A Survey of Functional Testing and Validation of Quantum Circuits

Emma Andrews, Aruna Jayasena, and Prabhat Mishra
University of Florida, USA

***Abstract*—Quantum computing has the potential to solve a wide variety of classically intractable problems. However, the inherent complexity, probabilistic behavior, and fragility of quantum systems pose significant challenges to ensuring the correctness and reliability of quantum programs. In this survey, we present a comprehensive overview of functional testing and validation techniques aimed at addressing these challenges for quantum circuits. We begin by outlining the unique characteristics of quantum systems that complicate traditional testing methodologies, including the absence of classical oracles, non-determinism due to quantum measurement, and the exponential growth of the state space. We then categorize and analyze a wide range of functional testing and validation techniques that have emerged to meet these challenges, including mutation testing, fuzz testing, and coverage testing. We further explore advanced semantic and white-box methods such as property-based testing, metamorphic testing, equivalence checking, fault testing, assertions, and concolic testing. Specifically, we discuss the applicability of these testing techniques to quantum circuits and examine their limitations in the current NISQ (Noisy Intermediate-Scale Quantum) era.**

***Index Terms*—Quantum computing, quantum testing**

I. Introduction

The emergence of quantum computing promises significant computational advantages over classical computing for a variety of problems in cryptography, optimization, simulation, and machine learning. However, realizing practical quantum applications remains a daunting challenge due to the fragile and error-prone nature of quantum hardware, the probabilistic behavior of quantum circuits, and the complexity of quantum software development. As quantum systems grow in scale and complexity, ensuring the correctness and reliability of quantum programs becomes critically important.

In classical computing, mature testing methodologies have enabled decades of robust software and hardware validation. However, quantum computing introduces a fundamentally different paradigm. Qubits can exist in superposition and become entangled, deterministic computations become non-deterministic through measurement and environmental factors, and direct observation of the internal state can irreversibly alter it. These properties render many classical testing assumptions inapplicable. For instance, a traditional test oracle that compares an output to a known correct result becomes challenging to define when outputs are probabilistic or depend on quantum state collapse, particularly in the presence of noise and imperfect measurements. Moreover, the exponential growth of quantum state space with the number of qubits severely limits exhaustive testing or simulation-based validation techniques. Simulating quantum circuits classically is limited, requiring extensive computational resources as the qubits increase.

A. Quantum Testing and Validation Challenges

Figure 1 shows an overview of a typical quantum computing framework that consists of three major phases. The first phase enables specification of quantum algorithms and associated constraints using a high-level language (e.g., Rust or Python). The second phase compiles a given specification to a quantum circuit. The third phase performs mapping of the quantum circuit to enable evaluation using a physical quantum computer or simulation of quantum systems using a classical computer. There are various validation opportunities, such as validation of the specification (quantum algorithms), equivalence checking between the specification and the implementation (quantum circuits), and validation of quantum quantum circuits to ensure that it satisfies the specification.

In this survey, we primarily focus on test generation methods for validation of quantum circuits. Figure 2 highlights quantum testing challenges to motivate the need for quantum testing across all design phases. During the specification phase, logical and semantic bugs may arise from incorrect circuit structures, improper use of entanglement, or unintended side effects. During the compilation phase, optimization routines may modify circuits in ways that subtly change their behavior, especially when reducing gate count or adapting to hardware constraints. During the execution phase, quantum hardware is inherently noisy and variable, making test reproducibility difficult and emphasizing the need for robustness testing against realistic noise models.

B. Survey Highlights

This survey provides a comprehensive overview of foundational testing techniques, starting from their classical origins and extending into the emerging domain of quantum computing. Prior surveys [1] explore testing methodologies in relation to classical software testing, however do not go into depth with those from classical hardware testing. As quantum circuits evolve in complexity and scale, systematic testing becomes important to ensure correctness, robustness, and hardware-level alignment. To help transition smoothly from familiar paradigms, we first present classical versions of each technique before introducing their quantum-specific adaptations and challenges. Specifically, we discuss the following topics in detail.

- Coverage testing and test adequacy for systematic exploration of quantum program behavior.
- Assertions, and property-based testing for detecting runtime errors and bugs in quantum circuits.
- Symbolic execution and formal methods for generation of test inputs for validation of quantum circuits.

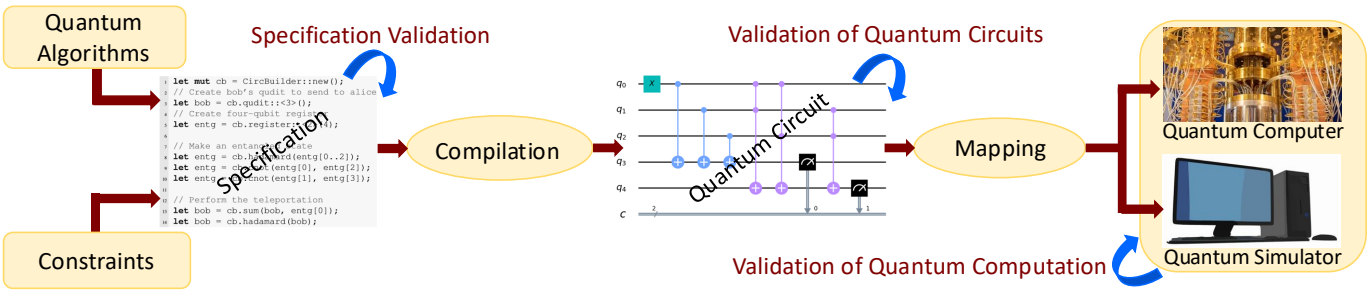


Fig. 1: An overview of a typical quantum computing framework that consists of three major phases (specification, compilation, and execution). Testing and validation are crucial at every phase of the design cycle to ensure reliable quantum computing.

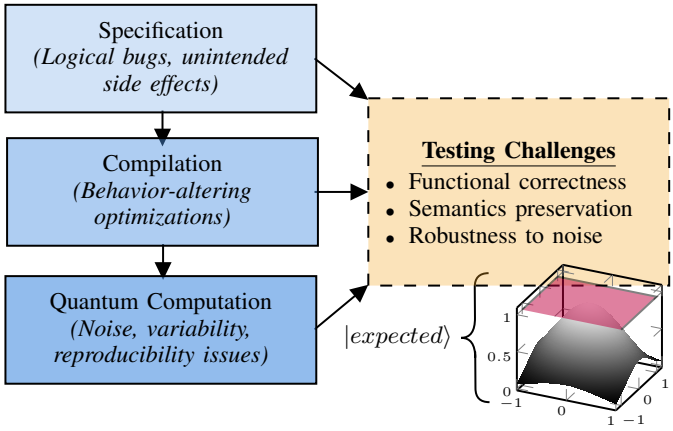


Fig. 2: Quantum testing challenges.

- Mutation testing and metamorphic testing for evaluating test suite robustness and generating adversarial test cases.

The remainder of the survey is organized as follows. Section II provides essential background on classical and quantum computation. Section III discusses coverage-based testing metrics for both classical and quantum computing. Section IV explores assertions and property-based validation techniques across classical and quantum computing domains. Section V discusses validation methods that utilize symbolic execution and formal methods. Section VI covers mutation and metamorphic testing for validating classical and quantum systems. Section VII examines differential testing. Finally, Section VIII concludes the paper with a discussion on testing strategies for future quantum systems.

II. Computing and Testing Fundamentals

This section introduces the fundamentals of quantum computing, including qubits and quantum operations. Next, it examines classical testing strategies that rely on predictable and reproducible execution, and contrast them with the new constraints introduced by quantum mechanics which necessitate a rethinking of test design, evaluation, and automation.

A. Quantum Computing Basics

Quantum computing departs from classical computing by leveraging the principles of quantum mechanics to represent and manipulate information. The basic unit of quantum information is the *qubit*. Unlike a classical bit, which can be either 0 or 1, a qubit can exist in a *superposition* of both states simultaneously. A qubit can be described by a linear combination of the computational basis states:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad \text{where } \alpha, \beta \in \mathbb{C},\ |\alpha|^2 + |\beta|^2 = 1.$$

The amplitudes α and β determine the probability of observing each basis state upon measurement: the qubit collapses to $|0\rangle$ with probability $|\alpha|^2$, and to $|1\rangle$ with probability $|\beta|^2$.

Quantum *entanglement* is a uniquely non-classical phenomenon in which the quantum states of two or more qubits become deeply correlated such that the state of each qubit cannot be described independently of the others, even when they are spatially separated. In an entangled system, the global state carries information that is not present in any individual qubit alone. A canonical example of entanglement is the two-qubit Bell state that can be illustrated as follows:

$$\left|\Phi^+\right\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle),$$

which represents a superposition where both qubits are simultaneously in the $|00\rangle$ and $|11\rangle$ states. The crucial property of this entangled state is that the outcome of a measurement on one qubit is perfectly correlated with the outcome of a measurement on the other. Entanglement introduces complexity into quantum testing, as the behavior of an entangled qubit cannot be predicted or validated in isolation, and the full system state must be considered as a whole.

Quantum algorithms are constructed as sequences of operations on qubits, where each operation is typically represented by a unitary transformation (gate) acting on one or more quantum states. These transformations manipulate the quantum state coherently, preserving the overall probability amplitude. Unlike classical circuits, which use irreversible logic gates, quantum gates are reversible and linear, enabling interference and entanglement to play a computational role.

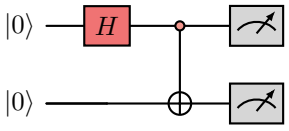


Fig. 3: Quantum circuit for Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

Figure 3 shows a simple quantum circuit that prepares the entangled Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. The circuit

begins with two qubits initialized in the $|0\rangle$ state. The first gate applied is a Hadamard gate (H) on the top qubit, which transforms it into an equal superposition state:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

Next, a CNOT (controlled-NOT) gate is applied, with the top qubit acting as the control and the bottom qubit as the target. This entangles the two qubits by flipping the target qubit only when the control qubit is in the $|1\rangle$ state. The resulting joint state is the Bell state $|\Phi^+\rangle$, an entangled state that cannot be separated into individual qubit descriptions. The final stage of the circuit involves measurement, indicated by the measurement symbols in the diagram. When measured in the computational basis, the qubits collapse to either $|00\rangle$ or $|11\rangle$, each with 50% probability.

This probabilistic and entangled behavior makes testing and validation of quantum circuits substantially more subtle than in classical systems. A single run may not reveal the correct behavior; instead, multiple executions and statistical analysis are often required to validate the correctness of quantum circuits. Moreover, measurement irreversibly collapses the quantum state, meaning intermediate inspection of a circuit (common in classical validation and debug) is not generally possible in quantum systems.

Beyond standard quantum circuits, advanced models have emerged to support more flexible and expressive computation. Parameterized quantum circuits incorporate gates with tunable parameters, which can be fixed or optimized during training, especially in hybrid quantum-classical machine learning workflows. Another powerful abstraction is the dynamic quantum circuit, which introduces classical control flow, such as 'if' and 'for' constructs, within a quantum circuit. These circuits enable expressive behaviors based on measurement outcome during execution while reducing circuit depth and width.

B. Testing in Classical Systems

In classical computing, testing is a fundamental practice to ensure that systems behave according to specifications [2]. Testing encompasses both software and hardware domains, and relies on deterministic, reproducible behavior to detect deviations from expected and unexpected outcomes. The objectives of testing classical systems span a wide range of quality concerns, from functional correctness to security assurance. These objectives guide the development of test strategies, automation frameworks, and validation tools across both software and hardware systems.

C. Challenges in Quantum Testing

While classical testing benefits from determinism and reproducibility, quantum systems introduce fundamental physical and operational constraints that challenge these assumptions. Quantum testing must contend with the unique behaviors of qubits and quantum mechanics: notably, the no-cloning theorem, probabilistic measurement, and system noise. Even challenges in classical testing, such as a large number of tests required for exhaustive testing of all possible input bits, these are compounded in quantum. Quantum computing must address entanglement and superposition in addition to all possible basis states.

No-Cloning: The *no-cloning theorem* asserts that it is not possible to create an identical copy of an arbitrary unknown quantum state. The no-cloning theorem restricts the ability to probe or duplicate quantum states during execution without disturbing them. This phenomenon poses a significant challenge for testing:

- It is impossible to clone a qubit to preserve its state while measuring it.
- The inability to duplicate state limits the use of classical test strategies such as snapshot-based testing or branching test cases from a common state.

Probabilistic Outputs and Oracle Challenges: Quantum measurement is inherently probabilistic: a qubit collapses into $|0\rangle$ or $|1\rangle$ based on the squared amplitude of its state components. This introduces uncertainty even when running the same quantum circuit repeatedly with the same input.

- A test may yield different outputs across executions due to quantum randomness.
- It is not possible to define an expected output; test oracles must specify distributions or expected probabilities.

This calls for statistical test evaluation, where the correctness of a quantum circuit is determined by comparing the observed distribution of measurement outcomes (collected over repeated executions) with the expected distribution.

Hardware Noise and Decoherence: Quantum hardware is still in its infancy and remains susceptible to various sources of noise, which can introduce errors at every stage of computation. These errors affect the fidelity of quantum operations and ultimately distort the output distribution, making reliable testing and debugging more complex.

- *Decoherence*: This refers to the loss of quantum information due to interactions between the qubits and their surrounding environment. Decoherence causes qubit values to degrade over time, impacting their superposition and entanglement as well as limiting the depth and duration of quantum computations. It is a major challenge in preserving quantum coherence during execution.
- *Readout errors*: Even after an ideal computation, the final measurement step can produce incorrect classical outcomes. These readout errors are often asymmetric (e.g., $|0\rangle$ being misread as $|1\rangle$ more frequently than vice versa) and may vary across qubits.

Such noise-induced alterations impact the distribution of measurement results and complicate the evaluation of correctness. As a result, a test failure may not always indicate a logical error in the quantum circuits, but could instead reflect a transient hardware fault or instability in the underlying device. Quantum testing frameworks must therefore disentangle true algorithmic bugs from stochastic physical effects, often relying on repeated executions, statistical validation, and noise-aware modeling to make meaningful assessments of correctness.

III. Coverage Metrics and Test Adequacy

Coverage metrics are essential for assessing how thoroughly a test suite exercises a program. Classical software relies on well-defined coverage metrics to guide test generation and evaluate adequacy. However, quantum circuits introduce unique challenges, such as superposition, entanglement, and environmental factors, that render many classical metrics insufficient. This section first reviews classical coverage foundations and then introduces quantum-specific approaches, including input/output and gate-level metrics, to address the distinct nature of quantum computation.

A. Classical Coverage Metrics

Classical software and hardware testing have long relied on well-defined coverage metrics to assess test adequacy. Statement coverage ensures that each executable instruction in the program is visited by at least one test case. Branch coverage, also known as decision coverage, ensures that every possible outcome (true and false) of each conditional statement is exercised. Gate-level coverage refers to ensuring that all logic gates (AND, OR, NOT, etc.) are activated during simulation.

B. Quantum Coverage Metrics

As classical coverage metrics fall short in capturing the probabilistic and entangled nature of quantum circuits, researchers have proposed quantum-specific alternatives. These include metrics defined at the quantum gate level, such as coverage over controlled operations like CNOT, as well as high-level test strategies such as combinatorial and search-based testing. Unlike classical branches, branching patterns in quantum can arise from multi-controlled gates and dynamic circuit constructs. Thus, gathering coverage metrics for a fixed set of executions is linear in circuit size. However, coverage definitions that require reasoning over the full quantum state space incur worst-case exponential complexity in the number of qubits.

Input/Output Level Metrics: Coverage can be measured in different ways, each considering a different facet of the quantum circuit under test. The most popular coverage metrics are input, output, input-output, and gate branch [3], [4]. Input coverage is defined as all possible basis state values for the number of qubits of the circuit acting as input. Conversely, output coverage is the output being in all possible basis state values for the number of qubits of the circuit. Input-output coverage indicates that for every input-output pair tested, all possible basis states within the pair are covered.

Quito is an automated framework for generating tests that achieve high coverage according to input, output, and input-output coverage metrics [3]. Tests are generated iteratively, with the previous iteration of tests and their coverage used as feedback to guide the generation of the next tests to improve overall coverage. Quito operates on a quantum program, and optionally, a specification of the quantum program. The specification details the expected output distributions for specific inputs and is expressed as

$$\mathrm{S}(i) = \left\{(o_1, p_1), \ldots, (o_{|O|}, p_{|O|})\right\}, \tag{1}$$

where $\mathrm{S}(i)$ is the specification under input i, $o_j \in O$ is the jth output from the set of possible output values O of the circuit, and p_j is the probability that output o_j occurred. The probabilities create a distribution, and thus $\sum_{j=1}^{|O|} p_j = 1$.

As quantum circuits can result in different outputs across different runs of the quantum circuit due to measurements and environmental factors, it is imperative that any test suite and coverage metric can determine when the generated outputs are a failure, indicating an issue with the quantum circuit. Popular metrics are as follows [3].

- *Incorrect output state:* every output generated by running the test suite is compared to the expected outputs in the specification. For every output generated with the test suite, it must also occur in the specification. In other words, if there is an output generated by running the test suite that is not listed in the specification, the test suite encountered a failure and contains a fault or other bug in the quantum circuit.
- *Incorrect output distribution:* it examines every output generated with the test suite and compares it to the expected outputs in the specification, but with a focus on the distribution of the outputs in both. The generated output distribution must be within a margin of error of the expected output distribution for a given input.

If there is no specification, the three coverages (input, output, input-output) are measured according to the encountered state values out of the total possible state values given the number of qubits.

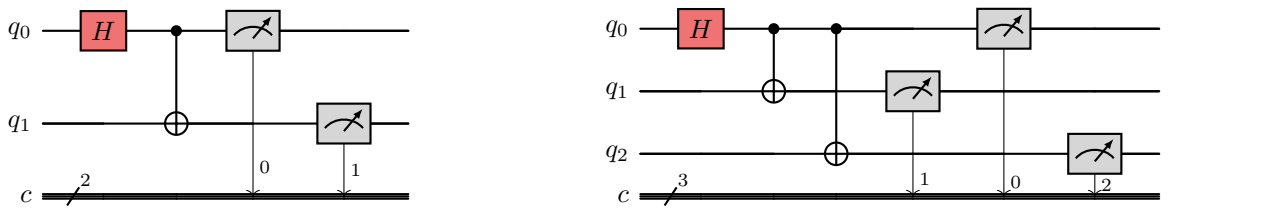
Gate Level Metrics: The classical notion of branches does not exist in traditional quantum circuits. There are some gates, collectively called controlled gates, that exhibit conditional behavior by activating a gate based on the value of another qubit. For example, CNOT, or controlled NOT, is a two-qubit gate that uses one qubit as the control signal for whether the second qubit has the NOT gate processed on its current state. Therefore, there are different definitions of branch coverage in quantum compared to classical.

Gate branch coverage is a branch coverage metric defined specifically for quantum circuits [4]. It measures the different paths taken from controlled gates, ensuring that the one or more control qubits of the controlled gate did (or did not) activate the gate on the other qubit. Specifically, the gate branch coverage can be expressed as

$$\mathrm{GBC}(G) = \frac{\sum_{g \in G} e(g)}{\sum_{g \in G} 1 + c(g)}, \tag{2}$$

where gate g in the set of controlled gates G of the quantum circuit has $e(g)$ exercised branches and $c(g)$ control qubits.

Figure 4 illustrates an example setup for getting the execution paths to inform gate branch coverage of the execution paths for the quantum circuit under test. In this example, the quantum circuit being tested is a two-qubit circuit as shown in Figure 4a, with a Hadamard gate acting on qubit q_0, followed by a CNOT gate on q_0 and q_1 such that q_0 is the control qubit for the gate. To get the execution path without using the measurement operator and collapsing the state of the qubit, the quantum circuit can be redefined such that an extra qubit,



(a) Quantum circuit to test.

(b) Quantum circuit to test with gate branch coverage measurements inserted.

Fig. 4: Gate branch coverage (b) of a basic quantum circuit with a Hadamard gate and a CNOT gate (a).

such as q_2 in Figure 4b, is inserted. Connecting a CNOT gate between q_0 and q_2 with q_0 as the control bit will cause the NOT gate to process on q_2 if q_0 is $|1\rangle$, meaning that if the measured value of q_2 after the CNOT gate indicates the NOT gate was processed, then it is known that the previous CNOT gate between q_0 and q_1 was also taken. This CNOT gate followed by a measurement with the extra qubit can provide an effective way to get the execution path of a circuit without changing its functionality at the cost of an extra qubit.

IV. Assertion and Property-Based Testing

This section first outlines assertion-based validation for classical software and hardware systems. Next, it explores assertions and property-based testing for quantum computing.

A. Classical Assertions

Assertions in classical computing, which span across both software and hardware domains, are critical tools for validating the correctness of computations, enforcing specifications, and detecting anomalies during simulation. Assertions are Boolean conditions inserted into programs or hardware designs that must evaluate to `true` during execution. If an assertion fails, it signals a deviation from the expected behavior [5].

B. Quantum Assertions

Due to the unique constraints of quantum computing, including non-deterministic outcomes, the no-cloning theorem, and measurement-induced collapse, traditional assertions used in classical programs do not directly apply to quantum programs. Instead, the inability to clone or observe quantum states without altering them introduces significant challenges for defining test oracles and writing assertions. To address these challenges, researchers have developed a variety of assertion techniques adapted to quantum contexts.

Statistical Assertions: Statistical assertions perform statistical tests on the measured quantum states as the assertion condition [6]. The statistical tests of the chi-square test and contingency table analysis are performed on three types of assertions related to the measured quantum state, as follows:

1) *Classical assertions* test whether the quantum state is a classical value, meaning a bit value of either 0 or 1, at a specific measurement.
2) *Superposition assertions* check the probability distribution of the superposition of a quantum state at a specific measurement.
3) *Entanglement assertions* examine the entanglement value of two or more entangled quantum states at a specific measurement.

Runtime Assertions: As statistical assertions require the measurement of the quantum states to check if the values are as expected, runtime assertions try to avoid using the measurement operation directly [7]. This is done to avoid collapsing the state of the qubit being measured, which can affect any circuit computations that are processed after the measurement. There are several types of runtime assertions, each testing different aspects of the quantum circuit:

1) *Dynamic assertions* allow for creating assertions around the value of qubits during runtime [7]. These can be used to measure the three different assertions within statistical assertions [6] through the use of an ancilla qubit [7]. This ancilla qubit uses CNOT gates to measure the current value of the circuit qubits. Measuring the ancilla qubit and thus collapsing its current state value does not impact the execution of the remainder of the circuit. This is similar to the strategy used to measure the gate branch coverage as described in Section III-B.
2) *Projection-based assertions* establish the assertions with projections based on Birkhoff-von Neumann logic [8]. The quantum state values to be analyzed are measured using projection measurements, which can then be compared through the assertion's projection. By doing so, the resulting projection-based assertions can provide more insight into the quantum circuit under test.
3) *SWAP assertions* adjust how the ancilla qubit is measured to allow for a broader range of states to be observed [9]. To do so, a SWAP-based circuit is used, consisting of a unitary gate U on the ancilla qubits, the inverse of the unitary gate U^{-1} on the real qubits, and finally, a SWAP gate between the real qubits and the ancilla qubits. This allows for the state in circuit under test to remain a known value as $U^{-1}U = I$ and the ancillas to be measured directly, containing a more precise state value.
4) *Approximate quantum state assertions* check if the set of states measured via the ancilla qubits is part of the set of expected state values for the circuit at that measurement point within the circuit execution [9]. This assertion is beneficial as it enables developers to approximate the results instead of having specific knowledge of the exact value the state needs to be.
5) *Concurrent assertions* run in a separate circuit to the

actual circuit under test when run on a real quantum device [10]. As both circuits are executing, a SWAP test is performed, similar to SWAP assertions, that checks if the states of both circuits that are measured are equivalent in the presence of noise. A SWAP test estimates the fidelity between two quantum states using an ancilla qubit.

Property-based Testing: Another automated testing strategy is property-based testing, which uses a model to generate test cases based on properties of the circuit under test. QSharpCheck adapts QuickCheck from classical to perform property testing on Q# programs [11]. It defines a test language for these programs, which includes definitions of the properties to test. Each property contains specific values as input and output to satisfy the property that are used in the generation of a test case for the property. These properties are often derived from the program specification. With the test cases generated from the properties, each are executed, gathering the output results from each test. Statistical analysis is performed on the resulting test output to determine the success of each test to locate potential issues. QuCheck is a similar tool that has been proposed for use with Qiskit programs [12].

The authors in [13] expand on property-based testing to include delta debugging for regression testing of the generated test cases. When a quantum circuit gets developed, it may be tested many times during the development cycle. The test cases are generated based on the properties as defined by the program specification for the final circuit. As development proceeds, these test cases are run. If a test that succeeded in the past is now failing, there was a regression in the quantum circuit, and it needs to be debugged to find the error. Delta debugging is useful, as it is a lightweight solution in pinpointing the potential cause of the regression [13]. It does so by having an oracle examine the deltas between the passing and failing circuits to find potential changes.

The adaptation of assertion and property-based testing to quantum computing represents a fundamental shift from classical verification paradigms. Quantum assertions must navigate unique challenges through innovative techniques such as statistical assertions and runtime assertions using ancilla qubits. This includes complexity, as assertion insertion and evaluation is linear for measurement-based assertions but worst-case exponential for state-based assertions. Automated frameworks discussed in this section demonstrate promising progress toward systematic quantum verification, though significant challenges remain in assertion placement, measurement trade-offs, and scalability to complex quantum systems.

V. Symbolic Execution and Concolic Testing

Symbolic and formal testing techniques play a central role in validating both classical and quantum programs by enabling systematic reasoning of quantum circuits. In classical systems, symbolic execution explores program paths using symbolic inputs and constraint solvers to uncover bugs and increase test coverage, while formal methods such as model checking and theorem proving provide strong guarantees about system correctness. As quantum programs grow in complexity, these techniques are being reimagined to account for the unique properties of quantum computation, such as superposition, entanglement, and probabilistic measurement. This section presents a comparative view of symbolic and formal testing in classical and quantum domains, beginning with classical foundations and then detailing the adaptations required for quantum systems, including path amplitude reasoning, state vector constraints, and hybrid concolic techniques suited for quantum behaviors.

A. Classical Symbolic Execution

Symbolic execution is a powerful program analysis technique used to explore multiple execution paths in a program by treating inputs as symbolic variables rather than concrete values [14]. It systematically generates path constraints and checks their feasibility using constraint solvers. The core idea is to reason about all possible execution paths that depend on symbolic inputs to uncover errors such as assertion violations, crashes, or security vulnerabilities.

Concolic testing in classical computing combines symbolic execution with concrete program execution to direct program execution into previously unexplored paths [15]. During a test run, the program is executed with concrete input values, and symbolic expressions are simultaneously tracked along the execution path. After the execution, the collected symbolic path constraints are negated one at a time to generate new input values that force the program execution through previously unexplored paths. The paths are generated one at a time, allowing for iterative steering of path generations and reducing the overall number of paths that must be analyzed.

B. Quantum Symbolic Execution

Similar to its classical counterpart, quantum equivalence checking involves comparing the behavior of two quantum circuits to determine whether they produce equivalent or identical outcomes. As quantum circuits are executed either on simulators or real quantum hardware, they often undergo compilation and transpilation into intermediate representations tailored to the target backend. This transformation process makes equivalence checking essential, not only to verify the correctness of compiler optimizations and hardware mappings, but also to ensure that the high-level quantum logic is preserved across different execution platforms.

Equivalence Checking: Burgholzer and Wille [16] design an equivalence checking framework that compares the original quantum circuit with its compiled equivalent to ensure that both circuits produce the same result. Figure 5 illustrates the overall architecture of the proposed framework. Quantum circuits have the reversibility property where the matrix associated with a gate, such as a unitary gate U, has an inverse U^{-1}. Therefore, multiplying the two gates together results in the identity matrix I, such that $UU^{-1} = I$. When the gates are reversible, the entire circuit can also be reversed resulting in the identity function $\mathbb{I}$, $GG^{-1} = \mathbb{I}$ with G as the original circuit and G^{-1} as its inverse. This idea is extended to equivalence checking by using the reversibility property.

With two quantum circuits G and G' to test, G' can invert all of its gates to result in $\mathbb{I}$ when applied with the gates of

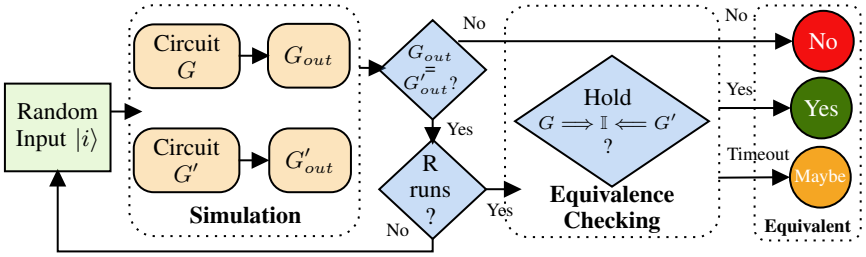


Fig. 5: Equivalence checking framework proposed by Burgholzer and Wille [16].

G. This needs to be done in a specific manner, and is aided by decision diagrams of the gates. Therefore, the decision diagram is constructed such that $G \Rightarrow \mathbb{I} \Leftarrow G'$, or the gates from G are applied normally and the gates from G' are applied as their inverses, resulting in $\mathbb{I}$. If applying the gates does not result in $\mathbb{I}$, then G and G' are not equivalent.

Equivalence, Identity, and Unitarity Checking: While equivalence checking often refers to the task of confirming that two quantum circuits produce identical outputs, Long et al. [17] provide a more nuanced breakdown of this verification task into three core problems:

1) *Equivalence Checking:* A common approach for equivalence checking between two quantum programs P and P' is the *Swap Test*, a standard quantum subroutine that estimates the overlap (fidelity) between their output states. The programs are applied to identical copies of input states sampled from the Pauli basis, chosen for its completeness, since any quantum state can be expressed as a linear combination of Pauli eigenstates. If the output states are nearly identical (within a small tolerance ϵ) across a representative set of such inputs, the two programs are considered equivalent.
2) *Identity Checking:* Verifies whether a program P performs the identity operation (i.e., $P = I$). This fidelity-based approach extends naturally to identity checking, where the goal is to determine whether a program P acts as the identity operation. After applying P to a randomly selected Pauli basis state $|\psi\rangle$, the result is passed through the inverse of the state preparation (mapping it back to $|0\rangle^{\otimes n}$), and the outcome is measured. Consistent measurement of $|0\rangle^{\otimes n}$ across trials indicates functional equivalence to the identity.
3) *Unitarity Checking:* Confirms whether P is a valid unitary operation, preserving orthogonality and state purity. To check whether a quantum program is unitary, a fundamental requirement for most quantum operations, two key properties are tested: preservation of orthogonality and consistency under superposition. This involves preparing pairs of orthogonal basis states (e.g., $|i\rangle$, $|j\rangle$) and superposition states (e.g., $\frac{1}{\sqrt{2}}(|i\rangle \pm |j\rangle)$), applying the program to them, and then using the Swap Test to verify that the output states remain orthogonal. A failure in either test suggests a violation of unitarity. Collectively, these checks ensure the program preserves linearity, orthogonality, and purity, which are the core principles of valid quantum transformations.

Concolic Testing: Quantum programs have been defined for use with concolic testing through the definition of quantum constraints, including generation techniques [18]. The overall workflow of quantum concolic testing is similar to classical with several extra steps to accommodate quantum logic, and the steps are summarized below.

1) Generation of initial test case, including inputs.
2) Creation of the symbolic objects.
3) Execution of the quantum circuit using the input values specified by the test case. The execution path is saved.
4) Generate a quantum path constraint using the quantum constraint generator on the execution path.
5) Determine a new path constraint through modification of the saved one.
6) Generate a test case by solving the new path constraint with an SMT solver.
7) Repeat from (3), checking that the desired path was executed, until coverage is satisfactory.

As such, several preliminaries must be defined in the quantum domain. These preliminaries try to match their classical counterparts as much as possible.

- *Quantum control statement* is the measurement of a specified set of qubits in the circuit and their resulting output.
- *Quantum condition* are the set of desired results from the quantum control statements.
- *Quantum symbolic object* represents the entire quantum circuit, recording the execution path and all operations along it.
- *Quantum constraint* determines if the output state from the quantum control statement satisfies all quantum conditions. It is defined by

$$\bigwedge_{a \in S \setminus S_r} (\langle a|\phi\rangle = 0) = \text{true}, \tag{3}$$

where $|\phi\rangle$ is the output state of quantum control statement, S is the set of all possible observable outputs as determined by the number of qubits, and S_r the set of output states from the quantum control statement that satisfy the quantum condition.

Quantum constraints must be generated specifically to account for the non-deterministic nature of executing quantum circuits. As one input can produce different outputs across different executions, just checking the resulting output states alone is not sufficient to determine correctness or obeying a de-

fined quantum condition. Therefore, three different constraint generation methods are used, as follows.

- *State equality* checks if the distribution of output states measured is equivalent within a margin of error to the expected output distribution, and is formalized as

$$\bigwedge_{a \in S} \left(\left| |\langle a|\phi\rangle|^2 - D_a \right| < \delta \right) = \text{true}, \tag{4}$$

  where δ is the margin of error and D_a is the expected probability distribution for a.
- *State lower limit* checks if the probability that a specific output occurred is greater than a defined lower limit within a margin of error, and is formalized as

$$\bigwedge_{s \in P \wedge s=(a_s,p_s)} \left(|\langle a_s|\phi\rangle|^2 > p_s - \delta \right) = \text{true}, \tag{5}$$

  where δ is the margin of error and p_s is the lower limit for state pair s in the set of expected state pairs P.
- *State upper limit* checks if the probability that a specific output occurred is less than a defined upper limit within a margin of error, and is formalized as

$$\bigwedge_{s \in P \wedge s=(a_s,p_s)} \left(|\langle a_s|\phi\rangle|^2 \leq p_s + \delta \right) = \text{true}, \tag{6}$$

  where δ is the margin of error and p_s is the lower limit for state pair s in the set of expected state pairs P.

In summary, symbolic execution and formal testing techniques are evolving from classical foundations to address quantum computing's unique challenges. While classical approaches rely on deterministic constraint solving, quantum adaptations must handle probabilistic measurements, superposition states, and unitary operations through novel methods like quantum constraint generation, equivalence checking via decision diagrams, and specialized concolic testing frameworks. These formal methodologies will be essential for ensuring quantum program correctness as quantum applications mature. In terms of complexity, symbolic execution techniques have worst case exponential computational complexity due to the exponential size of the quantum state space.

VI. Mutation and Metamorphic Testing

Metamorphic and mutation testing have proven effective across both classical hardware and software testing, serving as powerful techniques to assess test adequacy and uncover faults in systems where traditional test oracles may be insufficient. Mutation testing involves introducing small, systematic changes (mutants) to a program or hardware description to evaluate the ability of test cases to detect these faults [19]. Metamorphic testing leverages known relationships among inputs and outputs to validate correctness even when expected outcomes are unknown or difficult to compute. In this section, we examine the application of mutation and metamorphic testing in both classical and quantum contexts.

A. Classical Mutation Testing

Classical mutation testing is a fault-based testing technique used to evaluate the quality of a test suite. It operates by systematically introducing small changes called as *mutants* into a program's source code or a hardware description, with the goal of assessing whether the existing test cases can detect the injected faults. If a test case can distinguish the mutant from the original version (i.e., cause different outputs), the mutant is said to be *killed*; otherwise, it is considered *alive*, possibly indicating a weakness in the test suite. Software mutation testing, for example, can change operators in the code, such as altering a relational operator $>$ to $\geq$. Hardware mutation testing can alter gates, such as changing AND to OR.

B. Quantum Mutation Testing

Quantum mutation testing involves defining several mutation operations that can change the functionality of a quantum circuit minimally, with the hope of having the test suite catch the buggy execution of the resulting mutated quantum circuit. Muskit [20] and QMutPy [21] define three basic mutation operations on quantum circuits, centered around the gates. QMutPy defines two other mutation operations on the measurements within the quantum circuit.

1) *Quantum gate replacement*: Replaces a gate in the quantum circuit with another gate. The new gate must be syntactically equivalent, meaning it must use the same number of operands and produce the same number of results as the original.
2) *Quantum gate deletion*: Removes a gate from the quantum circuit.
3) *Quantum gate insertion*: Inserts a syntactically equivalent gate immediately before an existing gate.
4) *Quantum measurement insertion*: Inserts a measurement operator on a qubit.
5) *Quantum measurement deletion*: Removes a measurement operator from a qubit.

To check how well the introduced bugs into the mutants were able to be identified by the tests, a mutation score is calculated [21]. The mutation score measures the rate of how many mutants were killed against all mutants tested. Formally, this is expressed as

$$\sum_{o \in O} \frac{\frac{|K_o|}{|M_o|-|I_o|}}{|O|} \times 100\%, \tag{7}$$

where o is a mutation operator in the set of all mutation operators O, $|M_o|$ is the number of mutants o created, $|I_o|$ is the number of mutants that are incompetent meaning they did not contain changes, $|K_o|$ is the number of mutants created by o that were killed, and the number of incompetent mutants cannot be more than the number of working mutants. Mutation testing for quantum circuits is thus polynomial in the number of mutants but incurs worst-case exponential complexity due to the exponential cost of simulating n-qubit quantum states.

Fuzz Testing: Mutations can also be used to generate new inputs to test in a testing strategy called fuzz testing. An initial input can be generated following some criteria, such as the

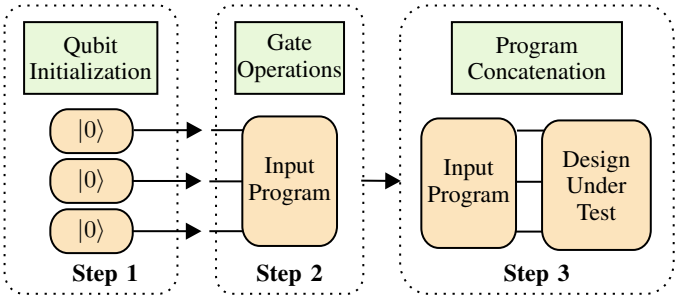


Fig. 6: Test generation process with QuraTest framework [23].

program specification, and several additional test inputs can be generated with minor modifications via mutation. QuanFuzz is a fuzz testing framework that utilizes a search-based input generator to generate inputs that achieve high coverage [22]. The inputs are taken as their density matrices, and along with their resulting measurement outputs, are analyzed for any sensitive branches. This is an execution path within the circuit, and the search-based framework mutates the input matrix to generate a new test case. The aim of this mutation is to activate the non-taken paths, until all possible paths are taken.

One drawback to standard input generation techniques is that they use basis states, such as $|0\rangle$ or $|1\rangle$, but quantum programs can exist in superposition and entanglement with other qubits. QuraTest adds an additional circuit before the quantum circuit to test to transform the original basis state input into input with superposition and entanglement [23]. These input changing circuits are generated as UCNOT using unitary and CNOT gates, IQFT using unitary gates and inverse quantum Fourier transform, and random using randomly selected gates from a specified list. Figure 6 contains an overview of the workflow with this framework. The additional input states that are able to be used with the insertion of the additional circuit allow for more dynamic testing of quantum circuits, aligning closer to realistic scenarios on real quantum devices.

C. Classical Metamorphic Testing

Metamorphic testing is a property-based testing technique designed to alleviate the *oracle problem*, where the correct output of a program is unknown or difficult to compute. Instead of requiring a precise expected output, metamorphic testing relies on *metamorphic relations* (MRs), necessary properties or transformations that relate multiple inputs and their corresponding outputs [24]. If a program violates these relations, a fault is likely present.

The workflow of metamorphic testing involves generating follow-up test cases from source test cases by applying known metamorphic relations. The results are then checked for consistency according to the expected transformation of outputs.

D. Quantum Metamorphic Testing

When a direct oracle is unavailable, which is the case when testing most of the quantum circuits, quantum metamorphic testing offers a promising alternative by defining new relationships between inputs and outputs rather than requiring exact expected values. Often, these relationships are defined from properties of quantum circuits, such as reversibility. This results in linear complexity in the number of applied metamorphic relations, but worst-case exponential due to exponential-size quantum state simulation and combinatorial growth from relation compositions.

Abreu et al. [25] defined a formal structure for specifying metamorphic relations for quantum circuits under test. The metamorphic rules are Boolean functions defining a relationship between the input and output of the quantum circuits. The implementation of the metamorphic rules in quantum circuits must be reversible. The metamorphic rules are thus crafted using unitary gates to ensure effective reversibility, such that $UU^{\dagger} = I$. These metamorphic rules can replace measurement operations within the quantum circuit to avoid collapsing the state while gaining important information about the execution of the quantum circuit. The resulting quantum circuit with the metamorphic rules inserted is referred to as an oracle quantum circuit.

MorphQ is an automated metamorphic testing framework that can establish metamorphic rules in quantum circuits under test while also being able to automatically generate the quantum circuits to test [26]. The framework begins by generating a valid quantum circuit in Qiskit to test, ensuring that it is syntactically correct and exhibits valid gate placements as illustrated by Figure 7. Next, the metamorphic transformations are inserted into the generated quantum circuit. Three categories of potential transformations can be inserted:

1) *Circuit transformations* change the circuit itself, modifying qubits and gates.
2) *Representation transformations* adjust the intermediate representation of the quantum circuit.
3) *Execution transformations* change transpilation and simulation settings within Qiskit for simulating the quantum circuit.

Once the transformations are inserted into the quantum circuit, both the original circuit and the transformed circuit are simulated, gathering their execution results. If there are any deviations in the results, it indicates the presence of a bug. We note that simulation in this work is carried out via statevector simulation, and that these relationships may need to be adjusted under the presence of tensor network simulators.

VII. Differential Testing

This section first presents classical differential testing. Then, quantum differential testing is examined.

A. Classical Differential Testing

Classical differential testing runs two different programs: a base program and a test program [27]. The base program is a known program producing correct output, whereas test program is a modified version that produces results that are unknown if correct. If the outputs differ, there is a potential for a fault in the test program. This notion can also be extended to the same program across different software stacks.

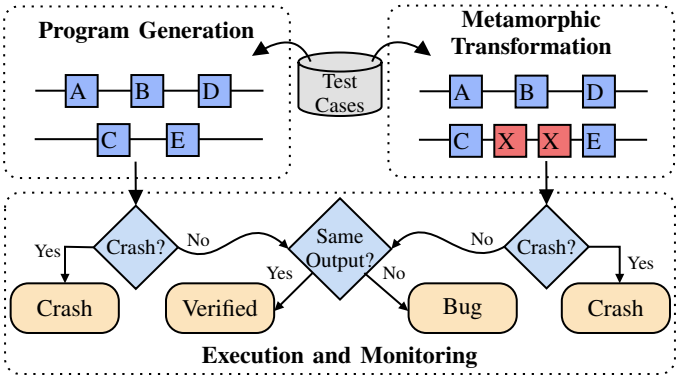


Fig. 7: Overview of MorphQ [26] quantum metamorphic testing framework with in-built quantum circuit generation.

B. Quantum Differential Testing

Quantum differential testing is a promising functional testing technique that analyzes the same quantum circuit in different quantum software stacks to ensure that equivalent output is observed. There are a growing number of quantum software stacks to simulate quantum circuits both classically and on real devices, thus, it is imperative that all quantum software stacks behave similarly for a given quantum circuit. QDiff is a differential testing framework for analyzing different quantum circuits across Qiskit, Cirq, and Pyquil [28]. In this framework, several mutation operations are used to alter the given quantum circuit and locate potential bugs within the quantum software stacks.

VIII. Conclusion

Classical computing has benefited from decades of mature testing and validation methods, including well-defined fault models, deterministic execution, and extensive tool support for test generation, mutation analysis, and formal verification. These techniques enable reliable validation even in large-scale systems, aided by benchmarks, coverage metrics, and automated oracles. Quantum computing inherits this rich legacy but also diverges significantly due to its probabilistic behavior, fragile hardware, and fundamentally different models of computation and observation. While many classical techniques, such as mutation testing, symbolic execution, and metamorphic testing, can be adapted to quantum systems, they face substantial challenges when applied to quantum circuits involving a large number of qubits due to scalability, noise, and reproducibility concerns. While advancements are being made to handle noise, such as using machine learning to learn a noise model to filter noise from output results [29], more testing techniques are needed. Another challenge is reproducibility, ensuring that the results from testing methods can be reproduced. This can be established through benchmark suites or through characterization techniques. These techniques can establish upper bounds on statistical distances such as the Hellinger distance for comparing the ideal and observed output distributions from the execution of a quantum circuit [30]. Advancements in testing techniques are thus needed to overcome these challenges and to establish a robust foundation for reliable quantum computing systems.

References

[1] N. C. Leite Ramalho, H. Amario de Souza, and M. Lordello Chaim, "Testing and Debugging Quantum Programs: The Road to 2030," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, pp. 155:1–155:46, May 2025.

[2] A. Jayasena and P. Mishra, "Directed test generation for hardware validation: A survey," *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–36, 2024.

[3] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Quito: A Coverage-Guided Test Generator for Quantum Programs," in *International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 1237–1241.

[4] D. Fortunato, J. Campos, and R. Abreu, "Gate Branch Coverage: A Metric for Quantum Software Testing," in *ACM International Workshop on Quantum Software Engineering: The Next Evolution*, ser. QSE-NE 2024, Jul. 2024, pp. 15–18.

[5] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–33, 2022.

[6] Y. Huang and M. Martonosi, "Statistical assertions for validating patterns and finding bugs in quantum programs," in *International Symposium on Computer Architecture*, Jun. 2019, pp. 541–553.

[7] J. Liu, G. T. Byrd, and H. Zhou, "Quantum Circuits for Dynamic Runtime Assertions in Quantum Computation," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2020, pp. 1017–1030.

[8] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Projection-based runtime assertions for testing and debugging Quantum programs," *Proceedings of the ACM Programming Languages*, vol. 4, no. OOPSLA, pp. 150:1–150:29, Nov. 2020.

[9] J. Liu and H. Zhou, "Systematic Approaches for Precise and Approximate Quantum State Runtime Assertion," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2021, pp. 179–193.

[10] H. Witharana, D. Volya, and P. Mishra, "QcAssert: Quantum Device Testing with Concurrent Assertions," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2024, pp. 491–496.

[11] S. Honarvar, M. R. Mousavi, and R. Nagarajan, "Property-based Testing of Quantum Programs in Q#," in *International Conference on Software Engineering Workshops*, Sep. 2020, pp. 430–435.

[12] G. Pontolillo, M. R. Mousavi, and M. Grzesiuk, "QuCheck: A Property-based Testing Framework for Quantum Programs in Qiskit," Mar. 2025.

[13] G. J. Pontolillo and M. R. Mousavi, "Delta Debugging for Property-Based Regression Testing of Quantum Programs," in *ACM/IEEE International Workshop on Quantum Software Engineering*, Aug. 2024, pp. 1–8.

[14] A. Jayasena and P. Mishra, "Firmwall: Directed symbolic execution of firmware binaries for defending against unauthorized system calls," *IEEE Transactions on Information Forensics and Security*, 2025.

[15] H. Witharana, A. Jayasena, and P. Mishra, "Incremental concolic testing of register-transfer level designs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 5, pp. 1–23, 2024.

[16] L. Burgholzer and R. Wille, "Advanced Equivalence Checking for Quantum Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 9, pp. 1810–1824, Sep. 2021.

[17] P. Long and J. Zhao, "Equivalence, identity, and unitarity checking in black-box testing of quantum programs," *Journal of Systems and Software*, vol. 211, p. 112000, 2024.

[18] S. Xia, J. Zhao, F. Zhang, and X. Guo, "Quantum Concolic Testing," Jun. 2025.

[19] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[20] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue, "Muskit: A Mutation Analysis Tool for Quantum Software Testing," in *International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 1266–1270.

[21] D. Fortunato, JOSÉ. CAMPOS, and RUI. ABREU, "Mutation Testing of Quantum Programs: A Case Study With Qiskit," *IEEE Transactions on Quantum Engineering*, vol. 3, pp. 1–17, 2022.

[22] J. Wang, M. Gao, Y. Jiang, J. Lou, Y. Gao, D. Zhang, and J. Sun, "QuanFuzz: Fuzz Testing of Quantum Program," Oct. 2018.

[23] J. Ye, S. Xia, F. Zhang, P. Arcaini, L. Ma, J. Zhao, and F. Ishikawa, "QuraTest: Integrating Quantum Specific Features in Quantum Program Testing," in *International Conference on Automated Software Engineering (ASE)*, Sep. 2023, pp. 1149–1161.
[24] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.
[25] R. Abreu, J. P. Fernandes, L. Llana, and G. Tavares, "Metamorphic testing of oracle quantum programs," in *Proceedings of the 3rd International Workshop on Quantum Software Engineering*, Feb. 2023, pp. 16–23.
[26] M. Paltenghi and M. Pradel, "MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform," in *International Conference on Software Engineering (ICSE)*, May 2023, pp. 2413–2424.
[27] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
[28] J. Wang, Q. Zhang, G. H. Xu, and M. Kim, "QDiff: Differential Testing of Quantum Software Stacks," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 692–704.
[29] A. Muqeet, T. Yue, S. Ali, and P. Arcaini, "Mitigating Noise in Quantum Software Testing Using Machine Learning," *IEEE Transactions on Software Engineering*, vol. 50, no. 11, pp. 2947–2961, Nov. 2024.
[30] S. Dasgupta and T. S. Humble, "Characterizing the Reproducibility of Noisy Quantum Circuits," *Entropy*, vol. 24, no. 2, p. 244, Feb. 2022.

Emma Andrews is a Ph.D. student in the Department of Computer and Information Science and Engineering at the University of Florida. Her research is focused on testing quantum circuits and developing new quantum machine learning strategies.

Aruna Jayasena is an Assistant Professor at the University of Tennessee at Chattanooga. He received his Ph.D. in the Department of Computer and Information Science and Engineering at the University of Florida in 2025. His research focuses on heterogeneous system design, applied cryptography, trusted execution, and hardware-firmware co-validation.

Prabhat Mishra is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include embedded and cyber-physical systems, hardware security and trust, and energy-aware computing. He is an IEEE Fellow, an AAAS Fellow, and an ACM Distinguished Scientist.