QMutPy: A Mutation Testing Tool for Quantum Algorithms and Applications in Qiskit

Daniel Fortunato
daniel.b.fortunato@tecnico.ulisboa.pt
Faculty of Engineering of University of Porto & INESC-ID, Portugal

José Campos
jcmc@fe.up.pt
Faculty of Engineering of University of Porto & LASIGE, Portugal

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

ABSTRACT

There is an inherent lack of knowledge and technology to test a quantum program properly. In this paper, building on the definition of syntactically equivalent quantum gates, we describe our efforts in developing a tool, coined QMutPy, leveraging the well-known open-source mutation tool MutPy. We further discuss the design and implementation of QMutPy, and the usage of a novel set of mutation operators that generate mutants for qubit measurements and gates. To evaluate QMutPy's performance, we conducted a preliminary study on 11 real quantum programs written in the IBM's Qiskit library. QMutPy has proven to be an effective quantum mutation tool, providing insight into the current state of quantum tests. QMutPy is publicly available at https://github.com/danielfobooss/mutpy. Tool demo: https://youtu.be/fC4tOY5trqc.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software maintenance tools.*

KEYWORDS

Quantum computing, Quantum software engineering, Quantum software testing, Quantum mutation testing

ACM Reference Format:
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 (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3533767.3543296

1 INTRODUCTION

The fast-approaching universal access to quantum computers is bound to break several computation limitations, but it is also bound to pose major challenges in many, if not all, computer science disciplines [23], e.g., *software testing*. Despite that, and the fact that in the classical computing realm *testing* has been extensively investigated and several approaches and tools proposed [7, 11, 12], such approaches for Quantum Programs (QPs) are still in their infancy [8, 14, 22].

Mutation testing [10, 19] has been shown to be an effective technique to improve testing practices, helping in guaranteeing the correctness of the program. Big tech companies, such as Google and Facebook, have conducted several studies [1, 18, 20] advocating for mutation testing and its benefits. The general principle underlying mutation testing is that the *bugs* considered to generate *buggy* program versions represent realistic mistakes that programmers often make. Such *bugs* are deliberately seeded into the original program by simple syntactic changes to create a set of *buggy* programs called mutants. To assess the effectiveness of a test suite in detecting mutants, these mutants are executed against the input test suite. If the result of running a mutant is different from running the original program, the mutant is considered *detected* or *killed*. The more *killed* mutants, the better is a test suite at detecting *bugs* that might occur.

Currently, to the best of our knowledge, there is no easy way to perform mutation testing on QPs automatically in a scalable manner. Since manually-written test suites are often prone to mistakes and carelessness, and given the added challenge of programming in a quantum language, we figured that implementing a tool that could help developers better assess the effectiveness and resilience of their test suites would be a valuable contribution to the quantum field. We argue that tools designed for this purpose would motivate developers to improve their quantum testing practices.

Thus, in this paper, we investigate the application of mutation testing in real QPs. We focus our investigation on the most popular open-source full-stack library for quantum computing [3], IBM's Qiskit. We propose QMutPy, a novel novel Python-based toolset, that automatically performs mutation testing for QPs written in Qiskit. QMutPy can perform automatic mutation testing for Python programs. In addition to the already implemented 20 classic mutation operators in MutPy, QMutPy supports the five novel quantum mutation operators we have presented in our previous work [4].

2 QMUTPY TOOLSET

2.1 Codebase

QPs written in Python using the Qiskit library are a mix of classic operations (e.g., initialization of variables, loops), as well as quantum operations (e.g., initialization of quantum circuits, measuring qubits). Thus, we searched for existing mutation tools that: (1) Support Python programs and the two popular testing frameworks for Python: `unittest` and `pytest`; (2) Support various classic mutation operators (e.g., Assignment Operator Replacement, Conditional Operator Insertion); (3) Support the creation of a report that could be shown to a developer or easily parsed by an experimental infrastructure; (4) Foster wide adoption. The learning curve to install, configure, and use the tool ought to be low.

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3543296

Mutatest [13], mutmut [9], MutPy [6], and CosmicRay [2] are the most popular mutation testing tools for Python that are available through *pip* (the package installer for Python). Albeit being open-source, fully automated, and support classic mutation operators, MutPy [6] was the only tool that met all requirements, and therefore we built QMutPy on top of MutPy.

QMutPy performs mutations through changes to the Python Abstract Syntax Tree (AST). MutPy had already implemented many AST operations (e.g. `ast.AugAssign`, `ast.Expr`). However, to implement our mutation operators, we required the use of the `ast.Call` operation, since quantum gates and measurements are function calls. In addition to a new AST operation, we created two new files (i.e., `qgates.py`, `qmeasurements.py`) which implement our five novel mutation operators [4].

2.2 Quantum Mutation Operators

We argue that our quantum mutants match real world bugs as (1) [15] described quantum mutation to be helpful to assess the correct behavior of QPs, and (2) 3 out of the 8 common bug patterns in Qiskit programs described in [24] are related to quantum gates as are the majority of our mutation operators.

Building on this and the previously introduced concept of syntactic equivalence of quantum gates [4] where a gate g is considered syntactically-equivalent to gate j if and only if the number and the type of arguments[1] required by both g and j are the same (e.g., the `h` (Hadamard) gate implemented in Qiskit has 10 syntactically-equivalent gates: `i`, `id`, `s`, `sdg`, `sx`, `t`, `tdg`, `x`, `y`, and `z`). The following subsections describe in detail each quantum mutation operator [4] and provide an example (based on the implementation of Shor's [21] algorithm available in the Qiskit-Aqua's repository[2]) on how each mutation operator creates mutants.

2.2.1 *Quantum Gate Replacement (QGR).*

This mutation operator first identifies each call to a quantum gate function (e.g., `circuit.x()`), and then replaces it with all syntactically-equivalent gates, e.g., `circuit.h()`, one at a time. For instance, for the `h` quantum gate, 10 mutants are generated as there are 10 syntactically-equivalent gates. Listing 1 exemplifies the QGR operator.

```
-   circuit.x(qubits[0])
+   circuit.h(qubits[0])
```

Listing 1: Example of the QGR operator.

2.2.2 *Quantum Gate Deletion (QGD).*

Adding and removing quantum gates from a QP can have a significant impact on its output. The QGD operation deletes an invocation to a quantum gate. Listing 2 exemplifies the QGD operator.

```
-   circuit.x(qubits[0])
+   pass
```

Listing 2: Example of the QGD operator. Note that in Python, a `pass` statement is a `nop` that when executed nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed [5].

[1]Optional arguments are not taken into consideration.
[2]https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/qiskit/aqua/algorithms/factorizers/shor.py

2.2.3 *Quantum Gate Insertion (QGI).*

This quantum mutation operator performs the opposite action of the QGD operator. That is, instead of deleting a call to a quantum gate, it inserts a call to a syntactically-equivalent gate. For each quantum gate in the source code, this mutation operator creates as many mutants as the number of each syntactically-equivalent gates. For example, for the `x` gate, which has 10 syntactically-equivalent gates, it creates 11 mutants, one per equivalent gate. Note that the `x` gate itself can be inserted into the source code, counting as a valid mutant. Listing 3 shows an example of the QGI operator.

```
    circuit.x(qubits[0])
+   circuit.y(qubits[0])
```

Listing 3: Example of the QGI operator.

2.2.4 *Quantum Measurement Insertion (QMI).*

In quantum computing, measuring a qubit breaks the state of superposition and therefore the qubit's value becomes either 1 or 0 (as in classical computing), which can be considered a mutation by design. That is the underline idea of the QMI operator. It adds a call to the `measure` function for each quantum gate call. Listing 4 shows an example of the QMI operator.

```
    circuit.x(qubits[0])
+   measurement_cr = ClassicalRegister(circuit.num_qubits)
+   circuit.add_register(measurement_cr)
+   circuit.measure(qubits[0], measurement_cr)
```

Listing 4: Example of the QMI operator.

2.2.5 *Quantum Measurement Deletion (QMD).*

Contrary to QMI, the QMD removes each measurement from a QP, one at a time. Without a `measure` call, the QP keeps the superposition state and as a consequence does not converge the qubit to either 1 or 0. Listing 5 shows an example of the QMD operator.

```
    up_cqreg = ClassicalRegister(2 * self._n, name='m')
    circuit.add_register(up_cqreg)
-   circuit.measure(self._up_qreg, up_cqreg)
+   pass
```

Listing 5: Example of the QMD operator.

2.3 Challenges

In this section, we describe the main challenges we faced to augment MutPy with the five novel mutation operators: QGR, QGD, QGI, QMD, and QMI.

2.3.1 *Replacement/Deletion of Existing Nodes in a Processed Python AST.*

The QGR mutation operator did not pose any difficulties to implement; it sufficed to alter the *node_to_mutate*. The QGD and QMD mutation operators however, at first, caused problems to implement, since we wanted to return the node as *null* (i.e., deleting the node). This was not possible for the AST to compute. We decided then to simply alter the *node_to_mutate* to a *pass* statement.

2.3.2 *Injection of New Nodes in a Processed Python AST.*

QMI and QGI (Listings 3 and 4, respectively) mutation operators posed some challenges to be implemented, since both these mutations do not require the modification of the *node_to_mutate*, but rather the injection of new nodes following the *node_to_mutate*. Inserting new nodes (i.e., new Lines of Codes (LOCs)) in the middle of a file means that all LOCs that follow must be bumped by the number of lines added. In order to overcome this problem, we modify the *node_to_mutate* to call a function that we added to the end of the file where we can insert new nodes without problems.

Listings 6 and 7 show the actual code created by QMutPy for the QMI and QGI mutation operators.

```
153 -   circuit.x(qubits[0])
153 +   __qmutpy_qgi_func__(circuit, qubits[0])
424 +   def __qmutpy_qgi_func__(circuit, qubit)
425 +       circuit.x(qubit)
426 +       circuit.y(qubit)
```

Listing 6: Example of the QGI operator in QMutPy.

```
153 -   circuit.x(qubits[0])
153 +   __qmutpy_qmi_func__(circuit, qubits[0])
424 +   def __qmutpy_qmi_func__(circuit, qubit)
425 +       circuit.x(qubit)
426 +       measurement_cr = ClassicalRegister(circuit.num_qubits)
427 +       circuit.add_register(measurement_cr)
428 +       circuit.measure(qubit, measurement_cr)
```

Listing 7: Example of the QMI operator in QMutPy.

2.4 Installation

Installing and using QMutPy is simple and straightforward. To get its latest version, one can clone the tool's repository:

```
git clone https://github.com/danielfobooss/mutpy
```

and compile/install QMutPy from it's source code:

```
python3 setup.py install
```

Note that although QMutPy requires Python version ≥ 3.4.0, Qiskit requires Python version ≥ 3.7.0, thus in our preliminary study (see Section 3) we used Python version 3.7.0.

2.5 Workflow

QMutPy's workflow is composed of four main steps. Given a Python program P, its test suite T, and a set of mutation operators M, QMutPy's workflow is as follows: (1) QMutPy firstly loads P's source code and test suite; (2) Executes T on the original (unmutated) source code; (3) Applies M and generates all mutant versions of P; (4) Executes T on each mutant and provides a summary of the results either as a yaml or html report.

2.6 Usage Example

Assuming we aim to perform mutation testing on the following quantum code which implements a Bell state, i.e., 4 entangled quantum states represented by 2 qubits:

```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, Aer,
       execute

def bell_state():
    q = QuantumRegister(2)   # Initialize qubits
    c = ClassicalRegister(2) # Initialize measurement bits
    qc = QuantumCircuit(q, c) # Initialize the circuit

    qc.h(q[0])                 # Apply Hadamard gate
    qc.cx(q[0], q[1])          # Apply CNOT gate
    qc.measure(q, c)           # Measure

    backend = Aer.get_backend('qasm_simulator')
    job = execute(qc, backend, shots = 1000)
    return job.result().get_counts(qc)
```

Listing 8: Code example to be tested (`bell-example.py`).

using the following test suite which tests if only 2 quantum states are obtained and if those states are '00' and '11' with a probability of 50% each (we give a margin of 5% error in our test cases):

```
from unittest import TestCase
from bell-example import bell_state

class TestBellState(TestCase):
    def test_is_00_observed_prob_50_percent(self):
        self.assertTrue(bell_state()['00']/1000 > 0.45)

    def test_is_11_observed_prob_50_percent(self):
        self.assertTrue(bell_state()['11']/1000 > 0.45)

    def test_has_only_2_measurements(self):
        self.assertTrue(len(bell_state()) == 2)
```

Listing 9: Code's test suite (`test-bell-example.py`).

We could perform quantum mutation testing using the following command and only enabling quantum mutation operators:

```
mut.py --show-mutants \
  --target bell-example --unit-test test-bell-example \
  --operator QGR QGD QGI QMI QMD
```

This command takes 3 seconds to run and once it has finished it reports a mutation score of 70.6% with 36 mutants killed, 15 survived, and 2 are considered incompetent. During its execution, QMutPy reports to the *stdout*, for each mutant, the mutation performed and its result, for example:

```
   - [#   1] QGD bell-example:
--------------------------------------------------------------
    7:     q = QuantumRegister(2)
    8:     c = ClassicalRegister(2)
    9:     qc = QuantumCircuit(q, c)
   10:
-  11:     qc.h(q[0])
+  11:     pass
   12:     qc.cx(q[0], q[1])
   13:     qc.measure(q, c)
   14:
   15:     backend = Aer.get_backend('qasm_simulator')
--------------------------------------------------------------
[0.00765 s] killed by test_is_11_observed_prob_50_percent
       (test-bell-example.TestBellState)
```

As an alternative, QMutPy can also report all data generated during the mutation testing session in YAML format (`-report REPORT_FILE`) which is easier to parse, in particular for large experiments.

3 PRELIMINARY STUDY

Our goal is to provide a tool that can perform mutation testing on QPs automatically in a scalable manner. To this end, we conducted a preliminary study[3] in which we considered 11 QPs written in the IBM's Qiskit library that range from 80 to 443 lines of code (245 on average). Our set of quantum mutation operators generated a total

[3]Fortunato et al. reports the preliminary study and discusses the results obtained, in more detail, in [4].

of 696 mutants for the 11 QPs, of which 325 (46.7%) were killed by the programs' test suites. The non-killed mutants either survived to the test suites (307, 44.1%), were not even exercised by the test suites (0.3%), or resulted in a timeout (62, 8.9%).

These results highlight the need for tools like QMutPy to the quantum world, in order to assist developers at designing better test suites, i.e., test suites that are able to *kill* more mutants and therefore more likely to detect *bugs*.

4 RELATED WORK

To the best of our knowledge, there are two tools in the literature that have performed quantum mutation testing on QPs: Muskit [16] and MTQC [17].

Mendeluze et al. [16] proposed Muskit, a Python mutation tool for Qiskit QPs, which can be used through the command line, its Graphic User Interface (GUI), or a web application. Muskit supports the mutation operator QGD as defined in Section 2.2.2 and the mutation operators QGI and QGR, but with no concept of syntactically equivalent gates. To use Muskit, one must provide the specification of the QP so that Muskit is able to assess whether a mutant has been killed by a test. This requires expertise in quantum computing and/or on Qiskit which is not available for any of the QPs used in our study. QMutPy, on the other hand, supports two additional quantum mutation operators, i.e., QMD and QMI, and is able to mutate 40 gates (+21 than Muskit). In addition, as the manually-written tests used in our study are equipped with test assertions, QMutPy does not require any program specification or any manual configuration to assess whether a test kills a mutant. It works out-of-the-box.

Pellejero [17] proposed MTQC, a Java quantum mutation tool that supports Qiskit and Q# QPs and is able to mutate 17 Qiskit's gates (vs. 40 in QMutPy). At the time of writing this paper, no study has been conducted with MTQC. Furthermore, we could not compare QMutPy's performance and effectiveness with MTQC as (1) MTQC does not support `unittest`, a requirement to run Qiskit-Aqua's manually-written tests, and (2) it requires one to manually use its GUI to perform the mutation analysis, one project at a time, which is time consuming and prone to mistakes.

5 CONCLUSIONS & FUTURE WORK

In this paper, we propose a mutation-based technique to test QPs, coined QMutPy, that is capable of mutating QPs for Qiskit, the IBM quantum framework. This is a first attempt to perform mutation testing on QPs with a tool that is easy to use, works out-of-the-box and at scale, and is available as an open-source project at https://github.com/danielfobooss/mutpy. Furthermore, QMutPy offers classic and more quantum mutation operators than the approaches / tools proposed in the literature.

As for future work, we plan to extend QMutPy with other mutation operators, offer it to other quantum frameworks (e.g., Cirq and Q#), and run our mutation analysis on real quantum computers. Moreover, combining QMutPy with techniques to automatically generate test suites for QPs [8, 14, 22] is an interesting venue for future work. In particular, QMutPy could be used to assert the effectiveness of the generated test suites.

ACKNOWLEDGMENTS

This work was supported in part by FCT/MCTES through projects ref. PTDC/CCI-COM/29300/2017 and CMU/TIC/0064/2019, and the research units LASIGE (ref. UIDB/00408/2020 and UIDP/00408/2020) and INESC-ID (ref. UIDB/50021/2020).

REFERENCES

[1] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry–A Study at Facebook. arXiv:2010.13464 [cs.SE]
[2] Austin Bingham. [n.d.]. Cosmic Ray: mutation testing for Python. https://github.com/sixty-north/cosmic-ray.
[3] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. 2018. Open source software in quantum computing. *PLOS ONE* (2018).
[4] Daniel Fortunato, José Campos, and Rui Abreu. 2022. Mutation Testing of Quantum Programs Written in QISKit. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 358–359. https://doi.org/10.1109/ICSE-Companion55297.2022.9793776
[5] Python Software Foundation. 2021. Python – pass statement. https://docs.python.org/3/tutorial/controlflow.html#pass-statements. Accessed: 2021-08-24.
[6] Konrad Hałas. 2011. MutPy: A Mutation Testing Tool for Python 3.x Source Code. https://github.com/mutpy/mutpy. Accessed: 2021-01-18.
[7] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. 2013. Achieving Scalable Model-Based Testing through Test Case Diversity. 22, 1, Article 6 (March 2013), 42 pages. https://doi.org/10.1145/2430536.2430540
[8] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. 2020. Property-Based Testing of Quantum Programs in Q#. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 430–435. https://doi.org/10.1145/3387940.3391459
[9] Anders Hovmöller. 2016. Mutmut: a Python mutation testing system. https://github.com/boxed/mutmut. Accessed: 2021-01-18.
[10] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.
[11] Natalia Juristo, Ana M Moreno, and Wolfgang Strigel. 2006. Guest editors' introduction: Software testing practices in industry. *IEEE software* 23, 4 (2006), 19–21.
[12] 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.
[13] Evan Kepner. [n.d.]. mutatest: Python mutation testing. https://github.com/EvanKepner/mutatest.
[14] 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
[15] P. Liu, S. Hu, M. Pistoia, C. R. Chen, and J. M. Gambetta. 2019. Stochastic Optimization of Quantum Programs. *Computer* 52, 6 (2019), 58–67.
[16] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2021. Muskit: A Mutation Analysis Tool for Quantum Software Testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1266–1270. https://doi.org/10.1109/ASE51524.2021.9678563
[17] Javier Pellejero. 2020. MTQC: Mutation Testing for Quantum Computing. https://javpelle.github.io/MTQC. Accessed: 2021-01-18.
[18] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 163–171. https://doi.org/10.1145/3183519.3183521
[19] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does Mutation Testing Improve Testing Practices?. In *Proc. of the 43rd IEEE/ACM ICSE*.
[20] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Practical Mutation Testing at Scale: A view from Google. *IEEE TSE* (2021).
[21] 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
[22] 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]
[23] Jianjun Zhao. 2020. Quantum Software Engineering: Landscapes and Horizons. arXiv:2007.07047 [cs.SE]
[24] Pengzhan Zhao, Jianjun Zhao, and Lei Ma. 2021. Identifying Bug Patterns in Quantum Programs. In *Proc. of the 2nd Q-SE.*