Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs

Yipeng Huang
yipeng@cs.princeton.edu
Princeton University

Margaret Martonosi
mrm@princeton.edu
Princeton University

ABSTRACT

In support of the growing interest in quantum computing experimentation, programmers need new tools to write quantum algorithms as program code. Compared to debugging classical programs, debugging quantum programs is difficult because programmers have limited ability to probe the internal states of quantum programs; those states are difficult to interpret even when observations exist; and programmers do not yet have guidelines for what to check for when building quantum programs. In this work, we present quantum program assertions based on statistical tests on classical observations. These allow programmers to decide if a quantum program state matches its expected value in one of classical, superposition, or entangled types of states. We extend an existing quantum programming language with the ability to specify quantum assertions, which our tool then checks in a quantum program simulator. We use these assertions to debug three benchmark quantum programs in factoring, search, and chemistry. We share what types of bugs are possible, and lay out a strategy for using quantum programming patterns to place assertions and prevent bugs.

CCS CONCEPTS

• **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → **Software testing and debugging**; *Patterns*; • **Hardware** → *Quantum computation*; • **Theory of computation** → Assertions.

KEYWORDS

quantum computing, correctness, program patterns, assertions, debugging, validation, chi-square test

ACM Reference Format:
Yipeng Huang and Margaret Martonosi. 2019. Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322213

1 INTRODUCTION

Quantum computing is reaching an inflection point [34, 41]. After years of work on low-level quantum computing (QC) devices, small but viable QC prototypes are now available to run programs. These QC prototypes are increasing in size, with much research attention being placed on improving their reliability and increasing the counts of qubits (quantum bits), the fundamental building block for QC [20, 24, 33, 50]. These advancements in QC hardware may soon lead to a demonstration of a quantum algorithm running on QC hardware that exceeds the performance of a classical computer system [3, 14]. Such a demonstration would move the world closer to a new era of computing where QC systems solve problems in chemistry [26, 36], optimization [7, 8, 37], and even cryptography [46] that are currently intractable with classical computers.

With small-scale machines available to run real code, a natural challenge lies in bridging the QC architectural gap between algorithms and hardware. One aspect of that gap is in creating correct programs to run on quantum computers [4, 13]. Until recently, QC algorithms existed only in the form of abstract specifications and equations, and were rarely programmed for actual execution or simulation, and therefore relatively little QC debugging has ever occurred. Furthermore, QC debugging faces challenges beyond that of classical computing. In particular, typical debugging approaches based on printing out variable values during program execution do not easily apply to QC programs, because program states in QC collapse to classical values when observed. Second, while programmers have more freedom to observe full program states in QC simulations on classical computers, the massive state spaces

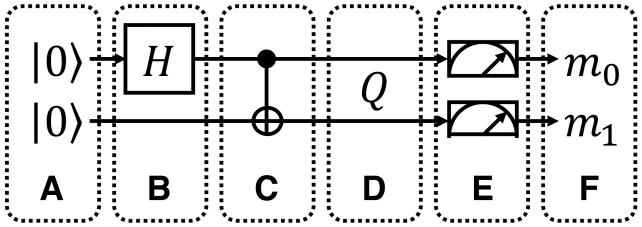

Figure 1: A Bell state creation quantum program. The sequence of operations in time flows left to right. A classical state consisting of two qubits (A) is manipulated into a superposition state by a quantum operation (B). A controlled-NOT gate (C) then induces entanglement between the two qubits to create an entangled state Q, which notably can no longer be factored into two separate pieces of information (D). Measurement of both of the qubits collapses the quantum state (E). Because the the qubits were entangled when they were measured, the measurement results m_0 and m_1 are correlated (F). Statistical tests on these measurements aid programmers in implementing and debugging quantum programs.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322213

of QC executions limits this approach to small programs. Finally, even when limited simulations are tractable, it can be difficult to interpret the simulation results.

While the problem of debugging and validating quantum programs has been extensively identified as a major barrier to useful quantum computation [4, 9, 13, 34, 39, 44, 49], little has been said about what actually constitutes a quantum program bug. Similarly limited detail has been shared about the inside story of translating QC algorithms in to working QC programs, even though the field is now making rapid progress in writing open source QC program benchmarks across several quantum programming languages [9, 17, 23, 27, 47, 48].

This work shares the detailed process of debugging quantum programs, with the help of a proposed set of quantum program assertions and breakpoints based on statistical tests. Using ensembles of classical observations taken from the intermediate state of quantum programs, these statistical tests are able to decide if the program state belongs to one of three important classes of quantum states: *classical*, *superposition*, and *entangled*. With this information, programmers can determine if the execution of the quantum program is valid up to each breakpoint. If the program state is invalid, the assertions guide the programmer in finding the bug inside subroutines and in program code up to that point.

For three quantum program benchmarks in factoring integers, database search, and quantum chemistry, we describe what kinds of bugs occurred in the process of bringing up the programs from unit tests to integration testing. We categorize the bugs according to where in the structure of quantum programs they may arise, and we lay out a strategy for placing statistical assertions that effectively catches them.

The rest of this paper is organized as follows: Section 2 provides relevant background for quantum programs and debugging. Section 3 details our statistical assertions and simulation framework for debugging quantum programs, which is then used in Section 4 for building and debugging an integer factorization quantum program. Section 5 evaluates the use of the assertions in two additional case studies. Section 6 discusses related approaches to writing correct quantum programs.

2 BACKGROUND ON QUANTUM STATES AND QUANTUM PROGRAMS

First, we review the principles of quantum computing [18, 28, 29, 35], in order to understand how debugging quantum programs is different from and more challenging than classical debugging.

2.1 Qubits, superpositions, and entanglement

The basic unit of information in QC is the *qubit*, which can take on values of $|0\rangle$ and $|1\rangle$ like bits in classical computing, but unlike classical bits, qubits can also be in a probabilistic *superposition* between the two values. Quantum computers can also *measure* the value of a qubit, forcing it to collapse out of superposition into a classical value such as '0' or '1'. Measurement disturbs the values of variables in a quantum computer. So unlike the case in classical computing, programmers cannot easily pause execution and observe the values of qubits as a quantum program runs. As a result of this limited visibility into program state, programmers must carefully choose what to measure and test when they debug quantum programs.

Aside from qubits being in superposition states, the other feature of data in QC is *entanglement*. For example, when the states of two qubits are entangled, the combined state of the two-qubit system must be viewed as a superposition of a larger set of elementary states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. An entangled state cannot be factored into independent pieces of information, and can no longer be viewed as a simple concatenation of two qubits. Likewise, a three-qubit system has potential superpositions of eight states, and so on. For this reason, as more qubits come into play in a quantum computer, the number of states that data can be in grows exponentially. This exponential growth of possible values due to superpositions and entanglement underlies the power of QC.

On the flip side, the huge state spaces involved in QC limits programmers' ability to use classical computers to simulate and debug QC programs. Naïve simulation of a 50-qubit quantum computer, for example, needs 2^{50} or roughly one quadrillion floating point numbers just to store the program state at any instant [11]. While more advanced techniques can decrease the memory requirement for simulating circuits [25, 30, 52, 55, 58], interactive programming and simulating quantum programs on a workstation is still limited to 20 to 30 qubits. For this reason, testing and debugging QC programs in simulation is only possible for toy-sized programs.

2.2 Quantum computer operations, programs, and a taxonomy for bugs

The process of quantum computing involves applying operations on qubits. Quantum computer scientists use diagrams such as Figure 1 to represent sequences of quantum operations. Looking at Figure 1 one sees that quantum programs consist of three conceptual parts:

(1) **Inputs** to quantum algorithms include quantum initial values for qubits and classical input parameters such as coefficients for rotations. Getting these inputs to be correct is the focus of Section 4.1.
(2) **Operations** include the specification of how to create an entangled state shown in Figure 1. Getting these basic operations to be correct is the focus of Section 4.2. Additionally, operations can be further composed according to patterns such as iteration, recursion, and mirroring. The correctness of these code patterns is the focus of Sections 4.3, 4.4, and 4.5.
(3) **Outputs** of quantum algorithms are the final classical measurement values of qubits such as m_0 and m_1. Furthermore, any temporary variables used in the course of a program have to be safely disentangled from the rest of the quantum state and discarded. Getting these final results to be correct is the focus of Section 4.6.

Bugs in quantum programs can crop up in any of these three parts of a QC program due to mistakes in converting algorithm specifications to program code. We will give examples of bugs in each of these places using detailed case studies of real quantum programs. To our knowledge, our work is the first study of such QC program patterns and anti-patterns [15].

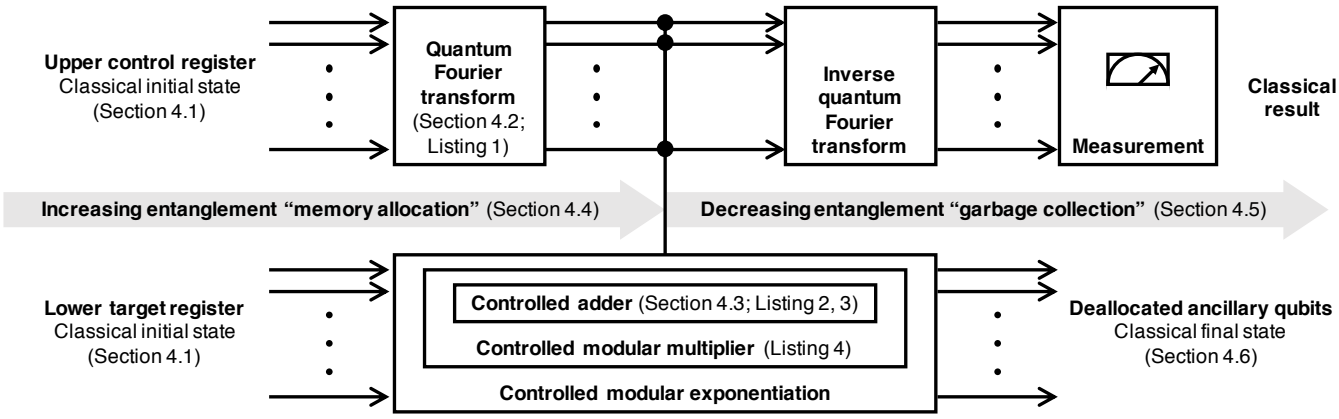

Figure 2: Roadmap for implementing and debugging Shor's algorithm.

3 OUR APPROACH TO STATISTICAL QUANTUM PROGRAM ASSERTIONS

Even though it is hard to have as much visibility into quantum program state as is the case in classical computing, limited but useful assertions checking is possible, particularly for the purposes of writing correct quantum programs. In this paper, we propose using statistical tests on measured outputs as a way to gain visibility into a quantum program (Section 3.1). We implement several quantum programs in a quantum programming language (Section 3.2). Using our "quantum breakpoints," programmers are able to check for expected values at various points in simulated quantum program runs, allowing them to debug the programs with the aid of our statistical assertions (Section 3.3).

3.1 Quantum assertions using statistical tests

In this subsection, we preview three basic types of assertions useful in quantum debugging, and we discuss the mechanics of using statistical tests as quantum assertions.

Following our overview of quantum states, superpositions, and entanglement in Section 2, one already sees that there are three kinds of possible assertions in a quantum program:

(1) **Classical assertions:** a quantum variable should take on a deterministic (classical) integer value upon measurement;
(2) **Superposition assertions:** a quantum variable in superposition should take on a probabilistic distribution of multiple values upon measurement;
(3) **Entanglement assertions:** two or more quantum variables in an entangled state should take on associated (correlated)[1] values once they are measured.

Statistical tests use ensembles of multiple measurements to decide to reject hypotheses. These serve as quantum programming assertions. With enough measurements, a statistical test is able to tell that an assertion does not hold, indicating a bug in the program. Otherwise, if the assertion holds, programmers can proceed cautiously knowing the quantum state so far is consistent with "no bug," given the number of measurements provided to the statistical test. While this approach is not powerful enough to decisively conclude that a quantum program state is correct, the debugging experience we share in this paper shows that detecting incorrect states is still useful enough to catch program bugs.

Specifically, our tool uses the **chi-square test** to check for classical and superposition quantum states, and it uses **contingency table analysis** coupled with the chi-square test to check for entangled states [42]. The assertions on classical and superposition quantum states are useful for quantum algorithm precondition checks and for unit testing, discussed in Sections 4.1, 4.2, 4.3, and 4.6. Similarly, the assertions on entangled states are useful for checking interaction between qubits, discussed in Sections 4.4 and 4.5.

3.2 Benchmark QC algorithms for debugging

To demonstrate using our assertions framework to debug quantum programs, we focus this paper on debugging three programs in factoring integers, database search, and quantum chemistry, each representing a different class of quantum algorithms.

Using the *Shor's integer factoring* quantum algorithm for factoring integers as a centerpiece example throughout Section 4, we show how the structure of quantum programs guides programmers where to put useful quantum assertions. We propose a complete taxonomy of where bugs can take place, and show assertions can catch all of the categories of bugs.

Then, using the *Grover's database search* and a *quantum chemistry* problem as additional case studies in Section 5, we discuss how different classes of quantum algorithms present different opportunities and challenges for debugging.

3.3 Simulation and assertions checking methodology

We implement the programs in the Scaffold quantum programming language developed by our research group [17],[2] now augmented

[1] Correlation is dependence between variables whose magnitude does matter. That compares with association which is dependence between nominal variables that are merely categories, and whose magnitude don't matter.

[2] https://github.com/epiqc/ScaffCC

Table 1: Correct and incorrect code for rotation decomposition. Using the Scaffold language [17] as an example, we code out Figure 3's controlled operation U, where U is a rotation in just one axis. Because only one axis is needed, we can drop either operation A or C, paying attention to the sign on the angles. Reordering the lines of code or signs results in a rotation in the wrong direction.

Correct, operation A unneeded	Correct, operation C unneeded	Incorrect, angles flipped
`Rz(q1,+angle/2); // C`	`CNOT(q0,q1);`	`Rz(q1,-angle/2);`
`CNOT(q0,q1);`	`Rz(q1,-angle/2); // B`	`CNOT(q0,q1);`
`Rz(q1,-angle/2); // B`	`CNOT(q0,q1);`	`Rz(q1,+angle/2);`
`CNOT(q0,q1);`	`Rz(q1,+angle/2); // A`	`CNOT(q0,q1);`
`Rz(q0,+angle/2); // D`	`Rz(q0,+angle/2); // D`	`Rz(q0,+angle/2); // D`

with the ability to specify and check for assertions. The assertions instruct the compiler where to stop execution and measure qubit states. Since premature measurement destroys the quantum state, the assertions effectively terminate the quantum program, splitting the quantum program into multiple breakpoints.

Our tool uses the ScaffCC compiler to compile Scaffold code with assertions into multiple versions of OpenQASM, a QC assembly language [6]. Each version of the compiled program has the program execution up to the quantum breakpoint, followed by an early measurement and assertions on expected values for the quantum variables.

Then, our tool simulates an ensemble of executions for each of the programs ending at each breakpoint, using the QX quantum simulator [19] running on a cluster compute infrastructure.

Finally, the tool gathers the results from the simulations to check for the assertions at each breakpoint. The measurement results feed into statistical tests, which check if the quantum variables have values that are consistent with being in one of classical, superposition, or entangled states. If the statistical tests reject the null hypotheses, that indicates the assertions were violated, which means there is a bug in the quantum program or that the assertion was an incorrect constraint.

Given the rapid growth of QC infrastructure, programmers now have the chance to test a variety of quantum algorithms written in many languages [23]. To validate our overall approach, we cross-validated our quantum programs and simulation results against equivalent programs written in other quantum programming languages, such as LIQUi|> [44], ProjectQ [12, 47], and Q# [48].

From this detailed debugging experience spanning multiple algorithms, languages, and simulators, we are able to concretely describe for the first time what types of bugs may crop up in quantum programs, and how assertions can aid in the debugging of quantum programs. As an added contribution, Section 5 discusses how language features of different QC programming languages can aid with the placement of quantum assertions, or otherwise prevent bugs in the first place.

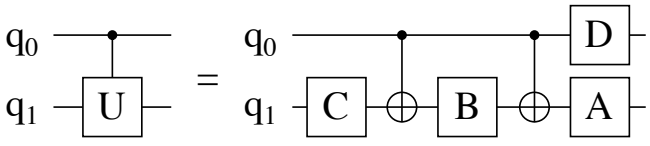

Figure 3: Decomposition of a simple QC program. Time flows left to right, showing sequences of operations applied to qubits q_0 and q_1. The left symbol is a *controlled* arbitrary operation U. Whether the operation U applies to the *target* qubit q_1 is dependent on the value of the *control* qubit q_0. The diagram on the right shows the decomposition into the equivalent sequence of more basic operations. The basic operations include single-qubit *rotations* A through D that alter the probability distribution of qubit values. The operations also include two two-qubit *controlled-NOT* operations that flip the target qubit (denoted $\oplus$) contingent on the value of the control qubit (denoted $\bullet$) [35].

4 QC DEBUGGING AND ASSERTIONS: SHOR'S ALGORITHM CASE STUDY

Using the Shor's quantum algorithm for factoring integers as a concrete case study, we show how the structure of the quantum program shown in Figure 2 aids programmers in using assertions to debug quantum programs. To defend against bugs in quantum program input, operations, and outputs, programmers can write assertions that check for preconditions, invariants, and postconditions. These constraints aid in the process of bringing up the program from unit tests to overall integration tests.

Shor's factorization algorithm uses a quantum computer to factor a composite number in polynomial time complexity, providing exponential speedup relative to the best known classical algorithms [46]. The algorithm works by estimating eigenvalues of a matrix, where the matrix is generated from the exponentiation of an integer representing a trial factor. The arithmetic is done in modular space with the modulus N set to be the integer one wants to factor. Here, we replicate results for factoring $N = 15$, the simplest example [21] [35, p. 235], by following an example for an implementation that minimizes the qubit cost [2]. Once the quantum part of Shor's algorithm is done finding the eigenvalues, those values are useful in a classical post-processing algorithm to find 3 and 5, the factors of 15.

We focus on debugging the Shor's factoring algorithm because it features in a single overall algorithm several important primitives (kernels) and program patterns common to many quantum algorithms. The primitives invoked in Shor's algorithm include order finding, eigenvalue estimation, state preparation, phase estimation,

and quantum Fourier transform. Our assertions and debugging techniques apply to several other QC applications that invoke similar primitives and patterns.

4.1 Classical and superposition precondition assertions on quantum initial values

Correct implementation and execution of QC programs begins with the right input states. Given their importance, it is worthwhile to check these preconditions by running or simulating programs up to the entry point of subroutines, and performing a premature measurement to check for these anticipated states.

Bug type 1: Incorrect quantum initial values. The type of quantum initial state that an algorithm needs depends on the type of algorithm. For eigenvalue estimation algorithms such as Shor's algorithm, the two major pieces of quantum data are an upper register and a lower register (far left of Figure 2): the upper register participates in a phase estimation subroutine; while the lower register is scratch space to implement a mathematical function such as modular exponentiation.

Here, the *lower target register* needs to be initialized to a strictly *classical integer* value, such as '1'. That means that if a quantum computer measured the qubit encoding the least significant bit of the quantum variable, it should return '1', while the measurements on the other qubits of the variable should return '0'.

On the other hand, the *upper control register* needs to be initialized to a uniform *superposition* of values. More concretely, if the upper register consists of, for example, 3 qubits, the measurement of the upper register at the beginning of the algorithm should return the eight values '000', '001', ... '111' with equal probability. That uniform superposition of values is created by the quantum Fourier transform (QFT). The QFT operation has the effect of taking integer inputs, and re-encoding them as quantum values that are distinct from each other by a quantum property known as *phase*.

The classical value in the upper register and uniform superposition in the lower register are the preconditions for Shor's algorithm. Other types of preconditions are possible for other types of algorithms. For example, quantum communications protocols often need entangled states as initial conditions.

Defense type 1: Assertion checks for classical and superposition preconditions. Our tool checks for both classical and superposition states using chi-square statistical tests on measured values.

To test for *classical integer* values, our tool gives the chi-square test the hypothesis that the distribution is unimodal with a peak at the expected value. If the test returns a small p-value (≤ 0.05), then the null hypothesis is rejected, meaning the initial state cannot be the expected value, indicating a violation of the precondition. If, on the other hand, the test returns a large p-value, typically close to 1.0, then the input state is consistent with being the expected value, though programmers cannot completely rule out a precondition violation. If there actually was a bug, programmers would only be able to detect the precondition violation using more measurements.

To test for *superposition* quantum states of n-qubits, the chi-square test uses as its hypothesis that the measurements should be a uniform distribution across all 2^n integer values. If the superposition precondition is violated, and there are sufficient measurements, the values would be concentrated enough for the chi-square test to reject the null hypothesis and raise an exception.

```
#include "QFT.scaffold"
#define width 4 // number of qubits
int main () {

  // initialize quantum variable to 5
  qbit reg[width];
  for ( int i=0; i<width; i++ ) {
    PrepZ ( reg[i], (i+1)%2 ); // 0b0101
  }

  // precondition for QFT:
  assert_classical ( reg, width, 5 );

  QFT ( width, reg );

  // postcondition for QFT &
  // precondition for iQFT:
  assert_superposition ( reg, width );

  iQFT ( width, reg );

  // postcondition for iQFT:
  assert_classical ( reg, width, 5 );
}
```

Listing 1: Test harness for quantum Fourier transform.

In prior work, the Q# quantum programming language has support for assertion checks for *integer* values, and is able to check for such assertions in simulations of quantum programs [48]. To our knowledge, this paper is the first proposal for quantum assertions on *superposition* and (later in this paper) *entangled* quantum states. Furthermore, this is the first work to discuss using statistical tests to check for these hypotheses.

4.2 Unit tests for a library of subroutines

Now that we have made sure the quantum initial states are valid, the next step in programming the Shor's algorithm is to build up the algorithm operations. We do so starting from elementary operations, which we exhaustively validate against their closed form solutions, and against implementations in other languages. Then we compose the elementary operations following code patterns—iterations, recursion, and mirroring—and test the composite subroutines using assertions (Sections 4.3, 4.4, 4.5).

Bug type 2: Incorrect operations and transformations. In order to correctly implement Shor's algorithm, programmers first have to build up the quantum subroutines such as the *controlled rotation* subroutine depicted in Figure 3. This subroutine is the building block for QFT and adder routines in Shor's algorithm (modules in Figure 2). Typically this task consists of translating quantum circuit diagrams, such as Figure 3, into quantum program

```
// outputs b <= a+b, where a is a `width' bit constant integer
// b is an integer encoded on `width' qubits in Fourier space
module cADD (
  const unsigned int c_width, // number of control qubits
  qbit ctrl0, qbit ctrl1, // control qubits
  const unsigned int width, const unsigned int a, qbit b[]
) {
  for ( int b_indx=width-1; b_indx>=0; b_indx-- ) {
    for ( int a_indx=b_indx; a_indx>=0; a_indx-- ) {
      if ( (a>>a_indx) & 1 ) { // shift out bits in constant a
        double angle = M_PI / pow ( 2, b_indx-a_indx ); // rotation angle
        switch (c_width) {
          case 0: Rz ( b[b_indx], angle ); break;
          case 1: cRz ( ctrl0, b[b_indx], angle ); break;
          case 2: ccRz ( ctrl0, ctrl1, b[b_indx], angle ); break;
}}}}}
```

Listing 2: Controlled adder subroutine using QFT.

```
#include "cADD.scaffold" // see Listing 2
#define width 5 // number of qubits
int main () {
  // control qubits unimportant here
  qbit ctrl[2];
  PrepZ ( ctrl[0], 0 );
  PrepZ ( ctrl[1], 0 );

  // initialize quantum variable to 12
  const unsigned int b_val = 12;
  qbit b[width];
  for ( int i=0; i<width; i++ ) {
    PrepZ ( b[i], (b_val>>i)&1 );
  }
  assert_classical( b, width, 12 );

  // perform the addition
  QFT ( width, b );
  const unsigned int a = 13;
  cADD (0,ctrl[0],ctrl[1],width,a,b);
  iQFT ( width, b );

  // assert a+b = 12+13 = 25
  assert_classical ( b, width, 25 );
}
```

Listing 3: Test harness for controlled adder subroutine.

code. Sometimes, programmers do not even have quantum circuit diagrams and must instead start with equation descriptions for the operations they need. This process of converting specifications to program code is unintuitive and tricky. For example, Table 1 lists multiple ways to code the decomposition of the controlled rotation, including a buggy one where a small mistake leads to the wrong operation.

Defense type 2: Assertion checks for unit testing. An obvious defense against coding mistakes in basic subroutines (such as controlled rotation, QFT, and addition subroutines) is to use a library of shared code. Doing so helps ensure program correctness by allowing programmers to exhaustively validate small subroutines, in order to bootstrap larger subroutines. Unit testing is especially important in QC as running or simulating large quantum programs is costly, making larger scale integration tests impossible.

As a concrete example, we use precondition and postcondition assertion checks inside a test harness to validate the QFT subroutine, another important building block. As shown in Listing 1, first the program prepares a classical integer state (Lines 5-9). Then, the program checks as a precondition of the QFT subroutine that the input is a classical integer value, in this case '5' (Line 12). The corresponding postcondition of the QFT subroutine is that the output should be a uniform superposition if the program collapses the quantum state and measure the values at that point (Line 18).

While these simple constraints are not enough on their own to validate that the QFT implementation and its sub-components are correct, they are valuable lightweight sanity checks. For the QFT subroutine, additional validation comes from cross checking its outputs against closed form mathematical solutions, and against implementations in other languages.

4.3 Numeric assertion checks for composing gates with iterations

From the basic subroutines, programmers typically compose the subroutines into quantum programs using patterns including iterations, recursion, and mirroring. Here we focus on iterations, a pattern commonly invoked in code related to the QFT for the purpose of manipulating qubits that represent numbers. Our tool can catch bugs in iteration code using assertions on integer inputs to and outputs from subroutines.

Bug type 3: Incorrect composition of operations using iteration. Now that we have validated code for the controlled rotation and QFT subroutines, the next more complex subroutine is

the controlled adder, which is itself a subroutine for the modular exponentiation part of Shor's algorithm (bottom module in Figure 2). Listing 2 shows the iteration code for the constant-value adder, showing tricky places in Lines 8 through 11 where bugs can crop up. These possible bugs include indexing errors in the two-dimensional loop, bit shifting errors, endian confusion, and mistakes in rotation angles.[3]

Defense type 3: Assertion checks for classical intermediate states. Our tool's assertions on classical integer values allows for unit testing of code that involve iterations. As an example in Listing 3, programmers can write assertions on the inputs (Line 15) and outputs (Line 24) of the controlled adder subroutine. With these assertions programmers can catch coding mistakes made in its constituent subroutines. For example the bug involving the incorrect version of the rotation operation in Table 1 is caught here when the output assertion returns $p-$value $= 0.0$, indicating the addition did not work as expected, due to a bug inside the controlled adder.

4.4 Entanglement assertion checks for composing gates with recursion

The next two types of bugs in quantum programs have to do with two more ways to compose basic operations, both of which have to do with the interaction between quantum variables; i.e., between two or more sets of qubits. That is in contrast to the previous two types of bugs in basic operations and iterating operations, which generally act on single variables (where the variables may comprise multiple qubits).

In quantum computing, the interaction between variables takes place through entanglement. For example, in Figure 2, the upper and lower registers interact when they are entangled through the controlled modular exponentiation operation. If two variables are entangled when a quantum computer measures them, the classical values that they collapse to will be correlated. Using statistical tests on the measurement results, programmers can write assertions to check whether variables are entangled as expected.

Bug type 4: Incorrect composition of operations using recursion. Entanglement is achieved using controlled operations, which is a common pattern in quantum programs that involves performing operations (e.g., modular multiply), contingent on a set of qubits known as control qubits. These controlled operations correspond to using recursion to compose basic operations. A multiply-controlled rotation, for example, is just a controlled rotation that is itself controlled by other qubits (Figure 4).

The process of coding recursive operation patterns may introduce bugs. That is because quantum algorithms often need varying numbers of control qubits in different parts of the algorithm, leading to replicated code from multiple versions of the same subroutine differing only by the number of control qubits. An example appears in Listing 2, where the addition operation is contingent on control qubits taken as parameters in Lines 4 and 5. Depending on how many control qubits are needed, the switch statement in Lines 12 through 15 applies the correct operation. The specific bug we are going to demonstrate catching next is if a programmer made a mistake in Line 15, where they accidentally use `ctrl1` twice instead of `ctrl0`, causing a mistake in how the control qubits are routed.

```
1  #include "cMODMUL.scaffold"
2  #define width 5 // number of qubits
3  #define N 15 // number to factor
4
5  // CALCULATE: b <= a*x+b mod N
6  int main () {
7
8    // control qubit in superposition
9    qbit ctrl[1];
10   PrepZ ( ctrl[0], 1 );
11   H ( ctrl[0] );
12
13   // initialize x variable to 6
14   const unsigned int x_val = 6;
15   qbit x[width];
16   for ( int i=0; i<width; i++ ) {
17     PrepZ ( x[i], (x_val>>i)&1 );
18   }
19   assert_classical ( x, width, 6 );
20
21   // initialize b variable to 7
22   const unsigned int b_val = 7;
23   qbit b[width];
24   for ( int i=0; i<width; i++ ) {
25     PrepZ ( b[i], (b_val>>i)&1 );
26   }
27   assert_classical ( b, width, 7 );
28
29   // ancillary qubits unimportant here
30   qbit ancilla[1];
31   PrepZ ( ancilla[0], 0 );
32
33   // perform modular multiplication
34   const unsigned int a = 7;
35   cMODMUL ( ctrl[0], width, a, x, b, N,
       ancilla[0] );
36
37   assert_entangled( ctrl,1, b,width );
38
39   // inverse modular multiplication
40   const unsigned int a_inv = 13;
41   cMODMUL ( ctrl[0], width, a_inv, x, b, N,
       ancilla[0] );
42
43   assert_product( ctrl,1, b,width );
44 }
```

Listing 4: Test harness for the controlled modular multiplier subroutine.

[3] One of the trickiest aspects of quantum programming is properly keeping track of how quantum variables map to qubit assignments. One way to prevent bugs altogether in this kind of code is to introduce QC data types for numbers, providing greater abstraction than working with raw qubits. For example, ProjectQ has quantum integer data types [47], while Q# [48] and Quipper [9, 51] offer both big endian and little endian versions of subroutines involving iterations. These QC data types permit useful operators (e.g., checking for equality) that help with debugging and writing assertions.

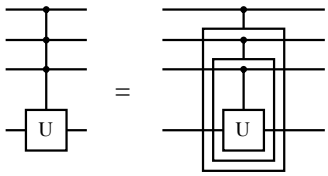

Figure 4: Controlled operations with multiple control qubits result in recursive code patterns.

Defense type 4: Assertion checks for entangled intermediate states. Programmers can check for these types of bugs in recursive code patterns for controlled operations using entanglement assertions, a new kind of quantum assertion that we introduce to test for dependence between measured values.

As a very simple example, we show how the entanglement assertion check works on the simplest example of entangled states. In the Bell state creation circuit we showed in Figure 1, the state of the two qubits Q in location (D) of that diagram are in a *Bell state*, a minimal example of an entangled state between qubits. The measurement results m_0 and m_1 are maximally correlated—either both return '0' or both return '1'. Using such observations one can build a contingency table:

Probability		m_0 measurement	
		0	1
m_1	0	1/2	0
measurement	1	0	1/2

Next we again use a chi-square statistical test on the table to determine a contingency coefficient. If the p-value is small (≤ 0.05), as is the case for this table, then the test rejects the null hypothesis and concludes the observations must be correlated, and therefore the quantum variables were entangled when they were measured. On the other hand, if the p-value fails to be significant, then the observations are consistent with the variables being independent and unentangled.

These entanglement assertions are powerful tools for catching bugs such as our example bug of mistaken control qubits in the controlled adder subroutine. Our tool catches the bug using entanglement assertions in the controlled modular multiplier test harness, shown in Listing 4. To prepare the contingency table, the programmer only needs to identify pairs of quantum variables that should be entangled with each other using the `assert_entangled` statement (Line 37), which takes four parameters specifying the control and target quantum variables and their bitwidths. Then, our debugging tool keeps track of which qubits those specified variables correspond to. The simulator then does an early measurement of the qubits for both variables. The debugging tool then maps the measurement results into columns and rows of a contingency table automatically, and a chi-square test checks to make sure the control qubits have an effect on whether the multiplier acts on the target qubits.

If the controlled add operation is bug-free, with the control qubits correctly routed, the first assertion returns $p-$value $= 0.0005$ for an ensemble size of 16, indicating the control and target register values are entangled at the point of the assertion. That means that whichever way the control qubit collapses out of its superposition state, it correctly controls whether the multiplication works on the target register. On the other hand, if the control qubits are routed incorrectly, the first assertion returns $p-$value $= 0.121$ for an ensemble size of 16. This indicates the control register value is not correctly toggling the operation of the multiplier, hinting the bug must be somewhere inside the multiplier implementation.

Table 2: Correct classical input a and a^{-1} to Shor's algorithm for factoring 15, using 7 as a guess.

k, **the algorithm iteration**	**0**	**1**	**2**	**3**	**...**
$a = 7^{2^k} \mod 15$	7	4	1	1	...
$a^{-1}; a \times a^{-1} \equiv 1 \mod 15$	13	4	1	1	...

4.5 Product state postcondition assertions for composing gates with mirroring

Contingency table analysis is also useful for checking for the third and final kind of pattern in quantum programs, the correct mirroring of operations. The reason this pattern appears in quantum programs is to allocate and deallocate qubits within a quantum subroutine, analogous to the allocation and garbage collection of memory in classical programs. For example, the Shor's algorithm in Figure 2 can be seen as allocating the bottom register of qubits (known as ancillary qubits) in the left half of the algorithm, performing the modular exponentiation, and then deallocating the bottom register of qubits in the right half of the algorithm. Product state assertions validate that the deallocation of these ancillary qubits is done correctly.

Bug type 5: Incorrect composition of operations using mirroring. In order to garbage collect ancillary qubits in quantum programs, programmers need to reverse all the operations they applied to the qubits.

The reason programmers have to do so is because garbage collection is different in quantum computing compared to that in classical computing. In classical computing, programmers can simply mark any memory as unneeded in order to free it, and that memory would be rewritten some time later in program execution. But in quantum computing, qubits can be entangled and therefore cannot be treated as independent pieces of information. Suppose a program is done with using the lower register in Figure 2, but they remain entangled with the upper register qubits. Then anything that happens to the ancillary qubits, such as measurement, re-initialization, or lapsing into incoherence, can have unintended effects on the output qubits in the upper register that the program user does care about.

To prevent these unintended side effects, programmers have to carefully undo any entanglement they have built up between qubits. To do this, programmers perform inverse operations in backward order from the order they originally performed them. This process is called *uncomputation* [13, 18, 35]. After uncomputation, ancillary qubits should be properly untangled from the rest of the program state, and are truly ready for reuse.

This process of uncomputation can be tricky if done manually. Take for example the controlled adder subroutine shown in Listing 2. Uncomputing the addition operation would need an inverse adder

Table 3: Probability of measuring values of outputs and ancillary qubits of Shor's algorithm, with incorrect inputs ($a^{-1} = 12$ instead of 13 on first iteration). If the ancillary qubits collapse to zero on measurement, the algorithm still succeeds, returning correct outputs of 0, 2, 4, 6 [35, p. 235]. However, the possibility of measuring non-zero for the ancillary qubits indicates a bug.

Probability		Output measurement							
		0	1	2	3	4	5	6	7
Ancillary qubits measurement	0	1/8	0	1/8	0	1/8	0	1/8	0
	2	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	7	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	8	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	13	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64

counterpart to the controlled adder. The code for the inverse adder would have each of the iterations in Lines 8 and 9 iterated in reverse order, and would have the rotation angles used in Lines 13 through 15 negated. Bugs in these inverse operations would impact the qubit deallocation process.

Defense type 5: Assertion checks for product state postconditions. As a counterpart to entanglement assertions, our tool offers product state assertions to make sure that ancillary qubits and output qubits are in a product state, meaning they have no entanglement. This kind of assertion would make sure that code for the pattern of mirroring operations is correct.

We demonstrate the use of product state assertions also in Listing 4. Following the controlled modular multiplier in Line 35, the program reverses that operation in Line 41. The way the program invokes the inverse operation in Line 41 is by multiplying by the modular inverse. In the example here, $7{\times}13 \equiv 1 \bmod 15$, so multiplying by $a^{-1} = 13$ inverts the operation of multiplying by $a = 7$. With the correct inverse computation, the `assert_product` statement in Line 43 returns $p-$value $= 1.0$, consistent with no entanglement between the upper control register and the bottom target register, indicating the bottom register is properly deallocated.

If on the other hand the program mistakenly multiplies by any number that is not the modular inverse, for example $a^{-1} = 12$, then the assertion returns $p-$value $= 0.0005$ (for an ensemble size of 16) indicating the two registers are still incorrectly entangled, meaning the bottom register was not correctly deallocated, which hints to a bug in the mirrored code.

4.6 Classical postcondition assertions on deallocated ancillary qubits

Finally, we are ready to run the Shor's algorithm in an overall integration test. To run Shor's algorithm, the programmer has to feed the algorithm pairs of modular inverse numbers as its input. For example, Table 2 shows the input pairs for factoring 15, using 7 as a trial divisor. Then, the algorithm should return 0, 2, 4, or 6, each with equal probability, from measuring the upper register [35, p. 235]. These numbers would go into a classical post-processing algorithm to find the factors of $3 \times 5 = 15$.

Typically, programmers would only measure the upper register of qubits (Figure 2) that carry program output, and ignore the bottom register of qubits as they are merely ancillary qubits and should carry no information. However, when a programmer is debugging a quantum program, these ancillary qubits often carry useful side channel information that informs the programmer whether the main outputs are valid. Our tool checks for this information using classical assertions on the expected values for these deallocated ancillary qubits.

Bug type 6: Incorrect classical input parameters. The final bug we study for Shor's algorithm stems from giving wrong input parameters to an otherwise correctly written quantum program. These mistakes can be difficult to debug, even though the bug is entirely in the classical inputs to the algorithm.

The specific mistake is the programmer supplies wrong pairs of numbers as modular inverses for the algorithm. Instead of using $(a, a^{-1}) = (7, 13)$ for the first iteration in Table 2, the programmers gives a wrong pair of numbers $(7, 12)$. We show our tool can debug this problem using assertions.

Defense type 6: Assertion checks for classical postconditions. The outputs of Shor's algorithm for this incorrect pair of inputs is recorded in Table 3. The table is a contingency table showing the joint probability for the output measurement and the ancillary qubit measurement. The table shows the ancillary qubits collapse to a non-zero value with probability $1/2$, which is incorrect because they should always return to their initial value of 0 after appropriate uncomputation. This symptom is to be expected because the incorrect pair of modular inverses fed to the algorithm has caused incorrect inversion of the multiplication operation inside the algorithm.

The programmer can use a classical assertion as a postcondition check on the deallocated ancillary qubits. The program should assert that the ancillary qubits should return their initial value of 0. If the postcondition assertion fails, the programmer knows there was a bug in the deallocation of qubits and therefore the outputs may be wrong. If the postcondition succeeds, then the Shor's factoring algorithm returns valid outputs.

5 QC PROGRAM DEBUGGING ACROSS ALGORITHM PRIMITIVES

This section shifts focus away from the Shor's algorithm case study and presents two additional debugging case studies. The goal is to understand whether the debugging techniques for Shor's algorithm generally apply to other classes of algorithms.

Table 4: Grover's amplitude amplification subroutine in two languages, showcasing QC-specific language syntax for reversible computation (rows 2 & 6) and controlled operations (rows 3 & 5), exposing structure that can guide placing assertions.

	Scaffold (C syntax) [17]	ProjectQ (Python syntax) [47]

1	`int j;`
	`qbit ancilla[n-1]; // scratch register`
	`for(j=0; j<n-1; j++) PrepZ(ancilla[j],0);`
		`# reflection across`
		`# uniform superposition`

2	`// Hadamard on q`
	`for(j=0; j<n; j++) H(q[j]);`
	`// Phase flip on q = 0...0 so invert q`
	`for(j=0; j<n; j++) X(q[j]);`
		`with Compute(eng):`
		` All(H) | q`
		` All(X) | q`

3	`// Compute x[n-2] = q[0] and ... and q[n-1]`
	`CCNOT(q[1], q[0], ancilla[0]);`
	`for(j=1; j<n-1; j++)`
	` CCNOT(ancilla[j-1], q[j+1], ancilla[j]);`
		`with Control(eng, q[0:-1]):`

4	`// Phase flip Z if q=00...0` `cZ(ancilla[n-2], q[n-1]);`	` Z | q[-1]`

5	`// Undo the local registers`
	`for(j=n-2; j>0; j--)`
	` CCNOT(ancilla[j-1], q[j+1], ancilla[j]);`
	`CCNOT(q[1], q[0], ancilla[0]);`
		`# ProjectQ automatically`
		`# uncomputes control`

6	`// Restore q`
	`for(j=0; j<n; j++) X(q[j]);`
	`for(j=0; j<n; j++) H(q[j]);`
		`Uncompute(eng)`

In the Shor's case study, we argued how the structure of the algorithm code guides the placement of assertions. Our methodology for debugging the algorithm was to bring up the subroutines from unit tests to full integration tests. We used assertions to check for preconditions, intermediate states, and postconditions of subroutines. Furthermore the code patterns of how subroutines are composed further guided what assertions to use. A natural question is whether that rigorous methodology is helpful for debugging other algorithms.

Many different quantum algorithms rely on a handful of QC algorithm primitives to get speedups relative to classical algorithms [5, 31, 32]. These algorithm primitives are akin to algorithm kernels in the context of classical algorithms. Each algorithm type has distinct pitfalls and features that lead to distinct bugs and possible defenses.

This section covers two more algorithms that use completely different algorithm primitives. The first is Grover's database search algorithm based on the amplitude amplification primitive. The second is a quantum chemistry problem that uses quantum operations to simulate a physical system. This represents a broad selection of different quantum algorithm primitives.

While we have not covered in this paper some algorithm primitives (such as adiabatic algorithms, approximate optimization algorithms, and much less prominent primitives such as quantum random walks), the three areas we have covered represent the most important and well-studied algorithm classes.

5.1 Case study: Grover's database search

This section uses the Grover's benchmark to discuss how language syntax support for reversible computation and controlled operations guides placement of assertions.

5.1.1 Language support for placement of entanglement assertions. Higher-level quantum programming language features can help *automatically* place `assert_entangled` and `assert_product` assertions. We concentrate on the placement of these two assertion types because they are assertions on the relationship between two or more quantum variables. As such they are powerful debugging tools, but they also need the most programmer insight to correctly place them.

As we discussed in Sections 4.4 and 4.5, entanglement assertions are closely related to the quantum program patterns of recursion and mirroring. In the Scaffold language, these patterns are not explicitly captured by the C-style syntax, but in higher-level quantum programming languages, such as ProjectQ [47] and Q# [48], these patterns are essential to the language design. With these language features, the placement of entanglement assertions becomes as natural as placing precondition and postcondition assertions.

5.1.2 The Grover's algorithm for database search. The Grover's search algorithm finds an entry that matches search criteria, among an input data set of size N, with a time cost on the order of $\sqrt{N}$. That represents a polynomial speedup relative to the linear time cost in a classical computer [10].

The Grover's algorithm comprises three parts. First, the input qubits representing the indices of the matching entries are put in a state of superposition, akin to querying all entries at once. A superposition assertion (Section 4.1) helps certify that this algorithm precondition is satisfied. Second, the queries are put through a subroutine that checks for the search criteria. In our case study, our criteria is to find the square root of a number in a Galois field of two elements, a simple abstract algebra setting. Finally in the critical third step, the *amplitude amplification* algorithm primitive amplifies the index that matches the criteria while damping out those that do not.

5.1.3 Entanglement program patterns in the amplitude amplification subroutine. Table 4 shows the reversible computation and controlled operations program patterns coded in two quantum programming languages Scaffold [17] and ProjectQ [47]. The ProjectQ language has syntax support for reversible computation that automatically mirrors and inverts sequences of operations. Likewise, syntax support for controlled operations automatically allocates the ancillary qubits needed for controlled operations.

These higher-level language support for these patterns allows automatic placement of assertions: the controlled operation statement in rows 3 through 5 indicates that register q should be entangled in row 4, so it would be the right place to place an entanglement assertion. Furthermore the compute-uncompute pattern in Rows 2 and 6 hints at a product state assertion at the end of uncomputation.

5.2 Case study: Quantum chemistry

Next, we discuss our experience building up and debugging a simple quantum chemistry program. Quantum chemistry problems entail finding properties of molecules from theoretical first principles [26, 36]. Researchers anticipate these will be the first applications for QC due to the relatively few number of qubits they need to surpass classical computer algorithms. Debugging these problems is distinctively challenging, due to the importance of getting a large number of classical input parameters all correct, and because of the dearth of physically meaningful intermediate states we can check in the course of algorithm execution.

5.2.1 Classical input parameters. A key part of quantum chemistry programs is in correctly building up a *Hamiltonian* subroutine that simulates inter-electron forces. The procedure for doing this was laid out in detail by Whitfield [54]. We followed this procedure to create a subroutine for simulating the hydrogen molecule, but we needed additional cross-validation from several other sources to get a bug-free subroutine [53]. These resources include raw chemistry data found in open source repositories for the LIQUi|> framework[4]. The final parameters for actual operations on qubits were validated against a follow-up paper [45] and an implementation in the QISKit framework[5]. Because the procedure for preparing these quantum chemistry models involves many steps and needs domain expertise, arguably this step in preparing classical input parameters is the hardest aspect to debug.

Once the Hamiltonian subroutine is built, we can use the model in a variety of quantum algorithms spanning different primitives. These include phase estimation (an application of quantum Fourier transforms) [38], variational quantum eigensolvers [40], and adiabatic algorithms [1]. In this case study, we use iterative phase estimation to find the ground state energy of our H_2 model, validating results published by Lanyon [22].

5.2.2 Assertions on quantum intial values and final states. The correct preparation of qubit initial values stands out as another challenging aspect of debugging quantum chemistry QC programs. Incorrect initial values would cause the program to find solutions to different problems altogether. In this quantum chemistry problem, the initial values control the locations of the two electrons in H_2. As shown in Table 5, precondition assertions check the qubit assignment for finding the ground energy of H_2, while other assignments lead to results for other energy levels.

The symmetry of H_2 allows us to perform a sanity check, to make sure the Hamiltonian and the iterative phase estimation subroutines are working correctly. Though there are six ways to assign two electrons to four locations, there are in fact only four distinct energy levels, as shown in the experimental data (Table 5). Postcondition assertions are useful for checking that the two different ways to obtain E1 (and E2) give the same energy levels. These assertions validate that the model correctly preserves symmetry.

5.2.3 Assertions on intermediate algorithm progress. Unlike the other two case studies in this paper, the debugging process for the quantum chemistry benchmark is coarse-grained. That is because the Hamiltonian subroutine is a monolithic block of code whose components do not have obvious expected outputs—its components represent pair-wise electron interactions, and do not have inherent physical meaning. So how do we debug this program? The preconditions in the last subsection make sure the inputs to the algorithm are correct; the other observable state we have for debugging is to check the behavior of the algorithm as a whole.

In this quantum chemistry program, we can check for two types of overall algorithm behavior. One is the solution should converge to a steady value as finer Trotter time steps (a kind of numerical approximation) are chosen; a lack of this type of convergence indicates a bug in the Hamiltonian subroutine. The other algorithm behavior is when we vary the precision of the phase estimation algorithm, the most significant bits of the measurement output sequences should be the same—in other words, rounding the output of a high-precision experiment should yield the same output as a lower-precision experiment. a lack of this convergence indicates a bug in the iterative phase estimation subroutine. These checks for expected algorithm progress also apply to other algorithms.

6 RELATED WORK

Approaches to writing correct quantum programs range from formal methods to less-formal pragmatic methods, much like in classical programming. Most of the prior work in quantum program correctness has been in formal methods–e.g., using theorem provers and type checking to verify programs correctly match algorithm specifications [16, 39, 43, 56, 57]. Such verification techniques consider the correctness problem from a top-down perspective. While

[4] https://github.com/StationQ/Liquid/blob/master/Samples/h2_sto3g_4.dat
[5] https://github.com/Qiskit/aqua/blob/master/test/H2-0.735.json

Table 5: QC calculated energy for H_2 (bond length = 73.48 pm) for different electron assignments.

	Electron assignments				QC calculated energy (relative)
	Bonding		Antibonding		
	↑	↓	↑	↓	
3rd excited state (E3)	0	0	1	1	-0.164
2nd excited state (E2)	0	1	1	0	-0.217
	1	0	0	1	
1st excited state (E1)	0	1	0	1	-0.244
	1	0	1	0	
Ground state (G)	1	1	0	0	-0.295

useful, formal methods should not be the only approach to correct programs; more traditional debugging strategies are also useful. This work considers the possibility of using pragmatic assertion checks to build code bottom-up from exhaustive unit tests up through integration testing.

Quantum program assertions exist in several quantum programming languages, though not in the same capacity as the statistical assertions presented in this work. First, the Quipper language [9] features *assertive termination*, which allows the programmer to annotate known program state, in order to drive compiler optimizations. Their use relies on the programmer to write correct code and assertions, and cannot be used as postcondition checks. Second, the Q# programming language [48] allows programmers to write assertions on classical, integer states. These assertions are then checked during the simulation of the quantum programs. This work extends that set of assertions with assertions on superposition and entanglement states.

The set of quantum states considered in these assertions, namely classical, superposition, and entangled states, are a subset of possible quantum states. In the general case we need quantum phase estimation, quantum state tomography, and quantum process tomography to be able to examine general quantum states [35]. However, those processes are extremely costly and cannot be used as efficient assertion checks.

7 CONCLUSION

For the first time, we have access to program benchmarks for several major areas of quantum algorithms, along with input datasets and outputs that are detailed enough to permit detailed debugging and cross-validation. Using our experience in building up and debugging these programs, we presented in this paper a strategy for deploying and checking quantum program assertions based on statistical tests. Drawing on the structure of quantum programs, we point to where and how program bugs may arise, and point to how the presented assertions can catch them.

ACKNOWLEDGMENTS

This work is funded in part by EPiQC, an NSF Expedition in Computing, under grant 1730082.

REFERENCES

[1] R. Barends, A. Shabani, L. Lamata, J. Kelly, A. Mezzacapo, U. Las Heras, R. Babbush, A. G. Fowler, B. Campbell, Yu Chen, Z. Chen, B. Chiaro, A. Dunsworth, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, P. J. J. O'Malley, C. Quintana, P. Roushan, D. Sank, A. Vainsencher, J. Wenner, T. C. White, E. Solano, H. Neven, and John M. Martinis. 2016. Digitized adiabatic quantum computing with a superconducting circuit. *Nature* 534 (08 06 2016), 222 EP –. http://dx.doi.org/10.1038/nature17658

[2] Stephane Beauregard. 2003. Circuit for Shor's Algorithm Using 2N+3 Qubits. *Quantum Info. Comput.* 3, 2 (March 2003), 175–185. http://dl.acm.org/citation.cfm?id=2011517.2011525

[3] Sergio Boixo, Sergei V. Isakov, Vadim N. Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J. Bremner, John M. Martinis, and Hartmut Neven. 2018. Characterizing quantum supremacy in near-term devices. *Nature Physics* 14, 6 (2018), 595–600. https://doi.org/10.1038/s41567-018-0124-x

[4] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. 2017. Programming languages and compiler design for realistic quantum hardware. *Nature* 549 (13 09 2017), 180 EP –. http://dx.doi.org/10.1038/nature23459

[5] Patrick J. Coles, Stephan Eidenbenz, Scott Pakin, Adetokunbo Adedoyin, John Ambrosiano, Petr M. Anisimov, William Casper, Gopinath Chennupati, Carleton Coffrin, Hristo Djidjev, David Gunter, Satish Karra, Nathan Lemons, Shizeng Lin, Andrey Y. Lokhov, Alexander Malyzhenkov, David Mascarenas, Susan M. Mniszewski, Balu Nadiga, Dan O'Malley, Diane Oyen, Lakshman Prasad, Randy Roberts, Philip Romero, Nandakishore Santhi, Nikolai Sinitsyn, Pieter Swart, Marc Vuffray, Jim Wendelberger, Boram Yoon, Richard J. Zamora, and Wei Zhu. 2018. Quantum Algorithm Implementations for Beginners. *CoRR* abs/1804.03719 (2018). arXiv:1804.03719 http://arxiv.org/abs/1804.03719

[6] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. 2017. Open Quantum Assembly Language. *ArXiv e-prints* (July 2017). arXiv:quant-ph/1707.03429

[7] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. *arXiv e-prints*, Article arXiv:1411.4028 (Nov 2014), arXiv:1411.4028 pages. arXiv:quant-ph/1411.4028

[8] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. 2000. Quantum Computation by Adiabatic Evolution. *arXiv e-prints*, Article quant-ph/0001106 (Jan 2000), quant-ph/0001106 pages. arXiv:quant-ph/quant-ph/0001106

[9] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 333–342. https://doi.org/10.1145/2491956.2462177

[10] Lov K Grover. 2001. From Schrödinger's equation to the quantum search algorithm. *Pramana* 56, 2-3 (2001), 333–348.

[11] Thomas Häner and Damian S. Steiger. 2017. 0.5 Petabyte Simulation of a 45-qubit Quantum Circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, Article 33, 10 pages. https://doi.org/10.1145/3126908.3126947

[12] Thomas Häner, Damian S. Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High Performance Emulation of Quantum Circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 74, 9 pages. http://dl.acm.org/citation.cfm?id=3014904.3015003

[13] Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. 2018. A software methodology for compiling quantum programs. *Quantum Science and Technology* 3, 2 (2018), 020501. http://stacks.iop.org/2058-9565/3/i=2/a=020501

[14] Aram W. Harrow and Ashley Montanaro. 2017. Quantum computational supremacy. *Nature* 549 (13 09 2017), 203 EP –. https://doi.org/10.1038/nature23458

[15] Yipeng Huang and Margaret Martonosi. 2019. QDB: From Quantum Algorithms Towards Correct Quantum Programs. In *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018) (OpenAccess Series in Informatics (OASIcs))*, Titus Barik, Joshua Sunshine, and Sarah Chasins (Eds.), Vol. 67. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany,

4:1–4:14. https://doi.org/10.4230/OASIcs.PLATEAU.2018.4

[16] Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. 2019. Quantitative Robustness Analysis of Quantum Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 31 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290344

[17] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. ACM, 1:1–1:10. https://doi.org/10.1145/2597917.2597939

[18] Phillip Kaye, Raymond Laflamme, and Michele Mosca. 2007. *An Introduction to Quantum Computing*. Oxford University Press, Inc.

[19] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels. 2017. QX: A High-performance Quantum Computer Simulation Platform. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '17)*. 464–469. http://dl.acm.org/citation.cfm?id=3130379.3130487

[20] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien. 2010. Quantum computers. *Nature* 464 (04 03 2010), 45 EP –. https://doi.org/10.1038/nature08812

[21] B. P. Lanyon, T. J. Weinhold, N. K. Langford, M. Barbieri, D. F. V. James, A. Gilchrist, and A. G. White. 2007. Experimental Demonstration of a Compiled Version of Shor's Algorithm with Quantum Entanglement. *Phys. Rev. Lett.* 99 (Dec 2007), 250505. Issue 25. https://doi.org/10.1103/PhysRevLett.99.250505

[22] B. P. Lanyon, J. D. Whitfield, G. G. Gillett, M. E. Goggin, M. P. Almeida, I. Kassal, J. D. Biamonte, M. Mohseni, B. J. Powell, M. Barbieri, A. Aspuru-Guzik, and A. G. White. 2010. Towards quantum chemistry on a quantum computer. *Nature Chemistry* 2 (10 01 2010), 106 EP –. http://dx.doi.org/10.1038/nchem.483

[23] Ryan LaRose. 2019. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum* 3 (March 2019), 130. https://doi.org/10.22331/q-2019-03-25-130

[24] Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. 2017. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3305–3310. https://doi.org/10.1073/pnas.1618020114 arXiv:http://www.pnas.org/content/114/13/3305.full.pdf

[25] I. Markov and Y. Shi. 2008. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM J. Comput.* 38, 3 (2008), 963–981. https://doi.org/10.1137/050644756 arXiv:https://doi.org/10.1137/050644756

[26] S. McArdle, S. Endo, A. Aspuru-Guzik, S. Benjamin, and X. Yuan. 2018. Quantum computational chemistry. *ArXiv e-prints* (Aug. 2018). arXiv:quant-ph/1808.10402

[27] J. R. McClean, I. D. Kivlichan, K. J. Sung, D. S. Steiger, Y. Cao, C. Dai, E. Schuyler Fried, C. Gidney, B. Gimby, P. Gokhale, T. Häner, T. Hardikar, V. Havlíček, C. Huang, J. Izaac, Z. Jiang, X. Liu, M. Neeley, T. O'Brien, I. Ozfidan, M. D. Radin, J. Romero, N. Rubin, N. P. D. Sawaya, K. Setia, S. Sim, M. Steudtner, Q. Sun, W. Sun, F. Zhang, and R. Babbush. 2017. OpenFermion: The Electronic Structure Package for Quantum Computers. *ArXiv e-prints* (Oct. 2017). arXiv:quant-ph/1710.07629

[28] N.D. Mermin. 2007. *Quantum Computer Science: An Introduction*. Cambridge University Press.

[29] Tzvetan S. Metodi, Arvin I. Faruque, and Frederic T. Chong. 2011. Quantum Computing for Computer Architects, Second Edition. *Synthesis Lectures on Computer Architecture* (2011). https://doi.org/10.2200/S00331ED1V01Y201101CAC013 arXiv:https://doi.org/10.2200/S00331ED1V01Y201101CAC013

[30] D. M. Miller, M. A. Thornton, and D. Goodman. 2006. A Decision Diagram Package for Reversible and Quantum Circuit Simulation. In *2006 IEEE International Conference on Evolutionary Computation*. 2428–2435. https://doi.org/10.1109/CEC.2006.1688610

[31] Ashley Montanaro. 2016. Quantum algorithms: An overview. *NPJ Quantum Information* 2 (2016), 15023.

[32] Michele Mosca. 2009. Quantum algorithms. In *Encyclopedia of Complexity and Systems Science*. Springer, 7088–7118.

[33] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. 2019. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, 1015–1029. https://doi.org/10.1145/3297858.3304075

[34] National Academies of Sciences, Engineering, and Medicine. 2019. *Quantum Computing: Progress and Prospects*. The National Academies Press, Washington, DC. https://doi.org/10.17226/25196

[35] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press.

[36] Jonathan Olson, Yudong Cao, Jonathan Romero, Peter Johnson, Pierre-Luc Dallaire-Demers, Nicolas Sawaya, Prineha Narang, Ian Kivlichan, Michael Wasielewski, and Alán Aspuru-Guzik. 2017. Quantum Information and Computation for Chemistry. *arXiv e-prints*, Article arXiv:1706.05413 (Jun 2017), arXiv:1706.05413 pages. arXiv:quant-ph/1706.05413

[37] Scott Pakin. 2019. Targeting Classical Code to a Quantum Annealer. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, 529–543. https://doi.org/10.1145/3297858.3304071

[38] S. Patil, A. JavadiAbhari, C. Chiang, J. Heckey, M. Martonosi, and F. T. Chong. 2014. Characterizing the performance effect of trials and rotations in applications that use Quantum Phase Estimation. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 181–190. https://doi.org/10.1109/IISWC.2014.6983057

[39] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 846–858. https://doi.org/10.1145/3009837.3009894

[40] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5 (23 07 2014), 4213 EP –. http://dx.doi.org/10.1038/ncomms5213

[41] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. https://doi.org/10.22331/q-2018-08-06-79

[42] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing* (3 ed.). Cambridge University Press.

[43] Robert Rand. 2018. *Formally Verified Quantum Programming*. Ph.D. Dissertation. University of Pennsylvania.

[44] M. Roetteler, K. M. Svore, D. Wecker, and N. Wiebe. 2017. Design automation for quantum architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 1312–1317. https://doi.org/10.23919/DATE.2017.7927196

[45] Jacob T. Seeley, Martin J. Richard, and Peter J. Love. 2012. The Bravyi-Kitaev transformation for quantum computation of electronic structure. *The Journal of Chemical Physics* 137, 22 (2012), 224109. https://doi.org/10.1063/1.4768229

[46] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (Oct. 1997), 1484–1509. https://doi.org/10.1137/S0097539795293172

[47] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: An open source software framework for quantum computing. *Quantum* 2 (Jan. 2018), 49. https://doi.org/10.22331/q-2018-01-31-49

[48] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. ACM, Article 7, 10 pages. https://doi.org/10.1145/3183895.3183901

[49] K. M. Svore and M. Troyer. 2016. The Quantum Future of Computation. *Computer* 49, 9 (Sept. 2016), 21–30. https://doi.org/10.1109/MC.2016.293

[50] Swamit S. Tannu and Moinuddin K. Qureshi. 2019. Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, 987–999. https://doi.org/10.1145/3297858.3304007

[51] Benoît Valiron, Neil J. Ross, Peter Selinger, D. Scott Alexander, and Jonathan M. Smith. 2015. Programming the Quantum Future. *Commun. ACM* 58, 8 (July 2015), 52–61. https://doi.org/10.1145/2699415

[52] George F. Viamontes, Igor L. Markov, and John P. Hayes. 2009. *Quantum Circuit Simulation* (1st ed.). Springer Publishing Company, Incorporated.

[53] Dave Wecker, Bela Bauer, Bryan K. Clark, Matthew B. Hastings, and Matthias Troyer. 2014. Gate-count estimates for performing quantum chemistry on small quantum computers. *Phys. Rev. A* 90 (Aug 2014), 022305. Issue 2. https://doi.org/10.1103/PhysRevA.90.022305

[54] J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik. 2011. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics* 109 (March 2011), 735–750. https://doi.org/10.1080/00268976.2011.552441 arXiv:quant-ph/1001.3855

[55] Xin-Chuan Wu, Sheng Di, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T. Chong. 2018. Amplitude-Aware Lossy Compression for Quantum Circuit Simulation. *arXiv e-prints*, Article arXiv:1811.05140 (Nov 2018), arXiv:1811.05140 pages. arXiv:quant-ph/1811.05140

[56] Mingsheng Ying. 2012. Floyd–Hoare Logic for Quantum Programs. *ACM Trans. Program. Lang. Syst.* 33, 6, Article 19 (Jan. 2012), 49 pages. https://doi.org/10.1145/2049706.2049708

[57] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. 2017. Invariants of Quantum Programs: Characterisations and Generation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 818–832. https://doi.org/10.1145/3009837.3009840

[58] A. Zulehner and R. Wille. 2019. Advanced Simulation of Quantum Computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (May 2019), 848–859. https://doi.org/10.1109/TCAD.2018.2834427