Muskit: A Mutation Analysis Tool for Quantum Software Testing

Eñaut Mendiluze
Simula Research Laboratory
Fornebu, Norway
enaut@simula.no

Shaukat Ali
Simula Research Laboratory
Fornebu, Norway
shaukat@simula.no

Paolo Arcaini
National Institute of Informatics
Tokyo, Japan
arcaini@nii.ac.jp

Tao Yue
Nanjing University of Aeronautics and Astronautics, China
Simula Research Laboratory, Norway
taoyue@ieee.org

***Abstract*—Given that quantum software testing is a new area of research, there is a lack of benchmark programs and bugs repositories to assess the effectiveness of testing techniques. To this end, quantum mutation analysis focuses on systematically generating faulty versions of Quantum Programs (QPs), called mutants, using mutation operators. Such mutants can be used as benchmarks to assess the quality of test cases in a test suite. Thus, we present `Muskit` – a quantum mutation analysis tool for QPs coded in IBM's Qiskit language. `Muskit` defines mutation operators on gates of QPs and selection criteria to reduce the number of mutants to generate. Moreover, it allows for the execution of test cases on mutants and generation of results for test analyses. `Muskit` is provided as command line interface, GUI, and web application. We validated `Muskit` by using it to generate and execute mutants for four QPs.**
`Muskit` code: https://github.com/Simula-COMPLEX/muskit
Web app: https://qiskitmutantcreatorsrl.pythonanywhere.com/
Video: https://youtu.be/EbPHJOK_AEA

***Index Terms*—quantum programs, software testing, mutation analysis, quantum circuits**

I. Introduction

Quantum programming languages (e.g., IBM's Qiskit and Microsoft's Q#) provide necessary constructs to program quantum computers that aim to solve extremely complex computational problems. Testing Quantum Programs (QPs) is challenging due to their special characteristics (e.g., superposition and entanglement) that require the development of new testing techniques. To this end, a few quantum software testing techniques have started to emerge, which focus on different testing aspects such as coverage criteria [1], property-based testing [2], fuzz testing [3], and runtime projections [4].

One way to assess the quality of test cases produced by a testing technique is to execute them on benchmark QPs. However, in the context of quantum programming, there do not exist bug repositories and benchmark programs to assess the quality of test cases produced by testing techniques. An alternative is to apply mutation analysis [5] to produce, using mutation operators, faulty versions of a correct QP, which serve as the baseline to assess the quality of test cases. Mutation analysis has been recently applied [2], [1] to assess the effectiveness of test cases developed for QPs; however, no tool was provided and so mutants were generated manually.

We present a mutation analysis tool called `Muskit` (MUtation testing for QisKIT), for QPs coded in IBM's Qiskit framework [6]. We envision that `Muskit` is mainly intended for quantum software testing researchers interested in assessing the quality of their developed techniques for testing QPs; moreover, it could also be used by practitioners who want to estimate the quality of their test suites. Given a QP, `Muskit` applies a set of mutation operators to systematically generate mutated versions of the QP (i.e., *mutants*), by focusing on the gates and locations of the quantum circuit. `Muskit` also allows to execute provided test cases on the generated mutants and produce execution results. These results can be used to perform various analyses depending on the requirements of a testing technique. To validate this aspect, we implemented a test analyzer based on an existing published work [1].

`Muskit` is available as command line interface or as GUI. Command line is recommended when one is interested in running a lot of experiments and can do so within their code. The GUI version is convenient when one wants to try out a few QPs. Finally, one can also try `Muskit` as a web application.

To demonstrate the working of `Muskit`, we also generated and executed mutants for four QPs and reported the results.

The rest of the paper is organized as follows: Sect. II presents the necessary background and a running example, Sect. III introduces the tool architecture and methodology, and Sect. IV provides the validation of the tool. We present discussions in Sect. V, and conclude the paper in Sect. VI.

II. Background and Running Example

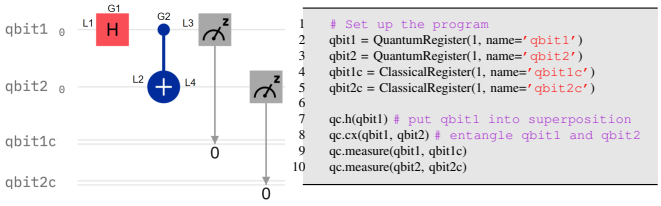
A. Quantum Program and Quantum Circuit

A *Quantum Program* (QP) works on *quantum bits* (*qubits*) in contrast to bits in classical computing. In a normal computer, the state is simply given by the bits values. In a quantum

This work is supported by the Qu-Test project (Project#299827) funded by Research Council of Norway. Paolo Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST.

TABLE I: Definitions of Some Basic Gates

Name	Description
Hadamard (h)	It puts a qubit into an equal superposition of states. For example, if in a one-qubit program, an HAD is applied, the program state will be an equal superposition of $\|0\rangle$ and $\|1\rangle$, i.e., it is 50% in $\|0\rangle$ and 50% in $\|1\rangle$.
NOT (x)	It flips a qubit (i.e., $\|0\rangle$ to $\|1\rangle$, and $\|1\rangle$ to $\|0\rangle$).
Rotation (R_z)	It is a phase gate that performs rotation across the z-axis with a provided angle (e.g., between 0.0 and 360.0).
Controlled-X (cx)	It is a two-qubit gate. It flips the *target* qubit, under the condition that the *control* qubit is $\|1\rangle$.



(a) Circuit Diagram

```
1  # Set up the program
2  qbit1 = QuantumRegister(1, name='qbit1')
3  qbit2 = QuantumRegister(1, name='qbit2')
4  qbit1c = ClassicalRegister(1, name='qbit1c')
5  qbit2c = ClassicalRegister(1, name='qbit2c')
6
7  qc.h(qbit1) # put qbit1 into superposition
8  qc.cx(qbit1, qbit2) # entangle qbit1 and qbit2
9  qc.measure(qbit1, qbit1c)
10 qc.measure(qbit2, qbit2c)
```

(b) Python code

Fig. 1: Entanglement in Qiskit

computer, instead, the state is defined by the values of the qubits and by an *amplitude* (α)—a complex number, which is described by its *magnitude* and *phase*. The square of the absolute value of amplitude of a state s (i.e., $|\alpha_s|^2$) indicates the probability of the program being in state s (if the values of the qubits are read). This probability is the magnitude. So, a program can be in *superposition* of different states $s_1, \ldots, s_n$ with different probabilities; this is represented in the bra-ket/Dirac notation [7] as $\sum_{i=1}^{n} \alpha_i |s_i\rangle$, with $\sum_{i=1}^{n} |\alpha_i|^2 = 1$.

A QP performs computations on qubits with *quantum gates*. A quantum gate takes qubits as input and performs computations resulting into changes of the state of the QP. Table I describes a list of gates commonly used in QPs. Note that this list is not complete.

Example 1. Fig. 1a shows the circuit diagram of the *entanglement* program, and Fig. 1b its equivalent Python code in IBM's Qiskit framework. The program performs *quantum entanglement*: given two input qubits, the program *entangles* them, i.e., when their values are read, they are the same (either both 0 or both 1). At Lines 2-3, the program initializes qubits *qbit1* and *qbit2* (by default, they are initialized to 0), while at Lines 4-5 it initializes two classical registers *qbit1c* and *qbit2c*. At Line 7, the program applies the Hadamard operation to *qbit1* (gate *h* in the circuit diagram in Fig. 1a): the effect of such operation is that the program enters in a superposition state in which *qbit1* can be both 0 and 1. At Line 8, the two qubits are entangled using the conditional not (*cx* gate in Fig. 1a). After this instruction, the program is in the superposition of 00 and 11 with equal probability; this means that reading the two qubits can lead to observe 00 or 11, both with 50% of probability. The *measure* operations at Lines 9-10 destroy the superposition, and register one of these two states in registers *qbit1c* and *qbit2c*.

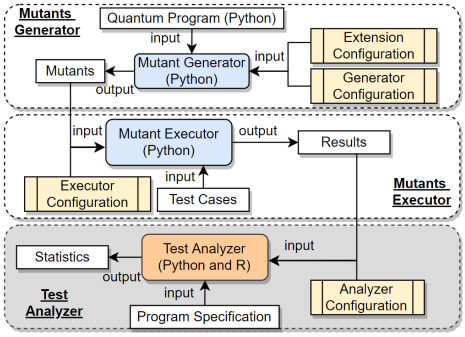


Fig. 2: `Muskit` Architecture

B. Mutation Analysis

A common way to assess the quality of test cases developed with testing techniques is through mutation analysis. With mutation analysis, we define a set of *mutation operators* that change a QP to produce a faulty version of it called a *mutant*. When we execute a test case on a mutant, and QP fails according to predefined criteria (e.g., a wrong output observed for a given input) based on *program specification* of the QP (including expected output), it means that the mutant is *killed*. The *mutation score* is commonly used to assess the quality of a test suite, where we calculate the number of mutants killed by the test suite out of the total number of mutants; if the number of equivalent mutants is known, it is deducted from the total number of mutants during the calculation.

`Muskit` focuses on mutating features of quantum circuits, in particular, the gates of a QP. To this end, we define two concepts: gate number and location. *Gate number* refer to a particular gate (e.g., *h* is *G1*) on a quantum circuit that we want to mutate via the *delete* or *replace* operators). As shown in Fig. 1a, *G1* and *G2* refer to an *h* gate and a *cx* gate, respectively. A location refers to a particular place in the quantum circuit, where we like to mutate by adding a new gate. L1-L4 in Fig. 1a indicate a set of locations where we can introduce gates. These two concepts will be used together to define mutation operators in Sect. III.

III. Tool Description and Methodology

`Muskit` has two main components, *Mutants Generator* and *Mutants Executor*, in addition to *Test Analyzer*, which can be implemented by the user depending on the required types of analyses. Fig. 2 shows an overview, and individual components are described below. The current implementation of `Muskit` is available as a command line application, a graphical user interface, and as a web application. All features are available in all implementation types, unless stated otherwise.

A. Mutants Generator

This component generates mutants for a given QP. In the following, for the ease of presentation, we define mutation operators at the circuit level; however, in the `Muskit` implementation, the Python code of QPs is mutated.

1) Mutation Operators: A mutation operator is defined based on three *Operator Types* described below:

- *Add Gate* (*AG*): adding a quantum gate at a particular location (e.g., L1-L4 in Fig. 1a).
- *Remove Gate* (*RemG*): deleting an existing quantum gate, e.g., deleting *cx*, i.e., *G2* in Fig. 1a.
- *Replace Gate* (*RepG*): replacing an existing gate with an alternative one which is compatible, e.g., replacing single-qubit gate *h* at *G1* (Fig. 1a) with single-qubit gate *x*.

In the current implementation, we support 19 gates. Thus, in total, we have 57 (3*19) mutation operators.

2) Mutation Selection Criteria: A large number of mutants can be generated for a QP. Thus, we provide selection criteria to reduce the number of mutants to be generated as:

- *All*. All possible mutants are generated.
- *Operator Selection*. A user selects one or more of the operator types, i.e., *AG*, *RemG*, or *RepG*.
- *Gate Selection*. A user controls mutant generation by selecting gates based on their number of input qubits. We define three categories, i.e., one qubit, two qubits, and more than two qubits.
- *Gate Number Selection*. A user selects particular gates by specifying the gate numbers (see Sect. II). This selection criterion is applicable only for the *RemG* and *RepG* operator types. In the command line, one specifies gate numbers in a configuration file, while in the GUI, one selects them from an automatically generated list.
- *Location Number*. A user specifies one or more location numbers (see Sect. II), on which to apply the *AG* operators. In command line, the user specifies the location numbers in the configuration file, whereas in the GUI, one selects them from an automatically generated list.
- *Phase Selection*. A user specifies the possible phase values that can be used to generate mutants for phase gates. Possible values can be specified between 0.0 and 360.0. This criterion is not supported in the web application.
- *Maximum Number*. One can limit the maximum number of mutants to be generated. Based on this, `Muskit` generates mutants by equally distributing them among the selection of operator types *AG*, *RemG*, and *RepG*. For example, if a user selected operators *AG* and *RemG*, and asked to generate 100 mutants, `Muskit` will generate 50 mutants of the *AG* type and 50 of the *RemG* type.

B. Mutants Executor

This component takes as input mutants and test cases, it executes the test cases on the mutants, and produces execution results. Given the probabilistic nature of QPs, each mutant is executed with a test case for a specified number of times. This component reads a configuration file (*ExecutorConfiguration*), where a user specifies the required number of repeated executions (i.e., also called *shots* in Qiskit).

Moreover, a user provides *Test Cases* (see Fig. 2). If a user sets *allInputs = True* in *Executor Configuration* (see Fig. 2), test cases corresponding to *All Input Coverage* [1] are used. Otherwise, the user can specify inputs they would like to use in *Test Cases* file as shown in Fig. 2. This file can contain test cases created by the user with any testing strategy. After test execution, results are produced in another file (*Results* in Fig. 2) that can be used by customized analyzers implemented in any implementation language of a user's choice. For each input, we produce the percentage of times that each output was observed. For example, if an input *a* was repeated 100 times, and we observed two outputs *o1* and *o2*, then the results file contains a percentage associated with each output (e.g., 40% of *o1* and 60% of *o2*). *Mutant Executor* is not supported in the web application version.

C. Test Analyzer

Test Analyzer is implemented by the user with their own test assessment criteria. To demonstrate the use of mutants, we implemented a simple test analyzer in *Python*. This component takes as input a configuration file (*AnalyzerConfiguration*), where a user specifies three things: 1) chosen level of p-value, e.g., 0.05; 2) qubit ids that are used as inputs; 3) qubit ids that will be measured. Given a program specification (e.g., specifying the expected outputs and their probabilities corresponding to each input) and a test results file from *Mutant Executor*, the component *Test Analyzer* can tell whether a mutant was killed by a test case or not. We employed two test assessment criteria from [1]. Simply speaking, we use the following two oracles:

1) Wrong Output Oracle (`WOO`): it checks whether a "correct" output is observed according to the program specification, i.e., whether the output can be produced with the given input. If not, the mutant is killed;
2) Output Probability Oracle (`OPO`): if all the observed outputs corresponding to an input are valid, then we compare their observed probabilities with the ones specified in the *Program Specification* file. If the differences are statistically significant (i.e., a p-value lower than the chosen significance level), the mutant is killed.

Based on the test assessments, detailed results are produced.

D. `Muskit` Extensions

In terms of extensions, one can specify new gates in the configuration file *ExtensionConfiguration*, which is read by `Muskit` to generate mutants. This functionality is supported in command line and GUI implementations. One can also checkout the code from GitHub and provide more advanced extensions to `Muskit`, such as integrating new test analyzers.

IV. Validation

To validate `Muskit`, we experimented four QPs with `Muskit`. Characteristics of the QPs are listed in Table II. IQFT implements inverse quantum Fourier transform, QRAM implements a simple quantum random access memory, BV implements Bernstein–Vazirani cryptography algorithm, and CE implements quantum conditional execution. For each QP, we show the number of qubits and number of gates it has. In addition, we show the numbers of single-qubit, multi-qubits and phase gates in columns *#s*, *#m*, and *#p* of Table II. To

TABLE II: Characteristics of QPs and Test Cases*

	#q	#g	#s	#m	#p	#tc
IQFT	6	9	6	3	0	64
QRAM	7	9	3	6	1	128
BV	6	9	6	3	0	64
CE	6	12	6	6	1	64

*#q: Num. of qubits; #g: Num. of supported gates; s#: Num. of single qubit gates; m#: Num. of multi-qubit gates; #p: Num. of phase gates; #tc: Num. of test cases

TABLE III: Results*

	Number of mutants				Mutation Score(%)	
	#AG	*#RemG*	*#RepG*	#Total	WOO	OPO
IQFT	285	9	78	372	0	100
QRAM	304	9	58	371	87.27	6.66
BV	255	9	79	343	77.35	0
CE	342	12	104	458	99.26	0.74

demonstrate `Muskit`, we generated test cases with the all inputs coverage [1], i.e., each possible input is a test case. For instance, for the 6 qubits program IQFT, there are 64 possible inputs (column *#tc* of Table II). Given the probabilistic nature of QPs, we executed each input 100 times.

Table III shows the results. It reports the total number of mutants generated, and the number for each mutation operator type, i.e., *AG*, *RemG*, and *RepG*. Moreover, it reports mutation score with respect to the two test oracles `WOO` and `OPO`. For example, for IQFT, a total of 372 mutants were generated and with 64 test cases, we reached 100% mutation score with respect to `OPO`. For BV, a total of 343 mutants were generated, and with 64 test cases, a mutation score of 77.35% was obtained with `WOO`. The rest of 22.65% of mutants weren't killed by any test case. Regarding the mutants not killed for QRAM and BV, it could be that either they are equivalent or the test suite does not contain enough tests to kill them. Indeed, regarding the latter case, in [1] we have shown that stronger criteria, such as Input-Output coverage, may be needed in some cases.

The generation of all the mutants for each program took less than 1 second. Execution of test cases on each mutant will vary from program to program and the computer used to execute them. However, in our experiments, executing one test case on one mutant took on average 3 seconds.

V. Discussion

Currently, we support mutation analysis only for IBM's Qiskit programs. However, we can see that transforming to other quantum programming languages is possible. For instance, there exist quantum programming frameworks (e.g., Quantum Programming Studio[1]) that can transform quantum circuits into diverse programming languages such as Rigetti Forest, IBM Qiskit, and Google Cirq.

Currently, `Muskit` is not able to detect equivalent mutants. This is mainly due to the reason that there is no current body of knowledge about equivalent mutants for QPs except for a few examples reported in [1]. As more knowledge becomes available about equivalent mutants and techniques to detect them, we will implement them in `Muskit`.

We here proposed different mutation operators, but we do not know how they relate each other in terms of killability of their mutants. As future work, we plan to theoretically study the mutation operators to check whether subsumption relations exist between some of them (i.e., killing a mutant of an operator always guarantee to kill a mutant of another operator). Moreover, we also plan to conduct an extensive empirical study to assess the difficulty to kill the different types of mutants, and to check whether some *practical* subsumption relations exist (i.e., although there is no theoretical subsumption relation, killing the mutant of one operator guarantees *most of the times* to kill a mutant of another operator).

In this paper, we implemented the test analyzer for an existing testing technique [1] to demonstrate the applicability of `Muskit`. However, users can implement test analyzers for their own purpose by reading the results file from `Muskit`'s Mutant Executor. Moreover, we only implemented two types of test assessment criteria from [1]. However, these can also be extended by users for their specific techniques.

VI. Conclusion

We presented `Muskit` – a mutation analysis tool for quantum programs (QPs) developed in IBM's Qiskit to help assessing quality of test cases developed for QPs. `Muskit` generates mutants based on gates in QPs and then has the functionality to execute mutants based on the provided test cases. Users can additionally implement their own test analyzers for their specific testing techniques using the results provided by `Muskit`. We demonstrated the applicability of `Muskit` by generating mutants for four QPs.

References

[1] S. Ali, P. Arcaini, X. Wang, and T. Yue, "Assessing the effectiveness of input and output coverage criteria for testing quantum programs," in *2021 IEEE 14th International Conference on Software Testing, Validation and Verification (ICST)*, 2021, pp. 13–23.

[2] S. Honarvar, M. R. Mousavi, and R. Nagarajan, "Property-based testing of quantum programs in Q#," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 430–435.

[3] J. Wang, F. Ma, and Y. Jiang, "Poster: Fuzz testing of quantum program," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 466–469.

[4] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Projection-based runtime assertions for testing and debugging quantum programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[5] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," ser. Advances in Computers. Elsevier, 2019, vol. 112, pp. 275–378.

[6] R. Wille, R. Van Meter, and Y. Naveh, "IBM's Qiskit tool chain: Working with and developing for real quantum computers," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1234–1240.

[7] M. Gimeno-Segovia, N. Harrigan, and E. Johnston, *Programming Quantum Computers: Essential Algorithms and Code Samples*. O'Reilly Media, Incorporated, 2019.

[1] https://quantum-circuit.com/