Property-based Testing *of* Quantum Programs in Q#

Shahin Honarvar
Mohammad Reza Mousavi
School of Informatics
University of Leicester
Leicester, United Kingdom
sh726@student.le.ac.uk
mm789@le.ac.uk

Rajagopal Nagarajan
Department of Computer Science
Middlesex University
London, United Kingdom
R.Nagarajan@mdx.ac.uk

ABSTRACT

Property-based testing is a structured method for automated testing using program specifications. We report on the design and implementation of what is to our knowledge the first property-based framework for quantum programs. We review various aspects of our design concerning property-specification, test-case generation, and test result analysis. We also provide an overview of the implementation and its way of working. Finally, we present the result of applying our framework to some examples.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**.

KEYWORDS

software testing, quantum programs, quantum computation, property-based testing

ACM Reference Format:
Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. 2020. Property-based Testing *of* Quantum Programs in Q#. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3387940.3391459

1 INTRODUCTION

Quantum computing and communication systems are becoming practical and are likely to revolutionise modern technology. Much progress has been achieved recently by companies in developing quantum computers. Quantum programming languages exist and programs can be run on today's quantum hardware or on classical computers using simulators.

It is well-known that quantum programs are non-trivial to design and programmers may struggle to develop a sound intuition of their behaviour [27]. Thus, we need a spectrum of rigorous and structured techniques for the quality assurance of quantum programs. Unfortunately, this is a very much understudied topic, with the most prominent existing results focusing on formal verification techniques [3, 17, 18, 28]. Model-based testing is a complementary technique to formal verification for quality assurance that has been widely applied to industrial-scale systems. While it is not exhaustive, the advantage of model-based testing is that it scales up to very large systems and can still provide rigorous results regarding conformance. It can provide quantitative guarantees in terms of coverage and diversity of the generated test cases.

Property-based testing is a model-based testing technique that was developed as a unit testing tool, called QuickCheck, for Haskell. It has been later extended to different programming languages and is used for testing large-scale reactive systems [4, 13]. The basic idea of property-based testing is not to write individual test cases, but to generate them from more general properties of the system under test. These properties are often described in a logical style of pre- and post-conditions. In this approach, the state of the program is abstracted away or represented at a high-level of abstraction as a model-state [12].

The state-of-the-art in testing quantum algorithms and protocols is still rudimentary and we are not aware of any model-based- or property-based testing framework for quantum programs. For instance, a basic unit testing framework is provided in Q# [21, 23] and Miranskyy and Zhang [19] report on a validation and verification framework for Quantum Computing. Other steps that could lead to more sophisticated techniques include a number of assertion languages for quantum programs, reviewed below.

The aim of this paper is to take the first step towards structured testing techniques for quantum programs. To this end, we propose a property-based testing framework for Q# and report on a prototype implementation of this framework. To examine its applicability, we experiment with our framework on a suite of small, yet representative set of quantum programs. Our techniques are formulated in the context of the procedural programming language Q#. However, we expect that it would be transferable to other quantum programming languages and frameworks in a similar fashion as the extensions of QuickCheck.

Related work. There is a reasonable amount of work on formal verification of quantum programs and protocols, e.g., using equivalence-checking and model-checking [2, 3, 10, 14, 17, 28] and theorem proving [5, 18]. Quantum Hoare Logic [27] and its applied variant [29] are extensions of Floyd-Hoare logic for quantum programs. In particular, Zhou, Yu, and Ying [29] promise a testing and debugging framework based on their Applied Quantum Hoare Logic. Our approach does borrow a number of ideas from Quantum Hoare Logic. Also, the assertion language proposed by Huang and

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3391459

Martonosi [11] has been influential in the design of our language for property-based testing. There is a fundamental difference between the approach proposed by Huang and Martonosi [11] and ours; namely, we focus on the input-output interfaces and hence, rely on measurements after method calls, while their approach relies on approximate cloning [7] to inspect the internal state without disturbing it.

Contributions. The contributions of this paper can be summarised as follows: (i) a test property specification language for Q# programs; (ii) a property-based testing method to generate concrete test cases, execute them, and perform statistical analysis on the test data; and (iii) a case study on the effectiveness of the proposed approach by applying mutation on two well-known programs from the literature.

Structure. The remainder of this paper is organised as follows. In Section 2, we describe our language for specifying properties of Q# programs and provide a number of examples of tests specified in our language. In Section 3, we describe the architecture of our tool for test case generation, test execution and the (statistical) analysis of the outcomes. In Section 4, we report on the application of our tool to two examples from the literature. Finally, in Section 5, we conclude the paper and present the directions of our ongoing research.

2 LANGUAGE OF TEST FOR QUANTUM PROGRAMS

Our test specification language is inspired by the syntax of Q#, by quantum predicates and predicate transformers by D'Hondt and Panangaden [9] and Quantum Hoare Logic by Minsheng Ying [27]. Also, noteworthy in this domain is the recent introduction of Applied Quantum Hoare Logic [29], where Quantum Hoare Logic is restricted in order to allow for more efficient verification and also in the future, testing and debugging.

In this section, we first fix a syntax of our basic language for Q# test properties and briefly and informally discuss its semantics. In our language, a test comprises the following four parts:

(1) **Test property name and parameters:** A test property is given a name and the following parameters used for test case generation, execution, and analysis: the number of concrete test-cases to be generated from the property (i.e., instantiation of variables used in the property); the statistical confidence level for the property to hold (in test analysis); and the number of measurements and experiments for each concrete test case to obtain the data for statistical tests. Note that the first and the last parameters are semantically different: the first parameter refers to the number of generated concrete test cases from each abstract property and the last parameter refers to the number of experiments performed for each concrete test case.
(2) **Allocation and setup:** In this phase a number of qubits or qubit arrays are allocated. Subsequently, a pre-condition is specified on the input qubits. Some initial operations may bring the state of the allocated qubits to a particular state or a region in the Bloch sphere, if at all needed. Note that QSharpCheck is in charge of concretising the inputs if their constraints do not define a unique value in this phase.
(3) **Function call:** The system under test will then be called and the allocated and setup qubits are passed to it as its arguments.
(4) **Assert and de-allocate:** Subsequently, assertions are used to relate the pre- and post-states of the qubits. QSharpCheck will take care of forming a statistical hypothesis that relates these two states and verify the hypothesis using repetitive measurements of pre- and post-states resulting from the respective function calls.

The syntax of test properties is illustrated below:

```
//Preamble
Property name;
(numberOfTestCases, confidenceLevel,
     numberOfMeasurements, trials);

//Precondition and initialisation
{q_0 ... q_n : Qubit (θ interval)(ϕ interval)};
{b_0 ... b_n : Bool};

//Invoking system under test
operation_name (arguments) : return type;

//Postcondition
//Any combination of the following assertions
[AssertProbability(q_0,0,proposed propability)];
[AssertEntangled(q_0, q_1)];
[AssertEqual(q_0, q_1)];
[AssertTeleported(q_0)];
[AssertTransformed(q_0,(θ interval)(ϕ interval))];
```

We illustrate this syntax by means of the following example.

Example 1. State Transformation. Consider a hypothetical program "TransformState" that applies a rotation to the θ of the input qubit state around the y-axis by 72 degrees. A property of this program is that it takes an input qubit in the subspace $36^\circ \leq \theta \leq 72^\circ$ and $0^\circ \leq \phi \leq 360^\circ$ to a qubit in the subspace $108^\circ \leq \theta \leq 144^\circ$ and $0^\circ \leq \phi \leq 360^\circ$. The corresponding test property in our syntax is defined as follows:

```
1 Transform_Property;
2 (10, 99, 500, 300);
3
4 {q : Qubit (36,72)(0,360)};
5 TransformState(q);
6 [AssertTransformed(q,(108,144)(0,360))];
```

As specified before, the first two lines are the property name and its parameters. The parameters are all optional. If they are left unspecified, then they are given default values (specified in the next section). The first parameter, in this case 10, refers to the number of test cases that should be generated. The second parameter, 99, is the confidence level for accepting or rejecting the statistical hypothesis. The third and fourth arguments, in this case 500 and 300, respectively, specify the number of measurements (per experiment) and the number of experiments per concrete test case.

Next we allocate and setup the input data. In this test, a single qubit, q, is allocated and is setup in a subspace of the Bloch sphere with the specified parameters: the two intervals correspond to the θ and ϕ value ranges (in degrees), respectively. Next, the system under test, i.e., "TransformState", is called with the prepared qubit as its argument.

Finally, using a built-in assertion method of the QSharpCheck library, it is checked whether the new qubit state lies in the expected subspace. This assertion method takes two arguments. The first argument is the qubit state following the transformation and the second is the destination subspace identified in the same manner as line 4. Ultimately, if all of the test cases are passed then the appropriate message will be displayed as follows:

```
Testing "Transform_Property"
Number of test cases:                  10
Confidence level:                      99%
Number of measurements:                500
Number of experiments:                 300
AssertTransformed was true in all test cases. Passed
      10 tests.
```

If any of the test cases do not satisfy the property, the test will fail. For example, changing the main code under test to rotate θ by 36 degrees instead of 72 degrees will not map the resulting state into the expected subspace anymore. Therefore, if we test the program with the previous test script, the result will be:

```
Testing "Transform_Property"
Number of test cases:                  10
Confidence level:                      99%
Number of measurements:                500
Number of experiments:                 300
After 1 test, AssertTransformed was falsified when:
θ = 38 and ϕ = 5
```

To describe the semantics of tests, we provide a brief and informal overview below.

The states of quantum programs are density matrices, intuitively representing the probability distribution of finding the qubits in one of the base vectors (e.g., $|0\rangle$ and $|1\rangle$) after measuring the state. The predicates, used to specify the pre- and post-conditions of tests, are observables that take their values in the interval $[0, 1]$. These represent the expectations on finding variables in certain states. The degree of satisfaction of predicates is specified by the expectation of value, i.e., the trace of the correct observation applied to the state. For a detailed mathematical treatment of these concepts (and also their differences with similar concepts in pure probabilistic programs), we refer to their accessible treatment by D'Hondt and Panangaden [9]. Instead of dealing with the formal semantics of programs and proving satisfaction or violation of traces, we rely on the Q# compiler to correctly implement the semantics and use the traces generated by the compiled program to evaluate the assertions using statistical inference as described below.

3 QSHARPCHECK TOOL

In this section, we present the architecture of our QSharpCheck tool and explain the process of test case generation, execution, and analysis.

3.1 Tool Architecture

To check the properties of quantum programs, we have implemented a statistical toolkit in our framework.

We start with an overview of the basic statistical method used within our tool. The following basic facts need to be taken into account when designing the tool [15]:

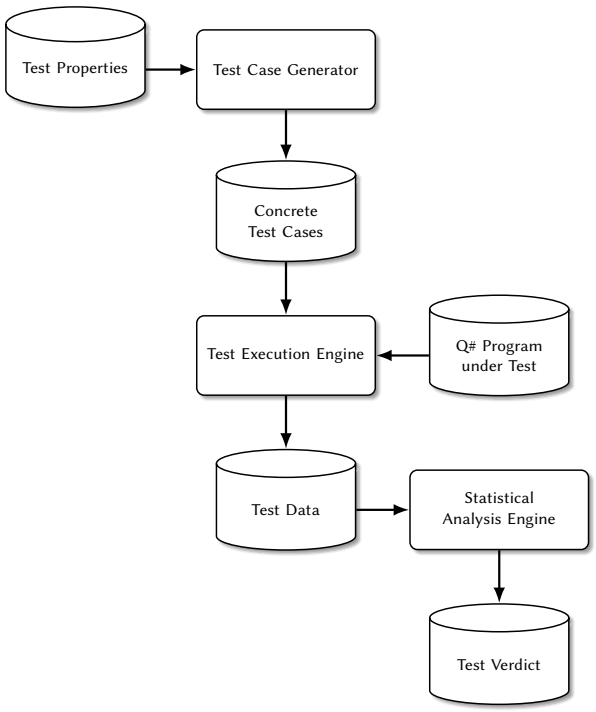

Figure 1: QSharpCheck Tool Architecture

- The exact state of the qubit before its measurement is unknown.
- We have access to a fixed number of measurements.
- There are only two possible discrete outcomes after each measurement: $|0\rangle$ or $|1\rangle$.
- Each measurement is independent of the others, which means the probability of one measurement outcome does not impact the probability of any other measurement result.

We use the statistical method of *hypothesis testing* [8], while assuming a *binomial distribution* of measurement probabilities. We have implemented this statistical check within our QSharpCheck framework. A type of hypothesis suggesting that no statistically significant difference between the observed counts and the predicted amounts is made in a set of observations is called a *null hypothesis* denoted by H_0[26]. The null hypothesis is presumed to be true until it is refuted by a statistical inference of its complement, the *alternative hypothesis* denoted by H_1. The hypothesis test is a method whereby one of the two complementary hypotheses should be rejected according to the sample values [8]. QSharpCheck considers the assertion provided in the last part of the test, to be the null hypothesis on the basis of confidence interval provided as the initial parameter and the measurement performed for each generated test case [25]. To find the aforementioned interval, QSharpCheck initially calculates the lower critical value and the higher critical value depending on the specified significance level during the two-tailed test implementation [6]. The testers can specify the confidence level as a parameter; however, if it is not provided, the framework will use 0.99 as the default value.

The overall architecture of our tool is depicted in Figure 1. The QSharpCheck tool parses the test scripts and using the initial parameters concretises a number of test cases. If the number of test cases is unspecified, it is set by default to 10. We experienced that tests for quantum programs are substantially more demanding than traditional computer programs and hence we have chosen the default number of test cases to be 10.

Subsequently, we organise a number of experiments and measurements: each experiment involves a fixed number of measurements and we treat each experiment as a separate data set to perform a statistical test across them. We set the default number of measurements to be 350 per experiment and 300 as the number of experiments per test case. The larger the values of the two arguments, the more reliable the test results will be, but since the cost of testing will increase, it is required to consider a reasonable trade-off between accuracy and cost. If the specified value is not sufficient for reaching a statistically significant verdict, an error message is produced beforehand and the testing campaign is terminated.

Based on the given variables, their data types and their intervals in the precondition part of the test script, the test case generator produces the required number of random input data. For instance, in Example 1, θ and ϕ intervals are (36,72) and (0,360), respectively. It is also stated that the number of test cases is 10. The test case generator generates 10 random θ values from 36 to 72 and the corresponding number of ϕ values from 0 to 360. The concrete test cases are then fed into the test execution engine to check the specified assertions after performing measurements.

Finally, we perform hypothesis testing on the obtained test data, as explained in the next section. We chose the default confidence level to 99 percent as it is a common value in experiments [25].

3.2 Different Types of Assertions

Our search for the type of assertions in our test language started with studying a number of typical quantum algorithms and their correctness properties. Subsequently, we abstracted a minimal set of assertions types that could cover these correct properties. These formed the basis of our postconditions and assertion types, specified further below. We expect that this basis will be extended further once we perform a more extensive inventory of quantum algorithms and their properties.

Measurement of qubits is common in quantum programming and due to the probabilistic nature of quantum program states, we need an assertion method that uses a robust statistical method to test the probability of observing a qubit in a given state. QSharpCheck is equipped with a method called **AssertProbability** which takes three arguments: a qubit, a state and the expected probability of observing the first argument (the qubit) in the second argument (the state) after measurement. For instance, for *AssertProbability(q, 0, 0.2)*, the suggested property claims that the probability of finding q in state $|0\rangle$ after measuring it, is 0.2.

Another important concept in the domain of quantum computation is quantum entanglement. The usual way of creating a two-qubit entanglement (*Bell State*) is by using a Hadamard gate followed by the application of a CNOT gate. Due to correlation between the entangled qubits, after measuring them the states to which they collapse are also correlated. Therefore, what QSharpCheck does is to check whether or not two qubits are correlated. **AssertEntangled** is the built-in assertion method of the library that takes two qubits as its arguments to test whether or not they are entangled. By making a sufficiently large number of measurements (whether set by QSharpCheck or the tester), a statistical testing technique, called "Chi-Square test for independence" [15], is used to analyse the correlation between the distributions of the measurement results of the two qubits. The null hypothesis in this case is that there is no correlation between the two qubits, that is, they are independent. This is complemented by the alternative hypothesis that there is indeed a correlation between them.

In quantum computation, there are cases where the equality of two qubits is of interest [16, 22, 24]. In order to test the equality of two states in quantum programs, our library has a method called **AssertEqual** which takes two qubits as its arguments to compare their states. In this case, QSharpCheck utilises the "independent samples t-test" to test whether the given two qubits states are equivalent or not. Based on the outcome of the comparison between the t critical value and the calculated t-value, the null hypothesis is either rejected or failed to reject. The null hypothesis in this test is that the two states are the same, while the alternative hypothesis is that they are not.

There is another assertion method in QSharpCheck called **AssertTeleported**. We have created this method specifically to test quantum teleportation as it is a significant protocol in the quantum realm. AssertTeleported takes two arguments: the sent and the received qubits, respectively. Initially, QSharpCheck starts allocating qubits with different random states. In each test case, since the amplitudes of the generated qubit state are known to QSharpCheck, the probability of observing the qubit after measurement in either states $|0\rangle$ or $|1\rangle$ is also known. For instance, if after measurement the probability of finding the qubit in state $|0\rangle$ is p_0 then based on p_0 the null hypothesis for checking the output qubit state is set. The statistical back-end calculations of this part of AssertTeleported is identical to that of AssertProbability method.

As illustrated in Example 1, the output of a quantum program may be the result of applying a transformation on the input arguments. To validate this, we have introduced **AssertTransformed** method in our library. To test the validity of any unitary transformation, firstly QSharpcheck allocates qubits with random states to generate concrete test cases. The amplitudes of these produced qubits states are specified, the scope of the destination state subspace is also established. On the basis of these details, QSharpCheck checks whether or not outcome qubits are in the predicted state subspace.

4 CASE STUDIES

To evaluate the effectiveness of our tool and technique, we took two typical quantum programs and defined a number of mutation operators to inject potential faults in them and studied which mutants could be killed by the test cases generated by our technique. In both case studies, the number of test cases was 10. In testing the teleportation system, since qubit calculations were required, the default values preset in the library were used for confidence level value, number of measurements and number of experiments. We obtained these programs source code from the online documentation

[20] and the repository [1] of Microsoft Q#. The types of mutation operators we used were statement deletion, statement duplication, replacement of Boolean relations, replacement of quantum operators, changing the order of quantum operators and replacement of a variable with another compatible one. The QSharpCheck test process is organised such that it stops if a test case fails. Otherwise, all test cases are executed. Therefore, the running time of a killed mutant is calculated up to the point where the first test case fails, while for live mutants it is the entire running time of all test cases.

Example 2. Teleportation. Initially, QSharpCheck tested the original Teleportation program and the tests passed. Next, QSharpCheck tested the mutants. The following table summarises the obtained results:

No.	Main Code	Mutant Code	Result	Run Time(ms)	Executed Test Cases
1	Z(there);	X(there);	killed	14826	1
2	X(there);	Z(there);	killed	15121	1
3	Z(there);	H(there);	killed	30997	2
4	X(there);	H(there);	killed	34816	2
5	H(here);	H(there);	killed	15890	1
6	H(msg);	H(here);	killed	14475	1
7	H(msg);	Y(msg);	not killed	179924	10
8	CNOT(msg, here);	CNOT(there, here);	killed	16910	1
9	CNOT(here, there);	CNOT(there, here);	killed	15542	1
10	M(msg) == One	M(msg) != One	killed	29448	2
11	M(here) == One	M(here) != One	killed	15437	1
12	if (M(here) == One) {X(there);}	deleted	killed	15577	1
13	CNOT(msg, here);	deleted	not killed	169646	10
14	CNOT(here, there);	deleted	killed	14080	1
15	CNOT(msg, here);	duplicate	killed	30213	2
16	Z(there);	duplicate	not killed	162513	10
17	X(there);	duplicate	killed	31163	2
18	CNOT(msg, here); H(msg);	H(msg); CNOT(msg, here);	killed	15547	1
19	H(here); CNOT(here, there);	CNOT(here, there); H(here);	killed	15229	1
20	CNOT(msg, here); –	CNOT(msg, here); let m = M(msg);	not killed	181276	10

The mutation score was 80% with the average execution time of 50932 milliseconds.

Example 3. Superdense Coding. No error was found by our framework after checking the main Superdense coding program. However, the library could kill a number of the generated mutants. The results are as follows:

No.	Main Code	Mutant Code	Result	Run Time(ms)	Executed Test Cases
1	H(q1); CNOT(q1, q2);	CNOT(q1, q2); H(q1);	killed	63	1
2	Z(qAlice);	Y(qAlice);	killed	71	1
3	Z(qAlice);	Y(qAlice);	not killed	76	10
4	Z(qAlice);	duplicate	killed	67	1
5	Z(qAlice);	duplicate	not killed	106	10
6	X(qAlice);	duplicate	not killed	62	10
7	if (b1) {Z(qAlice);}	deleted	killed	64	1
8	if (b1) {Z(qAlice);}	deleted	not killed	60	10
9	if (b2) {X(qAlice);}	deleted	killed	58	1
10	if (b2) {X(qAlice);}	deleted	not killed	61	10
11	Adjoint Entangle (qAlice, qBob);	deleted	killed	58	1
12	Adjoint Entangle (qAlice, qBob);	Entangle (qAlice, qBob)	killed	64	1
13	MResetZ(qAlice) == One	MResetZ(qAlice) == Zero	killed	57	1
14	MResetZ(qBob) == One	MResetZ(qBob) == Zero	killed	61	1
15	MResetZ(qAlice) == One	MResetZ(qAlice) == Zero	killed	65	1
16	Entangle (qAlice, qBob)	Entangle (qBob, qAlice)	killed	61	1
17	Entangle (qAlice, qBob)	Entangle (qBob, qAlice)	not killed	63	10
18	DecodeMessage (q1, q2)	DecodeMessage (q2, q1)	not killed	65	10
19	Entangle(q1, q2); EncodeMessage (q1, message);	EncodeMessage (q1, message); Entangle(q1, q2);	killed	69	1
20	Entangle(q1, q2); EncodeMessage (q1, message);	EncodeMessage (q1, message); Entangle(q1, q2);	not killed	70	10

In this example, the mutation score was 60% with the average execution time of 66 milliseconds.

Mutation scores in Examples 2 and 3 are indicative of the effectiveness of our approach. According to the results reported above, a majority of the mutants were killed, while we suspect that a number of remaining mutants are equivalent.

There are several threats to the validity of this study. One major threat to the validity of our claim to effectiveness stems from the mutation operators used in our study. We plan to provide more empirical evidence whether these mutation operators represent actual faults, by studying the faults introduced by programmers. We would also like to develop an automated and effective mutation tool for quantum programs that produces mutants with the right distribution of faults informed by actual data. Another threat to the validity of our result is due to the controlled simulation environment used for our experiments. Since the distribution of the outputs is considerably noisier on real quantum computers, a further study of the effect of the noise introduced by physical implementations remains as future work.

A lab package including the tool and the case studies will be available via: https://github.com/ShahinHonarvar/QSharpCheck.

5 CONCLUSIONS

We introduced QSharpCheck, a property-based testing framework for quantum programs written in Q#. We described our property specification language, the architecture of the tool and the statistical methods used to analyse test results. We have also applied our tool to two case studies and measured the effectiveness of our approach using a mutation score.

Much remains to be done in our ongoing research, as this is only our first step in property-based testing of quantum programs. Data generators that can efficiently satisfy preconditions and also provide a quantitative measure of test coverage or diversity are among our first priorities. Our property language is focused on stateless (pre- and post-condition type) properties. Extending the test property language is another important future extension of our approach. This will in turn enable shrinking strategies and effective debugging methods. Further studying the concept of mutation for quantum programs and developing an effective mutation tool for them remains another area for our future research.

REFERENCES

[1] 2019. Microsoft software developers, microsoft/QuantumKatas. https://github.com/microsoft/QuantumKatas/blob/master/SuperdenseCoding/ReferenceImplementation.qs.

[2] Ebrahim Ardeshir-Larijani, Simon J. Gay, and Rajagopal Nagarajan. 2013. Equivalence Checking of Quantum Protocols. In *Proceedings of TACAS 2013 (LNCS)*, Nir Piterman and Scott A. Smolka (Eds.), Vol. 7795. Springer, 478–492. https://doi.org/10.1007/978-3-642-36742-$7_3$3

[3] Ebrahim Ardeshir-Larijani, Simon J. Gay, and Rajagopal Nagarajan. 2018. Automated Equivalence Checking of Concurrent Quantum Systems. *ACM Trans. Comput. Log.* 19, 4 (2018), 28:1–28:32. https://doi.org/10.1145/3231597

[4] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *Proceedings of ICST 2015*. IEEE Computer Society, 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

[5] Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. 2015. Formalization of Quantum Protocols using Coq. In *Proceedings of QPL 2015 (EPTCS)*, Chris Heunen, Peter Selinger, and Jamie Vicary (Eds.), Vol. 195. 71–83. https://doi.org/10.4204/EPTCS.195.6

[6] Charles Henry Brase and Corrinne Pellillo Brase. 2016. *Understandable Statistics: Concepts and Methods*. Cengage Learning.

[7] Vladimir Bužek and Mark Hillery. 1996. Quantum copying: Beyond the no-cloning theorem. *Physical Review A* 54, 3 (1996), 1844.

[8] George Casella and Roger L. Berger. 2008. *Statistical Inference, Second Edition*. Thomson Learning.

[9] Ellie D'Hondt and Prakash Panangaden. 2006. Quantum weakest preconditions. *Math. Struct. Comput. Sci.* 16, 3 (2006), 429–451. https://doi.org/10.1017/S0960129506005251

[10] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. 2010. Specification and verification of quantum protocols. (2010), 414–472.

[11] Yipeng Huang and Margaret Martonosi. 2019. Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs. *CoRR* abs/1905.09721 (2019). http://arxiv.org/abs/1905.09721

[12] John Hughes. 2009. Software Testing with QuickCheck. In *Revised Selected Lectures of CEFP 2009 (LNCS)*, Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók (Eds.), Vol. 6299. Springer, 183–223. https://doi.org/10.1007/978-3-642-17685-2_6

[13] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *Proceedings of ICST 2016*. IEEE Computer Society, 135–145. https://doi.org/10.1109/ICST.2016.37

[14] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. 2008. QMC: A Model Checker for Quantum Systems. In *Proceedings of CAV 2008 (LNCS)*, Aarti Gupta and Sharad Malik (Eds.), Vol. 5123. Springer, 543–547. https://doi.org/10.1007/978-3-540-70545-$1_5$1

[15] Andrew A. Jawlik. 2016. *Statistics from A to Z, Confusing Concepts Clarified*. Wiley.

[16] B. Kraus. 2010. Local unitary equivalence of multipartite pure states. *Physical review letters* 104, 2 (2010), 020504.

[17] Yangjia Li, Nengkun Yu, and Mingsheng Ying. 2014. Termination of nondeterministic quantum programs. *Acta Inf.* 51, 1 (2014), 1–24. https://doi.org/10.1007/s00236-013-0185-3

[18] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Zhan Naijun. 2019. Formal Verification of Quantum Algorithms Using Quantum Hoare Logic. In *Proceedings of CAV 2019, Part II (LNCS)*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11562. Springer, 187–207. https://doi.org/10.1007/978-3-030-25543-$5_1$2

[19] Andriy V. Miranskyy and Lei Zhang. 2019. On testing quantum programs. In *Proceedings of ICSE (NIER) 2019*, Anita Sarma and Leonardo Murta (Eds.). IEEE / ACM, 57–60. https://doi.org/10.1109/ICSE-NIER.2019.00023

[20] Microsoft Q#. 2017. Putting It All Together: Teleportation. https://docs.microsoft.com/en-gb/quantum/techniques/putting-it-all-together#quantum-teleportation-code.

[21] Microsoft Q#. 2017. Testing and Debugging. https://docs.microsoft.com/en-us/quantum/techniques/testing-and-debugging?view=qsharp-preview&tabs=tabid-vs2019.

[22] Bao-Zhi Sun, Shao-Ming Fei, and Zhi-Xi Wang. 2017. On Local Unitary Equivalence of Two and Three-qubit States. *Scientific reports* 7, 1 (2017), 1–6.

[23] Krysta M. Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher E. Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. *CoRR* abs/1803.00652 (2018). http://arxiv.org/abs/1803.00652

[24] George F. Viamontes, Igor L. Markov, and John P. Hayes. 2007. Checking Equivalence of Quantum Circuits and States. *CoRR* abs/0705.0017 (2007). arXiv:0705.0017 http://arxiv.org/abs/0705.0017

[25] David L Weakliem. 2016. *Hypothesis testing and model selection in the social sciences*. Guilford Publications.

[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering*. Springer. https://doi.org/10.1007/978-3-642-29044-2

[27] Mingsheng Ying. 2011. Floyd-hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.* 33, 6 (2011), 19:1–19:49. https://doi.org/10.1145/2049706.2049708

[28] Mingsheng Ying and Yuan Feng. 2018. Model Checking Quantum Systems - A Survey. *CoRR* abs/1807.09466 (2018). arXiv:1807.09466 http://arxiv.org/abs/1807.09466

[29] Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. An applied quantum Hoare logic. In *Proceedings of PLDI 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1149–1162. https://doi.org/10.1145/3314221.3314584