Verification and Validation of Quantum Software

Daniel Fortunato, Luis Jiménez-Navajas, José Campos, and Rui Abreu

Abstract Quantum software—like classic software—needs to be designed, specified, developed, and, most importantly, tested by developers. Writing tests is a complex, error-prone, and time-consuming task. Due to the particular properties of quantum physics (e.g., superposition), quantum software is inherently more complex to develop and effectively test than classical software. Nevertheless, some preliminary works have tried to bring commonly used classical testing practices for quantum computing to assess and improve the quality of quantum programs. In this chapter, we first gather 16 quantum software testing techniques that have been proposed for the IBM quantum framework, Qiskit. Then, whenever possible, we illustrate the usage of each technique (through the proposed tool that implements it, if available) on a given running example. We showcase that although several works have been proposed to ease the burn of testing quantum software, we are still in the early stages of testing in the quantum world. Researchers should focus on delivering artifacts that are usable without much hindrance to the rest of the

D. Fortunato (✉)
Faculty of Engineering of University of Porto, Porto, Portugal

LIACC—Artificial Intelligence and Computer Science Laboratory (member of LASI LA), Porto, Portugal
e-mail: dabf@fe.up.pt

L. Jiménez-Navajas
aQuantum, Faculty of Social Sciences & IT, University of Castilla-La Mancha, Talavera de la Reina, Toledo, Spain
e-mail: Luis.JimenezNavajas@uclm.es

J. Campos
Faculty of Engineering of University of Porto, Porto, Portugal

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal
e-mail: jcmc@fe.up.pt

R. Abreu
Faculty of Engineering of University of Porto, Porto, Portugal

INESC-ID, Lisboa, Portugal
e-mail: rui@computer.org

I. Exman et al. (eds.), *Quantum Software*,
https://doi.org/10.1007/978-3-031-64136-7_5

community, and the development of quantum benchmarks should be a priority to facilitate reproducibility, replicability, and comparison between different testing techniques.

Keywords Quantum software · Verification and validation · Software testing

1 Introduction

In the last few years, quantum computing has evolved enormously in many aspects. It was not until 2019 that IBM unveiled its first commercial quantum computer with 20 qubits [1] and, in 2022, the same company developed a quantum computer with 433 qubits [2]. In addition, these hardware breakthroughs have been accompanied by software, where the largest companies in the world have created quantum programming languages [3] (such as Microsoft with Q# or IBM with OpenQASM), libraries to develop quantum software (such as Google with Cirq or IBM with Qiskit), or services to run and design quantum software (such as Amazon with Braket).

The entire ecosystem that quantum computing vendors have built allows users and organizations to develop and run quantum software in a straightforward manner [4]. This implies that, at some point, organizations that can take advantage of the potential benefits of this new technology will design and develop quantum components that can provide them with speedup. In other words, quantum software will be developed in a large-scale industrial context in the same way that classic software is nowadays produced [5].

Quantum software, as classical software, will, at some point in its development life cycle, need to be tested [6]. Apart from the evaluation of the functionality of the quantum software, concerns related to security vulnerabilities can also appear in this new programming domain [7].

However, we face three main challenges when testing quantum software [8]. First, unlike classical computing, with quantum computing, we cannot read the state of qubits at any time. If a qubit in superposition is measured, its state collapses. Second, the inherent nature of this new paradigm is non-deterministic. This implies that we will likely get a different result every time we run the quantum software. Third, the fact that current quantum computers are sensitive to noise and are fault-tolerant implies that when we run a quantum program and the result is different than expected, we cannot be sure whether the failure is caused by noise or by natural randomness.

Over the past few years, several approaches have been developed to alleviate the challenges associated with quantum testing. Regarding the verification of quantum programs, one can find works based on Hoare logic [9, 10, 11] or static analysis of source code [12, 13, 14, 15]. Concerning the validation of quantum programs, there are works related to the generation of data inputs aiming at detecting faults [16, 17,

18], oracle generation [19, 20], and a combination of both techniques [21, 22, 23, 24, 25, 26].

This chapter details current testing approaches used to help developers verify and validate their quantum software. More specifically, we focus our analysis on testing approaches designed to test quantum circuits since most quantum software is written through the application of quantum gates to quantum circuits. Consequently, we only present techniques and tools designed for circuit-based techniques. For instance, testing techniques for quantum annealing [27] are not included. Additionally, given that Qiskit [28], the circuit-based IBM framework, is one of the most popular quantum software development frameworks, we focus our analysis on works that use it.

This chapter is organized as follows. We present some concepts and definitions in Sect. 2. In Sect. 3, we discuss techniques that have been proposed for quantum software testing. Section 4 discusses current quantum fault benchmarks. We discuss some limitations of quantum software testing in Sect. 5 and conclude the chapter in Sect. 6.

2 Concepts and Definitions

2.1 Quantum Computing

Given that quantum computing is an emerging field, the definition of certain key concepts is warranted.

Qubit Unlike classical computers that use bits, quantum computers use the quantum bit (qubit for short) as their fundamental unit of memory. A qubit, just like the bit, has a state that can be $|0\rangle$ or $|1\rangle$, but contrary to the bit, those are just two possible states. The Dyrac notation, '$|\rangle$', is used to represent states in quantum mechanics. The difference between classic states and quantum states is that quantum states can be in superposition [29], meaning that it is possible to form linear combinations of states. A qubit can be expressed as $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$.

Unlike the classical bit, in which we can easily determine whether it is in state 0 or 1, we cannot determine a qubit's state [29]. We can only measure a qubit, and when we do, we obtain either 0 with $|\alpha|^2$ probability or 1 with $|\beta|^2$ probability. Another important qubit property is entanglement. Entanglement is, at the moment, still an ill-defined concept currently being subjected to heavy research, but its main idea is that the state of a qubit affects the state of other qubits in the system, meaning that there is a correlation between them.

Quantum Circuits A classical computer is built from electrical circuits containing wires and logic gates. Similarly, some quantum computers are built from quantum circuits (there are other types of quantum computers, although these are out of the scope of this chapter) containing wires and quantum gates that carry around and

operate on qubits. One of the quantum gates used throughout this chapter is the Not gate. Classically, this gate brings a bit from 0 to 1 and from 1 to 0. The quantum Not gate [29] interchanges the weights on α and β. It is represented by the following X matrix:

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1}$$

If we have the following quantum state $\alpha|0\rangle + \beta|1\rangle$, its vector notation would be

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{2}$$

and applying the Not gate to this state would yield the following output:

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}. \tag{3}$$

This is how gates are applied to qubits and how we can alter their state.

Quantum Programs A program is considered to be *quantum* when it initializes qubits and performs some operations that alter their state through the application of quantum gates. Quantum programs can be hybrid (i.e., they combine classical and quantum operations), the more common option, or pure (i.e., they only use quantum operations), the less common option.

2.2 Software Testing

As described by the IEEE Std. 610.12-1990, "Software testing is the process of operating a system or component under specified conditions, observing and recording the results, and making an evaluation." In other words, in software testing, a *test case* sets up a testing scenario that exercises software behavior and assesses whether the observed behavior matches the expected one; if not, a *fault* has been found. These faults, also known as *bugs* or *defects*, can cause failures in software systems.

Although a simple idea, it is far from easy—recent studies estimate that 20% to 80% of the total cost and time to develop a classical software system is fully dedicated to software testing and debugging [30], mostly because

(i) Assessing whether a piece of software performs correctly could be extremely complex due to the extremely large or even infinite number of possible tests that exist for any non-trivial system.
(ii) Software testing is traditionally a manual and tedious process that is subject to incompleteness and further errors.

The usage of some testing concepts throughout this chapter justifies their clarification.

Mutation Testing This testing technique refers to the change/mutation of statements in the source code (Fig. 1 is an example of a mutant) to check if tests can find errors in the source code. Mutation testing aims to ensure the quality of the source code's test suite. This is measured through the source code's mutation score, the number of killed mutants divided by the number of total mutants generated.

Coverage This is a testing metric that measures how thoroughly tests cover a given program. A test suite's coverage is the percentage of lines, branches, or paths of the code covered by at least one test case.

3 Automatic Verification and Validation of Quantum Software

Verifying and validating code is laborious, error-prone, and time-consuming in the classical realm. Given the added complexity of quantum programs, this endeavor is even more challenging in the quantum world [31, 5, 32]. Additionally, not all technologies are fully tailored to this new paradigm, and neither are the developers who would have to understand quantum physics/mechanics.

Nevertheless, some preliminary works are bringing commonly used classical testing practices for quantum computing [5] to assess and improve the quality of quantum programs. Regarding verification, there are works on the application of Hoare Logic [9, 10] and static code analysis [12, 14, 15, 11, 33]. And regarding validation, there are also techniques designed to automatically generate test inputs and/or full test cases based on mutation [34, 35, 36, 37, 18], metamorphic [19, 20], fuzzing [16], differential [17], projection [38], search-based [24, 23, 25, 26], and combinatorial testing [22, 21].

Table 1 shows the details of the collected research papers. These papers present tools that are 'Available' and can be used and experimented with, tools that are 'Unavailable' and do not provide any artifact with their paper, and tools that we considered 'Unusable' since they are not easily available or capable of testing any other program than the ones used in the empirical study of the tool. For instance, although LintQ's [33] source code is available online, it is stored in an anonymous repository that does not allow its download or cloning. QDiff [17] and Abreu et al. [19]'s tool only allows one to reproduce the experiments described in the paper, i.e., in order to run the proposed tool on any program, its source code would have to be adapted (which is out of the scope of this chapter). We discuss the "Available" tools in detail in the following subsections. It is also worth pointing out that there are several other works on verification and validation of quantum software applied on different quantum frameworks, test levels, or issues related to quantum software testing that are not included in this study as they target different quantum

Table 1 Details of the collected research papers

ID	Topic	Paper title	Tool	Year	Reference
Verification					
1	Hoare Logic	Floyd–Hoare Logic for Quantum Programs	Unavailable	2012	[9]
2	Hoare Logic	An Applied Quantum Hoare Logic	Unavailable	2019	[10]
3	Static analysis	QChecker: Detecting Bugs in Quantum Programs via Static Analysis	Available	2023	[12]
4	Static/Dynamic analysis	The Smelly Eight: An Empirical Study on the Prevalence of Code Smells in Quantum Computing	Available	2023	[51]
5	Static analysis	Quantum abstract interpretation	Unavailable	2021	[13]
6	Static analysis	Static Entanglement Analysis of Quantum Programs	Unavailable	2023	[14]
7	Static analysis	A Uniform Representation of Classical and Quantum Source Code for Static Code Analysis	Unavailable	2023	[15]
8	Static analysis	LintQ: A Static Analysis Framework for Qiskit Quantum Programs	Unusable	2023	[33]
Validation					
9	Data generation	QuanFuzz: Fuzz Testing of Quantum Program	Unavailable	2018	[16]
10	Data generation	QDiff: Differential Testing of Quantum Software Stacks	Unusable	2021	[17]
11	Data generation	Mutation-Based Test Generation for Quantum Programs with Multi-Objective Search	Unavailable	2022	[18]
12	Oracle generation	Metamorphic testing of oracle quantum programs	Unusable	2022	[19]
13	Oracle generation	MorphQ: Metamorphic Testing of Quantum Computing Platforms	Available	2022	[20]
14	Data/Oracle generation	Application of Combinatorial Testing To Quantum Programs	Available	2021	[21, 22]
15	Data/Oracle generation	Generating Failing Test Suites for Quantum Programs With Search	Available	2021	[23, 24]
16	Data/Oracle generation	Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs	Available	2021	[25, 26]

frameworks or used quantum physics knowledge that cannot be applied directly to software. The following paragraph briefly mentions them.

Muqeet et al. [39] propose a testing technique aware of the inherent problem of quantum computing related to noise. Zhang et al. [40] examine whether flaky tests (i.e., intermittently failing tests) affect quantum software development. They identify flaky tests in 12 out of 14 quantum software projects and note that quantum programmers need to start using flaky test countermeasures developed by software engineers. Long and Zhao [41, 42] address specific testing requirements of multi-subroutine quantum programs in their work. They present a systematic testing process tailored to the intricacies of quantum programming. They cover unit and integration testing, focusing on IO analysis, quantum relation checking, structural testing, behavior testing, and test case generation for Q#. Honarvar et al. [11] present a property-based framework applied for Q# derived from Hoare logic [43]. They review various aspects of design concerning property specification, test case generation, and test result analysis. Xia and Zhao [14] present a static analysis tool that constructs an interprocedural control flow graph for Q# programs and gathers the entanglement information within quantum programs. A similar tool is proposed by Yamaguchi et al. [44] for Qiskit; we detail it in Sect. 3.2. de la Barrera et al. [45] propose QuMU, a quantum mutation tool based on the Quirk[1] quantum circuit simulator. QuMU exports quantum circuits as JSON objects from Quirk and creates a circuit representation that shows the quantum operations of a quantum program. Mutation operators defined in QuMU can mutate the circuit representation of a quantum program, and their tool can then execute these mutants in Quirk.

3.1 Running Example

Let us introduce a running example for the remainder of this section. The quantum program in Fig. 1 implements a Bell state [46], the simplest example of quantum entanglement. Bell states are four entangled two-qubit states. We obtain a Bell state by applying the Hadamard gate to qubit 1 (line 13) and the Control-Not with qubit 1 as the control qubit and qubit 2 as the target qubit (line 16). This means that when the quantum program is executed, the qubits are dependent on each other, and one will obtain either 00 or 11 as a result, with a 50% chance of getting either one. Note that Qiskit initializes qubits as zero.

The quantum program listed in Fig. 1 follows the specification reported in Table 2. Note that although inputs 01 and 11 **do not** produce a Bell state, we still list them in the table to have the full program specification.

Suppose we introduce a fault in the program's source code to create a faulty version of the program. For instance, swap the Hadamard gate (h) in line 13 for the Not gate (x) in line 13. Note that this is a change (i.e., mutation) that a tool

[1] https://algassert.com/quirk, visited October 2023.

```
# Bell State quantum program example
from qiskit import *

def BellState(input='00'):
    # Create a Quantum Circuit object acting on a quantum and classical
    # register of two qubits/bits
    qr = QuantumRegister(2)
    cr = ClassicalRegister(2)
    circ = QuantumCircuit(qr, cr)
    circ.initialize(input, circ.qubits)

    # Add a H gate on qubit 0, putting this qubit in superposition
-   circ.h(qr[0])
+   circ.x(qr[0]) // Introduce a FAULT, swaped Hadamard gate for the Not gate
    # Add a CX (CNOT) gate on control qubit 0 and target qubit 1, putting the
    # qubits in a Bell state
    circ.cx(qr[0], qr[1])
    # Add measurement to the circuit
    circ.measure(qr, cr)

    # Execute the circuit
    backend = BasicAer.get_backend('qasm_simulator')
    job = execute(circ, backend, shots=1000)
    counts = job.result().get_counts()

    return counts
```

Fig. 1 Fault-free and faulty Bell state quantum program

Table 2 Specification of the Bell state quantum program in Fig. 1

Input	Output	Output Probability
00	00	50%
00	11	50%
01	00	50%
01	11	50%
10	10	50%
10	01	50%
11	01	50%
11	10	50%

like Muskit [37] or QMutPy [34, 35, 36] (described in Sect. 3.3.4) can produce. We then apply different verification and validation techniques on this faulty version of the running example in the following subsections to understand to what extent techniques can detect this fault. Note that if a tool of a specific technique is not available or usable, we do not apply it.

3.2 *Automatic Verification of Quantum Software*

Verification aims to assess whether developers have built the software correctly, i.e., it answers the question: *Does the software correctly do what has been specified?*

3.2.1 Hoare Logic

The Hoare logic testing [43] is a formal system with a set of logical rules for formal verification of the correctness of an algorithm against a formal specification. This logic is based on the idea of a specification as a contract between the implementation of a function and its client. To prove the correctness of a specification, it provides a mathematical framework using logical assertions, a pre- and post-condition, for describing the desired behavior of a program before and after its execution.

The central component of the Hoare logic is the Hoare triple. A Hoare triple is a notation used to express the relationship between a pre-condition, a program or program segment, and a post-condition. It is written as $\{P\}S\{Q\}$ where P is the pre-condition (predicate describing the condition the function relies on for correct operation), Q is the post-condition (predicate describing the condition the function establishes after correctly running), and S the statement implementing the function. The Hoare logic also provides a set of axioms and rules of inference that can be used in proofs of the properties of computer programs.

Regarding quantum software testing, Ying [9] derives from Hoare logic the Quantum Hoare Logic (QHL) for verifying the correctness of quantum programs. The correctness formula of QHL is also written as $\{P\}S\{Q\}$, but S is a quantum program, and both P and Q are quantum predicates on $\mathcal{H}_{all}$, which is the tensor product of the state spaces of all quantum variables.

Zhou et al. [10] further develop the work of Ying [9]. They propose aQHL, a new class of Hermitian operators (i.e., an operator that is equal to its conjugate transpose, e.g., $\mathcal{A} = \mathcal{A}\dagger$), which are used in the pre- and post-conditions and allow a simplification of the inference rules in case statements, and loops and computation of ranking functions in QHL. The authors prove that with aQHL they can verify the correctness of a well-known quantum algorithm for linear systems of equations, the HHL (Harrow-Hassidim-Lloyd) [47] algorithm. Zhou et al. [10] also propose several rules for reasoning about the robustness of quantum programs, i.e., error bounds of the output software programs, to prove that the outputs of a quantum program approximately satisfy a post-condition. They use these new rules to verify the quantum Principal Component Analysis (PCA) [48], a machine learning algorithm.

3.2.2 Static Analysis

Zhao et al. [12] propose QChecker,[2] a static analysis tool that generates warning messages to assist developers in pinpointing potential *faults* in their quantum programs. QChecker starts by extracting the abstract syntactic tree of a quantum program and parses it through a detection module equipped with a catalog of quantum faults patterns [49]. If the source code of a quantum program matches any

[2] https://github.com/Z-928/QChecker, visited October 2023.

of the patterns, a *true* fault might have been identified. The authors evaluate their tool on 20 real faults[3] from open-source quantum programs written in Qiskit [50] and their results attest to the efficiency and effectiveness of QChecker—all faults were detected.

Applying QChecker to our faulty running example (Fig. 1) we obtained two warnings (that might be *true* faults):

1. Incorrect initial state in lines 7 and 8. To fix it, one would have to create a variable `n = 2` and then reuse `n` in lines 7 and 8, i.e.,

```
7  -  qr = QuantumRegister(2)
8  -  cr = ClassicalRegister(2)
7  +  n = 2; qr = QuantumRegister(n)
8  +  cr = ClassicalRegister(n)
```

   The rationale is that one might initialize the `QuantumRegister` with a number of qubits and/or the `ClassicalRegister` with a different number of bits. This potential error is mitigated with a variable that defines the number of bits.
2. Parameter error in line 21. To fix it, one would have to hard code line 21 as the second parameter of the *execute* function in line 22, i.e.,

```
21 -  backend = BasicAer.get_backend('qasm_simulator')
22 -  job = execute(circ, backend, shots=1000)
21 +  job = execute(circ, BasicAer.get_backend('qasm_simulator'), shots=1000)
```

   We could not find any rationale for this QChecker warning and were inclined to label it as a false positive. Note that if we apply QChecker suggestion and the `execute` call starts to fail, we will not know whether the failure is due to `execute` or `get_backend`. This would make debugging more difficult.

It is worth noting that QChecker did not produce any warning regarding the fault we introduced in line 13 (in Fig. 1).

Chen et al. [51] define, for Qiskit programs, eight quantum-specific smells (which might lead to a fault) inspired by the best coding practices suggested by Google Cirq's team.[4] For example, LC (Long circuit) smell—the wider the circuit, the higher the probability of quantum noise affecting a quantum circuit's intended behavior. They also developed a tool named QSmell[5] that supports the proposed quantum-specific smells and empirically evaluated its effectiveness at detecting the smells in 15 quantum programs. Their results show that most quantum programs (73%) have at least one smell and, on average, a program has three smells; LC is the most common smell.

[3] Although the first version of the catalog proposed by Zhao et al. [49] is composed by only 36 real faults, Zhao et al. [12] used an augmented version of the catalog with 42 real faults of which only 22 can be detected by running the quantum program. Thus, Zhao et al. [12] only consider the remaining 20 in the evaluation conducted with QChecker.

[4] https://quantumai.google/cirq/google/best_practices, visited October 2023.

[5] https://github.com/jose/qsmell, visited October 2023.

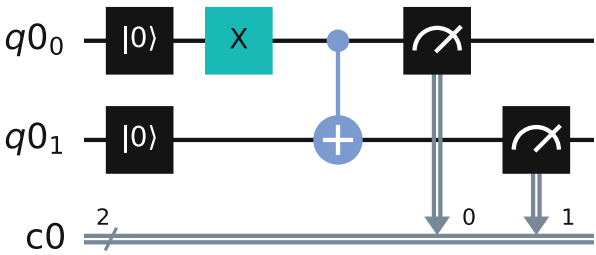

Fig. 2 Quantum circuit of the faulty Bell state quantum program

When we apply QSmell to our faulty running example (Fig. 1), one smell is reported by the tool, IdQ (Idle Qubits). With current quantum computers, it is only possible to ensure the correctness of a qubit's state for very short periods of time. This means that having idle qubits for too long enhances the loss of quantum information and might jeopardize the results of the running quantum programs. In a nutshell, QSmell reports that qubit 1 is idle between lines 10 and 16 (in Fig. 1) or between the first and third operations (Fig. 2), which might indicate a fault. In this case, and to the best of our knowledge, there is no other way to write the quantum circuit to avoid that. Thus, we consider this a false positive.

Yu and Palsberg [13] propose an abstract interpretation of quantum programs and use it to automatically verify whether a program might behave as expected in polynomial time. To achieve this, the authors take the density matrix of a quantum program and divide it into parts (i.e., reduced density matrixes). Then, they approximate each reduced matrix by a projection. Recall that a projection is the closest point/vector in a subspace to a given point in the space. This enables them to define abstract states to be a tuple of projections. To transition from abstract state to abstract state, the authors present a new abstract interpretation of quantum programs with new abstractions and concretization functions that form a Galois connection, and they use them to define abstract operations. Yu and Palsberg [13] evaluate their approach on three quantum programs. They first run the abstract interpretation, which produces an abstraction of the state of each quantum program. Then, they abstract the assertion (i.e., the circuit output desired) to the same format as the abstract states, and finally, they check that the abstract state satisfies the abstracted assertion. If the check succeeds, then the assertion is correct. For all three programs, the authors successfully verified their assertions.

Applying this technique would detect the fault in our running example. Starting with qubit state $|00\rangle$ and successfully generating the abstract states to be a tuple of projections through the application of the Not and Controlled-Not would not result in a successful assertion with our desired output, i.e., $\{|00\rangle, |11\rangle\}$. However, our correct running example would.

Paltenghi and Pradel [33] propose LintQ, a static analysis framework for detecting faults in quantum programs. LintQ receives a quantum program as input and extracts general information about Python code, such as control flow paths, data flow facts, and how to resolve imports. Then it represents the behavior of the quantum program using a set of reusable quantum programming abstractions, such

as qubits, gates, and circuits. Finally, LintQ contains a set of nine quantum analyses that detect potential faults. LinQ performs three main types of analysis:

1. Measurement-related and gate-related problems
2. Resource allocation problems
3. Implicit API constraints

The authors perform an empirical study applying LintQ to a quantum program dataset containing 7568 quantum programs where LintQ found multiple true positives with a precision of 80.5%. The authors also tried LintQ with the Bugs4Q [50] benchmark of real faults and obtained a recall of only 4.8%. The authors argue that the low recall achieved in the Bugs4Q benchmark programs is mainly due to the incomplete code snippets gathered from issues and forum questions provided by the benchmark.

Kaul et al. [15] extend the Code Property Graph (CPG) static code analysis technique [44] used in classical computing to quantum computing. CPG is a computer program representation that captures syntactic structure, control flow, and data dependencies in a language-independent property graph model. The authors extended this concept to quantum computing by modeling the memory and operations as well as dependencies between qubits and quantum registers. Their prototype supports Qiskit [28] and QASM [52] programs. It also includes information from the quantum realm in the graph (i.e., qubits, gates, gates arguments) and demonstrates CPG's ability to analyze classical and quantum source code. By combining all relevant information into a single detailed analysis, this tool can facilitate quantum source code analysis. To that end, the authors propose a series of eight queries that return specific information about the quantum program to the user, such as the quantum/classical parts of the program, constant conditions, or the result bits. This allows users to have a clearer picture of the implementation of the program.

3.3 Automatic Validation of Quantum Software

Validation aims to assess whether developers have built the correct software according to the user requirements, i.e., it answers the question: *Does the software do what it is supposed to do?*

To improve the effectiveness of software testing and to reduce its cost, researchers have devised approaches (in both the classical and quantum realm) to automate the generation of test cases and validate quantum software. Automating the creation of test cases offers several benefits over manually writing the test cases. In classical computing, it is computationally cheap to automatically generate test cases, and they are often more complete as they are generated systematically; there is no evidence that it would be otherwise for the automatic generation of test cases for quantum programs. Automatic test generation is a two-step process: (1) generation of *test data*, i.e., inputs to exercise the software, and (2) generation of

test oracles (also known as assertions) to verify whether the execution of the test data reveals any fault.

3.3.1 Test Data Generation

Wang et al. [17] propose QDiff, a differential testing approach for quantum programs, which can be used with three quantum frameworks: Qiskit, Cirq, and Pyquil. QDiff takes as input a quantum program and derives equivalent programs from it (i.e., programs that are supposed to produce identical behavior) that trigger unexpected behavior on the target quantum framework. To speed up their analysis, QDiff analyzes static program characteristics such as circuit depth (i.e., the longest sequence of applied gates to the circuit), the number of two-qubit gates, and known error rates. Finally, QDiff performs a statistical comparison between the measurements of the equivalent circuits. The empirical evaluation of QDiff found six sources of instability in the three quantum frameworks and managed to reduce compute-intensive simulation.

Fuzz testing [53, 54, 55, 56, 57, 58]—a set of software testing techniques implying the generation of a set of inputs aiming at finding errors/crashes and identifying security flaws—is gaining relevance in quantum software testing [16]. Wang et al. [16] adapt this technique to the quantum realm and present QuanFuzz, a search-based test input generator for quantum programs. In a nutshell, it can automatically find the input that triggers the quantum-sensitive branches. QuanFuzz was evaluated with seven programs and outperformed a random technique, increasing branch coverage by 20% to 60%.

Wang et al. [18] propose MutTG, a multi-objective and search-based approach to generate the minimum number of test cases that kill as many mutants as possible. The authors introduce a *discount factor* to tackle the equivalent mutants problem [59, 60, 61, 62, 63] ever-present in mutation testing (i.e., mutants that are equivalent to the source code and do not alter its result) to prevent their approach from repeatedly trying to kill those non-killable mutants. The authors employ NSGA-II as the multi-objective search algorithm and use five quantum programs for which they created 20 different versions (four mutants per program) with three distinct difficulty levels of killing mutants (easy, medium, difficult) to evaluate their approach. Results from their experimental evaluation show that NSGA-II [64] significantly outperforms the random search technique employed as a baseline for all the difficult benchmarks composed of subtle mutants (i.e., mutants that are killed by few inputs). Also, they show that their discount factor is effective in avoiding spending meaningless effort trying to kill non-killable mutants.

3.3.2 Test Oracle Generation

The well-known oracle problem [65] for classical testing becomes even more complex in this new programming paradigm. Test oracle automation is essential to

remove the bottleneck that inhibits greater overall test automation. In other words, without a formal specification of how software should behave, it is impossible to generate effective fault-revealing test oracles. Thus, techniques that generate tests usually generate regression tests.

To the best of our knowledge, two metamorphic approaches [19, 20] have attempted to address the oracle problem by substituting conventional oracles with mutated versions of the quantum program under test. Recall that metamorphic testing consists of injecting small mutations to the code that do not alter a program's execution (e.g., in classical computing, adding zero to a number; in quantum computing, introducing the identity gate to a circuit).

The approaches that Abreu et al. [19] and Paltenghi and Pradel [20] propose are similar in nature and define oracle quantum programs which validate a source quantum program's properties by doing mutations to its source code that do not alter the program output. Both of these approaches define a set of metamorphic rules and assert whether their mutated program behaves when executed. They empirically evaluate their metamorphic rules on quantum programs (i.e., they create a mutated version of a quantum program that is expected to produce the same result) and find that metamorphic rules are effective at finding crashes and incorrect outputs in quantum programs.

3.3.3 Test Data and Oracle Generation

Wang et al. [26] propose QUITO[6] (QUantum InpuT Output testing) consisting of three coverage criteria defined by the inputs and outputs of a quantum program:

1. Input coverage: checks that for a valid input, the quantum program produces a valid output. Only one execution of the program is necessary for this criterion. This is the least expensive (i.e., runs the smallest number of tests).
2. Output coverage: checks that all valid outputs are covered, iterating over all valid inputs until a wrong output value is detected or time runs out. This is the second most expensive criterion.
3. Input-Output coverage: checks that all possible output values are covered for all valid inputs, iterating over all valid inputs until a wrong output value is detected or time runs out. This is the most expensive criterion.

It also consists of two oracle generation strategies:

1. Wrong Output Oracle (WOO), which asserts whether the quantum program produced expected output values
2. Output Probability Oracle (OPO), which asserts whether the quantum program produced an expected output with its corresponding expected probability

To assess the effectiveness of the three coverage criteria, the authors perform an empirical study on 78 mutated versions of four quantum programs. They generate

[6] https://github.com/Simula-COMPLEX/quito, visited October 2023.

```
def run(circ):
  # Add (incorrectly) a X gate on qubit 0
  circ.x(0)
  # Add a CX (CNOT) gate on control qubit 0 and target qubit 1, putting the
  # qubits in a bell state
  circ.cx(0, 1)
  # Add measurement to the circuit
  circ.measure([0,1], [0,1])
```

Fig. 3 Faulty Bell state program adapted to be executed with QuCAT [21], QUITO [26], and QuSBT [23]

```
[program]
;The absolute root of your quantum program file.
root=bell_state.py
;The total number of qubits of your quantum program.
num_qubit=2
;The ID of input qubits.
inputID=0,1
;The ID of output qubits which are the qubits to be measured.
outputID=0,1

[program_specification_category]
;The category of your program specification. Choice: full/partial/no
ps_category=full

[quito_configuration]
;The coverage criterion you choose. Choice: IC/OC/IOC
coverage_criterion=IC

[program_specification]
;The program specification. Format: <input,output=probability>
00,00=0.5
00,11=0.5
01,00=0.5
01,11=0.5
10,10=0.5
10,01=0.5
11,01=0.5
11,10=0.5
```

Fig. 4 Configuration for the QUITO tool. In this figure, we only list the required parameters and which values we used. Other parameters were left with their default values. Consult QUITO's documentation (https://github.com/Simula-COMPLEX/quito/blob/main/README.md, visited October 2023) for more information

these mutants with the Muskit [37] tool. After generating a set of test cases for each mutant using the three coverage criteria, the authors evaluate them with WOO, stopping the testing if a failure occurs, and then the OPO. Results indicate that input coverage is more effective than the others.

We run QUITO with our faulty running example in Fig. 1. To test our example, we had to adapt it to the tool's requirements (Fig. 3) and create a configuration file where we define the number of input and output qubits our program would have, in our case, two input qubits and two output qubits (see Fig. 4). We also set input coverage (line 17 in Fig. 4) as the coverage criterion as it is the most effective according to QUITO's authors. Finally, we also detail the program specification (lines 21–29 in Fig. 4) for our example as shown in Table 2. QUITO generates 800

tests (total number of test suites, i.e., 200 by default × number of possible input states, i.e., four). For our example, all of our eight input/output qubit combinations fail with the OPO oracle.

Wang et al. [21, 22] proposed QuCAT[7] (QUantum CombinAtorial Testing), which attempts to trigger faults by particular input combinations of a given strength. These faults are found through the two oracles previously defined in QUITO (i.e., WOO and OPO). The strength of a combination is the number of input qubits used, meaning that two input qubits are a combination of strength two, three input qubits are a combination of strength three, and so on. QuCAT supports two test generation scenarios:

1. The generation of combinatorial test cases of a given strength
2. The incremental generation of combinatorial test cases of increasing strengths

The authors performed an empirical study on six Qiskit quantum programs, in which they manually introduced three faults in each. They found that their combinatorial technique of strength four (highest strength attempted) always detects the faults, tests of strength three have more difficulty in detecting all faults, and strength two only detects one fault consistently. Thus, with increased cost, this combinatorial technique increases in effectiveness. Also, results showed that combinatorial testing is always more effective than random testing in terms of generating test cases that expose program failure and performs better in 88% of the faulty programs.

Trying QuCAT was similar to QUITO. We include in its configuration file the same qubit and specification information as before. However, we also define the strength of the input combination as two as our program has two input qubits (see Fig. 5). This means that we execute QuCAT with the first test generation scenario (i.e., we generated combinatorial test cases of strength two). The tool generates four tests in a Python file and the results of the oracles in a separate text file, these bundled together in Fig. 6 for reading convenience. As we can see, the generated tests perform a print of the execution of the program with certain inputs. Although no explicit oracle (e.g., assert) exists in any test, all reveal the fault. If one compares the tests' output and the program's specification, one will notice that each output has only one result with 100% probability instead of two results with 50% each. To have fully automated tests, QuCAT should have generated the test oracles in Fig. 7 for `test_bell_state_0` (line 3 in Fig. 6). These test oracles would fail in lines 6 (as we obtained one pair of bits as output and not two), 7 (as there were no 00 results), 9 (since the probability of obtaining 11 is 100% which is superior to 55%), and 10 (because the probability of obtaining 00 is 0%, which is inferior to 45%). Line 8 does not fail; there are results of state 11.

Wang et al. [23, 24] propose QuSBT[8] (Quantum Search-Based Testing), a test generation tool for quantum programs that uses an evolutionary algorithm to search for the maximum set of tests that reveal the fault. The authors also use the WOO and

[7] https://github.com/Simula-COMPLEX/qucat-tool, visited October 2023.

[8] https://github.com/Simula-COMPLEX/qusbt-tool, visited October 2023.

```
[program]
;The absolute root of your quantum program file.
root=bell_state.py
;The total number of qubit of your quantum program.
num_qubit=2
;The IDs of input qubits.
inputID=0,1
;The IDs of output qubits which are the qubits to be measured.
outputID=0,1

[qucat_configuration]
;The maximum value of strength of a combination as the number of inputs used.
k=2

[program_specification]
;The program specification. Format: <input,output=probability>
00,00=0.5
00,11=0.5
01,00=0.5
01,11=0.5
10,10=0.5
10,01=0.5
11,01=0.5
11,10=0.5
```

Fig. 5 Configuration for the QuCAT tool. In this figure, we only list the required parameters and which values we use. We left all other parameters with their default values. Consult QuCAT's documentation (https://github.com/Simula-COMPLEX/qucat-tool/blob/main/README.md, visited October 2023) for more information

```
class bell_stateFun1K2Test(unittest.TestCase):

  def test_bell_state_0(self):
    input = '00'
    print(execute_quantum_program([0,1], [0,1], 2, input, "bell_state", 200))
    # output '11' | result OPO

  def test_bell_state_1(self):
    input = '01'
    print(execute_quantum_program([0,1], [0,1], 2, input, "bell_state", 200))
    # output '00' | result OPO

  def test_bell_state_2(self):
    input = '10'
    print(execute_quantum_program([0,1], [0,1], 2, input, "bell_state", 200))
    # output '01' | result OPO

  def test_bell_state_3(self):
    input = '11'
    print(execute_quantum_program([0,1], [0,1], 2, input, "bell_state", 200))
    # output '10' | result OPO
```

Fig. 6 Tests generated by the QuCAT tool [21, 22] for the faulty Bell state quantum program

OPO oracles in QuSBT. The authors evaluate QuSBT on six quantum programs, in which they manually introduce five faults in each and compare QuSBT's results with a random search strategy. The authors find that for the majority of the faulty programs (87%), QuSBT performs significantly better than the random approach. For the remainder of the faulty programs, no significant differences are detected.

```
-  print(execute_quantum_program([0,1], [0,1], 2, input, "bell_state", 200))
+  counts = execute_quantum_program([0,1], [0,1], 2, input, "bell_state", 200)
+  self.assertTrue(len(counts) == 2)
+  self.assertTrue('00' in counts)
+  self.assertTrue('11' in counts) // nao falha
+  self.assertTrue(0.45 < counts['00']/200 < 0.55)
+  self.assertTrue(0.45 < counts['11']/200 < 0.55)
```

Fig. 7 Ideal set of test oracles for the `test_bell_state_0` test (in Fig. 6)

```
[program]
;The absolute root of your quantum program file.
root=bell_state.py
;The total number of qubit of your quantum program.
num_qubit=2
;The IDs of input qubits.
inputID=0,1
;The IDs of output qubits which are the qubits to be measured.
outputID=0,1

[qusbt_configuration]
;A percentage of the inputs as the number of generated tests, 0.05 by default.
beta=1.0
;The confidence level for the statistical test, 0.01 by default.
;Although it is not required according to the official documentation, it is
;required at runtime.
confidence_level=0.01

[program_specification]
;The program specification. Format: <input,output=probability>
00,00=0.5
00,11=0.5
01,00=0.5
01,11=0.5
10,10=0.5
10,01=0.5
11,01=0.5
11,10=0.5
```

Fig. 8 Configuration for the QuSBT tool. In this figure, we only list the required parameters and the values we use. Other parameters are left with their default values. Consult QuSBT's documentation (https://github.com/Simula-COMPLEX/qusbt-tool/blob/main/README.md, visited October 2023) for more information

Running QuSBT requires the same initial configurations as QUITO and QuCAT (i.e., number of input and output qubits, program specification). Additionally, we set the `beta` parameter, a percentage of the inputs, as the number of generated tests, so that all possible tests are generated (see Fig. 8). The default value of `beta` is 0.05, which would mean, for our example, that only one test would have been generated. QuSBT generates two tests (similar to the first and third tests generated by QuCAT; see Fig. 9) that also fail with the OPO oracle. Note that extending QuSBT tests as we did for QuCAT in Fig. 7 would pass and fail for the same assertions.

```
class Bell_StateTest(unittest.TestCase):

  def test_bell_State_0(self):
    #Input: 00
    print(execute_quantum_program([0, 1], [0, 1], 2, 0, "bell_state", 200))

  def test_bell_State_1(self):
    #Input: 10
    print(execute_quantum_program([0, 1], [0, 1], 2, 2, "bell_state", 200))
```

Fig. 9 Tests generated by the QuSBT tool [23, 24] for the faulty Bell state quantum program

3.3.4 Test Adequacy Measurements

In the quantum realm, a few approaches and tools (based on the ideas borrowed from the classical realm) have been proposed to measure the effectiveness of manually written or automatically generated tests of quantum programs.

Structural Coverage

Code coverage and other source-code metrics used for classical software have not been adopted for quantum programs [66]. This may be because the differences in the importance between quantum code and classical code have not yet been fully explored. Also, thresholds for source code metrics and their significance as predictors of defects [67] cannot be used as a starting point for quantum programs since quantum programmers and their knowledge of this new programming paradigm are likely to be completely different.

Thus, recent studies propose other metrics besides traditional coverage. For instance, Kumar [68] proposes single-, two-, and three-qubit gate coverage and multiple controlled qubit gate coverage, which are defined by the total number of times test cases would execute these types of gates divided by the number of instances that gate is used in the code. Ali et al. [25] also propose three new types of coverage criteria previously discussed in Sect. 3.3.3: input coverage, output coverage, and input-output coverage.

Nevertheless, other studies still use classical coverage. For instance, the previously discussed work of Wang et al. [16] empirically evaluates whether their input generation technique increases coverage compared to random input generation (see Sect. 3.3.1). Also, Fortunato et al. [34] measure the coverage of 24 real Qiskit programs and find that tests covered on average 90% of the lines of code of a quantum program.

Fault Detection

Classically, mutation testing is often used as a practical substitute for real faults since mutant detection is positively correlated with fault detection [69]. Current

```
from qiskit import *

# Create a Quantum Circuit object acting on a quantum and classical
# register of two qubits/bits
qr = QuantumRegister(2)
cr = ClassicalRegister(2)
circ = QuantumCircuit(qr, cr)
circ.initialize('00', circ.qubits)

# Add a H gate on qubit 0, putting this qubit in superposition
circ.h(qr[0])
# Add a CX (CNOT) gate on control qubit 0 and target qubit 1, putting the
# qubits in a Bell state
circ.cx(qr[0], qr[1])
```

Fig. 10 Fault-free Bell state program implementation for Muskit

research in quantum testing uses mutants to artificially generate faults in programs and evaluate the effectiveness of their approaches at detecting the mutant, as seen in Sect. 3.3.3. Mendiluze et al. [37] propose Muskit[9] and Fortunato et al. [34, 36, 35] propose QMutPy[10] to perform mutation analysis. These tools are similar in nature since they perform mutations (i.e., artificial faults) to the input source program.

On the one hand, Muskit requires the raw circuit of a program to be able to execute. This means that real programs such as our running example in Fig. 1 would need to be transformed to the one in Fig. 10. Then to use Muskit, we have to:

- Create a configuration file to specify what we are going to mutate (i.e., which gates, which types of gates (1-qubit, 2-qubit, etc.), the maximum number of mutants to generate, what mutation operators we are going to use (Muskit mutation operators are Add, Remove, and Replace Gate), and the location of where to Add a new gate if the Add operator is selected).
- Create the executor file to specify the number of times we want to execute the circuit, if we are going to use all possible input values (in the case of Fig. 10, those would be 00, 01, 10, or 11) or not, and if we want to specify our input values we would need to create another custom test file where we specify which ones we want to use.
- Create an analyzer file to specify the number of qubits our program has that we want to measure (in our case, we have two qubits and want to measure both of them) and what is the significance level (p-value) for our tests.

To determine whether a mutant was detected, Muskit uses two oracles already explained in Sect. 3.3.3: the WOO (i.e., if the program output is wrong, the mutant is detected) and the OPO (i.e., if our p-value is lower than the chosen significance level the mutant is detected). Suppose we apply the Remove gate operator to both our gates with input values 00 (the default Qiskit qubit initialization value) to our running example (Fig. 1). After setting up all of the necessary files described above

[9] https://github.com/Simula-COMPLEX/muskit, visited October 2023.

[10] https://github.com/danielfobooss/mutpy, visited October 2023.

```
from unittest import TestCase

class BellStateTest(TestCase):

  def test(self):
    counts = BellState()
    self.assertTrue(len(counts) == 2)
    self.assertTrue('00' in counts)
    self.assertTrue('11' in counts)
    self.assertTrue(0.45 < counts['00']/1000 < 0.55)
    self.assertTrue(0.45 < counts['11']/1000 < 0.55)
```

Fig. 11 Manually written test for the Bell state quantum program

and running the tool, Muskit reports that two mutants were generated and that both were detected by the WOO. This is expected as the output value of our example without the Hadamard gate will always be 00 (i.e., only one correct output instead of two), and without the Controlled-Not gate, it will always be 00 with 50% probability, which is a correct output, and 10 with 50% probability, which is an incorrect output.

On the other hand, QMutPy only requires a Qiskit program and its set of test cases (either written in unittest[11] or pytest[12]). QMutPy allows us to select from five quantum mutation operators:

- QGD—Quantum Gate Deletion (equivalent to the Remove operator from Muskit)
- QGI—Quantum Gate Insertion (equivalent to the Add operator from Muskit)
- QGR—Quantum Gate Replacement (equivalent to the Replace operator from Muskit)
- QMD—Quantum Measurement Deletion
- QMI—Quantum Measurement Insertion

To run QMutPy, we simply execute a command where we select the target program file (i.e., the fault-free version in Fig. 1) and the target test file (Fig. 11) and select which operators we want to use. If we perform the same experiment (i.e., select the QGD operator), QMutPy will also report that both mutants were detected.

The empirical results from both studies [37, 34] show that both Muskit and QMutPy tools are efficient and effective at assessing the performance of programs' specifications or test cases. However, as we can see, QMutPy is far simpler to set up than Muskit since it does not require a formal specification of each quantum program. It only requires the program's source code and its corresponding tests. Also, in case we wish to re-run our experiment with different setups, we would have to manually alter our program specification files for Muskit, while for QMutPy we would only need to select additional or fewer operators to use. It is worth pointing out that Muskit could run a mutation analysis with different inputs, and for QMutPy to do this, it would be necessary to create more tests for the quantum program under

[11] https://docs.python.org/3/library/unittest.html, visited October 2023.

[12] https://docs.pytest.org, visited October 2023.

test. However, the QMutPy's authors left for future work the addition of an input mutation operator to the tool.

4 Benchmarks of Real Faults in Open-Source Quantum Programs

Although several techniques and tools have been proposed to verify and validate quantum programs (see Sect. 3), reproducing[13] previous studies or evaluating/-comparing new techniques is still challenging. The lack of widely accepted and easy-to-use databases of *real* quantum faults (i.e., faults that have occurred in *real* quantum projects) is one of the main challenges. For instance, Fortunato et al. [34] and Mendiluze et al. [37] proposed a similar tool for mutation analysis, but both conducted an empirical evaluation on a different set of Qiskit quantum programs. Hence, it is not possible to answer the question: Which tool performs better?

In classical computing, many databases of *real* faults have been proposed, e.g., Defects4J [70] for Java, BugsJS [71] for JavaScript, and BugsInPy [72] for Python. These benchmarks have allowed researchers to conduct empirical studies on *real* faults on different research topics, e.g., automatic test generation [73, 74], test prioritization [75, 76], fault localization [77, 78, 79, 80, 81], automatic program repair [82], on whether *artificial* faults might be a practical substitute for *real* faults [69], etc.

In quantum computing, to the best of our knowledge, only three benchmarks (not yet widely accepted or easy to use) have been proposed in quantum computing [83, 50, 84].

Campos and Souto [83] propose QBugs, a framework that includes a catalog of reproducible faults of real quantum programs and an infrastructure to enable empirical and controlled experiments in quantum software testing and debugging. QBugs is not available at the time of writing this chapter.

Zhao et al. [50] propose Bugs4Q,[14] a benchmark of 36 real and manually validated faults on programs written in Qiskit.[15] These faulty programs are not accompanied by, for example, any test that reproduces and reveals the faulty behavior (as, for example, in the Defects4J [70] benchmark). Furthermore, Bugs4Q

[13] ACM defines *reproducibility* as the measurement obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials for the same artifact. For computational experiments, this means that an independent group can obtain the same result as the author using the author's artifacts. https://www.acm.org/publications/policies/artifact-review-and-badging-current, visited October 2023.

[14] https://github.com/Z-928/Bugs4Q, visited October 2023.

[15] Since its release, Bugs4Q has been augmented with seven more faults on programs written in Qiskit, two faults on programs written in Q#, and seven faults on programs written in Cirq (October 2023).

only provides (for each fault) the faulty and fixed files. In other words, it does not provide fully faulty programs (i.e., including configuration files, build files, documentation, commit history, etc.) that might be relevant to some tools or other research venues. For instance, Paltenghi and Pradel [33] pointed out that the low precision of the LintQ tool in the Bugs4Q benchmark was due to incomplete faulty programs. Other research venues, for example, fault predictors that require the commit history of a program to predict which components (e.g., functions) are likely faulty [85, 86, 87, 88, 89, 75], might also perform poorly or not work at all due to the lack of such information.

Paltenghi and Pradel [84][16] present a catalog of 223 real-world faults mined from 18 open-source quantum computing platforms (including Qiskit, Cirq, and Q#) and perform an in-depth analysis of the types of faults most frequently found in quantum software. The authors make available the faults as a catalog, the type of faults found, and their fixes. Similar to the Bugs4Q benchmark, there is no interface to interact with the catalog of faults.

To the best of our knowledge, QChecker [12] and LintQ [33] are the only tools evaluated on *real* faults, i.e., that considered the Bugs4Q benchmark.

5 Discussion

Despite the many advances in the verification and validation of quantum programs, most approaches remain to be adopted or perfected. Based on our observations, we have compiled a list of limitations that researchers (Sect. 5.1), tool developers (Sect. 5.2), and benchmark developers (Sect. 5.3) should try to address in the future.

5.1 For Researchers

The approaches presented in Sect. 3 do not exercise to their full extent the underlying idiosyncrasies of the quantum programs under test [31, 5, 32], for example, the number of independent paths generated due to the superposition of each qubit [68].

5.2 For Developers of Testing Tools

Developing a quantum testing tool is not an easy endeavor. We highlight four key aspects for developers of testing tools to keep in mind.

[16] https://github.com/MattePalte/Bugs-Quantum-Computing-Platforms, visited October 2023.

- **Setup:** The installation and configuration of each tool require a huge amount of time to perform correctly. For instance, QuSBT [24], QuCAT [22], QUITO [26], and Muskit [37] require that we clone the tool from GitHub, set up the correct environment with the right packages and the right packages' versions, manually create a configuration file and set some parameters, execute a Python file, and then select options from a menu on the command line at runtime. All of these requirements and steps might discourage users from using these tools. Even tools like QMutPy [34] or QChecker [12] that only require the cloning, the environment setup, and the execution of a single command can be inconvenient and frustrating.
- **Usage:** Developers of tools should aim to, for example, integrate their tools with common Integrated Development Environments (IDEs) such as Visual Studio or IntelliJ IDEA to ease their usage. Tools such as EvoSuite [90] (a test generation automation tool for Java programs) increase their usability when integrated with an IDE. It should be no different for quantum tools.
- **Produce test suites source code:** Tools like QUITO [26] do not generate tests source code (i.e., written in Python) and therefore do not use any of the common testing frameworks (unittest[17] and pytest[18]). Without such functionalities, tests cannot be executed or integrated into any project. Thus, tests could not be used to, for example, (i) detect regressions in future versions of the quantum program or (ii) assist developers in localizing [91, 77] and repairing faults [82], as has been proposed in classical computing.
- **Produce test suites with an oracle:** Tools like QuCAT [22] and QuSBT [24] do generate tests source code (i.e., written in Python), but they do not generate an explicit oracle (i.e., assertion). Oracleless tests hold down the adoption of automatically generated tests as they would not be able to detect any fault in the program under test.

5.3 For Developers of Quantum Faults Benchmarks

Benchmarks, which are a pillar of reproducibility and applicability, allow one to compare the performance of different techniques with the same datasets. Currently, benchmarks are lacking in quantum software testing, and in regard to our focus of interest, more specifically, quantum faults benchmarks. This is mainly due to the fact that there are still few quantum programs to analyze, and fault patterns are still being extracted from real faulty quantum programs. As pointed out in Sect. 4, to support different venues of research in quantum software testing, benchmarks for quantum software testing should (1) provide an interface to interact with the fixed and faulty version of a quantum program, (2) provide fully fixed and faulty

[17] https://docs.python.org/3/library/unittest.html, visited October 2023.
[18] https://docs.pytest.org, visited October 2023.

programs, and (3) provide fault-revealing test cases (either manually written or automatically generated).

6 Conclusion

The field of quantum computing is developing at a very fast pace. Therefore, the development of tools to ensure the correctness of quantum programs is of the utmost importance. In this chapter, we presented and detailed novel techniques and tools researchers have proposed to verify and validate quantum programs. Based on our exploration of the many techniques, tools, and benchmarks that have been proposed in quantum verification and validation, we highlighted key aspects of what is still lacking in the field and offered suggestions for future work. In short, researchers should focus on further exploring the properties of quantum programs. Developers should work on delivering tools and quantum fault benchmarks in ways that promote their adoption and usefulness to the scientific community.

References

1. Russell, J.: IBM Quantum Update: Q System One Launch, New Collaborators, and QC Center Plans. HPC Wire
2. Collins, H., Nay, C.: IBM Unveils 400 Qubit-Plus Quantum Processor and Next-Generation IBM Quantum. IBM Newsroom
3. Ferreira, F.: An Exploratory Study on the Usage of Quantum Programming Languages. Available at http://hdl.handle.net/10451/56751
4. Hevia, J.L., Peterssen, G., Ebert, C., Piattini, M.: Quantum computing. IEEE Software **38**(5), 7–15 (2021). https://doi.org/10.1109/MS.2021.3087755
5. Barrera, A., Guzmán, I., Polo, M., Piattini, M.: Quantum software testing: state of the art. J. Software Evol. Process **35**(4), 2419 (2023). https://doi.org/10.1002/smr.2419
6. Weder, B., Barzen, J., Leymann, F., Salm, M., Vietz, D.: The Quantum Software Lifecycle. In: Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software. APEQS 2020, pp. 2–9. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3412451.3428497
7. Arias, D., García Rodríguez de Guzmán, I., Rodríguez, M., Terres, E.B., Sanz, B., Gaviria de la Puerta, J., Pastor, I., Zubillaga, A., García Bringas, P.: Let's do it right the first time: Survey on security concerns in the way to quantum software engineering. Neurocomputing **538**, 126199 (2023). https://doi.org/10.1016/j.neucom.2023.03.060
8. Tao Yue, P.A., Ali, S.: Quantum Software Testing: Challenges, Early Achievements, and Opportunities. ERCIM News
9. Ying, M.: Floyd–Hoare logic for quantum programs. ACM Trans. Program. Lang. Syst. **33**(6) (2012). https://doi.org/10.1145/2049706.2049708
10. Zhou, L., Yu, N., Ying, M.: An Applied Quantum Hoare Logic. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2019, pp. 1149–1162. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314584

11. Honarvar, S., Mousavi, M.R., Nagarajan, R.: Property-Based Testing of Quantum Programs in q#. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp. 430–435 (2020)
12. Zhao, P., Wu, X., Li, Z., Zhao, J.: QChecker: Detecting Bugs in Quantum Programs via Static Analysis (2023)
13. Yu, N., Palsberg, J.: Quantum Abstract Interpretation. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 542–558. ACM, Virtual Canada (2021). https://doi.org/10.1145/3453483.3454061. https://dl.acm.org/doi/10.1145/3453483.3454061
14. Xia, S., Zhao, J.: Static Entanglement Analysis of Quantum Programs (2023). https://doi.org/10.48550/arXiv.2304.05049. arXiv:2304.05049 [quant-ph]
15. Kaul, M., Küchler, A., Banse, C.: A Uniform Representation of Classical and Quantum Source Code for Static Code Analysis (2023). https://doi.org/10.48550/arXiv.2308.06113. arXiv:2308.06113 [cs]
16. Wang, J., Gao, M., Jiang, Y., Lou, J., Gao, Y., Zhang, D., Sun, J.: QuanFuzz: Fuzz Testing of Quantum Program (2018). arXiv:1810.10310 [cs]
17. Wang, J., Zhang, Q., Xu, G.H., Kim, M.: QDiff: Differential Testing of Quantum Software Stacks. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 692–704 (2021). https://doi.org/10.1109/ASE51524.2021.9678792
18. Wang, X., Yu, T., Arcaini, P., Yue, T., Ali, S.: Mutation-Based Test Generation for Quantum Programs with Multi-Objective Search. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1345–1353. ACM, Boston Massachusetts (2022). https://doi.org/10.1145/3512290.3528869. https://dl.acm.org/doi/10.1145/3512290.3528869
19. Abreu, R., Fernandes, J.P., Llana, L., Tavares, G.: Metamorphic Testing of Oracle Quantum Programs. In: Proceedings of the 3rd International Workshop on Quantum Software Engineering, pp. 16–23. ACM, Pittsburgh Pennsylvania (2022). https://doi.org/10.1145/3528230.3529189. https://dl.acm.org/doi/10.1145/3528230.3529189
20. Paltenghi, M., Pradel, M.: MorphQ: Metamorphic Testing of Quantum Computing Platforms (2022). https://doi.org/10.48550/arXiv.2206.01111. arXiv:2206.01111 [cs]
21. Wang, X., Arcaini, P., Yue, T., Ali, S.: Application of Combinatorial Testing to Quantum Programs. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 179–188 (2021). https://doi.org/10.1109/QRS54544.2021.00029
22. Wang, X., Arcaini, P., Yue, T., Ali, S.: QuCAT: A Combinatorial Testing Tool for Quantum Software (2023). https://arxiv.org/abs/2309.00119v1
23. Wang, X., Arcaini, P., Yue, T., Ali, S.: Generating Failing Test Suites for Quantum Programs With Search. In: O'Reilly, U.-M., Devroey, X. (eds.) Search-Based Software Engineering. Lecture Notes in Computer Science, pp. 9–25. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88106-1_2
24. Wang, X., Arcaini, P., Yue, T., Ali, S.: QuSBT: Search-Based Testing of Quantum Programs. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 173–177 (2022)
25. Ali, S., Arcaini, P., Wang, X., Yue, T.: 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), pp. 13–23 (2021). https://doi.org/10.1109/ICST49551.2021.00014
26. Wang, X., Arcaini, P., Yue, T., Ali, S.: Quito: A Coverage-Guided Test Generator for Quantum Programs. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1237–1241 (2021). https://doi.org/10.1109/ASE51524.2021.9678798
27. Rajak, A., Suzuki, S., Dutta, A., Chakrabarti, B.K.: Quantum annealing: an overview. Phil. Trans. Roy. Soc. A Math. Phys. Eng. Sci. **381**(2241), 20210417 (2023). https://doi.org/10.1098/rsta.2021.0417. https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2021.0417
28. Aleksandrowicz, G., Alexander, T., Barkoutsos, P., Bello, L., Ben-Haim, Y., Bucher, D., Cabrera-Hernández, F.J., Carballo-Franquis, J., Chen, A., Chen, C.-F., Chow, J.M., Córcoles-Gonzales, A.D., Cross, A.J., Cross, A., Cruz-Benito, J., Culver, C., González, S.D.L.P., Torre,

E.D.L., Ding, D., Dumitrescu, E., Duran, I., Eendebak, P., Everitt, M., Sertage, I.F., Frisch, A., Fuhrer, A., Gambetta, J., Gago, B.G., Gomez-Mosquera, J., Greenberg, D., Hamamura, I., Havlicek, V., Hellmers, J., Herok, Horii, H., Hu, S., Imamichi, T., Itoko, T., Javadi-Abhari, A., Kanazawa, N., Karazeev, A., Krsulich, K., Liu, P., Luh, Y., Maeng, Y., Marques, M., Martín-Fernández, F.J., McClure, D.T., McKay, D., Meesala, S., Mezzacapo, A., Moll, N., Rodríguez, D.M., Nannicini, G., Nation, P., Ollitrault, P., O'Riordan, L.J., Paik, H., Pérez, J., Phan, A., Pistoia, M., Prutyanov, V., Reuter, M., Rice, J., Davila, A.R., Rudy, R.H.P., Ryu, M., Sathaye, N., Schnabel, C., Schoute, E., Setia, K., Shi, Y., Silva, A., Siraichi, Y., Sivarajah, S., Smolin, J.A., Soeken, M., Takahashi, H., Tavernelli, I., Taylor, C., Taylour, P., Trabing, K., Treinish, M., Turner, W., Vogt-Lee, D., Vuillot, C., Wildstrom, J.A., Wilson, J., Winston, E., Wood, C., Wood, S., Wörner, S., Akhalwaya, I.Y., Zoufal, C.: Qiskit: An Open-source Framework for Quantum Computing. Zenodo (2019). https://doi.org/10.5281/zenodo.2562111

29. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press, Cambridge (2010). https://doi.org/10.1017/CBO9780511976667
30. Alaqail, H., Ahmed, S.: Overview of software testing standard iso/iec/ieee 29119. Int. J. Comput. Sci. Network Secur. (IJCSNS) **18**(2), 112–116 (2018)
31. Miranskyy, A., Zhang, L.: On Testing Quantum Programs. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), pp. 57–60 (2019). https://doi.org/10.1109/ICSE-NIER.2019.00023. http://arxiv.org/abs/1812.09261
32. De Stefano, M., Pecorelli, F., Di Nucci, D., Palomba, F., De Lucia, A.: Software engineering for quantum programming: How far are we? J. Syst. Software **190**, 111326 (2022). https://doi.org/10.1016/j.jss.2022.111326
33. Paltenghi, M., Pradel, M.: LintQ: A Static Analysis Framework for Qiskit Quantum Programs (2023). arXiv:2310.00718 [cs]
34. Fortunato, D., Campos, J., Abreu, R.: Mutation testing of quantum programs: a case study with Qiskit. IEEE Trans. Quant. Eng. **3**, 1–17 (2022). https://doi.org/10.1109/TQE.2022.3195061
35. Fortunato, D., Campos, J., Abreu, R.: Mutation Testing of Quantum Programs Written in QISKit. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 358–359 (2022). https://doi.org/10.1145/3510454.3528649
36. Fortunato, D., Campos, J., Abreu, R.: QMutPy: A Mutation Testing tool for Quantum algorithms and Applications in Qiskit. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 797–800. ACM, Virtual South Korea (2022). https://doi.org/10.1145/3533767.3543296 . https://dl.acm.org/doi/10.1145/3533767.3543296
37. Mendiluze, E., Ali, S., Arcaini, P., Yue, T.: Muskit: A Mutation Analysis Tool for Quantum Software Testing. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1266–1270 (2021). https://doi.org/10.1109/ASE51524.2021.9678563
38. Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y.: Projection-based runtime assertions for testing and debugging quantum programs (2020). Accepted: 2021-03-14T22:46:19Z
39. Muqeet, A., Yue, T., Ali, S., Arcaini, P.: Noise-Aware Quantum Software Testing (2023)
40. Zhang, L., Radnejad, M., Miranskyy, A.: Identifying Flakiness in Quantum Programs. Preprint (2023). arXiv:2302.03256
41. Long, P., Zhao, J.: Testing multi-subroutine quantum programs: From unit testing to integration testing (2023). arXiv:2306.17407 [quant-ph]
42. Long, P., Zhao, J.: Testing quantum programs with multiple subroutines (2023). arXiv:2208.09206 [cs]
43. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
44. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and Discovering Vulnerabilities with Code Property Graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604 (2014). https://doi.org/10.1109/SP.2014.44

45. Barrera, A.G., Guzmán, I.G.-R., Polo, M., Cruz-Lemus, J.A.: In: Serrano, M.A., Pérez-Castillo, R., Piattini, M. (eds.) Quantum Software Testing: Current Trends and Emerging Proposals, pp. 167–191. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-05324-5_9
46. Sych, D., Leuchs, G.: A complete basis of generalized bell states. New J. Phys. **11**(1), 013006 (2009). https://doi.org/10.1088/1367-2630/11/1/013006
47. Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. Phys. Rev. Lett. **103**(15), (2009). https://doi.org/10.1103/physrevlett.103.150502
48. Lloyd, S., Mohseni, M., Rebentrost, P.: Quantum principal component analysis. Nature Phys. **10**(9), 631–633 (2014) https://doi.org/10.1038/nphys3029
49. Zhao, P., Zhao, J., Ma, L.: Identifying Bug Patterns in Quantum Programs. In: 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE), pp. 16–21. IEEE, Madrid, Spain (2021). https://doi.org/10.1109/Q-SE52541.2021.00011. https://ieeexplore.ieee.org/document/9474564/
50. Zhao, P., Zhao, J., Miao, Z., Lan, S.: Bugs4Q: A Benchmark of Real Bugs for Quantum Programs. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1373–1376 (2021). https://doi.org/10.1109/ASE51524.2021.9678908
51. Chen, Q., Câmara, R., Campos, J., Souto, A., Ahmed, I.: The Smelly Eight: An Empirical Study on the Prevalence of Code Smells in Quantum Computing. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 358–370 (2023). https://doi.org/10.1109/ICSE48619.2023.00041
52. Cross, A.W., Bishop, L.S., Smolin, J.A., Gambetta, J.M.: Open Quantum Assembly Language (2017)
53. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: state of the art. IEEE Trans. Reliab. **67**(3), 1199–1218 (2018). https://doi.org/10.1109/TR.2018.2834476
54. Zhu, X., Wen, S., Camtepe, S., Xiang, Y.: Fuzzing: A survey for roadmap. ACM Comput. Surv. **54**(11s), (2022). https://doi.org/10.1145/3512345
55. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Trans. Software Eng. **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563
56. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1), 1–13 (2018)
57. Godefroid, P.: Fuzzing: Hack, art, and science. Commun. ACM **63**(2), 70–76 (2020)
58. Wang, Y., Jia, P., Liu, L., Huang, C., Liu, Z.: A systematic review of fuzzing based on machine learning techniques. PLOS ONE **15**(8), 1–37 (2020). https://doi.org/10.1371/journal.pone.0237749
59. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Software Test. Verif. Reliab. **7**(3), 165–192 (1997). https://doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U
60. Just, R., Kapfhammer, G.M., Schweiggert, F.: Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis? In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 720–725 (2012). https://doi.org/10.1109/ICST.2012.162
61. Madeyski, L., Orzeszyna, W., Torkar, R., Józala, M.: Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. IEEE Trans. Software Eng. **40**(1), 23–42 (2014). https://doi.org/10.1109/TSE.2013.44
62. Just, R., Kapfhammer, G.M., Schweiggert, F.: Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis. In: 2012 IEEE 23rd International Symposium on Software Reliability Engineering, pp. 11–20 (2012). https://doi.org/10.1109/ISSRE.2012.31
63. Just, R., Schweiggert, F.: Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators. Software Test. Verif. Reliab. **25**(5-7), 490–507 (2015). https://doi.org/10.1002/stvr.1561

64. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002). https://doi.org/10.1109/4235.996017
65. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The Oracle problem in software testing: a survey. IEEE Trans. Software Eng. **41**(5), 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785
66. Sicilia, M.-A., Sánchez-Alonso, S., Mora-Cantallops, M., García-Barriocanal, E.: On the Source Code Structure of Quantum Code: Insights from Q# and QDK. In: Shepperd, M., Abreu, F., Silva, A., Pérez-Castillo, R. (eds.) Quality of Information and Communications Technology. Communications in Computer and Information Science, pp. 292–299. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58793-2_24
67. Yamashita, K., Huang, C., Nagappan, M., Kamei, Y., Mockus, A., Hassan, A.E., Ubayashi, N.: Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density. In: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 191–201 (2016). https://doi.org/10.1109/QRS.2016.31
68. Kumar, A.: Formalization of structural test cases coverage criteria for quantum software testing. Int. J. Theor. Phys. **62**, (2023). https://doi.org/10.1007/s10773-022-05271-y
69. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are Mutants a Valid Substitute for Real Faults in Software Testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 654–665. ACM, Hong Kong China (2014). https://doi.org/10.1145/2635868.2635929. https://dl.acm.org/doi/10.1145/2635868.2635929
70. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 437–440. ACM, San Jose, CA, USA (2014). https://doi.org/10.1145/2610384.2628055. https://dl.acm.org/doi/10.1145/2610384.2628055
71. Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, A., Ferenc, R., Mesbah, A.: BugsJS: A Benchmark of JavaScript Bugs. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 90–101 (2019). https://doi.org/10.1109/ICST.2019.00019
72. Widyasari, R., Sim, S.Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J.E., Yieh, Y., Goh, B., Thung, F., Kang, H.J., Hoang, T., Lo, D., Ouh, E.L.: BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2020, pp. 1556–1560. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3417943
73. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 201–211 (2015). https://doi.org/10.1109/ASE.2015.86
74. Lukasczyk, S., Kroiß, F., Fraser, G.: An empirical study of automated unit test generation for Python. Empirical Software Eng. **28**(2), 36 (2023)
75. Paterson, D., Campos, J., Abreu, R., Kapfhammer, G.M., Fraser, G., McMinn, P.: An Empirical Study on the Use of Defect Prediction for Test Case Prioritization. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 346–357 (2019). https://doi.org/10.1109/ICST.2019.00041
76. Miranda, B., Cruciani, E., Verdecchia, R., Bertolino, A.: FAST Approaches to Scalable Similarity-Based Test Case Prioritization. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, pp. 222–232. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180210

77. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and Improving Fault Localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 609–620 (2017). https://doi.org/10.1109/ICSE.2017.62
78. Li, X., Li, W., Zhang, Y., Zhang, L.: DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2019, pp. 169–180. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3330574
79. Zou, D., Liang, J., Xiong, Y., Ernst, M.D., Zhang, L.: An empirical study of fault localization families and their combinations. IEEE Trans. Software Eng. **47**(2), 332–347 (2021). https://doi.org/10.1109/TSE.2019.2892102
80. Sarhan, Q.I., Beszédes, A.: A survey of challenges in spectrum-based software fault localization. IEEE Access **10**, 10618–10639 (2022). https://doi.org/10.1109/ACCESS.2022.3144079
81. Widyasari, R., Prana, G.A.A., Haryono, S.A., Wang, S., Lo, D.: Real world projects, real faults: evaluating spectrum based fault localization techniques on Python projects. Empirical Software Eng. **27**(6), 147 (2022)
82. Durieux, T., Madeiral, F., Martinez, M., Abreu, R.: Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019, pp. 302–313. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338906.3338911
83. Campos, J., Souto, A.: QBugs: A Collection of Reproducible Bugs in Quantum Algorithms and a Supporting Infrastructure to Enable Controlled Quantum Software Testing and Debugging Experiments (2021)
84. Paltenghi, M., Pradel, M.: Bugs in quantum computing platforms: an empirical study. Proc. ACM Program. Lang. **6**(OOPSLA1), (2022). https://doi.org/10.1145/3527330
85. Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., Whitehead Jr., E.J.: Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, pp. 372–381. IEEE Press, San Francisco, CA, USA (2013)
86. Freitas, P.A.F.: Software repository mining analytics to estimate software component reliability (2015)
87. D'Ambros, M., Lanza, M., Robbes, R.: Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Software Eng. **17**(4–5), 531–577 (2012). https://doi.org/10.1007/s10664-011-9173-9
88. Catal, C., Diri, B.: A systematic review of software fault prediction studies. Expert Syst. Appl. **36**(4), 7346–7354 (2009). https://doi.org/10.1016/j.eswa.2008.10.027
89. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the location and number of faults in large software systems. IEEE Trans. Software Eng. **31**(4), 340–355 (2005). https://doi.org/10.1109/TSE.2005.49
90. Arcuri, A., Campos, J., Fraser, G.: Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 401–408 (2016). https://doi.org/10.1109/ICST.2016.44
91. Campos, J., Riboira, A., Perez, A., Abreu, R.: Gzoltar: An Eclipse Plug-in for Testing and Debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ASE '12, pp. 378–381. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2351676.2351752