

Formal Verification vs. Quantum Uncertainty

Robert Rand 

University of Maryland, College Park, USA

<http://www.cs.umd.edu/~rrand/>


rrand@cs.umd.edu

Kesha Hietala 

University of Maryland, College Park, USA

<https://www.cs.umd.edu/people/khietala>

kesha@cs.umd.edu

Michael Hicks 

University of Maryland, College Park, USA

<http://www.cs.umd.edu/~mwh/>

mwh@cs.umd.edu

Abstract

Programming a quantum computer is difficult and writing a program that will execute successfully on quantum devices that exist today (or are likely to exist in the near future) is a daunting task. Not only is quantum computing inherently uncertain, the quantum computers that we have introduce a variety of novel errors that are difficult to predict or work around. Techniques from formal verification will allow us to quantify and mitigate these errors if we can bridge the gap between high level languages and machine specifications. In this paper, we review existing approaches to quantum program verification and propose a new approach focused not only on long term quantum programming, but on the quantum programs we can run today.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Hardware → Quantum error correction and fault tolerance

Keywords and phrases Formal Verification, Quantum Computing, Programming Languages, Quantum Error Correction, NISQ

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding All authors are funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040

Robert Rand: Also partly funded by a Victor Basili Postdoctoral Fellowship

Acknowledgements We would like to acknowledge our co-authors on work reviewed here, including Jennifer Paykin, Dong-Ho Lee, Steve Zdancewic, Shih-Han Hung, Shaopeng Zhu, Mingsheng Ying, and Xiaodi Wu.



© Robert Rand, Kesha Hietala and Michael Hicks;
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:13

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

[Kesha: Reviewer comments below.]

■ **Reviewer A:** I confess that I am unconvinced: I don't actually see what the "proposed vision" is. S2 summarizes quantum computing; S3 discusses "verification under ideal conditions" (using the authors' Qwire tool embedded in Coq); S4 points out that real world conditions are not ideal, so the problem is harder; S5 points out that you'd prefer a correct-by-construction approach to a construct-then-verify one. This material indeed sketches the issues, but does not to my mind put forward any of the "visionary ideas", "well-argued challenges", "unexpected connections" etc identified in the call for papers (let alone sketch any actual solutions, which anyway would be out of scope for SNAPL).

■ **Reviewer A:** For debugging quantum programs, perhaps it would be fruitful to make the comparison with debugging logic programs and lazy functional programs. There too you cannot rely on breakpoints and printlns, because the order of evaluation is hard to predict from the static source. You can't simply print an intermediate value in a quantum program, because that collapses the quantum state; similarly, you can't simply print an intermediate value in a lazy functional program, because that forces evaluation of something that might not need to be evaluated.

■ **Reviewer A:** [Re: Sec. 4.2] Your error estimates presumably also assume that errors across qubits are independent?

■ **Reviewer A:** [Re: Sec. 5] You could mention also Dijkstra's argument for developing the proof of correctness of a program hand in hand with the program itself: (a) you might as well use the proof to help you in constructing the program; and (b) if you do the program first, there is likely no proof anyway, because the program is likely wrong.

■ **Reviewer A:** [Re: Sec. 5] I should have thought a better analogy than exponential running time was numerical instability.

■ **Reviewer B:** [The paper] does not present a self-contained introduction to quantum programming. I came into the paper not completely clear on what a quantum program looks like, and I left the paper just as unclear. There is some attempt throughout the paper to give background, but it is not entirely clear that the background helps. For example, Section 2 on logical qubits introduces a little bit of math, but this doesn't really pay off in any useful way later on.

■ **Reviewer B:** Related to the previous point: The paper talks a lot, but is very light on concrete examples. I think adding some concrete examples would really help to make the paper more accessible.

■ **Reviewer B:** The paper does a good job of explaining why developing useful verification tools for quantum programs is hard, but I failed to understand what the authors are actually proposing as the way forward. There is Section 3, which introduces the previous work on Qwire, but it is very brief and high-level, and I wasn't really sure how it connected with the later sections. (It also has a paragraph about "ancillae", which I must confess I couldn't make any sense of.) Sections 4.1 and 4.2 make sense, but end by seemingly suggesting and then criticizing several potential approaches to error management. I wasn't sure what the way forward is. Finally, there is Section 4.3, which unfortunately was completely incomprehensible to me, perhaps again due to lack of examples.

■ **Reviewer B:** The paper induces some whiplash by seeming to go back and forth between saying, "Here is a problem, here is a potential solution, no it doesn't work, this is far from ideal, here is a new section with another problem..." I couldn't get a clear sense of what the authors are arguing for or against.

- 82 ■ **Reviewer C:** One point I did not see clearly discussed is the trade-off between # of
83 qubits and quality of qubits. Handling many bits is easier if the errors allowed are large.
- 84 ■ **Reviewer C:** Another point is the lack of influx of interesting current quantum algorithms
85 – this is a domain where coming up with an algorithm is very difficult and requires deep
86 expert knowledge – this is somewhat different recent Q# contests where the examples
87 are mostly trivial.
- 88 ■ **Reviewer C:** Also, many of the current frameworks (e.g., IBMs) provide standard
89 applications which run on top of libraries containing quantum algorithms (e.g., Grovers).
90 The question remains: what should be verified: the low-level quantum algorithm or the
91 entire application. It was unclear to me what the effect of low-level quantum algorithms
92 on the entire application are, especially given errors. One would expect type systems
93 (enerJ-like) that somehow encapsulate the algorithm.
- 94 ■ **Reviewer C:** I am familiar with QWire, but I am not sure QWire is a high level language
95 – I would consider it lower-level than say Quipper.
- 96 ■ **Reviewer C:** One point I was hoping to see addressed more is the compiler/optimization-
97 s/infrastructure business. There are many optimizations that optimize different criteria,
98 e.g., # of gates vs. types of gates, etc. And this also has connection to the underlying
99 simulator. A good simulator is also lacking these days (Project Q had one and it was
100 doing various optimizations some of which are not well justified or clean). I somehow see
101 the problem of Quantum Optimizing Compilers and Simulators as very important.
- 102 [**Kesha: End of reviewer comments.**]

Writing quantum programs is hard. Fundamentally, a quantum program corresponds to applying a limited set of operations to vectors of complex numbers, with the goal of producing a vector with the majority of its weight in a few meaningful indices. For instance, if you are trying to factor 77, you might want the 7th or 11th entry to contain a number close to 1 while the other indices contain numbers close to 0. As a result, designing a quantum program requires a fair amount of effort and mathematical sophistication. This difficulty is compounded by the fact that quantum programs are very difficult to test or debug.

Consider two standard techniques for debugging programs: breakpoints and print statements. In a quantum program, printing the value of a quantum bit entails measuring it and printing the returned value. This is akin to randomly and irreversibly coercing a floating point number to a nearby integer—it will give a weakly informative answer and corrupt the rest of the program. Unit tests are similarly of limited value when your program is probabilistic; repeatedly running unit tests on a quantum computer is likely to be prohibitively expensive. Simulating quantum programs on a classical computer holds some promise (and simulators are bundled with most quantum software packages) but it requires resources exponential in the number of qubits being simulated, so simulation can't help us in the general case.

Where standard software assurance techniques fail us, formal verification thrives. When we reason about a quantum program, we are reasoning exclusively about vectors, and vectors don't collapse when we analyze them. Formal verification is parametric in its inputs: Instead of reasoning about a 128-qubit implementation of an algorithm, we can prove properties of that algorithm for arbitrary values of n . Using techniques like induction, algebraic reasoning and equational rewriting we can verify the correctness of a broad range of quantum programs, as we previously showed [27] using the *QWIRE* programming language and verification tool.

Unfortunately, the challenges of measurement and simulation complexity are only the tip of the iceberg when it comes to near-term quantum computing.

For the foreseeable future, useful quantum computing will face a broad range of obstacles. The two major competing architectures for quantum computers are the superconducting qubit model used by IBM, Google, and Rigetti, and the trapped-ion model of IonQ and a number of academic labs. To varying extents (and the variance matters), each of these models suffers from the following issues:

1. Coherence times: Qubits can only maintain their state for a certain amount of time before they *decay*, effectively resetting themselves.
2. Gate errors: Quantum gates introduce errors and these errors vary with the gates being applied.
3. Connectivity: In general, you can't apply an operation to two or more qubits unless those qubits are physically adjacent to one another. We can use quantum gates to swap the values of qubits, and thereby bring two or more qubits together, but these operations take time and introduce additional errors.

These limitations aren't even uniform within a given machine, let alone across machines. On IBM's largest publicly available quantum computer, Melbourne, phase coherence times range from 22.1 to 106.5 microseconds depending on the qubit in question, and gate errors similarly vary by qubit [19].

Given the substantial challenges facing quantum computing, is formal verification even useful? We argue that can be.

To tackle the limitations of near-term quantum computers, we will need to tailor our verification efforts to the lowest level of the quantum software stack. We will need to incorporate information about the connectivity and error rates of a given machine in order to verify that a program can be run on that machine, and to bound the error of such an

execution. Such verification will be messy: It will have to take a lot of variables into account, including the ideal semantics of a given program, qubit by qubit decoherence and error rates, and specification that may differ substantially by platform. However, we argue that this verification is necessary, and that we have the tools necessary to perform it.

To make this case, we'll begin by introducing quantum computing and the semantics of error-free quantum programs (Section 2). In Section 3, we will survey the literature on formally verified quantum computing, which applies to an idealized, error-free setting. In Section 4 we get to the meat of this paper: What challenges does near-term quantum computing face, and how can we address them? We devote most of this discussion to the issues of architectural limitations (Section 4.3) and errors (Section 4.2). Finally, in Section 5 we reflect on how this discussion can inform the nascent field of quantum programming languages.

2 Logical Qubits

The approach to quantum computation taken by most textbooks, as well as quantum complexity theorists and (most) quantum algorithms designers assumes the existence of *logical qubits*, quantum bits which are not error-prone and behave according to a strict mathematical model. To be precise, a qubit corresponds to a two element vector of the form $\langle \alpha, \beta \rangle$ for complex numbers α and β , subject to the constraint that $|\alpha|^2 + |\beta|^2 = 1$.

Logical qubits obey strict rules but they are not deterministic. If we *measure* the qubit above, we will obtain one of the two *basis qubits* $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$ with probability $|\alpha|^2$ and $|\beta|^2$, respectively.

We can combine two qubits by taking their tensor product, where

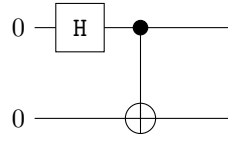
$$\langle \alpha, \beta \rangle \otimes \langle \gamma, \delta \rangle = \langle \alpha\gamma, \alpha\delta, \beta\gamma, \beta\delta \rangle.$$

Measuring the quantum system above will yield one of four basis vectors, with probabilities corresponding to the given entries. With probability $|\alpha\gamma|^2$ we will obtain $\langle 1, 0, 0, 0 \rangle$; with probability $|\alpha\delta|^2$ we will obtain $\langle 0, 1, 0, 0 \rangle$, and so forth.

Besides measuring (systems of) qubits, we can modify them by applying *unitary gates* which correspond to multiplication on the left by a restricted set of matrices called *unitaries*, which preserve the property that the sum-of-squares adds up to one. It's worth noting that if we apply certain gates (such as the controlled-not gate) to two or more qubits, it may no longer be possible to represent the outcome as the tensor product of multiple distinct 2-dimensional vectors. This state of affairs is called *entanglement* and is analogous to probabilistic dependence.

Let's give a simple example of entanglement in action. Imagine we start with the simple two qubit state $\langle 1, 0 \rangle \otimes \langle 1, 0 \rangle$. We then apply the Hadamard unitary $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ to the first qubit, obtaining $\langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \rangle \otimes \langle 1, 0 \rangle$, which we can also write as $\langle \frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}, 0 \rangle$. The controlled-not unitary exchanges the third and fourth elements of the vector, yielding $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}} \rangle$. This entangled vector cannot be decomposed into the tensor product of two smaller vectors, and is known as a *Bell pair*.

What happens if we measure our Bell pair? As we've seen, we will obtain either $\langle 1, 0, 0, 0 \rangle = \langle 1, 0 \rangle \otimes \langle 1, 0 \rangle$ or $\langle 0, 0, 0, 1 \rangle = \langle 0, 1 \rangle \otimes \langle 0, 1 \rangle$, each with probability $\left| \frac{1}{\sqrt{2}} \right|^2 = \frac{1}{2}$. After measurement, our two qubits are no longer entangled but their outcomes are correlated: Either both are $\langle 1, 0 \rangle$ or both are $\langle 0, 1 \rangle$. This correlation is an effect of entanglement, and one of the features that gives quantum computing its power.



(a) A circuit to produce a Bell pair

$$\begin{pmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{pmatrix}$$

(b) The density matrix for our Bell pair

■ **Figure 1** An example Bell pair.

Generally speaking, we will represent quantum programs as *circuits*. For instance, the circuit for constructing a Bell pair is shown in Figure 1, where 0 represents the vector $\langle 1, 0 \rangle$, H is the Hadamard matrix and the symbol connecting the two wires is the controlled-not. The circuit model is standard for quantum computing: Quantum programming languages like Quipper [14], Scaffold [20] and QWIRE are all circuit description languages; Microsoft's recent Q# tries to move away from this model by adding some abstractions, but all Q# programs can easily be read as describing circuits as well.

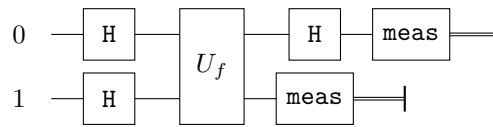
Conveniently, these quantum circuits have a straightforward denotational semantics: They correspond precisely to functions over complex vectors. We can also represent them as functions over *density matrices* which in turn correspond to distributions over complex vectors. A density matrix for our bell pair is given in Figure 1. The $\frac{1}{2}$ s in the first and fourth positions along the diagonal represent the probability of measuring both qubits as $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$, respectively. Embedding probabilities inside density matrices saves us from having to include probabilistic transitions in our denotational semantics. Once we have a denotational semantics for quantum circuits, we can begin to prove things about them.

3 Programming Languages and Verification Under Ideal Conditions

One of the simplest things to verify about a quantum program is that it doesn't attempt to duplicate qubits, which would violate the *no cloning* theorem of quantum mechanics. Non-duplication can be enforced by a linear type system, like that employed by the quantum lambda calculus [32], Proto-Quipper [31] or QWIRE [25]. A linear type system treats a function type $A \multimap B$ as something that *consumes* an A (precisely once) and produces a B (that may itself be used precisely once). Hence, it ensures that once we've done an operation on a qubit, the original qubit can no longer be used.

It's rather more complicated to ensure that a quantum program uses *ancillae* safely. Ancillae are spare qubits that are used in the computation of some result and then returned to their original state and discarded. They can be thought of as scratch space for intermediate quantum computations, but they have to be regularly garbage collected. Since qubits can be entangled with one another, operating on improperly discarded ancillae can corrupt the rest of our computation. Ancillae are a common feature of quantum algorithms, and hence appear in quantum programming languages like Quipper [14] and Q# [33]. Unfortunately, it's quite difficult to guarantee that ancilla qubits are properly garbage collected. Inspired by the REVERC [3] compiler for reversible programs, QWIRE allows the programmer to prove that ancillae are discarded correctly, while providing syntactic conditions for cases where this is trivially true [28]. In the general case, though, proving that we've appropriately disposed of ancillae requires us to reason about the behavior of complete quantum programs.

Doing whole-program analysis substantially increases the scope for formal verification,



■ **Figure 2** A circuit implementing Deutsch's algorithm

which inspired us to use *QWIRE* as a general-purpose verification tool [29]. One thing we may want to do is verify the correctness of a complete program, like Deutsch's algorithm [11, 8]. Deutsch's algorithm, shown in Figure 2, takes in an unknown function f , represented as a quantum gate, and returns the 0 qubit if and only if the function is constant. Using *QWIRE*, we can express the correctness of this algorithm as follows:

```

237 Lemma deutsch_constant : ∀ f, constant f →
238   [[deutsch (fun_to_gate f)]] I1 = |0⟩⟨0|.
239
240
241 Lemma deutsch_balanced : ∀ f, balanced f →
242   [[deutsch (fun_to_gate f)]] I1 = |1⟩⟨1|.
243

```

Here $|0\rangle\langle 0|$ represents a 0 qubit in density matrix form, and similarly for $|1\rangle\langle 1|$. I_1 is a 1×1 identity matrix, representing that the circuit has no input qubits (akin to `unit` or `void`). Deutsch's algorithm is relatively easy to prove correct by computation but in practice we can verify a broad range of algorithms, from families of quantum coin tossing circuits to algebraic properties of quantum circuits, like their being unitary [27].

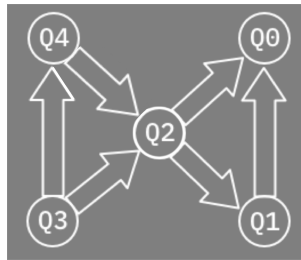
QWIRE isn't the only tool that attempts to tackle this problem. Amy [1] uses Feynman path integrals to check the correctness of a variety of concrete quantum circuits, including a quantum Fourier transform [10] using up to 31 qubits. A number of authors have also introduced Hoare-style logics for quantum programs and used them to verify Grover's algorithm [15, 38] and a quantum one-time pad [5, 35]. More specialized tools allow us to verifying the security of quantum protocols [36] and rewrite quantum programs expressed in the ZX calculus [9, 21], the latter of which we will address in more detail in Section 4.3.

4 Verification in the Real Quantum World

So far we have an interesting story and perhaps even a compelling one: Quantum programs are hard to write and debug and hence provide an excellent target for the techniques of formal verification, from program logics to proof assistants. However, these logics will be of limited use for the near future. The quantum computers that exist today and are likely to exist over the next ten to twenty years will be incapable of running arbitrary quantum circuits and will be very failure prone. Hence, for formal verification to be useful in the near term, it will need to be tied to the machines of tomorrow.

4.1 NISQ: The Era of Errors

In a recent keynote address [26], John Preskill coined the term Noisy Intermediate-Scale Quantum Computing (NISQ) to refer to quantum computing over the coming five or ten years. Preskill, like many in the field, suspects that quantum computing will soon have its first practical applications on computers with under 1000 physical qubits. On the other hand, it will take hundreds or thousands of physical qubits to construct one error-corrected logical qubit, and many thousands of logical qubits to beat classical computers at factoring numbers or performing a range of other tasks. In the near term, quantum programs will have to be



■ **Figure 3** Two-qubit gate connections on IBM's Tenerife machine. Taken from https://github.com/Qiskit/ibmq-device-information/blob/master/backends/tenerife/V1/version_log.md.

272 aimed at problems for which they are uniquely well-suited (like modeling quantum systems
273 in physics [7] and chemistry [30]) and tailored to the limitations of the available computers.

274 What do these limitations look like? One hard limitation is the number of qubits on the
275 machine. Another limitation is error rates. There are multiple sources of errors in quantum
276 circuits, stemming from *decoherence* (qubits tend to revert to their basis states with time) and
277 errors in gate application.¹ Today's machines also have a number of architectural limitations,
278 related to the connectivity of qubits: Instead of being complete undirected graphs, they tend
279 to resemble sparse directed graphs. For example, consider the diagram of IBM's 5-qubit
280 Tenerife machine shown in Figure 3. In this architecture, if you want to apply a controlled-not
281 gate from program qubit q_1 to program qubit q_2 , then you need to map those program qubits
282 to adjacent physical qubits in the machine (e.g. Q_4 and Q_2). This mapping may need to
283 be updated over the course of the program by adjusting the physical location of different
284 program qubits. This adjustment is both computationally expensive and error prone.

285 If formal verification is to guarantee the correctness of quantum programs in practice, it
286 will need to account for these crucial limitations on NISQ devices.

287 4.2 Verification in the Presence of Errors

288 Let us sketch out some possible approaches to dealing with the errors that are sure to arise
289 when we run our quantum computers.

290 One straightforward approach to verification in the presence of errors is to simply aggregate
291 errors along a quantum circuit's wire. Instead of the Hadamard gate (H) having the type
292 $\text{Qubit} \multimap \text{Qubit}$, it can have the type $\forall n, (\text{Qubit}, n) \multimap (\text{Qubit}, n+1)$, meaning that the gate adds
293 a single error to its wires. We could equally well include error probabilities along the wires,
294 though those would assume we knew the error rate for each gate and they were consistent
295 across qubits. Multiple-qubit gates are a bit trickier, as they take in and produce multiple
296 wires: We can either output the sum of all error terms along each output wire plus the
297 additional error introduced by the gate, take the max of the two wires, or (particularly in the
298 case of probabilities) use a more complex function over multiple inputs. A circuit then, would
299 likewise have an error term corresponding to the aggregated output of its wires. For a simple
300 example, depending on whether U_f takes the max or sum of its inputs' errors, Deutsch's

¹ The presence of these gate errors actually allows us to comfortably ignore a more fundamental issue in quantum computing. The so-called "universal gate sets" implemented by general purpose quantum computers are not actually universal in the sense that NAND gates are universal for classical computation: They only allow us to *approximate* arbitrary quantum operations. Unfortunately, in the near term, all gates will only loosely approximate their specified behaviors.

algorithm (Figure 2) would produce 2 or 3 errors, plus the errors introduced by U_f , assuming measurement doesn't introduce errors itself.

An advantage of this approach is that it's very easy to implement and sufficiently general that we can interpret the output in a number of different ways, depending on the setting. We can also automate it, allowing a built-in type inference algorithm to calculate the errors that occur in a given circuit.² It is also limited, though, in that errors only increase as the circuit size grows. This makes it difficult to analyze programs that include mechanisms for error mitigation, which will be important in near-term applications [22]. (In principle, we could have error-correcting gates that shrink the error, but error correction tends not to be so simple.)

We can also take ideas from our recent robustness logic [18], which draws on Carbin *et al.*'s Rely [6] and provides bounds for the errors in a quantum program. While powerful, this logic suffers from the same limitations as our error wire semantics—errors only increase throughout a program. It is also high level: It uses the same quantum while language as QHL [38], which doesn't directly describe circuits. By contrast, the ideal near-term logic would describe quantum circuits under a precise error model and allow us to apply general reasoning techniques to those circuit.

Ideally, the semantics for a quantum program would contain error terms, corresponding to possible failures. This would bring the advantage of allowing us to reason about and reduce error terms, through majority voting or the like. Unfortunately, it's still hard to know what the denotational model should be, without reference to the specific hardware. Hence, while doing high-level reasoning we want to leave the error term abstract (as in our robustness logic), and instantiate it for specific hardware models.

4.3 Verified Quantum Compilation

As we've seen, in order to make effective use of near-term quantum devices, we cannot entirely abstract away low-level architectural details. However, this is not to say that the programmer needs to consider every low-level detail when writing programs. In some cases, as in classical computing, the complexity of running on a particular architecture can be handled by the compiler.

For example, we have already discussed the challenge of limited connectivity on near-term machines. There has been significant work on developing automated transformations that map arbitrary quantum circuits into circuits that satisfy a particular machine's connectivity constraints [39]. Some of this work has also looked at how to perform this mapping in a way that reduces the error of the resulting circuit [23, 34].

Another way that near-term compilers for quantum programs can help is by performing optimizations that reduce resource usage [2, 16, 24]. Reducing resource usage is critical because near-term devices have access to a limited number of qubits and can support few operations before decoherence undoes the effect of any useful computation. These optimizations are often mathematically sophisticated, and thus vulnerable to programmer error. For example, in discussions with Nam *et al.* we learned that, while developing their optimizer, they found several bugs in their own implementation and also in the implementations that they compared against [24].

² It's worth noting that checking linearity, though harder, is an orthogonal problem, so we can infer the error terms and check linearity separately. This is somewhat surprising, since without linearity we would have to be concerned about adding errors to terms without using them up.

We envision a compiler from a high-level quantum programming language to low-level circuits that is *verified correct*. Testing compilers is hard [37], and testing compilers for quantum program and quantum architectures is even harder. Given the difficulty of testing quantum programs and the expense involved in running them, it is especially important, and useful, to verify compilers in the quantum setting.

The Quantomatic tool [21] can apply verified transformations to quantum computation expressed in terms of the ZX-calculus [9, 4], a diagrammatic approach to quantum computation. Unfortunately, not every ZX diagram corresponds to a valid quantum circuit, and an optimization in ZX may not optimize a corresponding circuit. Recent work [13] optimizes a restricted subset of ZX diagrams that do represent circuits, but these are limited to a subset of quantum circuits known as Clifford circuits.

[Robert: tentative, need sqire on arxiv] Our new intermediate representation for quantum circuits, called sqire [Robert: cite sqire], helps us to go somewhat further. sqire allows us to perform verified optimizations on circuits, with the goal of reducing the total circuit size. It also provides verified transformations that guarantee structural properties, like all controlled-not gates being applied to adjacent qubits. However, at the moment these optimizations have to be applied manually, rather than being part of a compilation procedure with knowledge of the target architecture.

We would like to go further. A verified compiler for quantum circuits should take the following three things into account:

1. The connectivity of the target machine;
2. the error rates for each qubit; and
3. the fidelity of individual gates applications.

It should use this information to compile an arbitrary quantum circuit to an equivalent circuit that can run on the target machine in a way that minimizes errors. This is difficult: Each of our desiderata imposes substantial constraint on the compiler and a burden on the verifier. However, each is necessary for our compiler to be both useful and reliable.

5 The Verification-Programming Loop

Throughout this exposition, we’ve treated verification as a task that is subsequent to programming, which guarantees that the program behaves as expected. However, this doesn’t quite reflect our intended use for formal verification in the quantum setting. In our experience, verification plays a major role in quantum programming itself, even ignoring machine errors. Even reproducing well-known quantum algorithms proves to be difficult, given the low level at which they are written. (For examples, see Huang and Martonosi’s [17] recent exploration of bugs in the implementations of common quantum algorithms.) In practice, we find ourselves writing quantum programs, attempting to prove their correctness, failing, and revising the original program. Often the failures can be quite informative: If the Coq proof assistant asks us to prove that $\frac{1}{\sqrt{2}} = 1$, it’s a sure sign that we’ve either left out a Hadamard gate or put in one too many (since $\frac{1}{\sqrt{2}}$ appears throughout the Hadamard matrix).

Following Dijkstra’s vision for formal verification in *A Discipline of Programming* [12], we expect error-aware quantum verification to assist us in writing quantum programs. On occasion it may make sense to simply write a quantum program, send it off to an optimizer, and execute it on a noisy quantum machine. But this won’t often be the case. Once you reach a point where your program’s output is indistinguishable from noise, optimizations probably aren’t going to make it work. At that point, you need to take a second look at your program, just as if you found that your scientific program was numerically unstable.

High error rates are a sign that you're doing something wrong, and the sooner you're made aware of that, the better off you'll be.

We should note that all this is far from ideal: In an ideal world we would write quantum programs in a high level language with appropriate abstractions, and never even think about quantum circuits. Unfortunately, we don't live in such a world and aren't likely to live in it for many years. Instead, our goal is to develop tools that will assist in developing efficient algorithms with correctness guarantees and precisely bounded errors and to be able to execute those algorithms on fundamentally limited machines. To that end, we have to expose everything down to the individual qubit decoherence to the programmer, so they can handle that decoherence. Providing these tools will allow us to do more with near-term quantum devices than we could possibly do today.

References

- 1 Matthew Amy. Towards large-scale functional verification of universal quantum circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018*, June 2018.
- 2 Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-time t-depth optimization of clifford+t circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33, 2013.
- 3 Matthew Amy, Martin Roetteler, and Krysta M. Svore. Verified compilation of space-efficient reversible circuits. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2017)*. Springer, July 2017. URL: <https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/>.
- 4 Miriam Backens. The zx-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, 16(9):093021, 2014.
- 5 P Oscar Boykin and Vwani Roychowdhury. Optimal encryption of quantum bits. *Physical review A*, 67(4):042317, 2003.
- 6 Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 33–52, 2013. doi:10.1145/2509136.2509546.
- 7 Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences of the United States of America*, 115 38:9456–9461, 2018.
- 8 Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 454, pages 339–354. The Royal Society, 1998.
- 9 Bob Coecke and Ross Duncan. Interacting quantum observables. In *International Colloquium on Automata, Languages, and Programming*, pages 298–310. Springer, 2008.
- 10 Don Coppersmith. An approximate fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067*, 1994.
- 11 David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 400, pages 97–117. The Royal Society, 1985.
- 12 Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Etats-Unis Informaticien, and Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.
- 13 Andrew Fagan and Ross Duncan. Optimising Clifford circuits with Quantomatic. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*, 2018.

- 439 **14** Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron.
 440 Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM*
 441 *SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013,
 442 pages 333–342, 2013.
- 443 **15** Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of*
 444 *the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–
 445 219, New York, NY, USA, 1996. ACM. URL: <http://doi.acm.org/10.1145/237814.237866>,
 446 doi:10.1145/237814.237866.
- 447 **16** Luke Heyfron and Earl T. Campbell. An efficient quantum compiler that reduces t count.
 448 *Quantum Science and Technology*, 4, 2017.
- 449 **17** Yipeng Huang and Margaret Martonosi. Qdb: from quantum algorithms towards correct
 450 quantum programs. *arXiv preprint arXiv:1811.05447*, 2018.
- 451 **18** Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi
 452 Wu. Quantitative robustness analysis of quantum programs. *Proc. ACM Program. Lang.*,
 453 3(POPL):31:1–31:29, January 2019. URL: <http://doi.acm.org/10.1145/3290344>, doi:10.
 454 1145/3290344.
- 455 **19** IBM. IBM quantum experience, 2017. URL: [https://quantumexperience.ng.bluemix.net/](https://quantumexperience.ng.bluemix.net/qx/devices)
 456 [qx/devices](https://quantumexperience.ng.bluemix.net/qx/devices).
- 457 **20** Ali Javadi-Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan
 458 Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold:
 459 Quantum programming language. Technical report, PRINCETON UNIV NJ DEPT OF
 460 COMPUTER SCIENCE, 2012.
- 461 **21** Aleks Kissinger. *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories*
 462 *and Applications to Quantum Computing*. PhD thesis, University of Oxford, 2011.
- 463 **22** Sam McArdle, Xiao Yuan, and Simon Benjamin. Error mitigated digital quantum simulation.
 464 *arXiv preprint arXiv:1807.02467*, 2018.
- 465 **23** Prakash Murali, Jonathan M Baker, Ali Javadi Abhari, Frederic T Chong, and Margaret
 466 Martonosi. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers.
 467 *arXiv preprint arXiv:1901.11054*, 2019.
- 468 **24** Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. Automated
 469 optimization of large quantum circuits with continuous parameters. *npj Quantum Information*,
 470 4(1):23, 2018.
- 471 **25** Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum
 472 circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming*
 473 *Languages*, POPL 2017, pages 846–858, New York, NY, USA, 2017. ACM. doi:10.1145/
 474 3009837.3009894.
- 475 **26** John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.
 476 URL: <https://doi.org/10.22331/q-2018-08-06-79>, doi:10.22331/q-2018-08-06-79.
- 477 **27** Robert Rand. *Formally Verified Quantum Programming*. PhD thesis, University of Pennsylva-
 478 nia, 2018.
- 479 **28** Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. ReQWIRE: Reasoning
 480 about reversible quantum circuits. In *Proceedings of the 15th International Conference on*
 481 *Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*, 2018.
- 482 **29** Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of
 483 quantum circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics*
 484 *and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, pages 119–132, 2017. URL:
 485 <https://doi.org/10.4204/EPTCS.266.8>, doi:10.4204/EPTCS.266.8.
- 486 **30** Markus Reiher, Nathan Wiebe, Krysta Marie Svore, Dave Wecker, and Matthias Troyer.
 487 Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy*
 488 *of Sciences of the United States of America*, 114 29:7555–7560, 2017.
- 489 **31** Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie
 490 University, 2015.

- 491 32 Peter Selinger and Benoît Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie,
492 editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University
493 Press, 2009.
- 494 33 Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina
495 Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#:
496 Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings*
497 *of the Real World Domain Specific Languages Workshop 2018*, page 7. ACM, 2018.
- 498 34 Swamit S. Tannu and Moinuddin K. Qureshi. A Case for Variability-Aware Policies for
499 NISQ-Era Quantum Computers. *arXiv e-prints*, page arXiv:1805.10224, May 2018. **arXiv:**
500 1805.10224.
- 501 35 Dominique Unruh. Quantum hoare logic with ghost variables. *arXiv preprint arXiv:1902.00325*,
502 2019.
- 503 36 Dominique Unruh. Quantum relational hoare logic. *Proc. ACM Program. Lang.*, 3(POPL):33:1–
504 33:31, January 2019. URL: <http://doi.acm.org/10.1145/3290346>, doi:10.1145/3290346.
- 505 37 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C
506 compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language*
507 *Design and Implementation*, PLDI ’11, pages 283–294, New York, NY, USA, 2011. ACM.
508 URL: <http://doi.acm.org/10.1145/1993498.1993532>, doi:10.1145/1993498.1993532.
- 509 38 Mingsheng Ying. Floyd-Hoare logic for quantum programs. *ACM Transactions on Programming*
510 *Languages and Systems (TOPLAS)*, 33(6):19, 2011.
- 511 39 Alwin Zulehner, Alexandru Paler, and Robert Wille. Efficient mapping of quantum circuits
512 to the IBM QX architectures. In *2018 Design, Automation & Test in Europe Conference &*
513 *Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 1135–1138, 2018. URL:
514 <https://doi.org/10.23919/DATE.2018.8342181>, doi:10.23919/DATE.2018.8342181.