arXiv:2001.10870v1 [cs.SE] 29 Jan 2020

# Is Your Quantum Program Bug-Free?

Andriy Miranskyy[1], Lei Zhang[1], and Javad Doliskani[1]

[1]Department of Computer Science, Ryerson University
Toronto, Canada
[1]{avm, leizhang, javad.doliskani}@ryerson.ca

**Abstract**

Quantum computers are becoming more mainstream. As more programmers are starting to look at writing quantum programs, they face an inevitable task of debugging their code. How should the programs for quantum computers be debugged?

In this paper, we discuss existing debugging tactics, used in developing programs for classic computers, and show which ones can be readily adopted. We also highlight quantum-computer-specific debugging issues and list novel techniques that are needed to address these issues. The practitioners can readily apply some of these tactics to their process of writing quantum programs, while researchers can learn about opportunities for future work.

## 1 Introduction

Quantum Computers (QC) are specialized devices that will be able to solve some problems faster than Classic Computers (CC) [3, 9]. This is known as a 'quantum advantage'.

The QC field is still in its infancy: the largest machines built to date are of the order of tens of qubits [1, 22], which is not sufficient for commercially viable applications. However, the power of the QC increases. We do not know when the quantum advantage will be reached: predictions of experts vary from months [17], to 3-5 years [20], to 7+ years [27]. This implies that QC may become practical within a decade.

The programming languages for QC are mainly low-level, operating at the level of QC register (e.g., OpenQASM [7]). However, higher-level languages are being developed (e.g., Scaffold [19]).

It is argued [26] that for the foreseeable future, the QC will be used in a System of Systems, where the majority of software features will be implemented on a CC, while the features involving quantum algorithms will be 'outsourced' to QC components.

To enable usage of the QC, libraries with pre-packaged quantum algorithms start to appear. For example, Qiskit Aqua [2] (an open source library written in Python) implements quantum algorithms for various domains, such as artificial intelligence, chemistry, and finance. Such a library enables a programmer to treat QC as a black-box and leverage quantum algorithms without having a deep understanding of the QC field.

Of course, the libraries themselves have to be developed by programmers with the expertise in the QC field. These programmers, inevitably, inject defects in their code (uniting CC and QC programming worlds). After that, the code has to be debugged. In this paper, we explore how existing debugging tactics can be applied to QC programs and which novel approaches have to be created.

## 2 Debugging Tactics

Debugging is a process of removing an error, once this error has been exposed [31]. While we hope that one day debugging will become an orderly and automated process (e.g., by automatically mapping bug reports to code where the defect resides, and then issuing a patch for this code [33]), currently it is an art more than a science [31].

The high-level tactics [28, Chapter 8] for debugging a software had not changed significantly over the last 40 years (when the first edition of the seminal work [28] was published), although integrated development environments and various automation tools have streamlined a lot of mundane tasks [36, 25]. The three common tactics [28, 31] are backtracking, cause elimination, and brute force, discussed below.

*Backtracking* debugging centers around examining the execution tree from the point of the error until a perpetrating code block is found. The analysis techniques for a code listing (such as code reviews and inspections) of a CC program can be readily applied to a QC program [26]. Thus, these tactics are transferable. Anecdotally, based on discussions with practitioners, code reviews and inspections are the most popular debugging techniques of quantum programs nowadays.

*Cause elimination* debugging formulates a hypothesis (using inductive or deductive reasoning), specifying a root cause for a bug under study. Then, data are devised, and experiments are conducted to refute or prove this hypothesis. This approach can be applied to QC. Given the probabilistic nature of the QC pro-

grams [29, 26], we will have to execute the program multiple times to obtain a distribution of the results and assess the accuracy of the answer. Thus, we may be able to extend the techniques used for testing probabilistic programs running on CC, such as [10, 11], to the QC domain.

*Brute force* debugging — centred around the analysis of runtime traces, memory dumps, and output statements — focuses on runtime data analysis. Of the three tactics, this is the most common one [31]. Some of the analyses of the runtime artifacts can be automated; however, a lot of the brute force debugging is still performed manually [31]. Can we transfer these tactics?

If we treat a QC program as a black-box, then the short answer is 'yes'. As mentioned in Section 1, if a QC program will be used as part of a System of Systems, then we can trace the input (passed from a CC component to the QC component) and the output (from the QC component to the CC component). The input and output data can be recorded in a log, and these data can be compared against the expected values.

But what if we would like to analyze a QC program at runtime using a white-box approach, e.g., to capture the execution trace of a QC program or perform interactive debugging of the code executed on the QC? In such a case, the short answer is 'it depends'. Let us elaborate on this answer below.

# 3 Debugging Quantum Programs

A quantum program executed on a modern gate-based QC leverages a register of qubits for performing quantum operations and a register of classic bits for recording the measurements of qubits' states and conditionally applying quantum operators [7]. Thus, a typical QC program mixes traditional instructions (to alter the state of bits and apply conditional statements) and quantum instructions (to alter the state of qubits and to measure qubit value).

At any point of execution, the state of a CC is given by a vector of bits taking the values of 0 and 1. A register of $m$ bits can represent $2^m$ states. The state of a QC is, however, given by a vector of qubits and bits. A qubit is a two-state system and, thus, is an element of the space $\mathbb{C}^2$, where $\mathbb{C}$ is the set of complex numbers. The quantum state of a qubit $|b\rangle$, can be captured as a *superposition* (linear combination) of the orthonormal basis states $|0\rangle$ and $|1\rangle$, where $|\cdot\rangle$ denotes a vector in a vector space using the bra-ket notation [21]. In the *computational basis* (i.e. a combination of the $|0\rangle$ and $|1\rangle$ states), a qubit is written as a normalized vector $|b\rangle = \alpha|0\rangle + \beta|1\rangle$, where $|\alpha|^2 + |\beta|^2 = 1$.

A vector of qubits resides in a Hilbert space, which is an euclidean complex vector space (see [34] for details). An $n$-qubits state is an element of $\mathbb{C}^{2^n}$. A

register of $n$-qubits is denoted by one of the equivalent notations $|b_1, \ldots, b_n\rangle$, or $|b_1\rangle|b_2\rangle \cdots |b_n\rangle$, or $|b_1\rangle \otimes |b_2\rangle \otimes \cdots \otimes |b_n\rangle$, where $\otimes$ denotes the tensor product of two vectors.

As mentioned above, a general quantum program consists of blocks of code each containing classical and quantum instructions. Quantum operations can be divided into two kinds: unitary and non-unitary. Unitary operations are reversible and preserve the norm of the operands. Non-unitary operations are not reversible and have probabilistic implementations.

The classical parts of a quantum program can be debugged using traditional methods. The quantum parts, however, can not be treated in the same way because of the properties of a QC — such as superposition, entanglement, and no-cloning — which are governed by the laws of quantum mechanics. The purpose of debugging a program is to present the user with human readable, i.e., classical, information about the runtime state of the system. Extracting classical information from a quantum state is done using measurement which is usually a non-unitary operation and results in collapse of the state, and hence an unintended behavior of the program. We shall describe, in the following, different scenarios in a QC to which classical debugging techniques cannot be applied, and discuss some potential solutions.

### 3.1 Superposition

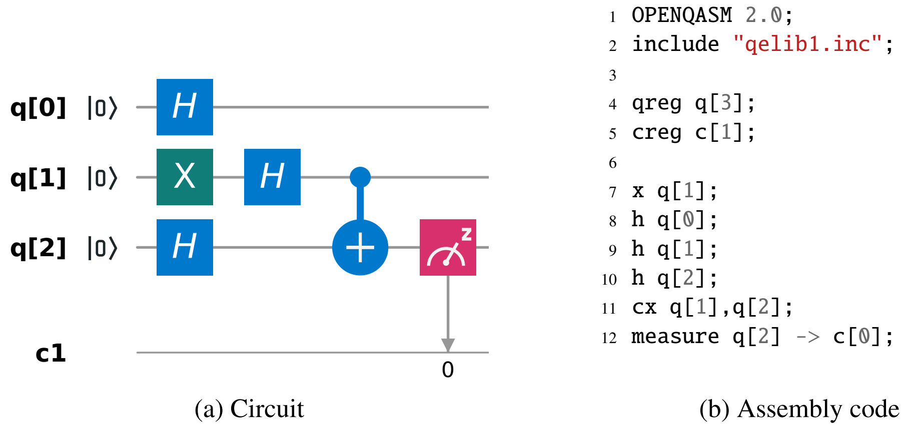
Let $|\psi\rangle$ be the state of an $n$-qubit register. We can uniquely write $|\psi\rangle$, in the computational basis, as

$$|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha_x |x\rangle,$$

where $\alpha_x \in \mathbb{C}$ and $\sum_{x \in \{0,1\}^n} |\alpha_x|^2 = 1$. We say that $|\psi\rangle$ is a superposition of the basis states $\{|x\rangle\}_{x \in \{0,1\}^n}$. By the measurement postulate of quantum mechanics, measuring the state $|\psi\rangle$ in the computational basis results in an outcome $x \in \{0,1\}^n$ with probability $|\alpha_x|^2$, and the state of the system after the measurement is $|x\rangle$. For example, consider the initial state $|010\rangle$ (which is set by applying NOT to qubit 2), and perform the following steps: first apply a Hadamard transform[1] to each qubit (creating superposition), then a controlled-not (CNOT)[2] to qubits 2 and 3, and finally measure qubit 3. If the measured qubit is 0 (which happens with probability 1/2), then the state collapses to $\frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$. An implementation of this example in OpenQASM 2.0 is shown in Figure 1.

[1]The Hadamard transform on one qubit is given by $|b\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + (-1)^b|1\rangle)$. A description of transforms and associated quantum gates are given in [21].

[2]The CNOT transform is defined on two qubits and is given by $|a, b\rangle \mapsto |a, a \oplus b\rangle$, where $\oplus$ is xor. Here, $|a\rangle, |b\rangle$ are called the control and input qubits, respectively.




(a) Circuit (b) Assembly code

Figure 1: Example of measuring a superposition. In OpenQASM, NOT is denoted by $x$, Hadamard — by $h$, and CNOT — by $cx$.

A natural feature of a debugger for quantum programs would be to check if the state of a variable is in superposition. There are the following two scenarios: when the input state is unknown (e.g., when it is generated as an output of another quantum program) and when the input state is known. Let us elaborate on these two cases.

#### 3.1.1 Unknown input state.

If the input to the program is an unknown state $|\psi\rangle$, then there is no known general algorithm that can efficiently decide if $|\psi\rangle$ is in a superposition. Not much can be done here in terms of a general method for debugging, different approaches should be considered for different problems.

For example, in the hidden subgroup problem [21, Chapter 7], if the group is abelian, then it can be efficiently decided if the coset state of a subgroup is in superposition. For non-abelian groups, however, the same problem is often hard. For example, the best known algorithm for the following problem has subexponential runtime [23]: let $N$ be a positive integer, and let $\mathbb{Z}_N$ be the group of integers mod $N$. For a random unknown $x \in \mathbb{Z}_N$ and fixed unknown $d \in \mathbb{Z}_N$, decide whether a given state is of the form $|x\rangle$ or $\frac{1}{\sqrt{2}}(|x\rangle + |x + d\rangle)$.

#### 3.1.2 Known input state.

If a state is the result of applying a unitary operation to a known initial state, i.e., $|\psi\rangle = U|x\rangle$ where $x$ is known, then $|\psi\rangle$ can be regenerated by the debugger. For

example, the state

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{h(x)} |x\rangle,$$

where $h(x)$ is the Hamming weight[3] of $x$, can be generated by applying the Hadamard transform to the $n$-qubit register $|11\ldots1\rangle$. In such cases, there are various methods (depending on the problem) to characterize the state $|\psi\rangle$. Often, one relies on *quantum state tomography*, which is the process of reconstructing a quantum state through a series of measurements [8, 6].

## 3.2 Entanglement

In a QC, a set of memory cells or registers is said to be in an entangled state if it is impossible to classically specify the correlations among them. More precisely, let $X_1, \ldots, X_n$ be the state spaces of a set of quantum systems that represent $n$ registers. The state space of the composite of these systems, that represents an array, is given by the tensor product $X = X_1 \otimes \cdots \otimes X_n$. A state $|\psi\rangle \in X$ that can be written in the form $|\psi\rangle = |\psi_1\rangle \otimes \cdots \otimes |\psi_n\rangle$, where $|\psi_j\rangle \in X_j$ for $j = 1, \ldots, n$, is called separable. A state that is not separable is called entangled. When debugging a program that operates on an entangled state, the following problems can be considered.

### 3.2.1 Checking for separability.

Given a state $|\psi\rangle \in X$, deciding whether $|\psi\rangle$ is separable is an NP-hard problem [13, 16]. This is called the *separability problem* in quantum information theory, see [34, Chapter 6] for details. There are a variety of methods (see [24, 15]) for separability/entanglement detection that can be implemented in practice, specially for lower dimensions. For example, if the debugger can generate several copies of $|\psi\rangle$, then one way to detect the nonlinear properties of $|\psi\rangle$ is via direct measurement. For the sake of brevity, we do not provide technical details here; see [24] for a numerical method for examining separability and [15] for other interesting methods and their implementations.

### 3.2.2 Extracting classical information.

Measuring a subsystem of a larger composite system that is in an entangle state will likely alter other subsystems. This prevents a debugger from presenting any classical information about a variable (whose state is entangled) to the user without disturbing its state.

[3] That is, the number of non-zero bits in $x$.

For example, consider the entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ of two qubits. Measuring any of the two qubits alters the result of the subsequent measurement on the other qubit. More precisely, if the first qubit is measured, then state collapses to $|00\rangle$ or $|11\rangle$ with probability $|1/\sqrt{2}|^2 = 1/2$; the outcome of measuring the second qubit is always 0 if the resulting state is $|00\rangle$, and it is always 1 if the resulting state is $|11\rangle$. Such a state is called maximally entangled.

A composite system, however, often has subsystems that are not entangled with any other subsystem. In this case, we can measure that subsystem without disturbing the whole state. For example, in the 3-qubit register

$$\frac{1}{2}(|000\rangle - |001\rangle + |110\rangle - |111\rangle) \tag{1}$$

the last qubit is not entangled with the first two while the first and the second qubits are entangled, see (2). The algorithms for separability detection (discussed in Section 3.2.1) could be used to identify separable subsystems. Things would be much simpler if the debugger could somehow estimate a given state with a state that is generated by applying some operation to a basis state, i.e., classical information. For example, the state in (1) can be generated as

$$\begin{aligned}(\mathrm{CNOT} \otimes H_2)(H_2 \otimes I_4)|001\rangle = \\ = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle),\end{aligned} \tag{2}$$

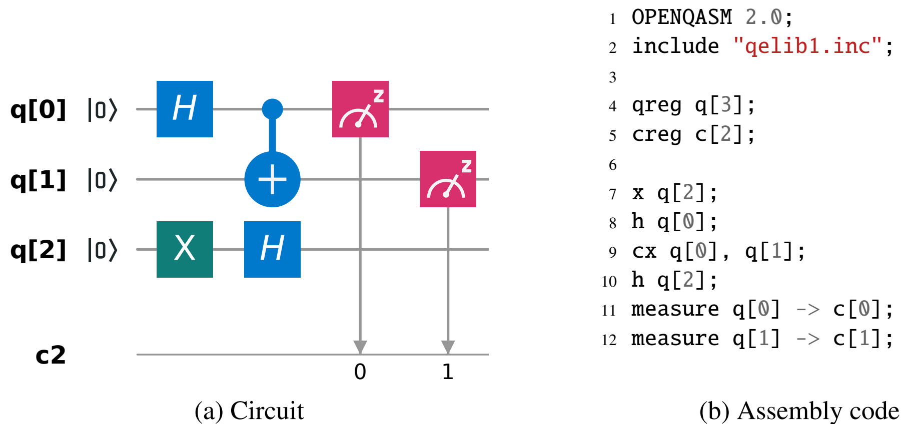
where $I_4$ and $H_2$ are the identity and Hadamard gates, respectively. Therefore, the state in (1) can be described by the debugger using the classical information $|001\rangle$ and the names of the above operators. An implementation of the sequence of operations in (2) is shown in Figure 2.

## 3.3 No-cloning

The most general method of obtaining information about a variable without disturbing its state is to make a copy of the variable and work on the copy. In the classical setting, this is often straightforward. In the quantum setting, however, the situation is much more complicated. In fact, it is impossible to make a copy of a given general unknown quantum state. More precisely, given an unknown state $|\psi\rangle$ and an arbitrary state $|\phi\rangle$, it can be shown [21, Theorem 10.4.1] that there is no unitary operator $U$ that can perform the following:

$$|\psi\rangle \otimes |\phi\rangle \xmapsto{U} |\psi\rangle \otimes |\psi\rangle.$$

In many practical scenarios, however, a debugger will only need to make an *approximate copy* of a state; a state that is "close enough" to the given state but




(a) Circuit (b) Assembly code

Figure 2: An implementation of state (2).

provides useful debugging information. For example, for a state $|\psi\rangle$ that encodes a probability distribution [14], such as the Gaussian distribution, an approximate clone would provide valuable information about the distribution. The possibility of approximate cloning was first discussed in [4]. Much research has been done on different cloning methods each optimizing particular aspects of a cloner that are desired for different situations, see [32] for a survey.

### 3.4 Discussion

In Sections 3.1–3.3, we discussed various issues preventing the application of the classic debugging techniques and identified some potential solutions.

As discussed in [26], if the input size and the amount of required qubits is small, we can run a quantum program in a simulator (running on a CC). However, the increase of the input size and the qubit register length may force us to run the program on a QC.

If we can generate multiple approximate copies of the state [4], then we can produce an empirical distribution of the qubit state and compare it against the expected distribution, to detect problems in the code. The generation of the multiple approximate copies can be readily implemented for moderate inputs sizes using universal cloning methods [35, 5, 12]. More efficient cloning can be achieved using state-dependent (i.e. non-universal) cloning methods [30, 32]. This would address issues related to superposition with known input state (discussed in Section 3.1.2), extraction of classical information (discussed in Section 3.2.2), and no-cloning (discussed in Section 3.3). A compiler can automatically generate the code for the approximate copying (akin to compilers for CC that can instrument the

code to add debugging information), translating higher-level language into quantum assembly [18]. The same principle of multiple approximate copies can be used to generate runtime assertions [37].

For the case of unknown input states, discussed in Section 3.1.1, no general solution exists and will require a programmer to make decisions on a case-by-case basis.

Finally, separability checking, discussed in Section 3.2.1, demands the implementation of numerical methods that will require changes to the QC and, hopefully, will be implemented in the future.

# 4 Conclusions

QC field is rapidly evolving, and the Software Engineering (SE) community should start bringing SE practices into the QC world. In this paper, we focus on the analysis of debugging tactics, highlighting classic ones that are readily applicable and showing that new ones have to be created. We believe that this work would be of interest to practitioners, creating quantum programs, as well as researchers, developing the next generations of tooling for QC.

# References


[1] [n. d.]. IBM Q Experience. `https://quantum-computing.ibm.com/` Accessed on 2019-09-14.

[2] H. Abraham et al. 2019. Qiskit: An Open-source Framework for Quantum Computing. `https://doi.org/10.5281/zenodo.2562110`

[3] E. Bernstein and U. Vazirani. 1997. Quantum complexity theory. *SIAM Journal on computing* 26, 5 (1997), 1411–1473.

[4] V. Bužek and M. Hillery. 1996. Quantum copying: Beyond the no-cloning theorem. *Physical Review A* 54, 3 (1996), 1844.

[5] V. Bužek and M. Hillery. 1998. Universal optimal cloning of arbitrary quantum states: from qubits to quantum registers. *Physical review letters* 81, 22 (1998), 5003.

[6] M. Cramer, M. B. Plenio, S. T. Flammia, R. Somma, D. Gross, S. D. Bartlett, O. Landon-Cardinal, D. Poulin, and Y.-K. Liu. 2010. Efficient quantum state tomography. *Nature communications* 1 (2010), 149.

[7] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. 2017. Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017).

[8] G. M. D'Ariano, M. G. Paris, and M. F. Sacchi. 2003. Quantum tomography. *Advances in Imaging and Electron Physics* 128 (2003), 206–309.

[9] D. Deutsch. 1985. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proc. R. Soc. Lond. A* 400, 1818 (1985), 97–117.

[10] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic. 2018. Testing probabilistic programming systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 574–586. https://doi.org/10.1145/3236024.3236057

[11] S. Dutta, W. Zhang, Z. Huang, and S. Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. 729–739. https://doi.org/10.1145/3338906.3338972

[12] H. Fan, K. Matsumoto, and M. Wadati. 2001. Quantum cloning machines of a d-level system. *Physical Review A* 64, 6 (2001), 064301.

[13] S. Gharibian. 2010. Strong NP-hardness of the quantum separability problem. *Quantum Information & Computation* 10, 3 (2010), 343–360.

[14] L. Grover and T. Rudolph. 2002. Creating superpositions that correspond to efficiently integrable probability distributions. *arXiv preprint quant-ph/0208112* (2002).

[15] O. Gühne and G. Tóth. 2009. Entanglement detection. *Physics Reports* 474, 1-6 (2009), 1–75.

[16] L. Gurvits. 2003. Classical deterministic complexity of Edmonds' Problem and quantum entanglement. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. ACM, 10–19.

[17] K. Hartnett. 2019. A New Law to Describe Quantum Computing's Rise? https://www.quantamagazine.org/does-nevens-law-describe-quantum-computings-rise-20190618/ Accessed on 2019-09-14.

[18] Y. Huang and M. Martonosi. 2019. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. 541–553. https://doi.org/10.1145/3307650.3322213

[19] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi. 2014. ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Proc. of Computing Frontiers Conference, CF'14*. 1:1–1:10. https://doi.org/10.1145/2597917.2597939

[20] C. Jeffrey. 2019. IBM VP says quantum computer commercialization coming in next 3–5 years. https://www.techspot.com/news/80222-ibm-vp-quantum-computer-commercialization-coming-next-3.html Accessed on 2019-09-14.

[21] P. Kaye, R. Laflamme, M. Mosca, et al. 2007. *An introduction to quantum computing*. Oxford University Press.

[22] J. Kelly. 2018. Google AI Blog: A Preview of Bristlecone, Google's New Quantum Processor. https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html Accessed on 2019-09-14.

[23] G. Kuperberg. 2005. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM J. Comput.* 35, 1 (2005), 170–188.

[24] J. M. Leinaas, J. Myrheim, and E. Ovrum. 2006. Geometrical aspects of entanglement. *Physical Review A* 74, 1 (2006), 012313.

[25] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: automated end-to-end repair at scale. In *ICSE (SEIP) 2019*. 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[26] A. V. Miranskyy and L. Zhang. 2019. On testing quantum programs. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2019*. 57–60. https://doi.org/10.1109/ICSE-NIER.2019.00023

[27] M. Mosca. 2018. Cybersecurity in an era with quantum computers: will we be ready? *IEEE Security & Privacy* 16, 5 (2018), 38–41.

[28] G. J. Myers, C. Sandler, and T. Badgett. 2011. *The art of software testing* (3 ed.). John Wiley & Sons.

[29] M. A. Nielsen and I. L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge Univ. Press.

[30] C.-S. Niu and R. B. Griffiths. 1999. Two-qubit copying machine for economical quantum eavesdropping. *Physical Review A* 60, 4 (1999), 2764.

[31] R. S. Pressman and B. R. Maxim. 2014. *Software engineering: a practitioner's approach* (8 ed.). McGraw-Hill.

[32] V. Scarani, S. Iblisdir, N. Gisin, and A. Acin. 2005. Quantum cloning. *Reviews of Modern Physics* 77, 4 (2005), 1225.

[33] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 25–36. https://doi.org/10.1109/ICSE.2019.00021

[34] J. Watrous. 2018. *The theory of quantum information*. Cambridge University Press.

[35] R. F. Werner. 1998. Optimal cloning of pure states. *Physical Review A* 58, 3 (1998), 1827.

[36] A. Zeller. 2009. Debugging debugging: acm sigsoft impact paper award keynote. In *ACM SIGSOFT 2009, Amsterdam, The Netherlands, August 24-28, 2009*. 263–264. https://doi.org/10.1145/1595696.1595736

[37] H. Zhou and G. T. Byrd. 2019. Quantum Circuits for Dynamic Runtime Assertions in Quantum Computation. *IEEE Computer Architecture Letters* 18, 2 (July 2019), 111–114.