Capstone Project - PIIV22
New Talents in Quantum Technologies

Asserting the correctness of Quantum Programs using Metamorphic Testing

Nuno Costa[1]

[1]Faculty of Engineering of the University of Porto

July 4, 2022

Abstract

An increasing sense of anticipation and enthusiasm surrounds quantum computing. However, several obstacles must be overtaken before quantum computing can be widely adopted. One of which is determining whether or not a quantum program is correct. There are several tried-and-true methods for identifying program flaws in the traditional world, but it is far from simple to adapt them to the quantum realm. In this project, we explore why Metamorphic Testing might be a good starting point in this transition and test a simple example on the well-known Shor's Algorithm, verifying the validity of this testing approach on impactful Quantum Algorithms.

1 Introduction

The world of computation is about to undergo a significant upheaval because of Quantum Computing (QC).

Honeywell [2] reported in July 2021 that they built the highest-ever quantum volume [1] quantum computer, with a volume of 1024. Alongside this, a computer intensive task that would take up to 8 years to complete concluded in just about 1.2 hours [1]. These and other significant accomplishments imply that QC may become widely used and have an industrial influence considerably sooner than anticipated.

Even though these technologies are on their way to becoming prominent, validation and verification (V&V) will be extremely difficult in quantum computers, according to researchers. For example, it is not possible to use conventional testing and debugging techniques, such as those that rely on break-point based monitoring, due to how quantum measurements work. In the same line of thought, quantum states are generally high-dimensional and difficult to interpret, limiting their usefulness for programmers to debug misbehaving quantum programs. Alongside this, the programs quantum computing seeks to solve will be, by nature, more complex.

As such, we seek to explore the challenges quantum testing imposes further. To do so, we intend to validate the already proposed approach to testing quantum programs by [3], which is based on the use of metamorphic relations. Learning the fundamentals of quantum computing and metamorphic testing is essential for this, as is correctly investigating the algorithms that can demonstrate the viability of such an approach to quantum testing.

The remainder of this report is organised as follows.

- In section 2, we introduce the fundamentals of Quantum Computing.
- In section 3, we introduce the Quantum Correctness Problem and the difficulties it brings to Quantum Testing
- In section 4, we introduce Metamorphic Testing and how it has been used and applied in Classical Computing

[1]Quantum volume is metric designed to indicate the fidelity of a quantum system.

- In section 5, we introduce Quantum Metamorphic Testing and an example of a methodology for it
- In section 6, we introduce the algorithms we studied with the perspective of testing them in a quantum setting
- In section 7, we introduce the Metamorphic Relations we defined for Shor's algorithm
- In section 8, we analyse the implementation of the used testing mechanism for Quantum Metamorphic Testing
- In section 9, we showcase the obtained results for the presented implementation

2 Quantum Computing

Standard computing has the bit as its smallest unit of information. That bit may hold one of two values: 0 or 1. Such representation is used in most electronic devices and is scalable for the vast majority of tasks.

The classical representation, however, limits the computation potential of some solutions, especially when they involve multiple variables [8]. In those cases, each variable needs to be stored and updated separately from the others - which intuitively would be the best case scenario.

Quantum Computing challenges this intuition. To do so, it uses a quantum mechanics concept: superposition.

Quantum Superposition *is the counter-intuitive capacity quantum objects - such as electrons - have that allows them to exist in more than one "state" simultaneously* [8]. Quantum Computing applies this concept to its smallest unit of information: the quantum bit or *qubit*.

The *qubit* is a two-state quantum-mechanical system. Instead of holding one of two values, like the bit does, it holds the probabilities of being each of the two values. This means that it may hold the two values at the same time, when there's a possibility of outputting either one of them with a given probability. In that case, the state $|0\rangle$ and state $|1\rangle$ are, in that case, *entangled*. [2]

One obtains the value of a qubit through measurement. This measurement, just like its classical counterpart, will output either 0 or 1. However, measuring the qubit interferes and irreversibly changes the state, which is a restriction and may cause some difficulties in testing - which will be explained further below.

With all of this in mind, the applications of Quantum Mechanics phenomena to computing can revolutionize how we think about the state of a program. This revolution may turn into reality computations that were, up until recently, thought to be impracticable.

3 Quantum Correctness Problem

As Quantum Computing and Quantum Software Development advance and evolves, difficult problems to tackle arise.

Firstly, the programs that QC seeks to solve will be, by nature, more complex. Defining test oracles [3] for these programs will be increasingly difficult, especially when concerning *clear-box testing*. This is known as the test oracle problem, which defines the *difficulty of distinguishing the corresponding desired, correct behavior from potentially incorrect behavior* in a program. It is already a relevant problem in Classical Computing and will be as relevant - if not more - in Quantum Computing.

Alongside the test oracle problem, *measurements destroy the quantum state*. This means that any measurement mid-computation will interfere with the result. The example below showcases how measurement can make a purely deterministic program enter a probabilistic state.

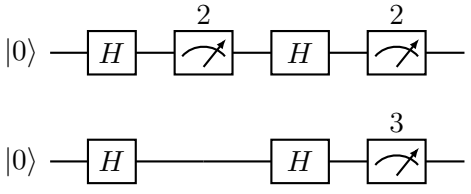


Figure 1: Example of a deterministic Quantum Circuit

[2] Entanglement refers to the quantum phenomenon that happens when two entangled particles, such as a pair of photons or electrons, remain connected even when separated by vasts distances.

[3] Mechanism for determining whether a test has passed or failed.

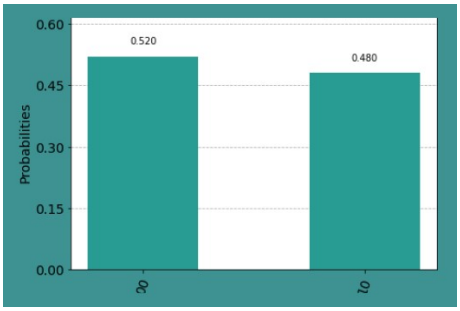


Figure 2: Simulation Output

The simple circuit shown previously has 2 qubits, each of which has 2 Hadamard (H) gates applied, one after another. Applying a Hadamard gate converts a computational basis state into an equal superposition state - in simple terms, there's a 50% chance of it being a 0 and a 50% chance of it being a 1 when measured. Since the Hadamard gate is a reversible quantum operation (can be represented by a unitary matrix), applying the Hadamard two times in a row should output the input value. This would be the same as applying no gates at all to the input.

In the first qubit, however, measurement is executed in-between the Hadamard gates. As the results in Figure 2 show, the output of the first qubit is compromised - it outputs 0 nearly 50% of the time for an input of 0, indicating it is in an equal superposition state. In fact, after measuring a superposition state, that same state falls to one of the basis states that form the superposition - either 0 or 1, with equal probability - thus destroying the original configuration.

The destruction of the state after measurement, which appears to be a seemingly simple problem, invalidates many V&V mechanisms. Those mechanisms include interactive debugging and mechanisms that rely on keeping track of the value of variables at run-time, for example.

Given the difficulties Quantum Computing poses to testing, it becomes clear that Quantum Programs will need different methods to assert correctness, as Classical Testing mechanisms will either be too complex to translate to this new paradigm or will not be practicable at all.

4 Metamorphic Testing in Classical Computing

Metamorphic Testing (MT) is a property-based testing technique. It approaches the software testing problem from a different perspective: it focuses on rules that inputs should respect about their outputs, instead of focusing on the value of the corresponding output of any given input. These same properties, known as metamorphic relations, should hold and are necessary for the program to be working properly and to do what's intended. In short, by asserting metamorphic rules, which can be a set of boolean functions relating the inputs and outputs of a function (telling whether it is satisfied or not), we can assert if a program is correct or faulty.

A great example that showcases how powerful metamorphic relations can be is present in search engines, such as *Google*. Searching for 'quantum', in *Google* outputs around 530.000.000 results. If, however, the search that we are looking for is more restrictive, i.e 'quantum computing', the new number of results should always be less than the number of results of their components – which should be, in principle, more broad in meaning. In fact, *Google* holds the relation for this test, as 'quantum computing' outputs around 170.000.000 results. As such, successfully passing tests of the presented metamorphic rule assures that the property *restricting a search query* is working as intended in a search engine.

The advantage of this kind of technique is clear. No prior knowledge of the output for a given input is needed, which is extremely useful when that same output is either costly or impractical to obtain, and as a consequence, the program depends uniquely on its properties. As such, it is a useful tool to alleviate the test oracle problem.

On the other hand, it may not detect faulty algorithms if not enough metamorphic rules are defined, which by itself is hard to avoid since it may be difficult to assert relevant metamorphic rules.

Nevertheless, MT is already thoroughly used in Classical Computing, being used and tested in several fields [4], ranging from compilers and numerical analysis to health and medical systems, as well as image processing and machine learning. Given the way it functions and the complexity of the problems it seeks to assess correctness on, its usage in Quantum Computing may be promising.

5 Quantum Metamorphic Testing

Having in mind the difficulties of Quantum Software Testing, it is easy to see why it would make sense to see Metamorphic Testing as a plausible solution.

In essence, it avoids the test oracle problem, as there is no need to define what the precise outputs are for a given input. Alongside that, it also avoids the quantum measurement problem, as the defined test oracles don't need to interfere with the quantum state mid-computation.

As such, Quantum Metamorphic Testing defines itself as asserting metamorphic rules for quantum programs to assert correctness. Although it makes sense conceptually, there are little to no references for frameworks of MT for Quantum Software, which may be deeply correlated with the fact that Quantum Testing itself is yet to be deeply explored and studied.

5.1 Testing Mechanism

Conceptually, a metamorphic testing mechanism in a quantum setting would respect the following phases:

1. Define the metamorphic rules of the algorithm at hand
2. Construct a circuit which verifies such rules
3. Apply such circuit and check the correctness of the algorithm

Given that some implementations are already proven as correct, one may add the following steps:

1. Introduce errors into the implementation
2. See if the chosen metamorphic rules fail and detect the errors in the faulty implementation

In [3], we can already find an example of this purely quantum approach to metamorphic testing, highlighting its validity. Aside from defining the Metamorphic Rule for a simple modular adder operator, they were also able to implement the rule in a purely quantum circuit, verifying the correctness of the operator. Alongside this, they added mutations to the code and, with the aid of Grover's Algorithm, it was possible to easily find failing tests.

However, such implementations are not simple, especially for harder problems, and given the introductory level of this paper - and the investigative initiation nature of it - a hybrid approach is explored.

Using the hybrid approach, instead of constructing a circuit that verifies the validity of the metamorphic rule, that same verification can be made through high-level code using Python, for example.

6 Explored Algorithms

In this paper, we seek to showcase the validity of the MT approach to Quantum Testing by defining metamorphic rules in a well-known quantum algorithm. The following two algorithms were thoroughly studied.

6.1 Grover's Algorithm

Grover's algorithm is a quantum search algorithm that runs quadratically faster than any equivalent classical algorithm [5]. For a given search space, viewed as a black box, it finds the exclusive entry that produces a defined output in that same black box. In classical computing, an equivalent algorithm would have a worst-case complexity of $\mathcal{O}(n)$, but Grover's approach reduces this complexity to $\mathcal{O}(\sqrt{n})$.

To do so, Grover's Algorithm follows the following steps:

1. Applies an Hadamard Gate to all qubits, making the state enter uniform superposition
2. Applies the Oracle (or black-box), negating the amplitude of the entry that produces the desired/defined output
3. Applies the Grover's Diffusion Operator, which amplifies the amplitude of the exclusive entry (the one with the negated amplitude)

The algorithm executes these steps r times, and at the end of each iteration, the probability of measuring the correct output (the sought value) is increased.

A simplified circuit representing Grover's Algorithm is shown below:

It is proven ([6]) that the number r that maximizes the probability of measuring the exclusive

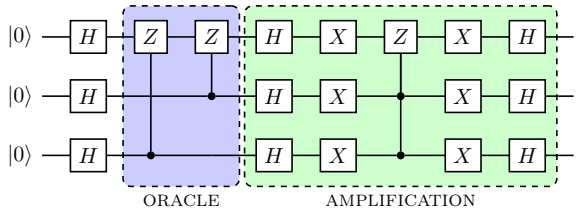


Figure 3: Grover's Algorithm Implementation

entry is equal to $\sqrt{n}$, leading to the $\mathcal{O}(\sqrt{n})$ complexity the algorithm has.

Grover's algorithm can have several applications, ranging from searching algorithms on large unordered lists - known as database bank searching -, to reverse engineering hash functions or even estimating the mean and median of a large set of numbers. As a consequence, its impact may be groundbreaking when quantum supremacy [4] is achieved for these applications.

6.2 Shor's Algorithm

Shor's algorithm is a quantum computer algorithm that runs almost exponentially faster than any equivalent classical algorithm. This algorithm attempts to find, for a given integer n a valid two-value factorization. In classical computing, an equivalent algorithm would have a worst case complexity of $\mathcal{O}(e^{1.9(\log N)^{\frac{1}{3}}(\log\log N)^{\frac{2}{3}}})$ - general number field sieve [9] - , but Shor's approach reduces this complexity to $\mathcal{O}((\log N)^2(\log\log N)(\log\log\log N))$.

To do so, Shors's Algorithm follows the following steps:

1. Create a quantum register $|reg1, reg2\rangle$
2. Load register 1 with an equally weighted superposition of all integers from 0 to $q-1$.
3. Load register 2 with all zeros.
4. Apply the transformation $a^x \bmod n$ for each number stored in register 1 and store the result in register 2.
5. Measure the second register, and observe some value k, collapsing register one into a equal superposition of each value x between 0 and $q-1$.
6. Compute the discrete Fourier transform on Register 1.
7. Measure the state m of register one, which has a very high probability of being a multiple of $\frac{q}{r}$, where r is the desired period.
8. Taking the value m, calculate on a classical computer the value of r based on m and q (explained below).
9. After attaining r, a factor of n can be determined by taking $\gcd(a^{\frac{r}{2}}+1, n)$ and $\gcd(a^{\frac{r}{2}}-1, n)$. If a factor of n was found, the algorithm has concluded; if not, it should go back to step 4.

A simplified circuit representing the Period Finding Function of Shor's Algorithm is shown in figure 4.

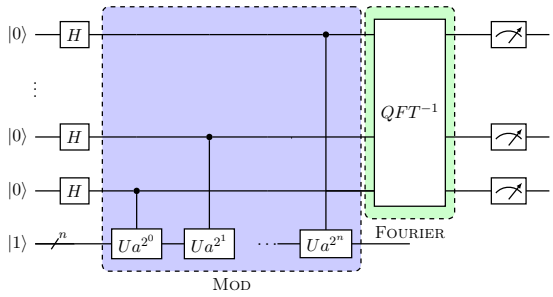


Figure 4: Period Finding Function Implementation

Some pre-calculations should be done before executing Shor. Firstly, the number n to be factored by Shor can't be a prime, an even number, or an integer power of a prime number for the algorithm to properly work. Alongside this, a power of 2 q must be chosen such that $n^2 \leq q < 2n^2$. Finally, a random number a that is coprime with n must also be chosen.

Finally, since there's a high probability that $m = \lambda\frac{q}{r}$, where λ is an integer, we can obtain r with the *convergent* of the execution of the *continued fraction expansion* on $\frac{m}{q}$, where r would be the denominator of the resulting fraction.

In short, Shor's algorithm relies on finding the period of a periodic function parameterized by n - the number to factorize - to actually compute a number $(x^{\frac{r}{2}}-1)(x^{\frac{r}{2}}+1)$, multiple of n. One of $(x^{\frac{r}{2}}-1)$ or $(x^{\frac{r}{2}}+1)$ - if neither is equal to 1 - will have a nontrivial common factor with n.

[4] Quantum supremacy describes the ability of a quantum computer to outperform its classical counterparts.

Quantum Parallelism [5], however, is the key to the improved runtime this algorithm has in comparison to its classical counterparts. It does so by allowing both the computing of $a^x \bmod n$ on each number in register 1 and the Quantum Fourier Transform (QFT) to be executed in one step each.

Shor's algorithm may have the impact of bringing an end to modern cryptography as we know it. The commonly-used RSA cryptosystem, for example, relies on the difficulty of factoring large integers, considering it unfeasible. However, with Shor's Algorithm that might not be the case and, as a consequence, Shor's impact, although exciting, can be seen as worrisome.

After understanding the inner workings of the algorithms to test, it is now possible to properly define rules and properties that relate the inputs and outputs - also referred to as metamorphic relations.

7 Metamorphic Relations

At first, Grover's Algorithm was explored in an attempt to find relevant metamorphic relations. However, that task was proven to be harder than expected. Given that the main goal was to apply palpable metamorphic relations to the algorithm and to prove its validity - within a short time window -, we decided to pivot to Shor's Algorithm.

While still being an interesting application of the metamorphic testing approach, Shor's Algorithm would be in line with the fully quantum MT approach taken by [3] if such a route was to be taken. This is the case because this approach used Grover's Algorithm in its implementation (testing Grover's Algorithm while using Grover's Algorithm for validation at the same time wouldn't make much sense).

For the development of these Metamorphic Rules for Shor's Algorithm, the template-based approach by [7] was used. All of them are shown below:

In the domain of *number factoring* where

- p is a given prime number
- q_1 is a given number
- $f(x)$ represents the set of factors of x

the following metamorphic relation should hold

- Add Prime:
  - *if* q_2 is equal to $q_1 \times p$
  - *then* $f(q_2) \setminus f(q_1) \subseteq \{p\}$

In the domain of *number factoring* where

- q_1 and q_2 are given numbers
- $f(x)$ represents the set of factors of x

the following metamorphic relation should hold

- Multiply Factors:
  - *if* q_3 is equal to $q_1 \times q_2$
  - *then* $f(q_3) = f(q_1) \cup f(q_2)$

Both of these metamorphic relations are good for testing. However, only the first one was implemented and thoroughly tested.

8 Implementation Analysis

After deciding the algorithm to test and defining its metamorphic relations, it was important to establish what platform would be used for the implementation of the testing mechanism for the chosen quantum program. For this, two different frameworks were used:

- Python with Qiskit's Shor's implementation
- Javascript with QCEngine's Shor's implementation

As previously stated, Shor's algorithm won't, under some circumstances, output a result, given the probabilistic nature of the period-finding subroutine. This would end up being a problem while implementing the testing mechanism.

Firstly, Qiskit's implementation was used, given Qiskit's ease of access and proper documentation. However, the algorithm provided by Qiskit attempts to solve the problem a built-in number of times. As such, when the a parameter is poorly chosen, instead of returning that no non-trivial results were found immediately - which could still happen with a proper value of a -, it attempts

[5] Ability of a quantum computer to encode multiple computational results into a quantum state in a single quantum computational step

again without user control. This ended up delaying the testing process since there was no way to define how many tries the algorithm should do.

As an example, factoring 33 with $a = 2$ would never output a non-trivial result, since both preconditions when choosing a fails ($a^r \equiv 1 \pmod{33}$ and r is not even, when $a = 2$ and $r = 5$). Although this could be asserted before choosing the value of a, it would result in a much more complex algorithm. As such, it would be useful that the used implementation allowed to define how many runs are done, so that, when no result is obtained after x amount of runs, a different value of a could be used - and repeat the process until a result is found.

After some research, QCEngine's implementation solved this problem. The algorithm itself was made available through O'Reilly's *'Programming Quantum Computers'* book ([1]) and allowed to control, programmatically, the number of executions, which resulted in a much more robust and generic hybrid algorithm for the testing of the metamorphic relations.

A simplified version of the developed code is shown below:

Algorithm 1 Metamorphic Rule Verification

Require: $q_1 > prime$

Ensure: $factors(q_1 \times prime) \setminus factors(q_1) = \{prime\}$

$q_2 \leftarrow q_1 \times prime$

$result_1 \leftarrow factorize(shor(q_1))$

$result_2 \leftarrow factorize(shor(q_2))$

$result_{diff} \leftarrow result_2 \setminus result_1$

assert $result_{diff} = \{prime\}$

Since Shor's algorithm factorizes the number given as input into two and only two coprimes, the $factorize$ function will just apply Shor's algorithm to all non-prime numbers given as output of Shor's.

It should be noted that it is not possible to run Shor's algorithm in a real quantum backend - that is, using real qubits - like the ones made available by IBM. This limitation is due to the number of qubits needed for Shor's algorithm.

For a number q that we wish to factor, Qiskit's implementation of Shor's algorithm would need $4 \times n + 2$ qubits to execute, where n corresponds to the number of bits that are needed to represent q in binary. As such, even for the lowest possible number to factor, which would be 1, the circuit would need at least $4 \times 1 + 2 = 6$ qubits. However, IBM only allows the use of at most 5 qubits for computations in real quantum machines, so there was no other option than to fall back on simulators.

9 Experimental Results

As Qiskit's implementation did not output relevant results for numbers greater than 30 - the computation times were very big, and for some numbers, it would not even return a proper result -, all results shown were obtained in the QCEngine's environment.

As previously stated, the **Add Prime** metamorphic relation was emphasised. Its results are described below.

$q1$	p	$q2$	$f(q1)$	$f(q2)$	$f(q_2) \setminus f(q_1)$
15	3	45	$\{3,5\}$	$\{3,3,5\}$	$\{3\}$
21	3	63	$\{3,7\}$	$\{3,3,7\}$	$\{3\}$
25	3	75	$\{5,5\}$	$\{3,5,5\}$	$\{3\}$
27	3	81	$\{3,3,3\}$	$\{3,3,3,3\}$	$\{3\}$
33	3	99	$\{3,11\}$	$\{3,3,11\}$	$\{3\}$
35	3	105	$\{5,7\}$	$\{3,5,7\}$	$\{3\}$
15	5	75	$\{3,5\}$	$\{3,5,5\}$	$\{5\}$
21	5	105	$\{3,7\}$	$\{3,5,7\}$	$\{5\}$
25	5	125	$\{5,5\}$	$\{5,5,5\}$	$\{5\}$
27	5	135	$\{3,3,3\}$	$\{3,3,3,5\}$	$\{5\}$
33	5	165	$\{3,11\}$	$\{3,5,11\}$	$\{5\}$
35	5	175	$\{5,7\}$	$\{5,5,7\}$	$\{5\}$
15	5	105	$\{3,5\}$	$\{3,5,7\}$	$\{7\}$
21	5	147	$\{3,7\}$	$\{3,7,7\}$	$\{7\}$
25	5	175	$\{5,5\}$	$\{5,5,7\}$	$\{7\}$
27	5	189	$\{3,3,3\}$	$\{3,3,3,7\}$	$\{7\}$
33	5	231	$\{3,11\}$	$\{3,7,11\}$	$\{7\}$
35	5	245	$\{5,7\}$	$\{5,7,7\}$	$\{7\}$

As expected, the metamorphic relation did hold for the default implementation.

However, to test that the metamorphic relations do detect faulty programs, some *mutants* were added to the code.

Since *O'Reilly*'s implementation allowed to change the code of the Quantum Operations themselves, an aditional CNOT operation on all output qubits after the Quantum Fourier Transform is applied.

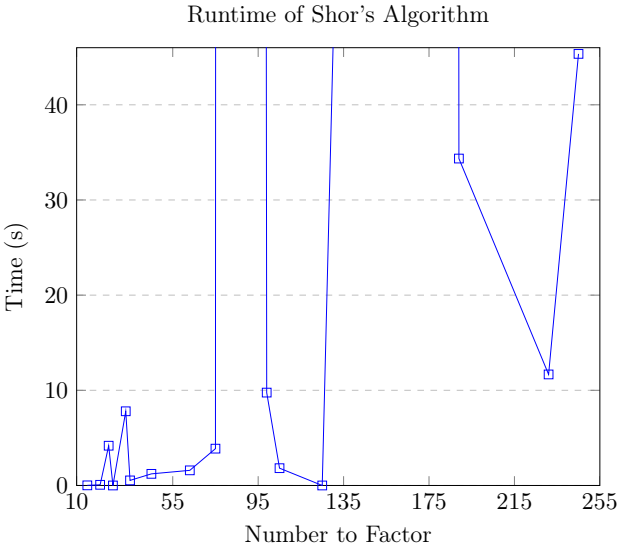
As Shor's Algorithm itself would not output any results - the *period finding function* had mutant code which changed its output - the values of m

which resulted from the quantum part of the algorithm can be compared before and after the mutant to verify that this mutant affected the Shor's result.

This can easily be seen for small numbers like 21, comparing the results of in 20 executions each. The function outputs several pairs of possible factors in each execution and it is possible to check if one of the pairs corresponds to a pair of coprimes of the number to factor. While the usual execution of the *period finding function* for 21 outputted a valid result in all 20 executions - that is, a valid pair of coprimes -, the mutant version struggled to do so, only outputting a valid result in of them - which can be explained due to the probabilistic nature of the algorithm.

In fact, metamorphic rules could have been made to only test the quantum component of the algorithm. Since this was not the case, the failure of the metamorphic rule - and the consequent detection of the quantum circuit mutant - happened because the algorithm itself failed to output any result.

Other interesting result concerns the time taken to factor out a number using Shor's Algorithm.



The plot above showcases the time of the execution of Shor's algorithm for different numbers. The runtime inconsistency is clear and may be explained due to two main factors:

- Shor's algorithm takes a random number as a parameter and, with it, it chooses several candidate 'results' for the previously mentioned *period finding function*. This random input can easily have an influence on the output of the algorithm since the value of a chosen might not be the best for the number which was to be factored.
- The algorithm is being run in a simulator, so the execution time has no real fidelity as it does not represent the actual runtime of the algorithm in a true quantum setting.

10 Conclusion

In this paper, we have approached the Quantum Correctness Problem and evaluated how Metamorphic Testing can be used as a possible solution. This solution avoids the Quantum Measurement Problem and the growing complexity of the problems Quantum Computing seeks to solve by relying on Metamorphic Rules which are intrinsical to the program to test.

Although the results do not fully demonstrate and prove the validity of the proposed testing mechanism, the conceptualized methodology shows promise to be a good way of tackling testing in the Quantum realm.

In future work, the metamorphic rules can be defined for each distinct part of the algorithm. In this case, such rules would be defined for Shor's Period Finding Function, allowing better control over failure detection. By defining rules for each component of the algorithm, in the quantum ones, it would also be possible to take advantage of Grover's Algorithm speedup for failed solution search (as proposed in [3]).

The proposed methodology can also be used in the future to obtain authentic experimental results - without using simulators -, which will, however, be always restricted to the state-of-the-art.

As a side note, this project will be submitted as a *Vision Paper*, which reports novel ideas about the application of quantum programming for software engineering, for the 1^{st} International Workshop on Quantum Programming for Software Engineering (SE4QP 2022). The workshop will be held in November 2022, in Singapore.

References

[1] Yulin Wu, Wan-Su Bao, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, Yajie Du, Daojin Fan, Ming Gong, Cheng Guo, Chu Guo, Shaojun Guo, Lianchen Han, Linyin

Hong, He-Liang Huang, Yong-Heng Huo, Liping Li, Na Li, Shaowei Li, Yuan Li, Futian Liang, Chun Lin, Jin Lin, Haoran Qian, Dan Qiao, Hao Rong, Hong Su, Lihua Sun, Liangyuan Wang, Shiyu Wang, Dachao Wu, Yu Xu, Kai Yan, Weifeng Yang, Yang Yang, Yangsen Ye, Jianghan Yin, Chong Ying, Jiale Yu, Chen Zha, Cha Zhang, Haibin Zhang, Kaili Zhang, Yiming Zhang, Han Zhao, Youwei Zhao, Liang Zhou, Qingling Zhu, Chao-Yang Lu, Cheng-Zhi Peng, Xiaobo Zhu, and Jian-Wei Pan. Strong quantum computational advantage using a superconducting quantum processor. *Phys. Rev. Lett.*, 127:180501, Oct 2021. doi: 10.1103/PhysRevLett.127.180501

[2] Retrieved Dec. 6, 2021, from https://www.honeywell.com/us/en/news/2021/07/honeywell-sets-another-record-for-quantum-computing-performance

[3] R. Abreu, J. P. Fernandes, L. Llana, G. Tavares, "Metamorphic Testing: A Simple Method for Alleviating the Test Oracle Problem," Q-SE 2022, Wed 18 May, *To Appear*

[4] T. Y. Chen, "Metamorphic Testing of Oracle Quantum Programs," 2015 IEEE/ACM 10th International Workshop on Automation of Software Test

[5] Retrieved Jun. 25, 2022, from http://twistedoakstudios.com/blog/Post2644_grovers-quantum-search-algorithm

[6] Retrieved Jan. 20, 2022, from http://math.uchicago.edu/ may/REU2012/REUPapers/Krahn.pdf - Proof

[7] Segura, S., Durán, A., Troya, J., & Cortés, A. R. (n.d.). A Template–Based Approach to Describing Metamorphic Relations. Retrieved from http://www.lsi.us.es/ jtroya/publications/MET17_at_ICSE17.pdf

[8] Retrieved Dec 15. 2021, from https://www.inria.fr/en/how-does-quantum-computer-work

[9] Retrieved June 26. 2022, from https://mathworld.wolfram.com/NumberFieldSieve.html

[10] Johnston, E. R., Harrigan, N., & Gimeno-Segovia, M. (2019). Programming Quantum Computers: Essential Algorithms and Code Samples (1st ed.). O'Reilly Media.

[11] Team, T. Q. (2022, June 24). Shor's algorithm. qiskit.org. Retrieved July 4, 2022, from https://qiskit.org/textbook/ch-algorithms/shor.html

[12] Programming Quantum Computers. (n.d.). Retrieved July 4, 2022, from https://oreilly-qc.github.io/