MorphQ++: A Reproducibility Study of Metamorphic Testing on Quantum Compilers

Linsey Kitt
Iowa State University
Ames, IA, USA
ljkitt@iastate.edu

Myra B. Cohen
Iowa State University
Ames, IA, USA
mcohen@iastate.edu

ABSTRACT

Quantum computing has been rapidly expanding, and many platforms for writing programs that can be compiled and run on quantum hardware (or simulated) are being developed. As with any compiler, transformation correctness is paramount as the machine readable code must maintain program semantics. Quantum programs are challenging to test given a lack of benchmark programs and the difficulty of defining an oracle. MorphQ solves both of these challenges by (1) generating syntactically correct quantum programs and (2) using metamorphic testing to avoid the oracle problem. However, Qiskit, the compiler it was built for, is rapidly evolving. This paper is a reproducibility study of the MorphQ platform, which we call MorphQ++. We have updated the core MorphQ engine to work on a newer version of Qiskit and added new metamorphic transformations. We find that our overall results are similar: we find a portion of the original faults (which were not yet fixed) and the distributions of types are not very different. Our new transformations lead to several new faults, suggesting there is room to expand the core framework. Additionally, we note the lack of power of the metamorphic relations in this context and suggest the need for more sophisticated relations and/or oracle evaluations.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Hardware** → **Quantum technologies**.

KEYWORDS

metamorphic testing, quantum, reproducibility

ACM Reference Format:
Linsey Kitt and Myra B. Cohen. 2024. MorphQ++: A Reproducibility Study of Metamorphic Testing on Quantum Compilers. In *Replications and Negative Results (RENE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3695750.3695823

1 INTRODUCTION

Quantum computing is a rapidly developing field with growing needs across many domains, including software engineering. Quantum software compilers such as IBM's Qiskit [1] or Google's Cirq [2] are fundamental to quantum computing, creating a space to write (and simulate) quantum programs which can be compiled and run on quantum computers. As such, they rely on strong software testing approaches to ensure their correctness [8, 10, 13, 16, 21].

One challenge is that quantum computing platforms may have nontraditional bug patterns which are difficult to identify due to the added complexities of quantum mechanics [12, 17, 21, 22]. These quantum computing platforms also must handle quantum programs which often do not have a known exact output due to the test oracle problem [3]. This problem is compounded by the relatively young age of the field, limiting the number of programs available to use as quality test inputs. Having a lack of test inputs can be solved by *program generators* such as the well known C-compiler testing tool which generates random C programs, Csmith[20]. To solve the oracle problem, metamorphic testing can be used (running sets of tests and comparing the relationships of their outputs) to determine program correctness [3, 4, 11, 14, 15]. Recent research by Paltenghi et al. on testing for quantum computing platforms has combined these ideas into a framework (and tool) called MorphQ [13].

MorphQ tests the IBM Qiskit quantum computing platform. It generates syntactically correct quantum programs and then performs metamorphic transformations to use as oracles for faults. MorphQ demonstrated its ability to tackle these problems with these methods, identifying 13 faults in 8k program pairs generated and tested in less than 48 hours. But with the rapid development seen in quantum computing platforms, this is only a first step: evidence of continuing effectiveness as systems evolve and the ability to upgrade and extend the platform is vital to make this a useful testing method.

To understand the needs of tools such as MorphQ for use in regression testing for developing quantum computing platforms, we designed a reproducibility study using MorphQ. The first part of the study was to upgrade MorphQ from the version of Qiskit it was written for, version 0.33.1, to the latest version at the time of this experimentation, 0.44.1. We then designed and added six new metamorphic transformations and extended two of the existing transformations included in MorphQ. We refer to this updated and extended MorphQ as MorphQ++. Our approach to creating MorphQ++ and the challenges faced are covered in the following sections. We then tested it against the original by running both the original MorphQ and MorphQ++ three times for 48 hours each.

The contributions of this work are

- A reproducibility study on MorphQ using an updated version of Qiskit;
- The addition of new metamorphic relations which find new bugs;
- Insights into future directions for the MorphQ framework

RENE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1270-8/24/10
https://doi.org/10.1145/3695750.3695823

2 REPRODUCIBILITY STUDY

In this section, we present an evaluation of the Qiskit upgrade and our new transformations. We first describe how we upgraded MorphQ, then discuss our new and extended metamorphic relations.[1]

2.1 Updating Qiskit

Our first focus was updating Qiskit and ensuring MorphQ could still be run on the newer version. Unfortunately, this was not as simple as just updating versions. Updating Qiskit caused several issues with backward compatibility, requiring modifications to several metamorphic transformations and even the base program generation, as fundamental pieces such as circuit gates and their inputs were changed. In addition, the update required an upgrade of the Python version and the update of many Python modules, several of which resulted in hidden faults in MorphQ. For example, Python's AST module had an update that removed a class being used to store values in the tree, resulting in a modification to the tree's structure. Some of MorphQ's transformations use this AST module to parse through the Qiskit program to apply the transformation, requiring careful debugging to resolve the new tree structure. These types of issues were not unexpected but were still a notable part of the process of our project.

After making these fixes, we performed a full run of MorphQ following the original paper for 48 hours as a validation step. While analyzing the results, we encountered several faults that had been found before the update but with their exception messages modified. This required overhead to track down exceptions that initially looked new but were actually known faults with more informative messaging. In the end, we found **one new fault** that we confirmed with developers.

2.2 New Transformations

With Qiskit updated, we started extending MorphQ by adding new transformations. We looked at Qiskit bug reports to identify the components causing a user's program to fail, following regression testing tenet which uses the fact that components with bugs are more likely to have more bugs during evolution [7, 9]. The other way we identified transformations was by focusing on the comments for workarounds. Qiskit developers often leave suggestions on workarounds until the bug gets fixed. We briefly describe each next.

(1) Added Three Basis Sets A basis or universal gate set is a set of gates that can represent any gate. When simulated, a circuit being transpiled with a specified basis gate set should have the same output as the original circuit. We added three new gate sets.

(2) QASM3 Open Quantum Assembly Language, or OpenQASM, is a method of representing quantum circuits in an intermediate representation [6]. Since 2017, two versions have been released, OpenQASM2 and OpenQASM3. Qiskit allows for conversion of quantum circuits to and from OpenQASM. Converted circuits should maintain their same outputs. We added the OpenQASM3 since MorphQ only translates to OpenQASM2.

(3) Change Transpilation Backend The transpilation process prepares quantum circuits for specific quantum computers optimizing for hardware. MorphQ only transpiles for the AerSimulator provided by Qiskit; however, other simulators are also available on Qiskit. This transformation transpiles for other simulators. The transpiled circuit is still tested on the AerSimulator, otherwise the outputs would be different and more difficult to verify

(4) To QPY and Back In addition to conversion to and from OpenQASM, Qiskit also has the ability to convert to and from QPY. QPY is a serialized format for storing and transferring quantum circuits from Qiskit. It is designed to be backwards compatible for all future versions of Qiskit. We added a new transformation to support this.

(5) Copy Circuit Another feature of Qiskit is a method that operates on a circuit by copying it and returning a new identical circuit. This transformation copies the circuit from the Qiskit program and verifies the copied version still runs as the original.

(6) Initialize Circuit Circuit initialization in Qiskit allows a circuit of qubits to be initialized with any legal value. Qiskit also has a separate function that allows for the generation of random circuits, which will randomly assign gates to the circuit. This transformation first turns the circuit in the Qiskit program into a random circuit, then re-initializes it to a circuit with all qubits set to $|0\rangle$. This is an equality relation.

(7) Reset Circuit Qiskit circuits have a function to reset all of the qubits to their default state, which should have the same effect as initializing all the qubits to $|0\rangle$. This transformation applies the circuit as usual, then resets all the circuits, and finally applies the circuit a second time before proceeding, verifying the reset circuit is the same.

(8) Add Barrier Qiskit allows for the addition of barriers to circuits, which divide operations on the circuit. This addition is both a visual change to the circuit diagrams, where diagrams have a barrier drawn in, and an operation that impacts the transpilation process, as the transpiler will consider each side of the barrier separately when compiling and optimizing: it will not optimize across the barrier. These barriers should not impact the output of the circuits, only the way they are set up and optimized.

3 RESULTS

We ran the original MorphQ for 48 hours, mimicking the original paper's 48 hour run, and MorphQ++ for 48 hours. We repeated this three times to ensure the results we gathered were not influenced by outliers in the random program generation. We then also ran MorphQ++ 10 times for two hours each to see how it navigated the search space given less time (we don't report those results). All experiments were performed on Red Hat Enterprise Linux 9.2 with Intel Xeon Gold 6144 CPUs with 32 cores and 366GB of RAM. We utilized the Jupyter notebooks provided by MorphQ developers [13] for our analyses and graphs in this paper.

Table 1 summarizes the faults found by MorphQ++, their crash messages, and the metamorphic transformations leading to them. The newly added metamorphic transformations involved in the crashes are underlined and bold. In total, 13 faults were found, five of which were new faults not previously reported to Qiskit. The table lists these as IDs 6, 7, 10, 11, and 12. These have all been confirmed as faults by the developers except for 12, which has not received a response at the time of writing. Notably, only one of these faults did not involve any new relations, ID 10, implying it was a fault introduced in the update from Qiskit 0.33.1 to 0.44.1. We confirmed this by running the test on 0.33.1, and it passed.

[1] Artifacts can be found at https://github.com/LavaOps/RENE24.

Table 1: Fault crash messages found and reported from running MorphQ++. New transformations are underlined and in bold

ID	Report	Status	Novelty	Crash message	Metamorphic transformations
1	#7700	confirmed	new in original MorphQ	too many subscripts in einsum (numpy)	Change of optimization level, Inject null-effect operations
2	#7641	confirmed	duplicate found in original MorphQ	Instruction id not found	Change of gate set
3	#7326	fixed	duplicate found in original MorphQ	Mismatch between parameter_binds	Inject parameters
4	#7748	fixed	new in original MorphQ	Cannot bind parameters not present in the circuit	Inject parameters
5	#10534	confirmed	duplicate found MorphQ++	Invalid param type <class \'complex\'> for gate ry	**Initialize circuit**, Change of gate set
6	#11560	confirmed	new in MorphQ++	unknown instruction: rxx_139974866104288	**Roundtrip conversion via QASM3**, Change of gate set, **Change of transpilation backend**
7	#11158	fixed	new in MorphQ++	Sum of amplitudes-squared is not 1, but 0.0	**Initialize circuit, Roundtrip conversion via QPY**
8	#8050	confirmed	duplicate found in MorphQ++	Cannot apply Operation: reset	**Initialize circuit**, Change of optimization level
9	#10345	confirmed	duplicate found in MorphQ++	eig algorithm (geev) did not converge	**Change of transpilation backend**
10	#10990	confirmed	new in MorphQ++	list index out of range	Roundtrip conversion via QASM2, Change of optimization level
11	#11558	confirmed	new in MorphQ++	only length-1 arrays can be converted to Python scalars	**Roundtrip conversion via QASM3**
12	#11559	reported	found in MorphQ++	KeyError: 0	**Change of transpilation backend** OR Change of coupling map, **Roundtrip conversion via QASM3**, **Roundtrip conversion via QPY**
13	#9746	confirmed	duplicate found in MorphQ++	unpack requires a buffer of bytes	**Roundtrip conversion via QPY**, **Initialize circuit**, Inject null-effect operations

In addition, four duplicate faults were found in MorphQ++, IDs 5, 8, 9, and 13. They are listed with their original bug reports made by other Qiskit users. Two of these, 9 and 13, were flaky tests [5, 18], i.e., they could not be replicated reliably; in 9, running the same program almost always succeeds – it is only in rare instances that it crashes. 13 was not able to be reproduced; among the one quarter million program pairs tested in this paper, this crash was only found once. We were unable to generate the same crash message when rerunning the program, instead getting the crash message from ID 7, "Sum of amplitudes-squared is not 1, but 0.0." Given this, we are also unable to verify which of the three metamorphic transformations of *Roundtrip conversion via QPY*, *Initialize circuit*, and *Inject null-effect operations* are truly required to generate the fault; it may have been caused by all three or a subset of these three transformations.

The other four remaining faults were found in the original MorphQ, signaling they were not fixed by the update of Qiskit. They are the first four faults listed in the table. Two were new faults found in the original MorphQ, while the other two were duplicates found originally by other users. The original paper found 13 faults, indicating the other nine not still found in the update were fixed.

Table 2 shows the number of faults each new transformation contributed towards finding. As can be seen, several transformations were productive towards finding faults, and each of these contributed towards finding at least three. In contrast, the remaining transformations did not find any. Through this research question, we conclude that MorphQ++ is successful at identifying new faults, especially through four of the added and modified metamorphic transformations. Some of these faults come from flaky tests, however, making them difficult to find and classify.

We next considered how our results compare against the original and MorphQ++ by running for 48 hours, three times each. We created several sets of figures and tables to analyze the data using Jupyter Notebooks provided alongside the original MorphQ code.

Table 2: Number of faults by new transformations

New metamorphic transformation	Unique bugs found
Roundtrip conversion via QASM3	3
Change of transpilation backend	3
Roundtrip conversion via QPY	3
Copy circuit	0
Initialize Circuit	4
Reset Circuit	0
Add Barrier	0

Table 3: Execution data on three original MorphQ runs

(a) First original run

	Number	Percentage
Tested program pairs	19,429	100.0%
↪ Crashes in source program	0	0.0 %
↪ Crashes in follow-up program	6,488	33.4 %
↪ Successful executions	12,941	66.6 %
↪ Distribution differences	169	0.9 %

(b) Second original run

	Number	Percentage
Tested program pairs	12,065	100.0%
↪ Crashes in source program	0	0.0 %
↪ Crashes in follow-up program	2,386	19.8 %
↪ Successful executions	9,679	80.2 %
↪ Distribution differences	163	1.4 %

(c) Third original run

	Number	Percentage
Tested program pairs	23,725	100.0%
↪ Crashes in source program	0	0.0 %
↪ Crashes in follow-up program	6,874	29.0 %
↪ Successful executions	16,851	71.0 %
↪ Distribution differences	305	1.3 %

3.1 Execution Data

Data from running the original MorphQ can be found in Table 3 and MorphQ++ in Table 4. The most obvious observation here is the difference in the volume of program pairs tested: MorphQ++ generated nearly four times as many program pairs as the original. There are many possible reasons for this difference; improvements to Qiskit's algorithms are likely a major cause. An obvious instance is the introduction of "Layer contains > 2-qubit gates" crash messages, which occur in the updated version but not the older version of Qiskit. These seem to occur when, in the older version, MorphQ would time out for taking too long, suggesting computations that were too complicated to be feasible. Now, the program quickly crashes rather timing out. There are likely many other optimizations made across Qiskit updates. There could similarly be speed increases from the updated Python version including optimizations,

Table 4: Execution data on three MorphQ++ runs

(a) First updated run

	Number	Percentage
Tested program pairs	79,336	100.0%
↪ Crashes in source program	0	0.0 %
↪ Crashes in follow-up program	12,118	15.3 %
↪ Successful executions	67,218	84.7 %
↪ Distribution differences	941	1.2 %

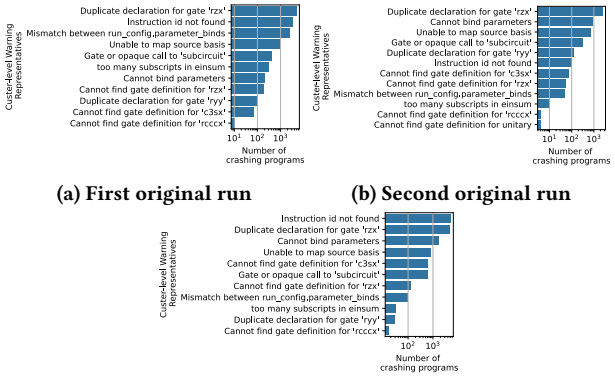
(b) Second updated run

	Number	Percentage
Tested program pairs	76,923	100.0%
↪ Crashes in source program	0	0.0 %
↪ Crashes in follow-up program	8,465	11.0 %
↪ Successful executions	68,458	89.0 %
↪ Distribution differences	872	1.1 %

(c) Third updated run

	Number	Percentage
Tested program pairs	64,097	100.0%
↪ Crashes in source program	0	0.0 %
↪ Crashes in follow-up program	6,934	10.8 %
↪ Successful executions	57,163	89.2 %
↪ Distribution differences	1,136	1.8 %

allowing more program pairs to be generated. One last possible contributor is the new metamorphic transformations: they may be faster to run on average than the original transformations, allowing the generated programs to run faster.

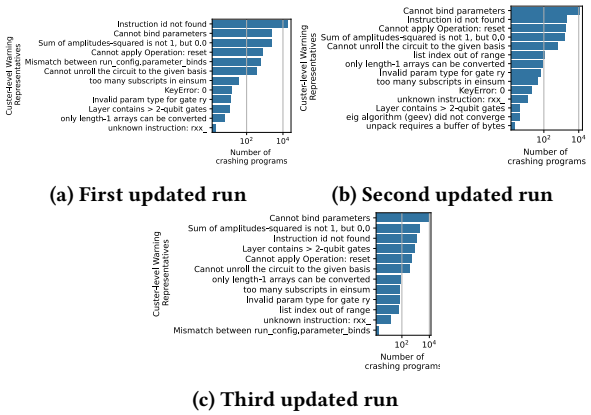


(a) First original run **(b) Second original run**

(c) Third original run

Figure 1: Frequency of crash clusters in original MorphQ

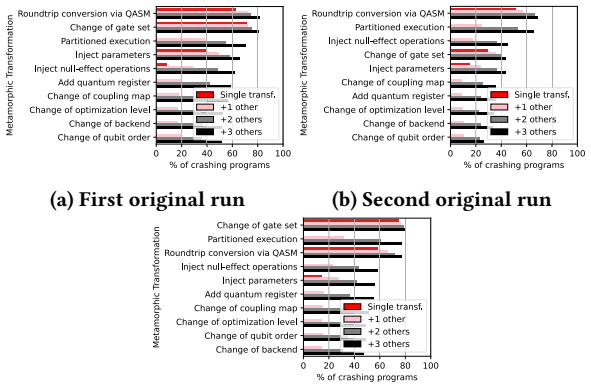
The next observation is the difference in percentage of crashing programs. Metamorphic transformations led to crashing programs 15% more often in the original MorphQ. In Table 1, nearly all new faults found in MorphQ++ required at least two specific transformations, with three requiring three different transformations. MorphQ



(a) First updated run (b) Second updated run

(c) Third updated run

Figure 2: Frequency of crash clusters in MorphQ++



(a) First original run (b) Second original run

(c) Third original run

Figure 3: Percent of crashing programs in original MorphQ broken down by transformation: the percentage of programs that crashed when the transformation was the only one applied in red, when the transformation was applied with one other in pink, two others in grey, and three or more, black

only applies up to four transformations at a time and will often apply fewer than four. From the original paper, faults found in the original MorphQ only require two or fewer transformations. This means faults will be found less often in MorphQ++ because they are less likely to have their specific transformations applied to the same program, and there will, therefore, be fewer crashes. These may, however, be more complex faults.

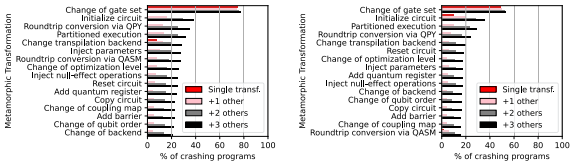
The distribution differences in the tables indicate the number of program pairs where the two programs had statistically significant differences in their outputs even though they were expected to closely match. As noted in the original paper, the statistical test used to determine these differences uses a 5% p-value, so up to 5% of programs are expected to have distribution differences even if all the programs are coming from the same distribution. None of the distribution differences across the runs go above 2%.

3.2 Crash Clusters

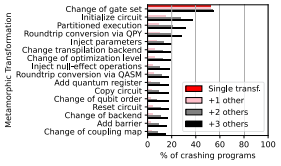
Figures 1 and 2 show the number of times each crash occurred in each run in the original and updated MorphQ, respectively. These graphs have many similarities, as many crashes occur with similar frequencies across runs in both the original and MorphQ++. There are a few notable differences, however. First, in both the original and MorphQ++, there are a few crashes that occur with wildly different frequencies. Looking specifically at the original, "Instruction id not found" occurred an order of magnitude more in the run in Figure 1a compared to the run in Figure 1b, meanwhile "Cannot find gate definition for unitary" never occurred at all in Figure 1a. In MorphQ++, "Mismatch between run_config.parameter_binds" occurred two orders of magnitude more in Figure 2a compared to Figure 2c and did not occur at all in Figure 2b. "KeyError: 0" also never occurred in Figure 2c and "eig algorithm (geev) did not converge" and "unpack requires a buffer of bytes" only occurred in Figure 2b. These inconsistencies may be explained by the randomness of program generation and transformation application and show the importance of using multiple runs.

3.3 Crashing Programs by Transformation

Figures 3 and 4 show the percentage of crashing programs when a transformation is applied by itself, with one other transformation, two others, or three or more. In both the original and MorphQ++, the main inconsistencies across runs come from the single transformation data. In the original, the *Inject null-effect operations* transformation only causes crashes on its own (Figure 3a). In MorphQ++, the *Change transpilation backend* transformation only finds crashes by itself (Figure 4a), while the *Initialize circuit* and *Roundtrip conversion via QASM* transformations only find crashes by themselves in Figure 4b. The latter found very few crashes in MorphQ++, which is in contrast to the original, where it consistently caused many crashes on its own, showing possible improvements to the updates to Qiskit – it now requires other transformations.



(a) First updated run (b) Second updated run

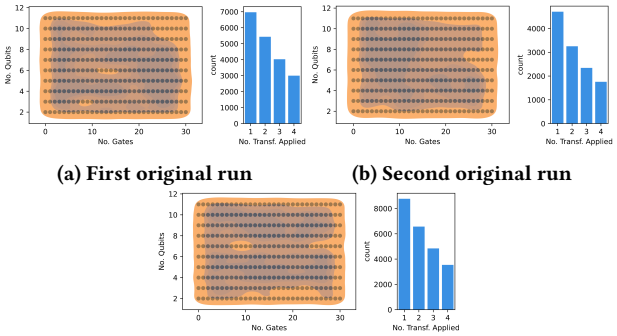


(c) Third updated run

Figure 4: Percentage of crashing programs in MorphQ++

3.4 Program Generation Distributions

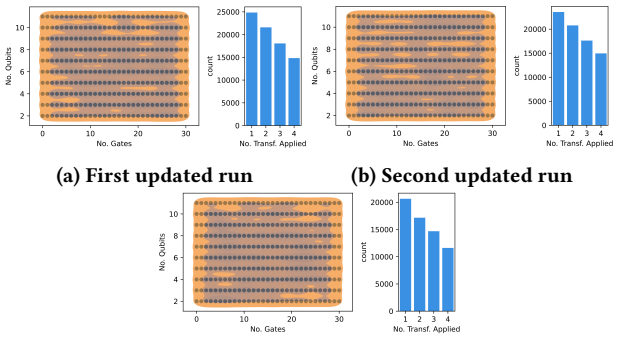
Figures 5 and 6 show that MorphQ++ has more uniform distributions across runs compared to the original. There is a smaller



(a) First original run (b) Second original run

(c) Third original run

Figure 5: Distribution of programs in original MorphQ: the right figure shows concentration of programs across the number of gates and qubits, while left shows the number of programs with a given number of transformations applied



(a) First updated run (b) Second updated run

(c) Third updated run

Figure 6: Distribution of programs generated by MorphQ++

relative difference in the number of programs with one transformation applied versus two in MorphQ++, and the number of gates versus the number of qubits is much more consistent. The generation of Qiskit programs involves randomly generating a number of gates and qubits to use within a configurable size, so the differences found here are likely due to the increase in sample size: MorphQ++ generates significantly more programs.

When looking at the results from the original MorphQ and MorphQ++, we found faults were coming from more complex interactions of Qiskit components in MorphQ++, and there were more programs generated but with fewer crashes. Neither version showed any distribution differences to be the result of faults, and both experienced many inconsistencies across their individual runs.

3.5 Discussion

A key advantage of metamorphic testing is the ability to make conclusions about the correctness of the output without having a test oracle by comparing related outputs. Although MorphQ (and MorphQ++) employs metamorphic testing, they did not find faults via this comparison. Rather, they found faults from unexpected crashes on valid programs. These faults, rather than being found through the added intelligence embedded in metamorphic testing, were essentially found via mutation-based fuzzing, another type of testing method where random inputs to a program are generated and mutated and run on the program to find faults [23], with a limited set of transformations or mutations. In recent work, Xia et al. [19] used fuzzing to test MorphQ, suggesting this is a viable option. Though metamorphic testing was not beneficial here, this is not to say it is not worth pursuing in quantum software or that it cannot find faults in any capacity. Rather, it indicates two directions future research can look to. One is to look towards different metamorphic strategies and build more robust metamorphic testing platforms to harness the strengths of metamorphic testing more effectively. A second research direction proposed is to improve the strategy for testing. Given the increased effort required to create metamorphic transformations and implement their equivalencies, it may be more efficient for these crashing faults, especially in the early phases of developing quantum platforms, to focus on the mutation-based fuzzing foundation of MorphQ.

Aside from considering metamorphic testing versus fuzzing strengths, there are still several challenges to be faced by these testing platforms. These platforms are undergoing a lot of development and therefore experience many updates, requiring a lot of work to keep the testing tools up to date. As discussed, upgrading MorphQ had several complications, requiring modifications to several transformations and elsewhere in the code. It is expected to face these types of challenges when working with developing platforms, and it emphasizes the importance of good code design to simplify the process of keeping up to date.

The last point of discussion here is looking at the inconsistencies across runs. As seen across the research questions, individual runs are unreliable at finding all faults and performing a uniform search across the search space. We may want to use a better strategy for shorter, multiple runs rather than one long run to ensure a more uniform exploration of the search space. We had some success with this strategy in a test run.

4 CONCLUSIONS AND FUTURE WORK

We have performed a reproducibility study of the MorphQ testing platform for quantum compilers. We updated MorphQ to work on a newer version of Qiskit and added new transformations, which we call MorphQ++. We found a similar set of faults with relatively similar distributions of faults; however, we also found new faults, primarily with our new transformations. One interesting observation is that the metamorphic relations themselves were not used to find real behavioral differences (e.g. subtle changes found between outcomes). Instead all of the bugs found were crashing type faults, meaning the metamorphic relations only provided manipulation of the data, not a change in the oracle. This indicates a fuzzing type approach of testing could also work.

ACKNOWLEDGMENTS

We would like to thank Jack Lutz for fruitful discussions and feedback on this work. We thank the authors of MorphQ for their valuable comments on the camera ready version. This work was supported in part by NSF grant #1909688.

REFERENCES

[1] 2023. Qiskit/Qiskit. https://github.com/Qiskit/qiskit. https://github.com/Qiskit

[2] 2024. quantumlib/Cirq. https://github.com/Qiskit/qiskit. https://github.com/quantumlib/Cirq

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[4] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (jan 2018), 27 pages. https://doi.org/10.1145/3143561

[5] Maxime Cordy, Renaud Rwemalika, Adriano Franci, Mike Papadakis, and Mark Harman. 2022. FlakiMe: laboratory-controlled test flakiness impact assessment. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 982–994. https://doi.org/10.1145/3510003.3510194

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

[7] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Portland, Oregon, USA) *(ISSTA '00)*. Association for Computing Machinery, New York, NY, USA, 102–112. https://doi.org/10.1145/347324.348910

[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] Jung-Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 119–129. https://doi.org/10.1145/581339.581357

[10] Peixun Long and Jianjun Zhao. 2024. Testing Multi-Subroutine Quantum Programs: From Unit Testing to Integration Testing. *ACM Trans. Softw. Eng. Methodol.* (apr 2024). https://doi.org/10.1145/3656339 Just Accepted.

[11] Christian Murphy, Gail Kaiser, Lifeng Hu, and Leon Wu. 2008. Properties of Machine Learning Applications for Use in Metamorphic Testing. 867–872.

[12] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 1–27. https://doi.org/10.1145/3527330

[13] Matteo Paltenghi and Michael Pradel. 2023. MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. https://doi.org/10.1109/icse48619.2023.00202

[14] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[15] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive Metamorphic Testing of Debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3293882.3330567

[16] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 692–704. https://doi.org/10.1109/ASE51524.2021.9678792

[17] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. Quito: a coverage-guided test generator for quantum programs. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 1237–1241. https://doi.org/10.1109/ASE51524.2021.9678798

[18] Chunqiu Steven Xia, Saikat Dutta, Sasa Misailovic, Darko Marinov, and Lingming Zhang. 2023. Balancing Effectiveness and Flakiness of Non-Deterministic Machine Learning Tests. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1801–1813. https://doi.org/10.1109/ICSE48619.2023.00154

[19] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Article 126. https://doi.org/10.1145/3597503.3639121

[20] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[21] 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

[22] Pengzhan Zhao, Jianjun Zhao, and Lei Ma. 2021. Identifying Bug Patterns in Quantum Programs. arXiv:2103.09069 [cs.SE]

[23] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (sep 2022), 36 pages. https://doi.org/10.1145/3512345