70.052328

[2] Rui Abreu, João Paulo Fernandes, Luis Llana, and Guilherme Tavares. 2022. Metamorphic Testing of Oracle Quantum Programs. In *2022 IEEE/ACM 3rd International Workshop on Quantum Software Engineering (Q-SE)*. 16–23. https://doi.org/10.1145/3528230.3529189

[3] Giovanni Acampora and Roberto Schiattarella. 2021. Deep Neural Networks for Quantum Circuit Mapping. *Neural Computing and Applications* 33, 20 (Oct. 2021), 13723–13743. https://doi.org/10.1007/s00521-021-06009-3

[4] Giovanni Acampora, Roberto Schiattarella, and Alfredo Troiano. 2021. A Dataset for Quantum Circuit Mapping. *Data in Brief* 39 (Dec. 2021), 107526. https://doi.org/10.1016/j.dib.2021.107526

[5] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. 2021. Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 13–23. https://doi.org/10.1109/ICST49551.2021.00014

[6] Matthew Amy. 2019. Towards Large-scale Functional Verification of Universal Quantum Circuits. *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 2019), 1–21. https://doi.org/10.4204/EPTCS.287.1 arXiv:1805.06908 [quant-ph]

[7] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[8] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. 2023. symQV: Automated Symbolic Verification of Quantum Programs. In *Formal Methods (Lecture Notes in Computer Science)*. Springer International Publishing, Cham, 181–198. https://doi.org/10.1007/978-3-031-27481-7_12

[9] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *PLDI 2020*. 286–300. https://doi.org/10.1145/3385412.3386007

[10] Benjamin Bichsel, Anouk Paradis, Maximilian Baader, and Martin Vechev. 2023. Abstraqt: Analysis of Quantum Circuits via Abstract Stabilizer Simulation. *Quantum* 7 (Nov. 2023), 1185. https://doi.org/10.22331/q-2023-11-20-1185

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

[12] Agustín Borgna, Simon Perdrix, and Benoît Valiron. 2021. Hybrid Quantum-Classical Circuit Simplification with the ZX-Calculus. In *Programming Languages and Systems*, Hakjoo Oh (Ed.). Springer International Publishing, Cham, 121–139. https://doi.org/10.1007/978-3-030-89051-3_8

[13] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. 2021. Variational Quantum Algorithms. *Nature Reviews Physics* (Sept. 2021). https://doi.org/10.1038/s42254-021-00348-9

[14] 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*. 148–177. https://doi.org/10.1007/978-3-030-72019-3_6

[15] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (May 2020), 1–36. https://doi.org/10.1145/3363562

[16] Qihong Chen, Rúben Câmara, José Campos, André Souto, and Iftekhar Ahmed. 2023. The Smelly Eight: An Empirical Study on the Prevalence of Code Smells in Quantum Computing. In *ICSE 2023*. https://doi.org/10.1109/ICSE48619.2023.00041

[17] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. https://doi.org/10.48550/arXiv.2002.12543 arXiv:2002.12543 [cs]

[18] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai. 2023. AutoQ: An Automata-Based Quantum Circuit Verifier. In *Computer Aided Verification*. 139–153. https://doi.org/10.1007/978-3-031-37709-9_7

[19] 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. *PLDI 2023* 7 (June 2023), 156:1218–156:1243. https://doi.org/10.1145/3591270

[20] Bob Coecke and Ross Duncan. 2011. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. *New Journal of Physics* 13, 4 (April 2011), 043016. https://doi.org/10.1088/1367-2630/13/4/043016

[21] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. 2019. On the Qubit Routing Problem. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.TQC.2019.5*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.TQC.2019.5

[22] Andrew Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. Bishop, S. Heidel, C. Ryan, P. Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *TQC* (Sept. 2022), 12:1–12:50. https://doi.org/10.1145/3505636

[23] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. *arXiv:1707.03429 [quant-ph]* (July 2017). arXiv:1707.03429 [quant-ph]

[24] Dimitrina S. Dimitrova, Vladimir K. Kaishev, and Senren Tan. 2020. Computing the Kolmogorov-Smirnov Distribution When the Underlying CDF Is Purely Discrete, Mixed, or Continuous. *Journal of Statistical Software* 95 (Oct. 2020), 1–42. https://doi.org/10.18637/jss.v095.i10

[25] Yongshan Ding, X. Wu, A. Holmes, A. Wiseth, D. Franklin, Margaret Martonosi, and Frederic T. Chong. 2020. SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation. In *ISCA 2020*. 570–583. https://doi.org/10.1109/ISCA45697.2020.00054

[26] Daniel Fortunato, José Campos, and Rui Abreu. 2022. QMutPy: A Mutation Testing Tool for Quantum Algorithms and Applications in Qiskit. In *ISSTA 2022*. 797–800. https://doi.org/10.1145/3533767.3543296

[27] Daniel Fortunato, Luis Jiménez-Navajas, José Campos, and Rui Abreu. 2024. Verification and Validation of Quantum Software. In *Quantum Software: Aspects of Theory and System Design*. Springer, 93–123.

[28] Felix Gemeinhardt, Antonio Garmendia, Manuel Wimmer, Benjamin Weder, and Frank Leymann. 2023. Quantum Combinatorial Optimization in the NISQ Era: A Systematic Mapping Study. *Comput. Surveys* 56, 3 (Oct. 2023), 70:1–70:36. https://doi.org/10.1145/3620668

[29] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

[30] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing (STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866

[31] M. Gupta, A. Pathak, R. Srikanth, and P. K. Panigrahi. 2007. General Circuits for Indirecting and Distributing Measurement in Quantum Computation. (Aug. 2007).

[32] Thomas Häner, Torsten Hoefler, and Matthias Troyer. 2020. Assertion-Based Optimization of Quantum Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–20. https://doi.org/10.1145/3428201

[33] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-Based Software Engineering: Trends, Techniques and Applications. *Comput. Surveys* 45, 1 (Dec. 2012), 11:1–11:61. https://doi.org/10.1145/2379776.2379787

[34] Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. 2020. Quantum Programming Languages. *Nature Reviews Physics* 2, 12 (Dec. 2020), 709–722. https://doi.org/10.1038/s42254-020-00245-7

[35] Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021. Proving Quantum Programs Correct. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2021.21*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2021.21

[36] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–29. https://doi.org/10.1145/3434318 arXiv:1912.02250

[37] Kesha Hietala, Robert Rand, Liyi Li, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2023. A Verified Optimizer for Quantum Circuits. *ACM Transactions on Programming Languages and Systems* 45, 3 (Sept. 2023), 18:1–18:35. https://doi.org/10.1145/3604630

[38] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969). https://doi.org/10.1145/363235.363259

[39] Kentaro Honda. 2015. Analysis of Quantum Entanglement in Quantum Programs Using Stabilizer Formalism. *Electronic Proceedings in Theoretical Computer Science* 195 (Nov. 2015), 262–272. https://doi.org/10.4204/EPTCS.195.19 arXiv:1511.01572 [quant-ph]

[40] Yipeng Huang and Margaret Martonosi. 2019. Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs. In *ISCA 2019*. 541–553. https://doi.org/10.1145/3307650.3322213

[41] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization. *ACM Transactions on Quantum Computing* 3, 3 (June 2022), 14:1–14:32. https://doi.org/10.1145/3491247

[42] Sakshi Jain, Sreraman Muralidharan, and Prasanta K. Panigrahi. 2009. Secure Quantum Conversation through Non-Destructive Discrimination of Highly Entangled Multipartite States. *Europhysics Letters* 87, 6 (Oct. 2009), 60008. https://doi.org/10.1209/0295-5075/87/60008

[43] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *CF 2014*. 1–10. https://doi.org/10.1145/2597917.2597939

[44] Ana Justel, Daniel Peña, and Rubén Zamar. 1997. A Multivariate Kolmogorov-Smirnov Test of Goodness of Fit. *Statistics & Probability Letters* 35, 3 (Oct. 1997), 251–259. https://doi.org/10.1016/S0167-7152(97)00020-5

[45] Maximilian Kaul, Alexander Küchler, and Christian Banse. 2023. A Uniform Representation of Classical and Quantum Source Code for Static Code Analysis. In *QCE 2023*. https://doi.org/10.1109/QCE57702.2023.00115

[46] Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (May 2020), 229–241. https://doi.org/10.4204/EPTCS.318.14 arXiv:1904.04735 [quant-ph]

[47] Abhoy Kole, Stefan Hillmich, Kamalika Datta, Robert Wille, and Indranil Sengupta. 2020. Improved Mapping of Quantum Circuits to IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (Oct. 2020). https://doi.org/10.1109/TCAD.2019.2962753

[48] Chris Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. https://doi.org/10.48550/arXiv.2002.11054 arXiv:2002.11054 [cs]

[49] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. QASMBench: A Low-Level Quantum Benchmark Suite for NISQ Evaluation and Simulation. *ACM Transactions on Quantum Computing* 4, 2 (Feb. 2023), 10:1–10:26. https://doi.org/10.1145/3550488

[50] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. In *ASPLOS 2019*. Association for Computing Machinery, New York, NY, USA, 1001–1014. https://doi.org/10.1145/3297858.3304023

[51] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 150:1–150:29. https://doi.org/10.1145/3428218

[52] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. 2023. Quarl: A Learning-Based Quantum Circuit Optimizer. https://doi.org/10.48550/arXiv.2307.10120 arXiv:2307.10120 [quant-ph]

[53] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. 2024. Quarl: A Learning-Based Quantum Circuit Optimizer. *OOPSLA 2024* (April 2024). https://doi.org/10.1145/3649831

[54] Ji Liu, Luciano Bello, and Huiyang Zhou. 2021. Relaxed Peephole Optimization: A Novel Compiler Optimization for Quantum Circuits. In *CGO 2021*. 301–314. https://doi.org/10.1109/CGO51591.2021.9370310

[55] Ji Liu, Max Bowman, Pranav Gokhale, Siddharth Dangwal, Jeffrey Larson, Frederic T. Chong, and Paul D. Hovland. 2023. QContext: Context-Aware Decomposition for Quantum Gates. In *ISCAS 2023*. https://doi.org/10.1109/ISCAS46773.2023.10181370

[56] Ji Liu, Gregory T. Byrd, and Huiyang Zhou. 2020. Quantum Circuits for Dynamic Runtime Assertions in Quantum Computation. In *ASPLOS 2020*. 1017–1030. https://doi.org/10.1145/3373376.3378488

[57] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal Verification of Quantum Algorithms Using Quantum Hoare Logic. In *Computer Aided Verification*. 187–207. https://doi.org/10.1007/978-3-030-25543-5_12

[58] Ji Liu and Huiyang Zhou. 2021. Systematic Approaches for Precise and Approximate Quantum State Runtime Assertion. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 179–193. https://doi.org/10.1109/HPCA51647.2021.00025

[59] Peixun Long and Jianjun Zhao. 2023. Testing Quantum Programs with Multiple Subroutines. https://doi.org/10.48550/arXiv.2208.09206

[60] Peixun Long and Jianjun Zhao. 2024. Equivalence, Identity, and Unitarity Checking in Black-Box Testing of Quantum Programs. *J. Syst. Softw.* 211, C (July 2024). https://doi.org/10.1016/j.jss.2024.112000

[61] Thomas Lubinski, Sonika Johri, Paul Varosy, Jeremiah Coleman, Luning Zhao, Jason Necaise, Charles H. Baldwin, Karl Mayer, and Timothy Proctor. 2023. Application-Oriented Performance Benchmarks for Quantum Computing. *TQE* (2023). https://doi.org/10.1109/TQE.2023.3253761

[62] Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. 2022. A Comprehensive Study of Bug Fixes in Quantum Programs. In *SANER 2022*. 1239–1246. https://doi.org/10.1109/SANER53432.2022.00147

[63] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2022. Muskit: A Mutation Analysis Tool for Quantum Software Testing. In *ASE 2021*. 1266–1270. https://doi.org/10.1109/ASE51524.2021.9678563

[64] Sara Ayman Metwalli and Rodney Van Meter. 2022. A Tool For Debugging Quantum Circuits. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE Computer Society, 624–634. https://doi.org/10.1109/QCE53715.2022.00085

[65] Giulia Meuli, Mathias Soeken, and Giovanni De Micheli. 2018. SAT-based {CNOT, T} Quantum Circuit Synthesis. In *Reversible Computation*, Jarkko Kari and Irek Ulidowski (Eds.). Springer International Publishing, Cham, 175–188. https://doi.org/10.1007/978-3-319-99498-7_12

[66] Ashley Montanaro and Ronald de Wolf. 2016. *A Survey of Quantum Property Testing*. Number 7 in Graduate Surveys. Theory of Computing Library. https://doi.org/10.4086/toc.gs.2016.007

[67] Yusei Mori, Hideaki Hakoshima, Kyohei Sudo, Toshio Mori, Kosuke Mitarai, and Keisuke Fujii. 2023. Quantum Circuit Unoptimization. https://doi.org/10.48550/arXiv.2311.03805 arXiv:2311.03805 [quant-ph]

[68] Asmar Muqeet, Tao Yue, Shaukat Ali, and Paolo Arcaini. 2024. Mitigating Noise in Quantum Software Testing Using Machine Learning. *IEEE Transactions on Software Engineering* (Sept. 2024). https://doi.org/10.1109/TSE.2024.3462974

[69] Prakash Murali, N. M. Linke, Margaret Martonosi, A.J. Abhari, N. H. Nguyen, and C. H. Alderete. 2019. Full-Stack, Real-System Quantum Computer Studies: Architectural Comparisons and Design Insights. In *ISCA 2019*. 527–540. https://doi.org/10.1145/3307650.3322273

[70] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated Optimization of Large Quantum Circuits with Continuous Parameters. *npj Quantum Information* 4, 1 (May 2018), 1–12. https://doi.org/10.1038/s41534-018-0072-4

[71] Jiang Nan, Wang Zichen, and Wang Jian. 2023. Quantum Symbolic Execution. *Quantum Information Processing* 22, 10 (Oct. 2023), 389. https://doi.org/10.1007/s11128-023-04144-5

[72] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *Comput. Surveys* 43, 2 (Feb. 2011), 11:1–11:29. https://doi.org/10.1145/1883612.1883618

[73] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (tenth ed.). Cambridge University Press, USA.

[74] Shin Nishio, Yulu Pan, Takahiko Satoh, Hideharu Amano, and Rodney Van Meter. 2020. Extracting Success from IBM's 20-Qubit Machines Using Error-Aware Compilation. *ACM Journal on Emerging Technologies in Computing Systems* 16, 3 (May 2020), 32:1–32:25. https://doi.org/10.1145/3386162

[75] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 10:1–10:32. https://doi.org/10.1145/3371078

[76] Alexandru Paler, Robert Wille, and Simon J. Devitt. 2016. Wire Recycling for Quantum Circuit Optimization. *Physical Review A* 94, 4 (Oct. 2016), 042337. https://doi.org/10.1103/PhysRevA.94.042337

[77] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum Computing Platforms: An Empirical Study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 86:1–86:27. https://doi.org/10.1145/3527330

[78] Matteo Paltenghi and Michael Pradel. 2023. MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform. In *ICSE 2023*. 2413–2424. https://doi.org/10.1109/ICSE48619.2023.00202

[79] Matteo Paltenghi and Michael Pradel. 2024. Analyzing Quantum Programs with LintQ: A Static Analysis Framework for Qiskit. *FSE 2024* (July 2024). https://doi.org/10.1145/3660802

[80] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: Synthesizing Uncomputation in Quantum Circuits. In *PLDI 2021*. 222–236. https://doi.org/10.1145/3453483.3454040

[81] Anouk Paradis, Benjamin Bichsel, and Martin Vechev. 2022. Reqomp: Space-constrained Uncomputation for Quantum Circuits. https://doi.org/10.48550/arXiv.2212.10395 arXiv:2212.10395 [quant-ph]

[82] Anouk Paradis, Benjamin Bichsel, and Martin Vechev. 2024. Reqomp: Space-constrained Uncomputation for Quantum Circuits. *Quantum* (Feb. 2024). https://doi.org/10.22331/q-2024-02-19-1258

[83] Tirthak Patel, Ed Younis, Costin Iancu, Wibe de Jong, and Devesh Tiwari. 2022. QUEST: Systematically Approximating Quantum Circuits for Higher Output Fidelity. In *ASPLOS 2022*. 514–528. https://doi.org/10.1145/3503222.3507739

[84] Jennifer Paykin, Albert T. Schmitz, Mohannad Ibrahim, Xin-Chuan Wu, and A. Y. Matsuura. 2023. PCOAST: A Pauli-Based Quantum Circuit Optimization Framework. In *QCE 2023*. https://doi.org/10.1109/QCE57702.2023.00087

[85] Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. 2022. QSSA: An SSA-based IR for Quantum Computing. In *CC 2022*. 2–14. https://doi.org/10.1145/3497776.3517772

[86] Tom Peham, Lukas Burgholzer, and Robert Wille. 2022. Equivalence Checking of Quantum Circuits With the ZX-Calculus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 3 (Sept. 2022), 662–675. https://doi.org/10.1109/JETCAS.2022.3202204

[87] Simon Perdrix. 2008. Quantum Entanglement Analysis Based on Abstract Interpretation. In *Static Analysis (Lecture Notes in Computer Science)*, María Alpuente and Germán Vidal (Eds.). Springer, Berlin, Heidelberg, 270–282. https://doi.org/10.1007/978-3-540-69166-2_18

[88] Matteo G. Pozzi, Steven J. Herbert, Akash Sengupta, and Robert D. Mullins. 2022. Using Reinforcement Learning to Perform Qubit Routing in Quantum Compilers. *ACM Transactions on Quantum Computing* 3, 2 (May 2022), 10:1–10:25. https://doi.org/10.1145/3520434

[89] John Preskill. 2018. Quantum Computing in the NISQ Era and Beyond. *Quantum* 2 (Aug. 2018), 79. https://doi.org/10.22331/q-2018-08-06-79 arXiv:1801.00862

[90] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* 7 (July 2023), 1062. https://doi.org/10.22331/q-2023-07-20-1062

[91] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2019. ReQWIRE: Reasoning about Reversible Quantum Circuits. *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 2019), 299–312. https://doi.org/10.4204/EPTCS.287.17 arXiv:1901.10118 [cs]

[92] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2018. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. *Electronic Proceedings in Theoretical Computer Science* 266 (Feb. 2018), 119–132. https://doi.org/10.4204/EPTCS.266.8 arXiv:1803.00699 [cs]

[93] Salonik Resch and Ulya R. Karpuzcu. 2021. Benchmarking Quantum Computers and the Impact of Quantum Noise. *Comput. Surveys* 54, 7 (July 2021), 142:1–142:35. https://doi.org/10.1145/3464420

[94] J.C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[95] Maria Schuld and Francesco Petruccione. 2021. *Machine Learning with Quantum Computers*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-030-83098-4

[96] Manuel A. Serrano, José A. Cruz-Lemus, Ricardo Perez-Castillo, and Mario Piattini. 2022. Quantum Software Components and Platforms: Overview and Quality Assessment. *Comput. Surveys* 55, 8 (Dec. 2022), 164:1–164:31. https://doi.org/10.1145/3548679

[97] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. 2014. Qubit Placement to Minimize Communication Overhead in 2D Quantum Architectures. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 495–500. https://doi.org/10.1109/ASPDAC.2014.6742940

[98] Peter W. Shor. 1999. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Rev.* 41, 2 (Jan. 1999), 303–332. https://doi.org/10.1137/S0036144598347011

[99] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintao Pereira. 2018. Qubit Allocation. In *CGO 2018*. 113–125. https://doi.org/10.1145/3168822

[100] Adrien Suau, Gabriel Staffelbach, and Aida Todri-Sanial. 2022. Qprof: A Gprof-Inspired Quantum Profiler. *ACM Transactions on Quantum Computing* 4, 1 (Oct. 2022), 4:1–4:28. https://doi.org/10.1145/3529398

[101] Bochen Tan and Jason Cong. 2020. Optimal Layout Synthesis for Quantum Computing. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3400302.3415620

[102] Bochen Tan and Jason Cong. 2021. Optimality Study of Existing Quantum Computing Layout Synthesis Tools. *IEEE Trans. Comput.* 70, 9 (Sept. 2021), 1363–1373. https://doi.org/10.1109/TC.2020.3009140

[103] Swamit S. Tannu and Moinuddin K. Qureshi. 2019. Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. In *ASPLOS 2019*. 987–999. https://doi.org/10.1145/3297858.3304007

[104] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2022. Giallar: Push-Button Verification for the Qiskit Quantum Compiler. In *PLDI 2022 (PLDI 2022)*. 641–656. https://doi.org/10.1145/3519939.3523431

[105] Teague Tomesh, Pranav Gokhale, Victory Omole, Gokul Subramanian Ravi, Kaitlin N. Smith, Joshua Viszlai, Xin-Chuan Wu, Nikos Hardavellas, Margaret R. Martonosi, and Frederic T. Chong. 2022. SupermarQ: A Scalable Quantum Benchmark Suite. In *HPCA 2022*. https://doi.org/10.1109/HPCA53966.2022.00050

[106] Dominique Unruh. 2019. Quantum Relational Hoare Logic. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 33:1–33:31. https://doi.org/10.1145/3290346

[107] Eñaut Mendiluze Usandizaga, Tao Yue, Paolo Arcaini, and Shaukat Ali. 2023. Which Quantum Circuit Mutants Shall Be Used? An Empirical Evaluation of Quantum Circuit Mutations. https://doi.org/10.48550/arXiv.2311.16913 arXiv:2311.16913 [cs]

[108] Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. 2018. Compiling Quantum Circuits to Realistic Hardware Architectures Using Temporal Planners. *Quantum Science and Technology* 3, 2 (Feb. 2018), 025004. https://doi.org/10.1088/2058-9565/aaa331

[109] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 692–704. https://doi.org/10.1109/ASE51524.2021.9678792

[110] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Application of Combinatorial Testing to Quantum Programs. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 179–188. https://doi.org/10.1109/QRS54544.2021.00029

[111] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Generating Failing Test Suites for Quantum Programs With Search. In *Search-Based Software Engineering*. 9–25. https://doi.org/10.1007/978-3-030-88106-1_2

[112] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. 2019. Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations. In *DAC 2019*. 1–6. https://doi.org/10.1145/3316781.3317859

[113] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W. Dueck, and Rolf Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *38th International Symposium on Multiple Valued Logic (Ismvl 2008)*. 220–225. https://doi.org/10.1109/ISMVL.2008.43

[114] W. K. Wootters and W. H. Zurek. 1982. A Single Quantum Cannot Be Cloned. *Nature* 299, 5886 (Oct. 1982), 802–803. https://doi.org/10.1038/299802a0

[115] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *ICSE 2024*. 1–13. https://doi.org/10.1145/3597503.3639121

[116] Shangzhou Xia and Jianjun Zhao. 2023. Static Entanglement Analysis of Quantum Programs. In *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*. 42–49. https://doi.org/10.1109/Q-SE59154.2023.00013

[117] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2023. Synthesizing Quantum-Circuit Optimizers. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 140:835–140:859. https://doi.org/10.1145/3591254

[118] Mingkuan Xu, Z. Li, O. Padon, S. Lin, J. Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: Superoptimization of Quantum Circuits. In *PLDI 2022*. 625–640. https://doi.org/10.1145/3519939.3523433

[119] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[120] Peng Yan, Hanru Jiang, and Nengkun Yu. 2022. On Incorrectness Logic for Quantum Programs. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 72:1–72:28. https://doi.org/10.1145/3527316

[121] Mingsheng Ying. 2012. Floyd–Hoare Logic for Quantum Programs. *ACM Transactions on Programming Languages and Systems* 33, 6 (Jan. 2012), 19:1–19:49. https://doi.org/10.1145/2049706.2049708

[122] Ed Younis and Costin Iancu. 2022. Quantum Circuit Optimization and Transpilation via Parameterized Circuit Instantiation. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 465–475. https://doi.org/10.1109/QCE53715.2022.00068

[123] Nengkun Yu and Jens Palsberg. 2021. Quantum Abstract Interpretation. In *PLDI 2021*. 542–558. https://doi.org/10.1145/3453483.3454061

[124] Xiao-Dong Yu, Jiangwei Shang, and Otfried Gühne. 2022. Statistical Methods for Quantum State Verification and Fidelity Estimation. *Advanced Quantum Technologies* 5, 5 (2022), 2100126. https://doi.org/10.1002/qute.202100126

[125] Charles Yuan, Christopher McNally, and Michael Carbin. 2022. Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 30:1–30:32. https://doi.org/10.1145/3498691

[126] Chi Zhang, Ari B. Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z. Zhang. 2021. Time-Optimal Qubit Mapping. In *ASPLOS 2021*. 360–374. https://doi.org/10.1145/3445814.3446706

[127] Jianjun Zhao. 2020. Quantum Software Engineering: Landscapes and Horizons. (July 2020).

[128] Pengzhan Zhao, Xiongfei Wu, Zhuo Li, and Jianjun Zhao. Q-SE 2023. QChecker: Detecting Bugs in Quantum Programs via Static Analysis. https://doi.org/10.1109/Q-SE59154.2023.00014

[129] Pengzhan Zhao, Xiongfei Wu, Junjie Luo, Zhuo Li, and Jianjun Zhao. 2023. An Empirical Study of Bugs in Quantum Machine Learning Frameworks. In *2023 IEEE International Conference on Quantum Software (QSW)*. IEEE Computer Society, 68–75. https://doi.org/10.1109/QSW59989.2023.00018

[130] Pengzhan Zhao, Jianjun Zhao, Zhongtao Miao, and Shuhan Lan. 2021. Bugs4Q: A Benchmark of Real Bugs for Quantum Programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE51524.2021.9678908

[131] Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. 2021. A Quantum Interpretation of Bunched Logic & Quantum Separation Logic. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–14. https://doi.org/10.1109/LICS52264.2021.9470673

[132] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. 2023. CoqQ: Foundational Verification of Quantum Programs. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 29:833–29:865. https://doi.org/10.1145/3571222

[133] Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. An Applied Quantum Hoare Logic. In *PLDI 2019*. https://doi.org/10.1145/3314221.3314584

[134] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* (Sept. 2022). https://doi.org/10.1145/3512345

[135] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. Efficient Mapping of Quantum Circuits to the IBM QX Architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1135–1138. https://doi.org/10.23919/DATE.2018.8342181

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009