Introduction to quantum software engineering

Yoann Marquer$^{1[0000-0002-4607-967X]}$ and
Domenico Bianculli$^{1[0000-0002-4854-685X]}$

University of Luxembourg, Luxembourg, Luxembourg
{yoann.marquer,domenico.bianculli}@uni.lu

Abstract. Quantum computing allows for processing information exponentially faster than classical computing, which opens opportunities in many software applications. Nevertheless, the transition to this completely different programming paradigm, with counterintuitive quantum concepts, presents substantial difficulties for software engineers when developing quantum programs. In this tutorial, we present an introductory course on quantum computing and quantum software engineering frameworks and propose exercises on writing, executing, and analyzing quantum circuits.

Keywords: Quantum computing · Quantum software engineering.

1 Introduction

Quantum computing (QC) uses quantum bits (qubits) instead of the bits used in classical computing, enabling massive parallel computation using quantum physics properties like superposition. This allows quantum computers to process information exponentially faster than any classical computer, with empirical evidence for quantum supremacy [2]. Thus, QC has an impact on many emerging technologies and industrial use-cases, especially regarding optimization problems [11]. For this reason, technology giants such as IBM, Google, and Microsoft have heavily invested in and committed to QC.

The availability of quantum computers with more and more qubits allows engineers to deploy more complex and diverse quantum programs at a larger scale. Engineering these programs is challenging, as they require expertise in various disciplines like physics and mathematics. Moreover, they involve unintuitive concepts like quantum superposition and entanglement, and are implemented in low-level, error-prone programming languages. Designing, implementing, testing, and maintaining such programs require the adoption of software engineering practices tailored to the specific domain (e.g., dealing with quantum-related bugs), leading to the establishment of *quantum software engineering* (QSE) as a discipline [17].

Furthermore, current quantum computers are called NISQ (noisy intermediate-scale quantum) computers, because they are larger than small-scale prototypes with a few qubits, but not large enough so that quantum error correction can be applied [11]. Thus, current quantum executions are noisy, which decreases the probability to obtain the expected output [5]. Hence the need for noise analysis and reduction techniques to improve the resilience of quantum computing.

In this paper, accompanying a tutorial with the same title, we provide an introduction to QC and touch upon some aspects of QSE, namely quantum programming, as well as analysis, specification, and testing of quantum programs.

2 Background: Quantum Computing

In quantum computing, a system usually has two classical states, denoted $|0\rangle$ and $|1\rangle$. A qubit is in a *superposition* of these states $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where α and β are complex numbers called *amplitudes*. In polar coordinates, $\alpha = |\alpha|\, e^{i\varphi_\alpha}$ and $\beta = |\beta|\, e^{i\varphi_\beta}$, where φ_α and φ_β are called phases; $|\alpha|$ and $|\beta|$ are called magnitudes and satisfy $|\alpha|^2 + |\beta|^2 = 1$. A *quantum state* is the combination of several qubits, e.g., in $|01\rangle$, the first bit is $|0\rangle$ and the second is $|1\rangle$. Qubits that have correlated values are *entangled*, e.g., in $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, the first qubit is $|0\rangle$ if and only if the second one is $|0\rangle$.

While in classical computing logic gates are the basic blocks of circuits, in quantum computing *quantum gates* are the basic blocks of *quantum circuits*. Quantum gates perform unitary transformations on one or several qubits at a time, updating the state of these qubits in a reversible way. A quantum state can be *measured*, in which case it collapses to a classical state, with a probability depending on the magnitude. For instance, a qubit $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ can collapse either in state $|0\rangle$ with probability $|\alpha|^2$ or in state $|1\rangle$ with probability $|\beta|^2$.

Quantum operations are performed in a *quantum processing unit* (QPU), which can have various implementations. The *qubit connectivity* of the QPU is the physical connection allowing qubit interactions, with each qubit typically connected to only a few neighbors. As NISQ computers are noisy, QSE approaches have to consider quantum noise. It is possible to analyze QPUs or use models in quantum simulators to determine the largest sources of noise [21].

3 Quantum Programming

Quantum programming frameworks are very diverse. Quantum circuits are usually described using low-level programming languages embedded in classical languages [1]. Some ideas for higher-level quantum programming languages have been proposed, but they are not ready yet [6,22,26]. Common quantum frameworks, like Qiskit, Cirq, Tket, and PennyLane, implement a quantum platform on top of Python code, using a dedicated library for quantum-specific operations [20].

These frameworks also support essential steps in quantum software development. *Transpilation* is the translation of a circuit to another circuit (written at the same level of abstraction) that takes into account the topology of and the gates supported by the target QPU. *Uncomputation* consists in performing inverse quantum operations in backward order to undo any previously built entanglement, so qubits are ready for the next execution.

As quantum executions can be costly and noisy, quantum *simulators* can be useful to test quantum circuits on classical computers. During a simulation, the execution can be stopped at any moment so intermediate states as well as phase information can be observed at will, facilitating debugging. However, since quantum simulations do not benefit from the speed-up due to actual quantum superposition, simulators can be used only for small circuits, with a few qubits.

4 Quantum Software Engineering

QSE approaches need not only to adapt techniques from classical computing, but also to consider specificities of quantum computing. For instance, a *quantum bug* is a bug on a quantum algorithm (not on the language or implementation) that happens because of quantum considerations (as opposed to classical ones, like API usage) [3]. They often require domain-specific knowledge to fix [19]. Most of them occur in components independent of the execution environment, hence they can be triggered in simulators. They tend to manifest though unexpected outputs, i.e., a silent misbehavior, as opposed to crashes, which are more common for classical bugs. Several studies have investigated quantum bug patterns [14,19,29], e.g., the incorrect usage of quantum gates and incorrect measurement handling. Actually, most quantum bugs are API- or math-related, indicating a lack of proficiency by developers [14].

QSE approaches must face several challenges regarding quantum programs:

- C1: Quantum states cannot be duplicated (as per the no-cloning theorem); therefore one may not extract a value and then study it while the execution resumes.
- C2: Quantum states, when observed through a quantum measurement, collapse into (usually) classical states; consequently, it is challenging to investigate intermediate states to find quantum bugs.
- C3: Intermediate states have a large size; this make them hard to interpret, even in simulation where they can be observed.
- C4: Quantum state space is large: making it challenging to identify the input capable of triggering a quantum bug.
- C5: Quantum programs have probabilistic outcomes, which means that quantum bugs may not manifest in the same way as in classical programs.

4.1 Quantum Program Analysis

Static analysis techniques can be applied to quantum programs without executing them. *Static analyzers* (like Qchecker [28] and LintQ [20]) are based on bug patterns or quantum abstractions; they generate bug reports with bug location and description. *Slicing techniques* address C3 by dividing the circuit in small subcircuits and removing qubits which are not involved in a particular slice [15]. *Probabilistic cloning* [9] tackles C1 as follows. While the no-cloning theorem prevents cloning quantum states using only unitary transformations, this technique performs unitary transformations and quantum measurements to produce, with a chance of failure, a clone that is not entangled with the state of interest. Such a state can then be used to obtain information about it without collapsing the computation.

4.2 Quantum Program Testing

To tackle C4, multiple approaches have been applied to generate and execute input qubits able to trigger quantum bugs. QuanFuzz mutates qubit inputs using common quantum gates [23]. QuBST uses a genetic algorithm to determine a subset of the initial test suite with enough failing tests [24]. QuCAT tests combinations of input qubits [25]. QuraTest generates small circuits to generate qubit inputs with diverse magnitude, phase, and entanglement [27].

One way to tackle C5 is repeating runs to determine the underlying output distribution; a test failure is either an unexpected output or an incorrect output distribution. *Statistical tests* on the output can be used to determine, e.g., if two qubits are equal or are entangled [7]. A recent work [18] has proposed to test qubit states after a *change of basis* (mixed Hadamard basis), depending on the expected state [18]. Such an approach can be used to tackle C3, as it leads to reduced output distributions (and thus smaller test cases), as well as to reduced sample size, reducing execution time while improving fault detection.

4.3 Specifications of Quantum Programs

Similarly to tests on the final state, *specifications* can be written and tested on intermediate states.

Huang and Martonosi [8] addressed C5 by proposing *statistical assertions*, i.e., measurements performed at breakpoints across multiple runs to approximate the underlying distribution. They can be used to determine, for example, whether a qubit is classical (i.e., it has almost always the same value), is in superposition (i.e., it consistently takes several values), or if two qubits are entangled (i.e., they tend to be correlated). Nevertheless, each quantum measurement stops the program execution while each assertion requires many runs to reach statistical significance.

To tackle C2, Liu et al. [12,13] proposed to instrument the source program with circuits checking for *dynamic assertions*, covering any state comparison and some entangled states. Li et al. [10] proposed a generalization of dynamic assertions with *projective predicates*, expressing that a state is in a given projective subspace. These predicates can then be combined using connectives to write more complex assertions. In the context of formal verification, Minh Do and Ogata [16] proposed to use linear temporal logic (LTL) to define properties of quantum programs and verify them using symbolic model checking.

5 Conclusion

In this paper, we have briefly discussed some challenges in QC (the no-duplication of quantum states, their collapse after observation, their size, their space size, and their probabilistic outcome) and reviewed how state-of-the-art approaches in QSE have tackled them.

The growing availability of powerful quantum computers capable to run complex and diverse programs calls for further research in QSE. This research is required to equip more software developers with the skills to master this novel computing paradigm. Given the interdisciplinary nature of the QC community, it is crucial to reach consensus on the *quantum abstractions* [4] that should be adopted when defining high-level quantum programming languages. These languages will facilitate improved compatibility among various quantum frameworks and simplify the development of quantum programs.

Acknowledgment

Acknowledgments. This project has received funding from SES and the Luxembourg National Research Fund under the Industrial Partnership Block Grant (IPBG), ref. IPBG19/14016225/INSTRUCT.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Akbar, M.A., Khan, A.A., Mahmood, S., Rafi, S.: Quantum software engineering: A new genre of computing. In: Proc. of QCE-NE. ACM (2024)
2. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J., Barends, R., Biswas, R., Boixo, S., Brandao, F., Buell, D., Burkett, B., Chen, Y., Chen, Z., Chiaro, B., Collins, R., Courtney, W., Dunsworth, A., Farhi, E., Foxen, B., Martinis, J.: Quantum supremacy using a programmable superconducting processor. Nature **574** (2019)
3. Di Matteo, O.: On the need for effective tools for debugging quantum programs. In: Proc. of Q-SE. ACM (2024)
4. Di Matteo, O.: The art of abstraction in quantum software. In: 2025 IEEE/ACM International Workshop on Quantum Software Engineering (Q-SE). pp. 25–26 (2025)
5. Ding, Y., Gokhale, P., Lin, S.F., Rines, R., Propson, T., Chong, F.T.: Systematic crosstalk mitigation for superconducting qubits via frequency-aware compilation. In: Proc. of MICRO (2020)
6. Esposito, M., Sabzevari, M.T., Ye, B., Falessi, D., Khan, A.A., Taibi, D.: Classiq: Towards a translation framework to bridge the classical-quantum programming gap. In: Proc. of QSE-NE. ACM (2024)
7. Honarvar, S., Mousavi, M.R., Nagarajan, R.: Property-based Testing of Quantum Programs in Q#. In: Proc. of ICSEW. ACM (2020)
8. Huang, Y., Martonosi, M.: Statistical assertions for validating patterns and finding bugs in quantum programs. In: Proc. of ISCA. ACM (2019)

9. Jiang, N., Wang, Z., Wang, J.: Debugging quantum programs using probabilistic quantum cloning (2023)
10. Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y.: Projection-based runtime assertions for testing and debugging quantum programs. Proc. ACM Program. Lang. **4** (2020)
11. Liimatta, P., Taipale, P., Halunen, K., Heinosaari, T., Mikkonen, T., Stirbu, V.: Research versus practice in quantum software engineering: Experiences from credit scoring use case. IEEE Software **41** (2024)
12. Liu, J., Byrd, G.T., Zhou, H.: Quantum circuits for dynamic runtime assertions in quantum computation. In: Proc. of ASPLOS. ACM (2020)
13. Liu, J., Zhou, H.: Systematic approaches for precise and approximate quantum state runtime assertion. In: Proc. of HPCA. IEEE (2021)
14. Luo, J., Zhao, P., Miao, Z., Lan, S., Zhao, J.: A comprehensive study of bug fixes in quantum programs. In: Proc. of SANER. IEEE (2022)
15. Metwalli, S.A., Van Meter, R.: A tool for debugging quantum circuits. In: Proc. of QCE. IEEE (2022)
16. Minh Do, C., Ogata, K.: Symbolic model checking quantum circuits in maude. PeerJ Computer Science **10** (2024)
17. Murillo, J.M., Garcia-Alonso, J., Moguel, E., Barzen, J., Leymann, F., Ali, S., Yue, T., Arcaini, P., Pérez-Castillo, R., García-Rodríguez de Guzmán, I., Piattini, M., Ruiz-Cortés, A., Brogi, A., Zhao, J., Miranskyy, A., Wimmer, M.: Quantum software engineering: Roadmap and challenges ahead. ACM Trans. Softw. Eng. Methodol. **34**(5) (May 2025)
18. Oldfield, N.H., Laaber, C., Yue, T., Ali, S.: Faster and better quantum software testing through specification reduction and projective measurements (2024)
19. Paltenghi, M., Pradel, M.: Bugs in quantum computing platforms: an empirical study. Proc. Program. Lang. **6** (2022)
20. Paltenghi, M., Pradel, M.: Analyzing quantum programs with lintq: A static analysis framework for qiskit. Proc. Softw. Eng. **1** (2024)
21. Sarovar, M., Proctor, T., Rudinger, K., Young, K., Nielsen, E., Blume-Kohout, R.: Detecting crosstalk errors in quantum information processors. Quantum Journal **4** (2020)
22. Varga, T., Aragonés-Soria, Y., Oriol, M.: Quantum types: going beyond qubits and quantum gates. In: Proc. of Q-SE) (2024)
23. Wang, J., Gao, M., Jiang, Y., Lou, J., Gao, Y., Zhang, D., Sun, J.: Quanfuzz: Fuzz testing of quantum program (2018)
24. Wang, X., Arcaini, P., Yue, T., Ali, S.: Qusbt: search-based testing of quantum programs. In: Proc. of ICSE. ACM (2022)
25. Wang, X., Arcaini, P., Yue, T., Ali, S.: Qucat: A combinatorial testing tool for quantum software. In: Proc. of ASE. IEEE (2024)
26. Waseem, M., Mikkonen, T., Ahmad, A., Khan, M.T., Haghparast, M., Stirbu, V., Liang, P.: Qadl: Prototype of quantum architecture description language (2024)
27. Ye, J., Xia, S., Zhang, F., Arcaini, P., Ma, L., Zhao, J., Ishikawa, F.: Quratest: Integrating quantum specific features in quantum program testing. In: Proc. of ASE. IEEE (2024)
28. Zhao, P., Wu, X., Li, Z., Zhao, J.: Qchecker: Detecting bugs in quantum programs via static analysis. In: Proc/ of Q-SE (2023)
29. Zhao, P., Zhao, J., Ma, L.: Identifying bug patterns in quantum programs. In: Proc. of Q-SE (2021)