Gate Branch Coverage: A Metric for Quantum Software Testing

Daniel Fortunato
Faculty of Engineering of the University of Porto
Porto, Portugal
INESC-ID
Lisboa, Portugal
dabf@fe.up.pt

José Campos
Faculty of Engineering of the University of Porto
Porto, Portugal
LASIGE, Faculty of Science of the University of Lisbon
Lisboa, Portugal
jcmc@fe.up.pt

Rui Abreu
Faculty of Engineering of the University of Porto
Porto, Portugal
INESC-ID
Lisboa, Portugal
rui@computer.org

ABSTRACT

The inherent lack of technologies and knowledge from software developers about the intricacies of quantum physics constitutes a heavy hindrance in the development of correct quantum software. Therefore, quantum computing testing techniques are currently under heavy research. This paper proposes a new testing metric, Gate Branch Coverage. This metric aims to provide insight into the verification process status of quantum programs and enhance the quantum testing process overall. Gate Branch Coverage explores the properties of quantum controlled-type gates, measuring their number of exercised branches during the execution of quantum programs.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Quantum Computing, Quantum Software Testing, Quantum Gate Branch Coverage

ACM Reference Format:
Daniel Fortunato, José Campos, and Rui Abreu. 2024. Gate Branch Coverage: A Metric for Quantum Software Testing. In *Proceedings of the 1st ACM International Workshop on Quantum Software Engineering: The Next Evolution (QSE-NE '24), July 16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3663531.3664753

1 INTRODUCTION

The imminent arrival of quantum computers with universal accessibility is bound to break numerous computational constraints. However, this advancement also presents significant challenges across various computer science domains [30], such as *software testing* [5, 10]. Despite the extensive exploration and proposition of approaches and tools for testing in classical computing [12–14], these methodologies remain nascent for quantum programs [8, 16, 26].

Software testing has emerged as a vital activity in the software maintenance life cycle, assessing and enhancing software quality [19]. Test coverage [22, 31], a key component of testing activities, measures the extent to which a program has been exercised and provides a percentage of the lines of code being covered. When coverage falls short of 100%, additional tests can be designed to target missed lines of code and improve overall coverage.

Moreover, measuring test coverage serves various purposes in enhancing the testing process. It provides insights into the verification process's status, identifies uncovered areas, and aids in activities such as regression testing [14], test case prioritization [24], and test suite augmentation [20]. Test coverage measurement and analysis for classical software has been the subject of heavy research [21] and has been applied to quantum software *as is* [8, 26, 28].

In this paper, we propose a novel coverage metric tailored for quantum programs named Gate Branch Coverage (GBC). Leveraging the idea of gate branches [15] for quantum controlled-type gates, we formally define how to calculate GBC for any given quantum program. Also, to obtain its GBC, we illustrate how to instrument a QASM [4] program, the quantum equivalent to classical Assembly utilized by most quantum frameworks to execute quantum circuits.

We argue and demonstrate how GBC can provide additional insight into the verification process status of quantum programs and enhance the quantum testing process overall. Moreover, we show how we can improve GBC for a given quantum program.

2 BACKGROUND

Quantum frameworks (e.g., IBM's Qiskit [1] and Google's Cirq [6]) provide libraries for the creation and manipulation of quantum circuits. These frameworks provide us with the means to execute quantum circuits using a simulator or a quantum computer. Currently, given the difficult access and highly volatile nature of quantum computers, it is preferable to use simulators as the default backend for quantum circuit execution.

QASM [4], the quantum low-level language equivalent to Assembly for classical programs, is the language used by quantum computers or simulators to execute a quantum circuit and is universally used in many frameworks (e.g., IBM's Qiskit and Google's Cirq). Therefore, the metric we propose is tailored to instrument QASM programs and can be used in all QASM-based frameworks.

There are two ways to build a quantum circuit in these frameworks: (1) building the circuit in a high-level language (i.e., usually Python) and use the frameworks' transpiler that transforms the high-level code to QASM [4] or (2) building the circuit in a QASM file and import this file as a circuit to a high-level language program.

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3 qreg q[2];
4 creg c[2];
5 cx q[0], q[1];
6 measure q[0] -> c[0];
7 measure q[1] -> c[1];
```

Figure 1: Motivational example quantum circuit.

In quantum computing, circuit operations are defined as gates. Many types of gates exist that perform distinct operations. For this work, we look into controlled-type gates. Controlled-type gates are multi-qubit gates that perform some operation (depending on the gate used) on one target qubit depending on the value of the control qubit(s). There are many types of gates in quantum computing that can be controlled [2]:

- Pauli Gates (i.e., Controlled-Not (cx), Controlled-Y (cy), Controlled-Z (cz), Controlled-H (ch))
- Rotation Gates (i.e., Controlled-r-z (crz), Controlled-r-x (crx), Controlled-r-y (cry))
- Phase Gates (i.e., Controlled-p (cp))
- U gates (i.e., Controlled-u (cu))

As we already stated, these types of gates can either perform some operation or not. This property effectively means that a controlled-type gate can follow two different branches: one where some gate operation is performed onto the target qubit, and another where nothing happens.

3 GATE BRANCH COVERAGE

3.1 Motivational Example

We present a motivational example in Figure 1 to better visualize and understand how controlled-type gates function. Lines 3 and 4 define a quantum and classical register with two qubits and bits, respectively. Line 5 applies a Controlled-Not gate with q[0] as the control qubit and q[1] as the target. Finally, we measure both qubits in lines 6 and 7. This quantum circuit can be translated into the flow graph presented in Figure 2. Recall that QASM circuits are executed sequentially, meaning that every line in the circuit will always be executed, which, from a classical point of view, signifies that the circuit in Figure 1 is always 100% covered. However, in Figure 2, the Controlled-Not box depicted in red divides the flow of the program through two branches. The value of the control qubit is calculated in the yellow box. If the control qubit equals 1, the gate follows the green branch and flips the target qubit's value by applying the Not gate (x). Otherwise, it follows the blue branch and proceeds to the next step of the circuit. Note that all controlled-type gates behave this way; only two aspects differ from gate to gate: (1) the operation performed in the green box and (2) the number of control qubits of the gate. For each control qubit, the equivalent number of yellow boxes will be present (with one being the minimum, as shown in Figure 2), and only if all of them equal 1 is the operation in green performed.

3.2 Formal Definition

We propose Gate Branch Coverage (GBC), a new metric tailored for quantum programs to measure the coverage of controlled-type gates branches. Formally, we define GBC as follows,

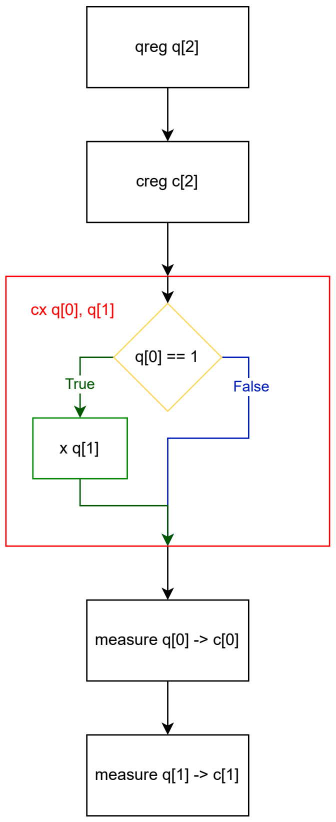


Figure 2: Flow graph of the motivational example in Figure 1.

$$GBC(G) = \frac{\sum_{g \in G} e(g)}{\sum_{g \in G} 1 + c(g)} * 100\% \qquad (1)$$

where G is the set of controlled-type gates of the program, $e(g)$ is the number of exercised branches of gate g, and $c(g)$ is the number of control qubits of gate g. The computation of $e(g)$ is described in Section 3.3.

3.3 Implementation

One of the main properties of quantum computing is *superposition* (i.e., a qubit can simultaneously be in both states 0 and 1 and its result collapses to either one upon measurement). This property completely nullifies the ability of a developer to inspect a program during runtime without ruining its result. Therefore, to obtain $e(g)$ to calculate the GBC of a given quantum circuit, it is required to add two lines of code and alter another two per controlled-type gate to the quantum circuit. This instrumentation presented in Figure 3 is detailed below:

(1) One extra qubit to the quantum register (line 3), used to store the value of the control qubit. Note that qubits are initialized in state 0.
(2) One bit to the classical register (line 4), used to store the measurement of the control qubit (i.e., the branch which the Controlled-Not gate passed through).

```
1  OPENQASM 2.0;
2  include "qelib1.inc";
3  qreg q[3];
4  creg c[3];
5  cx q[0], q[1];
6  cx q[0], q[2];
7  measure q[2] -> c[2];
8  measure q[0] -> c[0];
9  measure q[1] -> c[1];
```

Figure 3: Motivational example from Figure 1 instrumented with the implementation described in Section 3.3.

(3) One Controlled-Not gate (line 6), used to flip to the opposite state our additional qubit and, therefore, get the branch undertaken by the control qubit. This means that if the measurement of our additional qubit is 0 (i.e., it remained unchanged) then, from Figure 2, we can say that the blue branch was taken; otherwise, the green branch was taken.
(4) One measurement operation (line 7), used to measure our additional qubit.

After instrumenting our motivational example with these changes, we executed both circuits to calculate the circuit's GBC. Note that to execute all our circuit examples, we used the *qasm-simulator* backend in the publicly accessible IBM Quantum Platform web interface[1]. Executing them 1000 times each we obtain 1000 measurements of states 00 and 000, respectively. These results are to be expected, given that we kept the initial values of both qubits (i.e., 00), the controlled-not gate in line 5 (Figures 1 and 3) took the blue branch (Figure 2). The first qubit in 000 informs us that the execution went through the blue branch. Since this is the only measurement output our program recorded, this means the blue branch was executed 100% of the times. Considering that only one controlled-type gate with one control qubit (i.e., $c(g) = 1$) was present in our motivational example, and that only one branch was exercised (i.e., $e(g) = 1$), using Equation (1) we calculate the GBC of Figure 1 as $\frac{1}{1+1} * 100 = 50\%$.

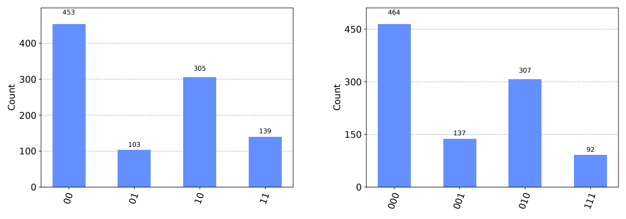
3.4 Improving GBC Score

Classically, to improve test coverage we would augment the program's test suite. In quantum computing this can also be done. In this case, we can initialize qubits with a different state until the GBC is improved; this would be the equivalent of performing fuzz testing [17] in the classical world but applied to quantum programs. Note that fuzzing on quantum programs has already been done and proved effective by Wang et al. [25]. Furthermore, QuraTest, a quantum input generation tool, was recently published by Ye et al. [29], which we use to generate the inputs in lines 5 and 6 of Figure 4.

After adding these inputs to the original example (Figure 1) and its instrumented version in Figure 3 (the result is shown in Figure 4), executing both circuits yielded the results presented in Figure 5. Figures 5a and 5b show the results for each circuit, respectively. This time we obtained four distinct measurement outputs: 00, 01, 10, and 11. The instrumented circuit informs us that the blue branch was executed 908 times (464+307+137) and the green branch 92 times. Note that the executions of Figures 5a and 5b yielded different yet

```
1  OPENQASM 2.0;
2  include "qelib1.inc";
3  qreg q[3];
4  creg c[3];
5  u(-0.9709814728188881,7.895822393995231,0) q[0];
6  u(1.4095460539185645,0,-3.1088748636690364) q[1];
7  cx q[0], q[1];
8  cx q[0], q[2];
9  measure q[2] -> c[2];
10 measure q[0] -> c[0];
11 measure q[1] -> c[1];
```

Figure 4: Motivational example from Figure 3 with inputs generated by QuraTest [29].



(a) Default **(b) Instrumented**

Figure 5: Circuit output of the motivational example in Figure 1 with the inputs used in Figure 4, and circuit output of the motivational example in Figure 4.

similar results. This is to be expected given the probabilistic nature of quantum programs. Given that both branches were taken during execution, the GBC score of Figure 4 is 100%.

4 RELATED WORK

Quantum software testing is an emerging field still in its infancy [10, 30]. de la Barrera et al. [5], Zhao [30] present current debugging and testing of quantum software research progress in their works. A common practice has been applying well-established classical testing approaches to quantum computing. Wang et al. [25] developed QuanFuzz, a fuzz testing approach to generate test cases for quantum programs, QuCAT [28], a combinatorial testing approach that utilizes various inputs to generate test suites with the intent to maximize the number of failing tests. Both these techniques could potentially be used to increase GBC score. Additionally, Wang et al. [27] present QuSBT, a test generation tool for quantum programs that uses an evolutionary algorithm to search for the maximum set of tests that reveal faults.

Kumar [15] applied cyclomatic complexity [18], a quantitative metric for finding the number of branches in a control flow graph, to quantum circuits. The author formally defines how to calculate the number of branches of a given control flow graph of a quantum circuit. While cyclomatic complexity measures the complexity of the code structure itself, GBC measures how well the quantum circuit has been tested. They are related in the sense that reducing cyclomatic complexity can often lead to code that is easier to test and understand, potentially increasing GBC, but they are not directly interchangeable metrics. It is generally beneficial to strive for both low cyclomatic complexity and high GBC to improve code quality and reliability.

[1] https://quantum.ibm.com/composer/.

5 CONCLUSION AND FUTURE WORK

In this paper, we proposed Gate Branch Coverage (GBC), a new metric tailored for quantum circuits. We formally define how to calculate the GBC of a given quantum circuit and illustrate with a simple example how to instrument a quantum circuit to compute its GBC. We show how to improve the GBC of a given program and how it can help provide additional information for developers during the verification process of a quantum program and enhance their quantum testing process.

For future work, we intend to perform a large-scale study on a larger sample of quantum programs. Quantum benchmarks such as Veri-Q [3] provide an extensive number of implementations of some of the most famous quantum algorithms, such as Shor's [23] and Grover's [11]. Furthermore, we also intend to investigate whether there is any correlation between GBC and, for instance, mutation score [7–9].

ACKNOWLEDGMENTS

This work was supported by FCT through grant ref. 2023.02439.BD, project DOI:10.54499/UIDB/50021/2020, and the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

REFERENCES

[1] Gadi Aleksandrowicz et al. 2019. *Qiskit: An Open-source Framework for Quantum Computing.* https://doi.org/10.5281/zenodo.2562111

[2] Jean-Luc Brylinski and Ranee Brylinski. 2002. Universal quantum gates. In *Mathematics of quantum computation.* Chapman and Hall/CRC, 117–134.

[3] Kean Chen, Wang Fang, Ji Guan, Xin Hong, Mingyu Huang, Junyi Liu, Qisheng Wang, and Mingsheng Ying. 2022. VeriQBench: A Benchmark for Multiple Types of Quantum Circuits. arXiv:2206.10880 [quant-ph] https://arxiv.org/abs/2206.10880

[4] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. arXiv:1707.03429 [quant-ph] https://arxiv.org/abs/1707.03429

[5] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and José A. Cruz-Lemus. 2022. *Quantum Software Testing: Current Trends and Emerging Proposals.* Springer International Publishing, Cham, 167–191. https://doi.org/10.1007/978-3-031-05324-5_9

[6] Cirq Developers. 2023. *Cirq.* https://doi.org/10.5281/zenodo.10247207

[7] Daniel Fortunato, José Campos, and Rui Abreu. 2022. Mutation Testing of Quantum Programs: A Case Study With Qiskit. *IEEE Transactions on Quantum Engineering* 3 (2022), 1–17. https://doi.org/10.1109/TQE.2022.3195061

[8] Daniel Fortunato, José Campos, and Rui Abreu. 2022. Mutation testing of quantum programs written in QISKit. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22).* Association for Computing Machinery, New York, NY, USA, 358–359. https://doi.org/10.1145/3510454.3528649

[9] Daniel Fortunato, José Campos, and Rui Abreu. 2022. 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* (Virtual, South Korea) *(ISSTA 2022).* Association for Computing Machinery, New York, NY, USA, 797–800. https://doi.org/10.1145/3533767.3543296

[10] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. 2023. Quantum software testing: State of the art. *Journal of Software: Evolution and Process* 35, 4 (2023), e2419. https://doi.org/10.1002/smr.2419 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2419

[11] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96).* Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866

[12] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. 2013. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.* 22, 1, Article 6 (mar 2013), 42 pages. https://doi.org/10.1145/2430536.2430540

[13] N. Juristo, A.M. Moreno, and W. Strigel. 2006. Guest Editors' Introduction: Software Testing Practices in Industry. *IEEE Software* 23, 4 (2006), 19–21. https://doi.org/10.1109/MS.2006.104

[14] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. *ACM Comput. Surv.* 50, 2, Article 29 (may 2017), 32 pages. https://doi.org/10.1145/3057269

[15] Ajay Kumar. 2023. Formalization of structural test cases coverage criteria for quantum software testing. *International Journal of Theoretical Physics* 62, 3 (2023), 49.

[16] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-based runtime assertions for testing and debugging Quantum programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 150 (nov 2020), 29 pages. https://doi.org/10.1145/3428218

[17] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476

[18] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[19] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing.* John Wiley & Sons.

[20] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2008. Test-Suite Augmentation for Evolving Software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* 218–227. https://doi.org/10.1109/ASE.2008.32

[21] Muhammad Shahid and Suhaimi Ibrahim. 2011. An evaluation of test coverage tools in software testing. In *2011 International Conference on Telecommunication Technology and Applications Proc. of CSIT*, Vol. 5. sn.

[22] Muhammad Shahid, Suhaimi Ibrahim, and Mohd Naz'ri Mahrin. 2011. A study on test coverage in software testing. *Advanced Informatics School (AIS), Universiti Teknologi Malaysia, International Campus, Jalan Semarak, Kuala Lumpur, Malaysia* 1 (2011).

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

[24] Praveen Ranjan Srivastava. 2008. Test case prioritization. *Journal of Theoretical & Applied Information Technology* 4, 3 (2008).

[25] Jiyuan Wang, Ming Gao, Yu Jiang, Jianguang Lou, Yue Gao, Dongmei Zhang, and Jiaguang Sun. 2018. QuanFuzz: Fuzz Testing of Quantum Program. arXiv:1810.10310 [cs.SE] https://arxiv.org/abs/1810.10310

[26] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Quito: a Coverage-Guided Test Generator for Quantum Programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1237–1241. https://doi.org/10.1109/ASE51524.2021.9678798

[27] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. QuSBT: search-based testing of quantum programs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22).* Association for Computing Machinery, New York, NY, USA, 173–177. https://doi.org/10.1145/3510454.3516839

[28] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2023. QuCAT: A Combinatorial Testing Tool for Quantum Software. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2066–2069. https://doi.org/10.1109/ASE56229.2023.00062

[29] Jiaming Ye, Shangzhou Xia, Fuyuan Zhang, Paolo Arcaini, Lei Ma, Jianjun Zhao, and Fuyuki Ishikawa. 2023. QuraTest: Integrating Quantum Specific Features in Quantum Program Testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1149–1161. https://doi.org/10.1109/ASE56229.2023.00196

[30] Jianjun Zhao. 2021. Quantum Software Engineering: Landscapes and Horizons. arXiv:2007.07047 [cs.SE] https://arxiv.org/abs/2007.07047

[31] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 4 (dec 1997), 366–427. https://doi.org/10.1145/267580.267590

Received 2024-04-25; accepted 2024-05-06