A Multi-Lingual Benchmark for Property-Based Testing of Quantum Programs

Gabriel Pontolillo
Department of Informatics, King's College London
London, United Kingdom
gabriel.pontolillo@kcl.ac.uk

Mohammad Reza Mousavi
Department of Informatics, King's College London
London, United Kingdom
mohammad.mousavi@kcl.ac.uk

ABSTRACT

We present a multi-lingual benchmark for (property-based) testing of quantum programs. We report on the methodology used to design our benchmark and the rationale behind its design decisions.

Our benchmark covers three major quantum programming languages, namely Qiskit, Cirq, and Q#. We curate our benchmark from languages documentations, open source repositories, and academic papers. In order to demonstrate the common logic of the algorithms included in our benchmark, we start from an implementation in one language (often Qiskit) and produce comparable implementations in the other two languages. We produce several properties and mutants for each program as a benchmark to measure the effectiveness of property-based testing frameworks. We reflect on the high-level quantum programming concepts offered in the three languages of our benchmark and their possible impact on testability and quality assurance.

ACM Reference Format:
Gabriel Pontolillo and Mohammad Reza Mousavi. 2022. A Multi-Lingual Benchmark for Property-Based Testing of Quantum Programs. In *The 3rd International Workshop on Quantum Software Engineering (Q-SE'22), May 18, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3528230.3528395

1 INTRODUCTION

Quantum computing is a rapidly expanding field of research, yet our ability to design and develop complex quantum software is still limited. Classical software engineering practices need to be adapted to quantum software engineering, enabling a more structured approach for building high-quality quantum software [35]. Testing is a major component of quality assurance in software development and it is comparatively understudied in the quantum community [24]. Quantum programs are difficult to comprehend and even more challenging to test; hence, we need advanced testing and quality assurance methodologies across various quantum and classical programming languages as quantum programs and protocols increase in complexity.

1.1 Problem statement

As a step to facilitate and support further research in testing quantum programs, a multi-lingual benchmark of programs, with their properties, and faulty variants (i.e., mutations), is needed. Benchmark programs can be used to evaluate general testing methodologies and procedures across multiple languages. A concrete set of programs that can be easily tested across multiple languages will help researchers that are either adapting classical techniques or developing new ways to test quantum programs. Particularly, property-based testing has potential for testing quantum software and would benefit from equivalent programs to appraise its efficacy and versatility and has the potential of bringing together model-based and search-based testing techniques in this domain.

1.2 Contributions

In this paper we present a set of benchmark programs, developed in Qiskit, Cirq, and Q#. The benchmark programs were translated to make them comparable to each other, making it easier to understand where the languages differ in addition to facilitating the development and translation of tests across the languages. In our repository, we provide a usage example for our benchmark using simple property-based tests, evaluating their efficacy with mutation testing with several mutants of the provided benchmark programs. We share insight in the languages whilst translating quantum algorithm code. The similarities and differences that were encountered are laid out and substantiated with examples from the benchmark code. These observations could inform further research as well as decision making regarding the language used for quantum programming and its consequences for testing. The current benchmark is available on an open Github repository via the following link https://bit.ly/3qa6cbJ.

Note that the development of the benchmark is an ongoing and long-term effort, but we decided to inform the community about our effort in its early stages through the present paper, so that the community may start using the benchmark and contributing to it, leading to a synergistic effect.

1.3 Structure

The rest of the paper is organised as follows: Section 2 provides an overview of the related work. Section 3 lists the decisions made when choosing the languages and evaluating the different information sources used to develop the benchmarks. The structure of the repository is explained in Section 4. We share our findings while developing the benchmark in Section 5, comparing the languages with the concept of testability in mind. Section 6 concludes the paper and presents the directions of our ongoing research.

Q-SE'22 , May 18, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9335-5/22/05.
https://doi.org/10.1145/3528230.3528395

2 RELATED WORK

The challenge of extending and adapting rigorous software engineering method to the area of quantum computing has been identified in the literature [5, 24] . In the recent years, several researchers have responded to this challenge and extended techniques such as fuzz testing [33], runtime assertion checking [15], search-based testing [34], mutation testing [17], and property-based testing [13] to quantum software.

Despite these recent advancements, there is still a major gap concerning benchmark programs [4]. The QBugs framework [4] identifies the gap concerning benchmark programs and example in quantum computing and proposes to curate a set of realistic bugs from GitHub repositories. Once realised our benchmark can be extended by injecting such real faults into the multilingual algorithms. This will lead to a richer evaluation framework for future testing and program analysis techniques.

Muskit [17] provides a method to automatically generate and analyse mutants of quantum programs. The mutants in our benchmark are very much inspired by the ideas presented in Muskit. We could not use Muskit directly to generate mutants as it requires our code to be structured monolithicly, without function definitions.

3 DESIGN DECISIONS

This section goes through the methodology and decisions made when designing our benchmark, including the choice of languages, and the sources used to curate the algorithms and their implementations.

3.1 Choice of Languages:

In order to choose the quantum programming languages for our benchmark, we defined two objectives:

(1) Popularity: we performed an analysis of open source repositories on Github and gathered data on the popularity of quantum programming languages in open source projects. To this end, we performed a search using the following query on Github:
"Quantum programming" OR
"Quantum computing" OR
"Quantum computation" OR
"Quantum algorithm"
We analysed the top 100 results based on the three ranking criteria provided by its search functionality. We then analysed the results based on the language or library used in each repository. We did count the result for all programming languages if multiple languages were used for a single repository. The results of our analysis are depicted in Figure 1; for reproducibility, we have also included the results in the benchmark repository. It appears from our analysis that Qiskit, Q#, and Cirq are among the most popular languages and libraries; other contenders that we would like to include in the future are Rigetti's PyQuil and Microsoft LiQui|⟩. Note that in these figures, among the other most popular languages and libraries, OpenQASM is a circuit-based abstraction for quantum programming in Qiskit (and hence, can be seen as a part of Qiskit) and PennyLane is a Python library mostly used for quantum machine-learning. The latter deserves its specific type of benchmark and test sets.
(2) Robust support and infrastructure: We would like our investment to sustain as a resource for the research community and hence, would like to invest our effort on programming languages that are backed by considerable corporate- or community effort. To start with we considered the technologies developed by IBM, Google, and Microsoft and are considering further languages by Amazon and Rigetti.

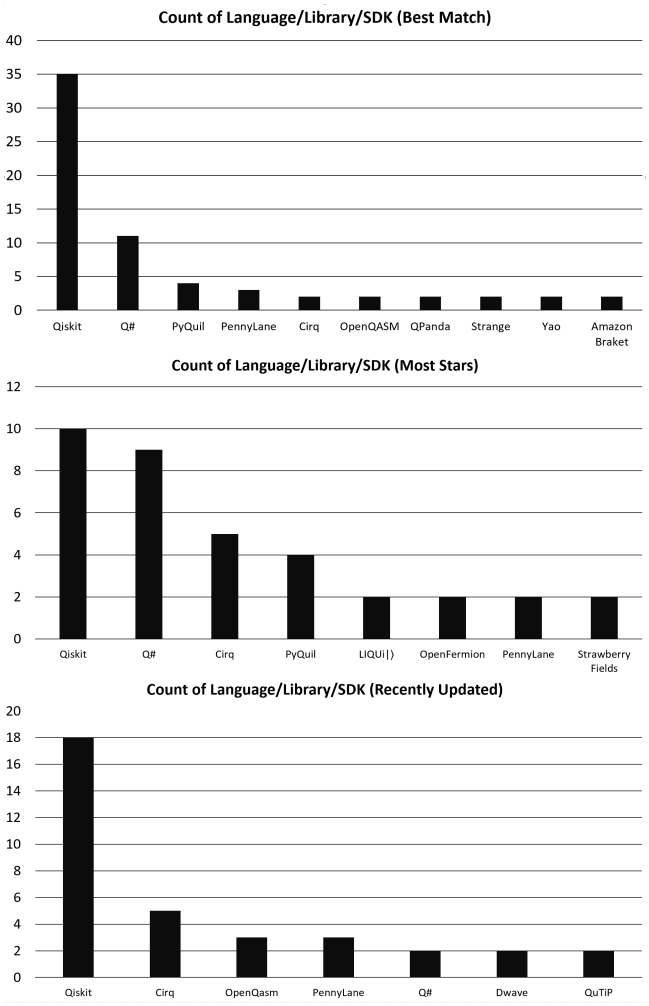

Figure 1: Relative popularity of different programming languages on Github repositories (data last retrieved on 15 January, 2022).

Based on these objectives, we settled on IBM's Qiskit, Google's Cirq, and Microsoft's Q# as our first set of quantum programming languages for our benchmark programs.

Table 1: The Different Sources Used for the Benchmark Algorithms

Algorithm	Language documentation and tutorials	Open repositories	Academic Papers
Superdense Coding	Qiskit Tutorial [29], Q# Tutorial [32]	Cirq github [12]	How to implement SDC from the inventors [2]
Quantum Fourier Transform	Qiskit Tutorial [26], Cirq Tutorial [6]	Q# Quantum Kata [18]	Description and circuit diagram of QFT in sec. 5.1 [25]
Quantum Phase Estimation	Qiskit Tutorial [28]	Cirq github [11], Q# Quantum Kata [20]	QPE circuit diagram in sec. 5 [8]
Quantum Key Distribution	Qiskit Tutorial [27]	Cirq github [10], Q# Quantum Kata [19]	QKD step-by-step in sec. 2 [3]
Shor's Algorithm	Qiskit Tutorial [30], Cirq Tutorial [7]	Q# github [21]	Shors' algorithm paper [31], Step-by-step implementation of the algorithm [16]

3.2 Algorithm Sources:

Benchmark quantum algorithms were selected by searching through academic papers, open source repositories and language documentation. We started with a set of common algorithms was included in the suite. We considered the code from the identified sources, chose one that clearly reflected the specification (or provided our own implementation in case none fit that criterion), translated it across the languages line-by-line and kept as close as possible to the original as the syntax of each library or language permitted, and performed conformance testing to make sure that the different implementations are consistent.

Below we list the sources used to create our benchmarks and how they contributed:

(1) **Language documentation and tutorials**: They have proven to be the most fruitful source of quantum algorithms. Programming language documentation usually contains common algorithms, though most of their focus tends to be in simpler programs for beginners to learn their language. Code from IBM's Qiskit tutorials [1] was heavily utilised for the benchmarks as the 'original' copy to convert to Cirq and Q#, due to its great documentation and simplicity.
(2) **Open source repositories**: Open source algorithm repositories often contain multiple quantum algorithms to use, they are a good source to get used to common coding practices within a language and see what is possible within a language. Microsoft's QuantumKatas are very good tools to learn quantum algorithms in Q#, they provide annotated Jupyter notebooks with step-by-step tasks to develop an algorithm, and tests to check if the implementation is correct.
(3) **Academic papers**: When searching through academic papers, the quantum algorithm description or implementation tends to be either limited or complex. Therefore, we have used academic papers to find new algorithms to add to the benchmark suite, though the implementation has been sourced from other locations. For example, 'Quantum Algorithm Implementations for Beginners' [9], is a large collaboration project containing a multitude of quantum algorithms, it contains many figures, circuit diagrams, algorithm descriptions and mathematical explanations to learn from.

An overview of the sources used to curate the algorithms are reported in Table 1.

4 BENCHMARK STRUCTURE

The repository containing the benchmark algorithms is structured as follows:

- **Introduction to Quantum Computing**: A detailed document on how to get started with quantum computing. It is split into five chapters, with the intent to provide a good starting point for people who have never used quantum programming languages. The document includes useful links and guidance to learn theory, install the languages, documentation and tutorials. The file also explains the structure of the repository and provides a short guide on running the provided code and analysing the results.
- **Algorithms**: The algorithms that have been implemented across the languages are listed.
  - **Implementations**: Under each algorithm, in three subfolders, we provide three implementations of the algorithm, in Qiskit, Cirq, and Q#. One of the implementations, the "base" is sourced from another location as discussed in Section 3. Much care went into translating the algorithms as closely as possible to clearly show the differences in the languages. Conformance testing was performed on each translation of the algorithms to provide some assurance that the programs are equivalent; the test results and analyses to that effect are available in the repository.
  - **Properties, Mutants, and Test Results**: In addition to the three implementations, we provide in a separate folder, several properties of the implemented algorithms (to enable property-based testing and use them for comparing different testing frameworks) and fifteen mutants. The properties we provide are formatted in the following order: **pre-condition**, **operation**, and **post-condition**. *Pre-conditions* describe the conditions that must be met by the input variable valuations in order to correctly test the property. *Operations* describe the sequence of functions that need to be executed. *Post-conditions* contain assertions on the attributes of the results.

Following the same principles as those in Muskit [17], for mutation testing, we used the following mutation operators for each and every algorithm: adding, removing, and replacing gates.

- **Code Comparison and description**: We reflected on our experience with the multi-lingual benchmark and in a separate folder provided a comparison of some of the common examples, pointing out the similarities and the differences among the different quantum programming frameworks. We spell out our reflections in the remainder of this paper.

These mutant versions of algorithms were then tested using the property-based tests that statistically validate classical outputs and qubit properties induced by functions in the quantum program. The results of the mutation testing were recorded and tabulated to view the property based tests' efficacy at killing mutants.

5 LESSONS LEARNED

In this section, we reflect on the important differences between the languages that we have encountered while translating algorithms across each of the benchmark programs; we organise the observed differences in the following structure:

- Low-level language constructs:
  - Quantum Circuits
  - Support for Frameworks
- High level language support
  - Custom gate creation
  - Gate modification
  - Mixed quantum and classical constructs
  - Quantum state introspection
- Testability
  - Test framework support
  - Performance

5.1 Low level language support:

Quantum Circuits: In Qiskit and Cirq, there is a direct correspondence between the algorithms provided in terms of quantum circuits and the code. However, in Q# the correspondence is more blurred with the use of object-oriented concepts and operations that directly manipulate qubits. Moreover, it does not currently offer a practical way to print or work with the quantum circuit. (IQsharp provides a command %trace to print the quantum circuits in notebook files, which we could not reproduce in our experiments.) The additional complexity of the Q# constructs, may be a drawback for programmers who have learned programming using quantum circuits. For such programmers, a visual description of a recursive or iterative quantum algorithm via a quantum circuit is likely to be more intuitive. With the inability to print the quantum circuit, it is challenging to quickly check whether the algorithm is iterating correctly without performing temporary measurements or using dumpmachine() to retrieve the quantum state, both of which may not work well depending on the complexity of the algorithm or quantum state.

Example. Instantiation of a quantum circuit is shown in Table 4, lines 2 and 3. The Python libraries differ slightly when we specify a quantum register of length four for the circuit in Qiskit whereas we separately instantiate four qubit objects in Cirq to be used as registers. In Q# we only need to instantiate the qubits and the quantum circuit object is left implicit.

The mathematics for producing a controlled phase is comparable in Qiskit and Q# is the same; however, due to the notational differences, the implementation of the mathematics in Q# is less readable in our view. (We comment on the scarcity of standard mathematical functions in Q# below.)

Support for Frameworks: We experienced a considerable difference between the three languages in terms of the available support for mathematical functions and programming frameworks. In our experience, the available libraries for Q# are much more limited (due to its specific language design as well as its recent introduction) while there are abundant Python libraries for various types of mathematical analysis for Cirq and Qiskit.

Example. At line 14 in Table 3, we can see the application of a quantum gate that applies a phase across the languages. The gates are named quite differently, but they all apply a user specified rotation about the Z-axis. In Table 3 the simple task of applying a measurement gate is a single line of code for the Python libraries, line 4. The Q# version spans through lines 4-7, with usage of its own mutable and set statements.

Formatting the QFT results as dictionaries that contain the result string and count as key value pairs (or similar), can be done with all languages, though it is more challenging for Q#. In Table 4 this can be done in two lines with Qiskit and Cirq (lines 7 and 8). To the best of our knowledge: the best approach for Q# is to use a host C# program that calls the Q# operation multiple times, while counting the results and placing them in a dictionary (lines 9-23).

5.2 High level language support:

Custom Gate Creation: It is possible to create custom-gates through decomposition across all languages; the main difference concerns the creation of gates in Cirq as classes, which should implement a standard interface of methods. In Qiskit and Q#, however, custom gates are implemented as functions and operations, respectively.

Example. Table 5 and Table 6 contain code for modular exponentiation gate definition and application respectively.

- **Qiskit:** We use a normal Python function, where we create a quantum circuit and append quantum gates to registers. Calling the "to_gate()" (line 5) method will convert the defined quantum circuit into a quantum gate.
- **Cirq:** We create Python class that inherits from "Cirq.Gate" (line 1), implementing the "_decompose()" (line 10) method allows us to define a quantum circuit composed of other quantum gates.
- **Q#:** We simply define an operation that applies quantum gates to qubits passed in.

Gate modification: The higher-level quantum focus of Q# is exemplified by the "Adj" and "Ctl" labels that can be added to operation type signatures to specify what functors are supported. The "Adjoint" functor is then added to an operation, which will cause the automatically generated adjoint of the operation to be called. It

Table 2: Rotation to change basis of measurement

Line	Qiskit	Cirq	Q#
1	def set_measure_x(circuit, n):	def set_measure_x(circuit, qubits, n):	operation set_measure_x(qubits: Qubit[]): Unit{
2	for num in range(n):	for num in range(n):	for index in 0 .. Length(qubits)-1 {
3	circuit.h(num)	circuit.append(cirq.H(qubits[num]))	H(qubits[index]);
4			}
5			}

Table 3: Quantum Fourier Transform recursive circuit

Line	Qiskit	Cirq	Q#
1	def qft_rotations(circuit, n):	def qft_rotations_cirq(circuit, qubits, n):	operation GenerateQFT(qubits: Qubit[], num: Int) : Result[] {
2	if n == 0:	if n == 0:	if (num == 0){
3	set_measure_x(qc, 4)	set_measure_x(circuit, qubits, 4)	set_measure_x(qubits);
4	qc.measure_all()	circuit.append(cirq.measure(*qubits, key='this'))	mutable results = [];
5			for index in 0 .. Length(qubits) - 1 {
6			set results += [M(qubits[index])];
7			}
8	return circuit	return circuit	return results;
9			}
10	n -= 1	n -= 1	mutable n = num-1;
11	circuit.h(n)	circuit.append(cirq.H(qubits[n]))	H(qubits[n]);
12	for qubit in range(n):	for qubit in range(n):	for index in 0 .. (n - 1) {
13			let divisor = PowD(2.0, IntAsDouble(n-index));
14	circuit.cp(pi/2**(n-qubit), qubit, n)	circuit.append((cirq.CZ**(1/2**(n-qubit)))(qubits[qubit],qubits[n]))	(Controlled R1)([qubits[index]], ((PI()/divisor), qubits[n]));
15			}
16	return qft_rotations(circuit, n)	return qft_rotations_cirq(circuit, qubits, n)	return GenerateQFT(qubits,n);
17			}

Table 4: Quantum Fourier Transform driver code

Line	Qiskit	Cirq	Q#
1	backend = Aer.get_backend('aer_simulator')	simulator = cirq.Simulator()	operation runQFTGenerator() : Result[] {
2		qubits = cirq.LineQubit.range(4)	use qubits = Qubit[4];
3	qc = QuantumCircuit(4)	circuit = cirq.Circuit()	
4	qc.x(0)	circuit.append(cirq.X(qubits[0]))	X(qubits[0]);
5	qft_rotations(qc,4)	qft_rotations_cirq(circuit, qubits, 4)	return GenerateQFT(qubits, Length(qubits));
6			}
7	job = execute(qc, backend, shots=1000000)	results = simulator.run(circuit , repetitions =1000000)	// inside the main of host.cs
8	print(job.result().get_counts())	print(results.histogram(key="this"))	// use a for loop to get more shots
9			using (var qsim = new QuantumSimulator()){
10			Dictionary<string, int>resCount = new Dictionary<string, int>();
11			for (int i = 0; i<100000; i++){
12			String resString = runQFTGenerator.Run(qsim).Result.ToString();
13			int value;
14			if (resCount.TryGetValue(resString, out value)){
15			resCount[resString] = value+1;
16			} else {
17			resCount.Add(resString, value+1);
18			}
19			}
20			foreach (KeyValuePair<string, int>vals in resCount){
21			Console.WriteLine("Key = 0, Value = 1", vals.Key, vals.Value);
22			}
23			}

Table 5: Shor's algorithm modular exponentiation gate

Line	Qiskit	Cirq	Q#
1	def c_amod15(a, power):	class aMod15Gate(cirq.Gate):	operation c_amod15(a: Int, power: Int, qubits: Qubit[]): Unit is Ctl{
2	...	def __init__(self, a, power):	...
3	U.swap(0,1)	super(aMod15Gate, self)	SWAP(qubits[0],qubits[1]);
4	...	self.a = a	...
5	U = U.to_gate()	self.power = power	}
6	U.name = "%i%i mod 15" % (a, power)		
7	c_U = U.control()	def _num_qubits_(self):	
8	return c_U	return 4	
9			
10		def _decompose_(self, qubits):	
11		...	
12		yield cirq.SWAP(q0,q1)	
13		...	
14			
15		def _circuit_diagram_info_(self, args):	
16		return "a mod 15"	

Table 6: Applying Shor's modular exponentiation gate

Line	Qiskit	Cirq	Q#
1	qc.append(c_amod15(a, 2**q),[q] + [i+n_count for i in range(4)])	qc.append(aMod15Gate(a, 2**q) .on(qubits[8],qubits[9],qubits[10],qubits[11]) .controlled_by(qubits[q]))	(Controlled c_amod15)([qubits[q]], (a, PowI(2,q), [qubits[n_count],qubits[n_count+1], qubits[n_count+2],qubits[n_count+3]]));

is also possible to call the inverse of a gate in Qiskit if the decomposed gates constituents have the inverse already implemented. The inverse of a gate is also obtainable in Cirq, provided that the gate implements the "pow" function with parameter -1. The "Ctl" label allows the "Controlled" functor to be added to a operation call, which allows control qubits to be added to any quantum operation in Q#. The same functionality is also available in Cirq and Qiskit through "Operation.controlled_by()" and "Gate.control()" respectively.

Example. Table 5 shows how to specify that a custom gate can be "controllable" and Table 6 shows how to pass the control qubits into a gate.

- **Qiskit:** We specify that a gate can be controlled by another qubit calling ".control()" on a gate (Table 5 line 7). The control qubit is the first qubit passed in through "append()" (Table 6),
- **Cirq:** We do not need to specify that a gate is controllable in the class definition, after specifying the qubits that are passed through the circuit using ".on()", the "controlled_by()" method can be called to specify the control qubits (Table 6 line 1).
- **Q#:** We can specify that an operation is controllable using the "Ctl" label in the type declaration (Table 5 line 1). To use the modified operation, we write "Controlled" before the function call, and pass in an extra list of qubits as the first parameter, these qubits will be the set of control qubits for the operation (Table 6 line 1).

Quantum state introspection: While these operations are not possible on real quantum computers, retrieving the exact quantum states of the qubits is useful when implementing and debugging quantum algorithms. It is possible to print the state vector of the exact quantum state with all three languages, additionally Qiskit also provides options to also visualise the quantum state across multiple qubits and states, through plotting the q-sphere or Bloch vector.

5.3 Testability:

Apart from the recent research effort reported in Section 2, there is very little domain-specific support for testing quantum programs. Below we compare the three programming languages included in our benchmark from the testing framework support and performance perspectives. Performance is particularly important, because due to the statistical nature of tests, testing is an extremely resource consuming part of development in this domain.

Test framework support: Cirq and Qiskit inherit the wealth of generic testing frameworks available for Python. However, such support is much more limited for Q#. Cirq and Qiskit can import frameworks such as Hypothesis to run property-based tests (which should be extended to cater for the statistical nature of quantum properties), while for Q# only a research prototype for property-based testing is available [13], which is less mature, and is missing features like shrinking and test case reproducibility.

Performance: There is a disparity in the performance of the languages, Q# falls behind in comparison to Cirq and Qiskit, running equivalent programs in the different languages takes the longest amount of time. Figure 2 provides an overview of our performance comparison using the Quantum Phase Estimation example in our benchmark. Note that although the diagram is drawn in logarithmic scale the differences are immense and even the time of executing the test program wrapper and the post-processing time required to check properties is negligible (and hence excluded from the analysis) compared to the simulation time. The details of the experiment are also included in the Github repository.

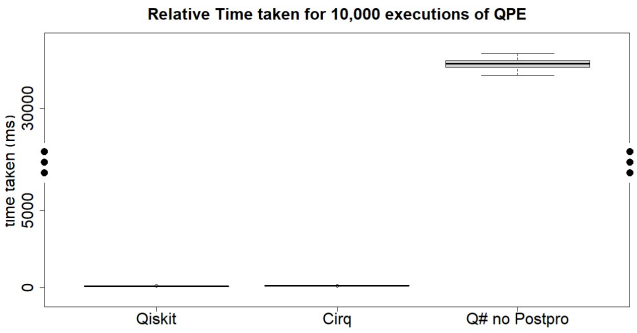

Figure 2: Performance Comparison of the Three Implementations of the Quantum Phase Estimation in Our Benchmark in the Three Programming Languages: Qiskit, Cirq, and Q

6 CONCLUSIONS

We presented a multi-lingual benchmark for quantum programming and property-based testing of quantum programs. Our benchmark comprises comparable code (in content and structure) across three major quantum programming languages, namely, Circq, Qiskit, and Q#. We reflect upon the insight we gained about differences across these programming languages.

This is the first step towards a community effort in order to support the evaluation of research in quantum program testing and analysis. We aim to develop both this benchmark and our property-based testing framework for the major quantum programming languages.

There are other examples [14] of online code. Providing properties for such programs and incorporating them into our benchmark is a part of our planned future work. Additionally, we did not incorporate specific concerns of noise and error correction such as those for noisy intermediate-scale quantum algorithms. Incorporating these concerns into our benchmark is another avenue for our future work.

Acknowledgments. Mohammad Reza Mousavi and Gabriel Pontolillo have been partially supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/2. We thank the anonymous reviewers for their helpful and constructive comments; we also thank José Miguel Rojas for his effort in compiling the initial list of examples for the benchmark.

REFERENCES

[1] S. Andersson, A. Asfaw, A. Corcoles, L. Bello, Y. Ben-Haim, M. Bozzo-Rey, S. Bravyi, N. Bronn, L. Capelluto, A. C. Vazquez, J. Ceroni, R. Chen, A. Frisch, J. Gambetta, S. Garion, L. Gil, S. D. L. P. Gonzalez, F. Harkins, T. Imamichi, H. Kang, A. h. Karamlou, R. Loredo, D. McKay, A. Mezzacapo, Z. Minev, R. Movassagh, G. Nannicini, P. Nation, A. Phan, M. Pistoia, A. Rattew, J. Schaefer, J. Shabani, J. Smolin, J. Stenger, K. Temme, M. Tod, E. Wanzambi, S. Wood,

and J. Wootton. Learn quantum computation using qiskit, 2020. URL http://community.qiskit.org/textbook.

[2] C. H. Bennett and S. J. Wiesner. Communication via one- and two-particle operators on einstein-podolsky-rosen states. *Phys. Rev. Lett.*, 69:2881–2884, 11 1992. doi: 10.1103/PhysRevLett.69.2881.

[3] C. H. Bennett, F. Bessette, G. Brassard, L. Salvail, and J. A. Smolin. Experimental quantum cryptography. *J. Cryptology*, 5:3–28, 1992. doi: 10.1007/BF00191318.

[4] J. Campos and A. Souto. Qbugs: A collection of reproducible bugs in quantum algorithms and a supporting infrastructure to enable controlled quantum software testing and debugging experiments. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pages 28–32, 2021. doi: 10.1109/Q-SE52541.2021.00013.

[5] C. Chareton, S. Bardin, D. Lee, B. Valiron, R. Vilmart, and Z. Xu. Formal methods for quantum programs: A survey. *CoRR*, abs/2109.06493, 2021.

[6] Cirq Documentation Tutorials. Textbook algorithms in cirq, 2021. URL https://quantumai.google/cirq/tutorials/educators/textbook_algorithms.

[7] Cirq Documentation Tutorials. Cirq shor's algorithm, 2021. URL https://quantumai.google/cirq/tutorials/shor.

[8] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca. Quantum algorithms revisited. *Proc. of the Royal Society*, 454(1969):339–354, 01 1998. doi: 10.1098/rspa.1998.0164.

[9] P. J. Coles, S. J. Eidenbenz, S. Pakin, A. Adedoyin, J. Ambrosiano, P. M. Anisimov, W. Casper, G. Chennupati, C. Coffrin, H. N. Djidjev, D. Gunter, S. Karra, N. Lemons, S. Lin, A. Y. Lokhov, A. Malyzhenkov, D. D. L. Mascarenas, S. M. Mniszewski, B. Nadiga, D. O'Malley, D. Oyen, L. Prasad, R. Roberts, P. Romero, N. Santhi, N. Sinitsyn, P. Swart, M. Vuffray, J. Wendelberger, B. Yoon, R. J. Zamora, and W. Zhu. Quantum algorithm implementations for beginners. *CoRR*, abs/1804.03719, 2018. URL http://arxiv.org/abs/1804.03719.

[10] Google's official Cirq repository. Cirq quantum key distribution, 11 2021. URL https://github.com/quantumlib/Cirq/blob/master/examples/bb84.py.

[11] Google's official Cirq repository. Cirq quantum phase estimation, 11 2021. URL https://github.com/quantumlib/Cirq/blob/master/examples/phase_estimator.py.

[12] Google's official Cirq repository. Cirq superdense coding, 11 2021. URL https://github.com/quantumlib/Cirq/blob/master/examples/superdense_coding.py.

[13] S. Honarvar, M. R. Mousavi, and R. Nagarajan. Property-based testing of quantum programs in q#. In *Proc. of ICSE '20*, pages 430–435. ACM, 2020. doi: 10.1145/3387940.3391459.

[14] E. Johnston, N. Harrigan, and M. Gimeno-Segovia. *Programming Quantum Computers: Essential Algorithms and Code Samples*. O'Reilly Media, Incorporated, 2019. ISBN 9781492039686. URL https://books.google.co.uk/books?id=LZY1vgEACAAJ.

[15] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proc. of OOPSLA 2020*, 4:150:1–150:29, 2020. doi: 10.1145/3428218.

[16] S. J. Lomonaco. Shor's quantum factoring algorithm, 2000.

[17] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue. Muskit: A mutation analysis tool for quantum software testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1266–1270, 2021. doi: 10.1109/ASE51524.2021.9678563.

[18] Microsoft's Quantum Katas. Qft reference implementation, 01 2021. URL https://github.com/microsoft/QuantumKatas/blob/master/QFT/ReferenceImplementation.qs.

[19] Microsoft's Quantum Katas. Q# quantum key distribution, 08 2021. URL https://github.com/microsoft/QuantumKatas/tree/master/KeyDistribution_BB84/ReferenceImplementation.qs.

[20] Microsoft's Quantum Katas. Q# implementation of quantum phase estimation, 01 2021. URL https://github.com/microsoft/QuantumKatas/blob/master/PhaseEstimation/ReferenceImplementation.qs.

[21] Microsoft's Quantum Katas. Q# Shor's algorithm, 08 2021. URL https://github.com/microsoft/Quantum/blob/main/samples/algorithms/integer-factorization/Shor.qs.

[22] A. Miranskyy and L. Zhang. On testing quantum programs. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '19, page 57–60. IEEE Press, 2019. doi: 10.1109/ICSE-NIER.2019.00023. URL https://doi.org/10.1109/ICSE-NIER.2019.00023.

[23] A. Miranskyy, L. Zhang, and J. Doliskani. Is your quantum program bug-free? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, page 29–32, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371261. doi: 10.1145/3377816.3381731. URL https://doi.org/10.1145/3377816.3381731.

[24] A. V. Miranskyy, L. Zhang, and J. Doliskani. On testing and debugging quantum software. *CoRR*, abs/2103.09172, 2021. URL https://arxiv.org/abs/2103.09172.

[25] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge, 2011. ISBN 1107002176.

[26] Qiskit Documentation Tutorials. Qiskit quantum fourier transform, 2021. URL https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html.

[27] Qiskit Documentation Tutorials. Qiskit quantum key distribution, 2021. URL https://qiskit.org/textbook/ch-algorithms/quantum-key-distribution.html.

[28] Qiskit Documentation Tutorials. Qiskit quantum phase estimation, 2021. URL https://qiskit.org/textbook/ch-algorithms/quantum-phase-estimation.html.

[29] Qiskit Documentation Tutorials. Qiskit superdense coding, 2021. URL https://qiskit.org/textbook/ch-algorithms/superdense-coding.html.

[30] Qiskit Documentation Tutorials. Qiskit shor's algorithm, 2021. URL https://qiskit.org/textbook/ch-algorithms/shor.html.

[31] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 10 1997. ISSN 1095-7111. doi: 10.1137/s0097539795293172.

[32] D. Vaughan. Quantum messaging with q# and blazor, 2019. URL https://docs.microsoft.com/en-us/archive/msdn-magazine/2019/september/quantum-computing-quantum-messaging-with-qsharp-and-blazor.

[33] J. Wang, F. Ma, and Y. Jiang. Poster: Fuzz testing of quantum program. In *Proc. of ICST 2021*, pages 466–469. IEEE, 2021. doi: 10.1109/ICST49551.2021.00061.

[34] X. Wang, P. Arcaini, T. Yue, and S. Ali. Generating failing test suites for quantum programs with search. In *Proc. of SBSE 2021*, pages 9–25, Cham, 2021. Springer.

[35] J. Zhao. Quantum software engineering: Landscapes and horizons. *CoRR*, abs/2007.07047, 2020.