Contract-based verification of a realistic quantum compiler

Yunong Shi*
The University of Chicago
yunong@uchicago.edu

Xupeng Li*
Columbia University
xupeng.li@columbia.edu

Runzhou Tao
Columbia University
runzhou.tao@columbia.edu

Ali Javadi-Abhari
IBM T.J. Watson Research center
Ali.Javadi@ibm.com

Andrew W. Cross
IBM T.J. Watson Research center
awcross@us.ibm.com

Frederic T. Chong
The University of Chicago
chong@cs.uchicago.edu

Ronghui Gu
Columbia University
ronghui.gu@columbia.edu

ABSTRACT

In this paper, we present CertiQ, a mostly-automated verification framework for the Qiskit quantum compiler. To our knowledge, CertiQ is the first effort to apply formal verification and SMT reasoning to a real-world quantum compiler. Qiskit is currently the most complete and widely-used open-source quantum software stack from low-level compilation to high-level quantum algorithms. With growing community contributions, the Qiskit compiler is in need of code quality control and verification down to the compilation level to guarantee reliability of scientific work that uses it. CertiQ is deeply integrated into the Qiskit compiler (called Terra), providing abstract specifications for quantum compiler data structures and offering verifiable contracts that specify the behaviors of compilation phases with heavy optimizations. CertiQ enables verification of existing implementation of the Qiskit compiler and future code submissions in a mostly-automated manner using invariant-guided contracts and contract continuation. With these CertiQ techniques in place, developers need to provide limited inputs only where function contracts and loop invariants cannot be inferred automatically.

The CertiQ verification procedure discovers several critical bugs, some of which are unique to quantum software. Our extensive case studies on four compiler phases of Qiskit demonstrate that CertiQ is effective for verification of quantum compilers with a low proof burden.

1 INTRODUCTION

The development of NISQ [29] (Noisy Intermediate-Scale Quantum) devices has transformed quantum computing from an academic pursuit to a realistic goal for the realization of practical quantum applications. NISQ devices like IBM's quantum machine with 20 qubits and Rigetti's quantum machine with 19 qubits has led to the emergence of cloud-based quantum services and associated computing software stacks [1, 6, 31].

Qiskit [1] is currently the most complete and widely-used open-source software stack. Qiskit lets users design and run quantum programs on the IBM Q cloud [19], a cloud based service for near-term quantum computing applications and research. With more than 100K users from 171 countries, Qiskit has accommodated over 5.3M experimental runs on quantum devices and 12M virtual simulations to date. Qiskit is also influential in the open-source community: with 180k downloads, 1500 Github forks (with 2nd place Cirq [6] < 500) and Github "usedby" of 122 (with 2nd place Qutip [34] with 59). Over 190 academic articles are based on IBM's cloud service, pushing progress in many different scientific disciplines, including: validation of properties of electron structure [41]; demonstration of error detection schemes [46]; demonstration of quantum machine learning algorithms [38, 50].

The increasing numbers of quantum computations have revealed numerous errors at all levels in the Qiskit toolchain, which can corrupt the scientific results performed with it. Specifically, the different nature of quantum computations along with heavy optimizations performed in the Qiskit compiler (called Qiskit Terra) makes the compilation error-prone. The high number of bug reports [44] related to the compilation process highlights the crucial need for effective, reliable and automated methods to verify near-term quantum computations down to the compilation level.

*Both authors contributed equally to this research.

We introduce CertiQ, a mostly-automated verification framework for Qiskit Terra. To our knowledge, CertiQ is the first effort to apply formal verification and SMT reasoning to real-world quantum compiler. The design philosophy underpinning CertiQ is motivated by three practical challenges that arise when providing correctness guarantee to Qiskit Terra.

The first challenge is that checking the equivalence of quantum circuits is generally intractable [20]. To mitigate this problem, in CertiQ we define a verifiable small-step operational semantics for quantum circuits that we call *primitive circuit moves* that circuit equivalence and the correctness of compiler transformation can be efficiently proved. Our semantics is proven to be sound and therefore faithful to the underlying quantum computation.

The second challenge for correctness guarantee is that the different nature of quantum computation can cause unexpected behavior of components when interacting with each other in a large and rapidly growing quantum software. Using the Design-by-Contract methodology, CertiQ offers contracts to specify the behaviors of quantum data structures and library functions in Qiskit Terra, for example, the Bloch sphere representation of qubit states. With the contract-based design, we are able to reason about the invocation of a verified function using its contract without looking into its concrete implementation. We design, specify, and verify a library of functions that perform primitive circuit transformations with *invariant-guided contracts*. Compilation phases implemented using this library can be easily verified. The reason is that these function calls can be treated as "transparent" in symbolic execution because their contracts imply semantics-preserving.

The third challenge is that compiler implementations in future code submission can be very complicated, making mostly-automated verification hard. By "mostly-automated", we mean that programmers have to write little to no annotations to assist the verification. To achieve this level of automation, CertiQ first pre-processes code by replacing its calls to library functions with their abstract specifications and contracts. CertiQ then uses symbolic execution to generate verification conditions in the form of satisfiability modulo theories (SMT) problems fed into a SMT solver, e.g., Z3 [11]. CertiQ introduces *contract continuation* to automatically infer the contracts of developers (contributors) defined functions and simplify contracts in verification. CertiQ also provides several types of commonly used contracts for developers to specify their functions. Further, with careful design of function calls and their contracts, the effect of CertiQ verification on compilation performance is not noticable.

For several important types of compiler extensions (optimizations), if an implemention by developers cannot be verified by the CertiQ verifier, CertiQ introduces a translation validator to validate the correctness of each compilation output at runtime with small overhead. The translational validator itself is also verified.

We verified four compiler phases and seven transpiler pass implementations of Qiskit Terra in four case studies. With these verified CertiQ implementations, we successfully identify three bugs of Qiskit Terra, two of which are unique in quantum software.

This paper makes the following contributions:

- We provide small-step semantics for quantum circuit equivalence and formal specifications of quantum circuit transformations that efficiently prove compiler optimizations are semantics preserving.
- We introduce a contract-based design that specifies the behavior of other operations, thereby facilitating modular verification of quantum compiler implementations.
- We build a transformation library verified with respect to its contract, which guarantees that the provided circuit moves preserve the circuit equivalence. This library can be used to build verified quantum compilers.
- We combine proof techniques including static analysis, SMT theory reasoning, and contract continuation that make CertiQ scalable and mostly-automated.
- We verify a series of Qiskit Terra optimizations and discover five critical bugs. Some of these bugs are unique to quantum software.

The paper is organized as follows: section 2 introduces background on quantum computing and Qiskit Terra; section 3 provides an overview of the CertiQ framework; section 4 discusses technical contributions of CertiQ; section 5 presents case studies demonstrating how CertiQ works in the realistic settings; section 6 evaluates the correctness and performance; section 7 discusses previous works; section 8 concludes.

2 BACKGROUND

2.1 Quantum Computing

Principle of quantum computation. The qubit (quantum bit) is the basic element of a quantum computing system. In contrast to classical bits, qubits are capable of living in a superposition of the logical states $|0\rangle$ and $|1\rangle$. The general quantum state of a qubit is represented as $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ (or in its vector form α, β^T), where α, β are complex coefficients with $|\alpha|^2 + |\beta|^2 = 1$. When

Fig. 1. Examples of circuit decomposition: SWAP gate decomposition (left); A circuit for preparing the 3 qubit GHZ state:$|\psi_{\mathrm{GHZ}}\rangle = \frac{1}{\sqrt{2}}|000\rangle + |111\rangle$. (right).

measured in the 01 basis, the quantum state collapses to $|0\rangle$ or $|1\rangle$ with probability of $|\alpha|^2$ and $|\beta|^2$, respectively.

The number of quantum logical states grows exponentially with the number of qubits in a quantum system. For example, a system with 3 qubits lives in the superposition of 8 logical states: $|000\rangle$, $|001\rangle$, $|010\rangle$, ..., $|111\rangle$. This property sets the foundation of quantum speedup over classical computation—an exponential number of correlated logical states can be stored and processed simultaneously by a quantum system with a linear number of qubits. However, this also brings great challenges for simulating and verifying quantum computations.

Quantum gates. The basic quantum operations are called quantum gates, which are unitary transformations on the qubit space. Some of the quantum gates commonly used in quantum algorithms include X gate, Y gate, Z gate, H gate, T and CNOT gate.

2.2 Compilation of Quantum Programs

Quantum compilation is the process of translating high-level description of quantum algorithms to a circuit consisting of a series of quantum gates. A quantum compilation process can be divided into four stages: 1) circuit decomposition; 2) system-independent optimization; 3) technology mapping; and 4) system-dependent optimization.

In the first stage, quantum algorithms are decomposed into quantum circuits. Figure 1 illustrates two simple examples of circuit decomposition: decomposition of a SWAP gate to 3 alternating CNOTs; decomposition of the circuit for preparing the 3-qubit GHZ state $|\psi_{\mathrm{GHZ}}\rangle = \frac{1}{\sqrt{2}}|000\rangle+|111\rangle$, a quantum state used widely in quantum computation and communication.

The system-independent optimization stage (stage 2) involves post-decomposition optimizations like circuit rewriting[42], template matching [26], commutativity detection [40].

Stage 3 technology mapping makes sure that the decomposed quantum circuit conforms to the topological constraints of the QIP platform. This process is usually executed by introducing quantum SWAP gates into the circuit.

The system-dependent optimization stage 4 further decomposes the gate set used in quantum algorithms to the gate set that can be executed on the QIP platform. Stage 4 also performs optimizations like single qubit rotation merge [9].

2.3 The Qiskit Terra Compiler

The Qiskit Terra compiler is the foundation of the Qiskit framework, upon which other Qiskit components are built. Qiskit Terra consists of a set of tools for composing quantum programs at the level of circuits and pulses, optimizing them for the constraints of a particular physical quantum processor, and managing the batched execution of experiments on remote-access backends. The optimizations in Qiskit Terra are crucial for successful execution of quantum programs since quantum resources are scarce and qubit coherence time is very limited.

We describe the main components of the Terr compiler (fig. 3), to which the CertiQ framework provides abstract specifications and contracts.

Quantum register. A quantum register is a collection of qubits that provides certain functionality in a quantum algorithm. Every qubit lives in a quantum register.

Coupling map and layout. Coupling map is the description of the connectivity of qubits on the physical device. It stores the edges of the qubits in a list. For example, `coup = [[0,1],[1,2],[2,3]]` describes a device of 3 physical qubits with linear connectivity. A layout is a Python dictionary from the virtual qubits in the quantum register to the physical qubits on the device. For example, the implementation in Terra is,

```
class Layout():
  def __init__(self):
    self._p2v = dict() # Physical to virtual qubit map
    self._v2p = dict() # Virtual to physical qubit map
```

QuantumCircuit. QuantumCircuit is the class that stores quantum circuits as a series of instructions on classical and quantum registers. It provides an interface to input quantum circuit description and for visualization.

DAGCircuit. The DAGCircuit class is another description of a quantum circuit and is equivalent to QuantumCircuit. Compared to QuantumCircuit description, DAGCircuit provides more flexible APIs for circuit transformations and to explicitly express the dependence between individual gates in circuits. For example, it provides a method `topological_op_nodes()` that allows users to traverse gates in the DAG in topological sort, easing out lots of circuit optimization algorithms.

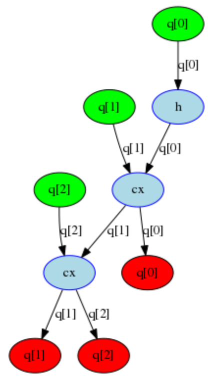


Fig. 2. The DAGCircuit representation of the circuit that prepares the GHZ state(See also fig. 1 on the right). Green nodes are input nodes, blue nodes are operation nodes and red nodes are output nodes. Arrows represent dependency and are specified by qubit number. DAG representation is equivalent to a quantum circuit description, thus, throughout the paper we will use circuit diagram for visualization for readability.

Transpiler. The transpiler is the circuit rewriting module in Qiskit Terra responsible for stage 2, 3 and 4 in the quantum compilation process. Because transpiler is the critical part of the compiler and also the fastest iterating component, the need for automated verification is thus pressing. The design language of the transpiler is similar to that of LLVM [22]. It consists of modular components called transpiler passes that can be assembled by the transpiler pass manager with respect to their dependency constraints. Input and output of transpiler passes are both DAGCircuit. There are two classes of passes: analysis passes and transformation passes. Analysis passes compute useful quantities for the input DAGCircuit while preserving the DAGCircuit. Transformation passes performs circuit optimization or constraint resolving on the DAGCircuit, either returning the modified DAGCircuit or returning a new DAGCircuit.

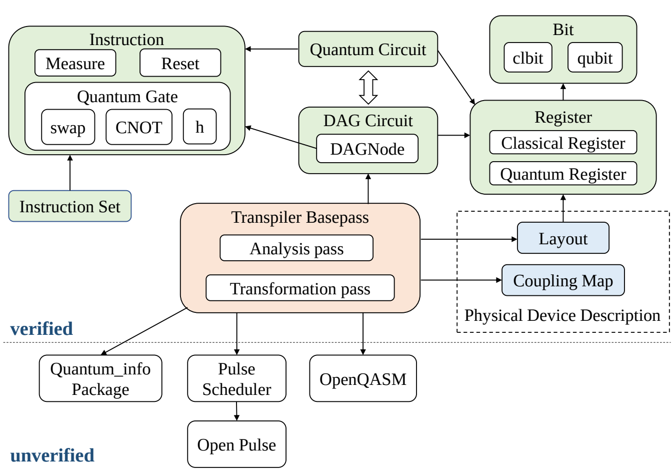


Fig. 3. Qiskit Terra components and call graph. Green boxes are quantum data structures. Blue boxes are physical devices related data structures. The red box is the transpiler. CertiQ gives specifications to components in blue and green boxes and verifies the transpiler in red box. White boxes under the horizontal dotted-line are parts that are not verified or just partly verified by CertiQ.

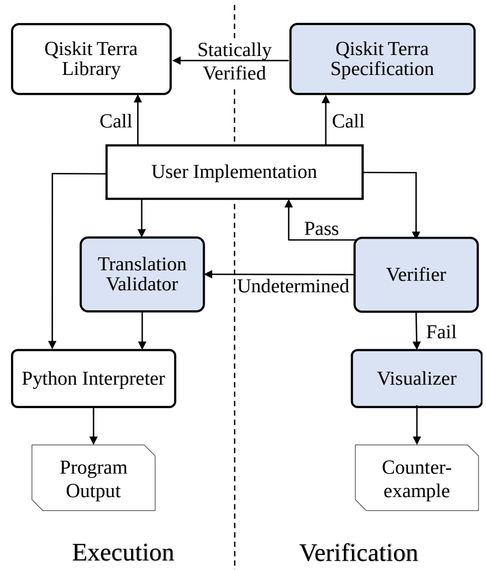


Fig. 4. The workflow of CertiQ framework. Blue boxes indicate CertiQ component. User code can be executed at two different backends. The usual Python backend and the verification engine share the same unified interface. In most cases, the user can be unaware of the verification engine when writing code.

3 THE CERTIQ WORKFLOW

This section gives an overview of the CertiQ framework by walking through the verification of a simple but less quantum-related transpiler pass. The integration of CertiQ and Qiskit Terra as a quantum software will be addressed in §4. The verification engine of CertiQ consists of four parts: specifications for Qiskit Terra quantum data structures/transformations, the verifier, the visualizer, and the translation validator. We use a simple transpiler pass named `basic_swap` as a running example to introduce the CertiQ verification workflow (see fig. 4).

3.1 Specifications

Specifications in CertiQ can be viewed as alternative implementations of quantum data structures and library functions in Qiskit Terra. They share the same interface with the original implementation but are re-implemented to support symbolic execution. Thus, we can verify contributors' code by replacing the data structure operations and invocation of Terra library functions with CertiQ specifications, and then feed the specifications into our verification engine. There are two kinds of specifications: *trusted specifications* and *abstract specifications*.

Trusted specifications. When replacing basic Python data structures with the ones in the SMT solver, e.g., Z3, the equivalence of the transformation is in our trusted computing base (TCB). These specifications are called trusted specifications. For example, the specification of the Layout data structure in CertiQ is

```
class Layout(): # Layout specification
  def __init__(self):
    # Physical to logical qubit map spec
    self._p2v = Map(IntSort(), IntSort())
    # Logical to physical qubit map spec
    self._v2p = Map(IntSort(), IntSort())
```

In this case, the specification in CertiQ is almost identical to the Layout implementation in Terra with the only difference that Layout specification uses Z3 `Map` rather than Python `dict`. Here, we trust that `Map` in Z3 correctly specifies `dict` in Python.

Abstract specifications. CertiQ allows users to lift some library implementations into abstract specifications to speed up the verification. For example, the abstract specification of the DAGCircuit class is an array of gates,

```
class DAGCircuit: # dagcircuit specification
  def __init__(self, gates=None, size=None):
    self.dag = (Array(IntSort(), GateType)
                if gates is None else gates)
```

The equivalence between all abstract specifications and their implementations is verified using *contextual refinement* (see §4.2).

3.2 Contracts

The concept of contract is deeply rooted in the design of CertiQ, primarily for specifying the behavior of quantum data structures/operations that might be unexpected in classical software. CertiQ offers contracts for every object and method that their specifications must comply with. In CertiQ, a contract consists of three parts: pre-conditions, post-conditions, and invariants. We give several simple examples of contracts that will be used in our `basic_swap` pass. More quantum-related contracts will be discussed in §4.2. For example, the contract of the Layout object is

```
class Layout():
  ... # omitted code
  # Contract of the Layout object
  def precondition(self): return True
  def invariant(self): return True
  def postcondition(self):
    i = fresh_int()
    return ForAll([i], self.p2v(self.v2p(i))==i)
```

The non-trivial part of this contract is its post-condition, which specifies that both the two `maps` in `Layout` must be bijections and the inverse of each other. For functions, their contracts are encoded in separate contract functions. For example, for the function `simple_layout` that generates an initial mapping of a circuit onto the coupling map of a physical device,

```
def simple_layout(qreg, coupling_map):
  layout = Layout()
  for i, qubit in enumerate(qreg):
    layout[qubit] = i
  return layout
```

its pre-condition function is,

```
def simple_layout_pre(qreg, coupling_map):
  return And(coupling_map.postcondition,
             qreg.postcondition,
             coupling_map.size >= qreg.size)
```

which states that the inputs must comply with the post-condition of their classes and the size of the coupling map must be equal or larger than the size of the quantum register. Its post-condition function simply returns `layout.postcondition`. To prove the function `simple_layout` complies with its contracts, in CertiQ we verify the following

```
# evaluate pre-condition
pre = simple_layout_pre(qreg, coupling_map)
# evaluate invariant before execution
inv_before = simple_layout_inv(qreg, coupling_map)
# symbolic execution
layout = simple_layout(qreg, coupling_map)
# evaluate post-condition
post = simple_layout_post(layout)
# evaluate invariant after execution
inv_after = simple_layout_inv(qreg, coupling_map)
# {pre-condition} Execution {post-condition}
certiq_prove(Implies(pre, post))
# pre-condition => invariant
certiq_prove(Implies(pre, inv_before))
# invariant before => invariant after
certiq_prove(Implies(inv_before, inv_after))
```

Another example is the function `shortest_path` in the `CouplingMap` class to find the shortest path of two physical qubits on the coupling map. Its pre-condition function returns `And(p1 < self.size, p2 < self.size, self.postcondition)` (here `self` is the coupling map). Its post-condition function returns

```
And(ForAll([i],
           Implies(And(i >=0, i < self.size - 1),
                   self.distance(
                   self.shortest_path[i],
                   self.shortest_path[i+1]) == 1)),
    self.shortest_path[0] == p1,
    self.shortest_path[1] == p2)
```

Which states the two neighboring physical qubits on the path must have distance 1 and the two ends of the path are the two input physical qubits.

The `basic_swap` pass. With the previous function definitions and their contracts, we can look into the implementation of the `basic_swap` pass (see fig. 5). This pass brings the qubit operands together for every 2-qubit gate in a DAGCircuit so that the 2-qubit gate operations can be done in two connected qubit locations on

```
# @swap
class BasicSwap(TranformationPass):
  def __init__(self, layout=None, coupling_map):
     self.layout = layout
    self.coupling_map = coupling_map

  def __run__(self, dag):
    if self.layout == None:
      self.layout = simple_layout(dag.qreg,
                                  self.coupling_map)
    # Traverse the dag
    for gate in dag.topological_op_nodes():
      if gate.op_num == 2:
        p1 = self.layout[gate.op1]
        p2 = self.layout[gate.op2]
        s_path = self.coupling_map.shortest_path(p1, p2
     )
        for i in range(len(s_path)-1):
          dag.swap_and_update_gate(i, i+1)
    return dag
```

Fig. 5. Implementation of the `basic_swap` pass.

the coupling map. The implementation invokes a function `swap_and_update_gate(i,j)` to insert quantum gate `swap(i,j)`. In §4, we will see it is an invariant-guided contract design, whose invariant is `equiv(input_dag, output_dag)`, stating that the DAGCircuits before and after inserting `swap` are equivalent.

In Fig.5, comment `# @swap` in the first line is a developer annotation informing CertiQ that this is a swap pass such that the CertiQ engine can automatically generate its corresponding contract as below,

```
# It should preserve the semantics
def basic_swap_inv:
  return equiv(input_dag, output_dag)
```

which is the same with the invariant of the `swap_and_update_gate` method. For simplicity, here we just list the invariant, we will discuss other parts of its contract in §5.1.

3.3 The verification goal

The verification goal of the CertiQ engine is to prove that **every function call in the submitted code complies with its contract**. For example, when our `basic_swap` pass is submitted for code review, the verification engine will prove that the `basic_swap` pass and (specifications of) all the functions called by it comply with their contracts if they possess one. Any calls to the Qiskit library will be directed to their abstract specifications. CertiQ has already verified these library calls with respect ot their contracts.

3.4 Static analysis and code transformations

Before we hand over the `basic_swap` pass to the Z3 SMT solver, we statically analyze [30] its code and perform transformations to assist and speed up the symbolic execution. For example, the invocation of `simple_layout` will be replaced by,

```
# Check pre-condition
certiq_prove(simple_layout_pre(dag.qreg,
                               self.coupling_map))
# Impose post-condition
simple_layout_post(layout)
```

In this way, the symbolic execution does not need to unfold the implementation of `simple_layout` every time it is invoked.

Another powerful transformation that can further reduce the verification complexity is called *contract continuation*. Intuitively, if a function does nothing but calls another function in its function body, then the contract of the callee can be continued to the caller function if the caller does not have a contract. This applies to the `for` loops in the example of `basic_swap`. Since the inner loop only has one child `swap_and_update_gate`, thus, the invariant in its contract continues to the loop function. In CertiQ, the static analyzer progressively propagates the contract and tests if the contract holds. Then, by this continuation, the `basic_swap` pass have the invariant of `swap_and_update_gate` propagated to it, which validates its contract and finish the proof.

3.5 The visualizer

If there is a bug in the user code, the verifier will invoke the visualizer and generate a counter-example. If the verification cannot finish in a certain amount of time or there are unsupported external library calls in the submitted code, we exit with "undetermined."

3.6 The translation validator

For two important types of transpiler passes, `swap` insertion passes and CNOT gate optimization passes, if CertiQ cannot verify the code (either verification does not stop for a certain period of time or exits with "undetermined"), a translation validator will be invoked to validate the compilation results at runtime. The translation validator itself is verified by the CertiQ verification engine.

4 THE CERTIQ FRAMEWORK

In this section, we discuss the main technical contributions in CertiQ and how they are integrated into Qiskit as a quantum verification framework. Most of the proof techniques we introduce are for the purpose of reducing

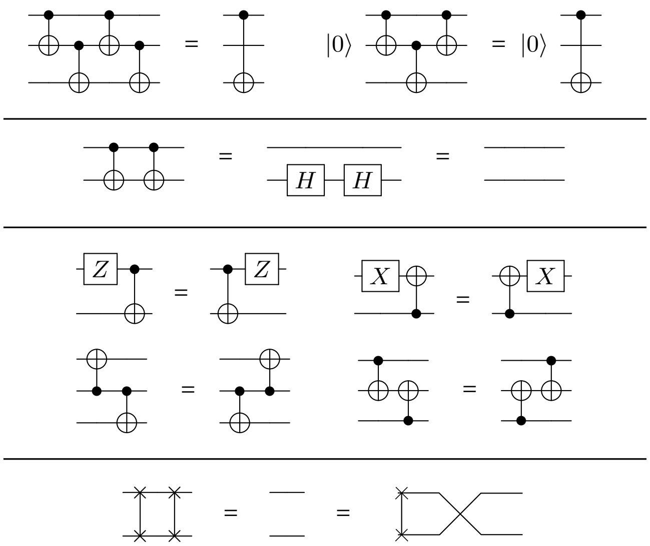


Fig. 6. Examples of certified primitive circuit moves implemented by `equiv` in CertiQ as small step semantics. They are bridging rules (above), cancellation rules (2nd line), commutativity rules (3rd, 4th line), swap rules (bottom).

the verification complexity, either of the underlying quantum computation or of potential state explosion from future code submission. We start a description of the formal semantics of quantum circuits defined in CertiQ, with which can be verified efficiently if a transformation is semantic preserving. We then discuss specifications for quantum data structures followed by the introduction of the invariant-guided contract design for library functions and contract continuation for user defined functions.

4.1 Primitive circuit moves

The problem of equivalence checking of quantum circuits falls into the complexity class of QMA (the quantum version of NP) [20], thus it is intractable to verify the equivalence of quantum circuits in their denotational semantics (unitary matrices). For efficient verification, in CertiQ we define small step operational semantics for DAGCircuits (quantum circuits). We call these small step semantics *primitive circuit moves.* These moves are implemented in CertiQ by the function `equiv` `:DAG` $\rightarrow$ `DAG` $\rightarrow$ `Boolean`. The input and output of primitive circuit moves are evaluated to be `True` by `equiv`. For example, `equiv`([`CNOT`(q_1, q_2), `Z`(q_1)], [`Z`(q_1), `CNOT`(q_1, q_2)]) $=$ `True` gives us the commutativity rule between the CNOT and the Z gate (see fig. 6). We still define the denotational semantics of quantum gates to be their unitary matrices so that the soundness of primitive circuit moves can be proved by direct simulation. Thus we can say that the equivalence relations we define are faithful to the underlying quantum computation. In this way, we define formal semantics that allow us to conduct efficient verification.

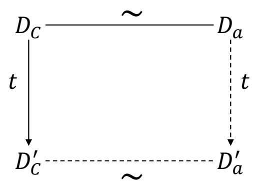


Fig. 7. The simulation diagram for refinement saying that, starting from a pair of equivalent concrete DAG D_C and abstract DAG D_a, for any transformation t, if t transforms D_C to D'_C, then t must transform D_a to an abstract DAG D'_a that is equivalent to D'_c.

4.2 Specification refinement

We say a specification and an implementation are equivalent if they satisfy the refinement property. Continuing with the example of DAGCircuit in §3, we describe how the equivalence of abstract specifications and implementation is proved in CertiQ. The refinement for DAGCircuit can be informally stated as: any transformation performed on a pair of equivalent abstract DAG (in specification) and concrete DAG (in implementation) should yield another pair of equivalent abstract DAG and concrete DAG. More precisely, in CertiQ we define the equivalence relation $\sim$ between a concrete DAG D_c and an abstract DAG D_a as,

$$D_c \sim D_a \iff \forall i. D_c.\text{topological\_op\_node} i = D_a i$$

where `topological_op_node` will return a list of gates in the DAG sorted by their dependence. Notice that $\sim$ is defined differently from `equiv` since that $\sim$ is a relation between abstract DAGs and concrete DAGs. With $\sim$ defined, if we perform any transformation, for example appending a gate, on D_c and D_a where $D_c \sim D_a$, then it must yield D'_c and D'_a respectively, where $D'_c \sim D'_a$ as well. This property can be illustrated by a simulation diagram [23] in Fig. 7.

In CertiQ, we prove the refinement property for `DAGCircuit` by symbolic execution in Z3 solver, thus proving the equivalence between the abstract specification and the real implementation.

Bloch sphere representation. Bloch sphere representation [4] gives us an example of how contextual refinement can be used to specify the behavior of quantum objects. The Bloch sphere representation can be viewed as an abstract specification of the qubit states, in which $|0\rangle$ is mapped to the north pole (see fig. 8). Specifically, Bloch sphere representation is a projection where the global phase of a qubit state $|\psi\rangle$ is omitted and $|\psi\rangle$ and $e^{i\gamma}|\psi\rangle$ are mapped to the same qubit state. The main part

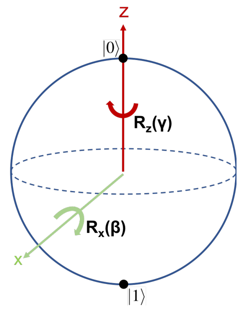


Fig. 8. The Bloch Sphere represents a single qubit. The $|0\rangle$ state is on the North Pole, the $|1\rangle$ state is on the South pole, and superposition states are in between. Single qubit gates correspond to rotations on the Bloch sphere. For instance, the Z gate rotates a qubit by angle π about the Z-axis.

of this projection can be seen trivially in the following Python code,

```
def Bloch_rep(gamma, theta, phi):return (theta, phi)
```

where a general qubit state $|\psi\rangle = e^{i\gamma}\cos\theta 2 \, |0\rangle + e^{i\gamma}e^{i\phi}\sin\theta 2 \, |1\rangle$ parameterized by γ, θ, ϕ is projected to the spherical coordinate θ, ϕ of a unit sphere. The Bloch sphere representation can be seen as an abstract specification of qubit states, where the equivalence relation $\sim$ is defined by the above `Bloch_rep` function. However, the contextual refinement property does not hold for it, since there is a transformation `tensor_product` together with other multi-qubit operations that breaks the diagram in fig. 7. Contextual refinement breaks because there is an untracked phase difference between qubits beyond the 1-qubit case. The relative phase change will induce non-trivial quantum computation that is not revealed in this representation. To address this issue, in CertiQ we explicit exclude any multi-qubit operations in the contract of any transformations conducted in the Bloch sphere representation. This restriction also applies to other qubit state representation in Qiskit Terra, for example the unit quarternion representation that will be discussed in §5.2.

4.3 Invariant-guided contracts

We describe how contracts in CertiQ assist the symbolic execution in verification and how invariant-guided contract design further reduces the complexity of symbolic execution. We also discuss user functions and show how to perform contract continuation and for user defined functions for which we cannot pre-define contracts.

Contracts in symbolic execution. We consider the general case in symbolic execution,

```
... # execution
terra_call(para)
... # continue execution
```

where `terra_call` is a Qiskit Terra library function, with call parameter *para*, pre-condition *pre(para)*, post-condition *post(para)*. As shown in §3, the contract is also parameterized by *para* (we leave out the invariant for now). We denote the program state before entering `terra_call` as s, then *para* also depends on s. If the program is correct, we have $s \Rightarrow pre\left(para\left(s\right)\right)$. By Modus ponens and s, we have $post\left(para\left(s\right)\right)$. Instead of continuing the symbolic execution inside the call, we directly conclude that the current program state is

$$s \wedge post\left(para\left(s\right)\right) \tag{1}$$

This execution can be accomplished because CertiQ preverifies the contract of library function `terra_call`.

Our approach of applying contracts to symbolic execution can be viewed as fine-grained predicate abstraction [8, 15] that minimizes the possibility of spurious counterexamples and greatly reduces verification complexity.

Invariant-guided contracts. If we assume further that `terra_call` is called in a loop,

```
... # execution
for i in range(n):
    terra_call(para)
... # continue execution
```

After 1 iteration, program state becomes eq. (1) described above. After 2 iterations, program state becomes

$$s \wedge postparas \wedge postparas \wedge postparas \tag{2}$$

This state can be seen by replacing s with $s \wedge postparas$ in eq. (1). After several iterations, the program state will become prohibitively complicated for efficient symbolic execution.

To reduce the verification complexity, CertiQ builds up a library of function calls with a desgin we call *invariant-guided contracts*. In the above case, an invariant *invs* becomes a fixed point. In a symbolic execution, a called method with contracts containing only invariant becomes "transparent" where the model checker has a lower verification burden. Invariant-guided contract design tries to implement interfaces that contains needed invariants.

The invariant-guided design is particularly interesting for CertiQ because for every primitive circuit moves in fig. 6, we can find an implementation that have an invariant. For example, the `swap_and_update_gate` method described in §3 has an invariant-guided contract.

To compare with the `basic_swap` function we wrote in the last section, we can look at the current implementation of the `basic_swap` pass in Qiskit Terra:

```python
for layer in dag.serial_layers():
    subdag = layer['graph']
    for gate in subdag.twoQ_gates():
        ... # Other codes

        # create the swap operation
        for swap in range(len(path) -2):
            swap_layer.apply_operation_back(SwapGate(),
                                args=[q1, q2],cargs=[])
        # layer insertion
        edge_map = current_layout.combine_into_edge_map
    (trivial_layout)
        new_dag.compose_back(swap_layer, edge_map)

        # Update layout
        for swap in range(len(path) -2):
            current_layout.swap(path[swap], path[swap +
                               1])
```

As we can see, there are two `for` loops in the outer loop and we cannot perform contract continuation to either loops because the `apply_operation_back` method and layout `swap` method does not have invariants in their contract. In general, implementation like this does not scale in symbolic execution.

4.4 Contracts for user defined functions

For user provided functions, CertiQ cannot not specify their contracts. Then the symbolic execution can fail because of the complexity of the code. CertiQ solves this problem in the following order: 1) CertiQ defines several commonly used contracts. With the user writing one line of function annotation, the corresponding pre-defined contract will be applied to the function and be tested if the function complies; 2) If the user does not provide annotation, CertiQ will perform contract continuation for the user function; and 3) If both 1 and 2 fail, the user function will continue with symbolic execution.

Pre-defined contracts for user functions. CertiQ provides several pre-defined contracts that users can apply to their functions, some of which can be automatically inferred. For example, if a function returns a layout, CertiQ will automatically add the bijection requirement for the returned layout. Some of the pre-defined contracts have to be hinted at by users. For example, if users provide the annotation "@coupling_map_path" at the beginning of a function, the CertiQ verification engine will apply the corresponding contract. Thus, the post-condition of a coupling map is applied, which is that the distance between the ith element and $i + 1$th element of the returned path must be 1 on the input coupling map.

Contract continuation. If no contract is assigned for a user function, CertiQ will perform contract continuation (invariant continuation). Similar to replacing real implementations with verified contracts, contract continuation can also be viewed as predicate abstraction, but with the possibility of failure. In the following general case, if `host_function` and `guest_func1()` and `guest_func2()` are user-defined functions, and the user provides a contract for the `host_function` that needs to be verified,

```python
# @host_contract
def host_function():
    ... # symbolic execution
    guest_func1()
    ... # symbolic execution
    guest_func2()
    ... # other calls
```

if we denote the program state before entering `guest_func1()` as s, the continued contract for `guest_func1()` is given by the continuation step at s (denoted as α),

$$\alpha s = \{c \in host\_inv : cs \wedge gs \models cgs\}$$

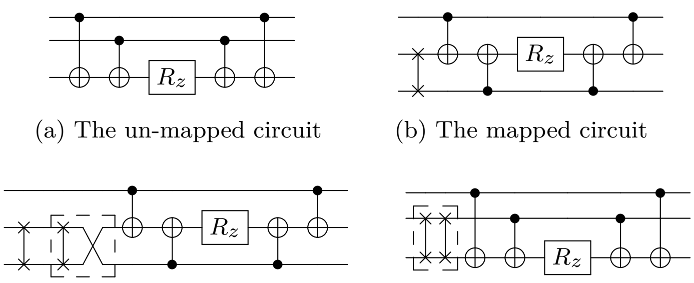
where $host\_inv$ is the invariant in the known host contract, gs is the execution of `guest_func1()` on the program state s. If αs is empty, it will be evaluated to $\perp$, which means the continuation fails. If the continuation for `guest_func1()` succeed, the verification engine will try contract continuation on `guest_func2()`. If contract continuation fails at some point, the verification engine will proceed with symbolic execution. Another contract continuation strategy is that verified contracts can be passed from guest functions to host functions, as described in the example of `basic_swap` pass in §3. In CertiQ, user is allowed to choose continuation strategies.

4.5 Loops and conditional statements

In CertiQ, we view loops and `if` statements as user defined functions. Users can provide loop-invariants and `if`-invariants for their contracts with a single line of annotation. For loops, if no loop invariant is given, CertiQ will try contract continuation. If no contract is assigned at the end of contract continuation, the verification will exit with "fail." For `if` statements, if no user provided invariant is given and contract continuation fails, in the static analysis stage, we generate two copies of the code with the if-else code block replaced by the "if" branch and the "else" branch, respectively. The strongest post-condition will then be derived with Z3 tactics to continue the symbolic execution.

4.6 The translation validator

CertiQ also provides a translation validator for `swap` passes and CNOT optimization passes if the symbolic execution fails. The translation validator can be viewed as a transformation pass that reverts the changes of the pass it validates and it is also verified by the CertiQ verifier. As an example, we illustrate the validation process of `swap` passes with an input circuit of 3 qubits in fig. 9. We assume the 3 qubits are coupled linearly, *i.e.*, the coupling map can be represented as $q_1 - q_2 - q_3$. For validating `swap` passes with the intput in fig. 9 (a), the validator finds `swap` gates in the mapped circuit and insert another `swap` gate right behind each with the `swap_and_update_gate` method (which is semantics preserving). After all `swap`s getting cancelled out by applying the `cancel_swap` method, we recover the input circuit. For validating the output of CNOT cancellation pass, a similar algorithm is performed.



(a) The un-mapped circuit (b) The mapped circuit

(c) Validation phase 1 (d) Validation phase 2

Fig. 9. Translation validation of `swap` passes. (a) and (b) show the input circuit and the output circuit of the `swap` pass. First, the validator performs a `swap_and_update_gate` (the dashed-line box in (c)) behind every `swap` in (b), which is a primitive circuit move. Then the validator performs the primitive circuit move `cancel_swap` (the dashed-line box in (d)). Then we recover the input circuit.

5 CASE STUDIES

Here we present case studies to show how CertiQ detects bugs and safety issues in realistic settings. We find a counter-example circuit that the `lookahead_swap` pass does not terminate on, we point out a bug in the `optimize_1q_gate` pass that can be addressed by specifying correct contracts for the pass and the function calls inside, and we identify two bugs in the `commutative_analysis` and `commutative_cancellation` passes when specifying their contracts.

5.1 `swap` insertion passes

`Swap` insertion passes are a fundamental stage of compiling for NISQ machines, where qubits are not fully connected and must be swapped on a chip to communicate. Because swap gates are error-prone and can corrupt the whole computation, these passes try to minimize the number of swap operations.

In the case study described here, we show the process of designing contracts for our verification goals and how it helps identify a bug in the `lookahead_swap` pass.

There are three verification goals for `swap` passes:

- The pass must be semantics preserving.
- The output DAGCircuit of the pass must conform to the coupling map of the physical device.
- The pass must terminate.

To prove each verification goal, we design a contract and perform a symbolic execution to verify the contract. The contract for the first goal is demonstrated in §3.2.

For the second verification goal, to ensure that the pass correctly accomplishes its goal of transforming the circuit to match device coupling constraints, we must verify that every 2-qubit gate in the output circuit operates on two neighboring physical qubits. Therefore, the following post-condition is added,

```
def swap_post(dag, layout, coup):
  n = dag.num_2q_gates()
  i = fresh_int()
  gate = dag.2q_gates[i]
  p1 = layout.v2p(gate.op1())
  p2 = layout.v2p(gate.op2())
  return ForAll([i], Implies(And(0 <= i, i < n),
                     coup.distance(p1, p2) == 1))
```

For the third verification goal, CertiQ does not try to solve it completely because it is undecidable. Instead, CertiQ aims to provide sound termination analysis for practical implementations. First, CertiQ concretizes the problem to verify the termination of passes with input circuit of bounded depth on a given coupling map. Termination can be proved by constructing strictly monotonic functions in a finite domain. For program states that are not in a loop or a recursive function, the program counter is a monotonic function to provide termination guarantee. For variable-length loops and while loops, CertiQ allows users to provide the monotonic function, for example,

```
gates_remaining = dag.topological_op_node()
while gates_remaining != 0:
# @mono: -gates_remaining.size
  ... # implementation code
```

Then in the backend SMT solver, the verifier solves for the circuit input that keeps `gates_remaining.size` unchanged and gives it as a counter example.

We verified three `swap` insertion passes: `basic_swap`, `lookahead_swap` and `noise_adaptive_swap`. We report that all three passes comply with the first two contracts. However, we find a counter-example circuit on coupling map of the IBM 16 qubit device, where the `lookahead_swap` pass does not terminate on (see fig. 10). The `lookahead_swap` pass greedily finds the next best 4 `swap` gates to minimize the totl distance of the unmapped

2-qubit gates. However, the counter example we found shows that the 4 `swap` gates can cancel each other out with the swap rules in fig. 6, thus `gate_remaining.size` will not update.

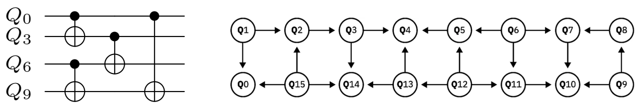


Fig. 10. (left) A counter-example generated by CertiQ that shows Qiskit's `lookahead_swap` pass does not always terminate on the IBM 16 qubit device. (right) The coupling map of the IBM 16 qubits device. Arrows indicate available CNOT directions (which does not affect the swap insertion step).

5.2 The `optimize_1q_gate` pass

We next focus on verifying the `optimize_1q_gate` pass and show that, with contract-based design, we can reveal bugs only arise in quantum software.

We verify the re-implemented `optimize_1q_gate` pass using a primitive `merge_1q_gate` method. This pass collapses a chain of single-qubit gates into a single, more efficient gate [39], to mitigate noise accumulation. It operates on u_1, u_2, u_3 gates, which are native gates in the IBM devices. These gates can be naturally describe as linear operations on the Bloch sphere, for example, u_1 gates are rotations with respect to the Z axis. For clarity, we list their matrix representations in table 1.

$$u_1\lambda = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}, \quad u_2\phi, \lambda = \frac{\sqrt{2}}{2}\begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i\lambda+\phi} \end{pmatrix}$$

$$u_3\theta, \phi, \lambda = \frac{\sqrt{2}}{2}\begin{pmatrix} \cos\theta & -e^{i\lambda}\sin\theta \\ e^{i\phi}\sin\theta & e^{i\lambda+\phi}\cos\theta \end{pmatrix}$$

Tab. 1. Matrix representation of physical gates u_1, u_2 and u_3. u_1 is a Z rotation on the Bloch sphere.

The `optimize_1q_gate` pass has two function calls. First, it calls the `collect_runs` method to collect groups of consecutive u_1, u_2, u_3 gates. Then it calls `merge_1q_gate` to merge the gates in each group. `merge_1q_gate` first transforms the single qubit gates from the Bloch sphere representation to the unit quaternion representation [17], then the rotation merges are performed in that representation.

As described in §4.2, the contracts of these two representations allow only single-qubit operations required by the refinement property. However, every Qiskit gate can be modified with a `c_if` or `q_if` method to condition its execution on the state of other classical or quantum bits. When the transpiler pass attempts to optimize these conditional gates, it can lead to a wrong circuit. For this reason, in the contract and implementation of `collect_runs`, we have to include that `gate1.q_if == False` and `gate1.c_if == False`.

The bugs described above, which relate to how quantum circuit instructions can be conditioned, have been observed in Qiskit in the past [32, 33]. In the absence of rigorous verification like this work, such bugs are hard to discover. In practice, this is usually done via extensive randomized testing of input/output circuits, which does not provide any guarantee. The results demonstrate that our contract for `merge_1q_gate` based on contextual refinement is effective for detecting quantum-related bugs.

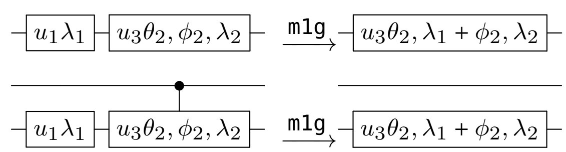


Fig. 11. Correct execution (top) and incorrect execution (bottom) of `merge_1q_gate`.

5.3 `commutation` passes

`commutation_analysis` and `commutative_cancellation` is a pair of Transpiler passes that optimizes DAGCircuits using the quantum commutation rules and the cancellation rules in fig. 6. First, `commutation_analysis` transforms the DAGCircuit to a representation called commutation groups [40], where nearby gates that commute with each other are grouped together. Then `commutative_cancellation` performs cancellation inside groups. In fig. 12, we give a working example.

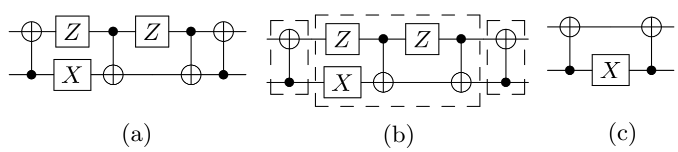


Fig. 12. A working example of `commutation_analysis` and `commutative_cancellation`. (a) the un-optimized circuit; (b) `commutation_analysis` forms the commutation group (c) `commutative_cancellation` cancels self-inverse gate inside groups.

We find two bugs when verifying this pair of passes. First, the commutation group can be viewed as an abstract specification for the DAGCircuit. However, when specifying the contract of `commutation_analysis`, we find that the commutation group representation violates contextual refinement property defined in §4.2. This violation comes from the fact that the commutation relation is in general not transitive. For example, if we denote the commutation relation as $\sim$ and there is 3 quantum gates, A, B, C where $A \sim B$, $B \sim C$, then $A \sim C$ is not guaranteed to be true. For this reason, gates with pairwise commutation relation cannot be grouped together. We propose two solutions to this bug. First, we can make sure the circuits that these passes operate on have a limited gate set where $\sim$ is indeed transitive. For example, in the gate set $\{$CNOT, X, Z, H, T, $u_1, u_2, u_3\}$, $\sim$ is transitive. Second, we can use a new algorithm that does not assume transitivity but is potentially less efficient.

The second bug we find is when specifying the contract of `commutative_cancellation`. The bug is the same to the bug in §5.2, where the pass cancels gates with a classical control. This bug can be fixed by asserting that `gate.c_if==False` and `gate.q_if==False`.

6 EVALUATION

We performed verification of four transpiler phases with seven implementations. The seven implementations are: the `collect_2q_block` pass, the `commutative_analysis` pass, the `commutative_cancellation` pass, the `lookahead_swap` pass, the `basic_swap` pass, the `noise_adaptive_swap` pass, the `optimize_1q_gate` pass. These passes are all re implemented with invariant-guided contract design. With seven successful verifications (none exit with "undetermined"), we report three bugs. We evaluate CertiQ based on the verification of these implementations.

Verification performance. With invariant-guided contract design, all transpiler passes can be verified within seconds. However, most of the original implementations can not be verified or take long time to verify.

Run-time performance. When executing, verified code still calls functions from the Terra library, not from the specifications in CertiQ, so its performance is not affected. However, the interface of primitive circuit moves indeed adds some constraints on the allowed operations. For example, when using the `swap_and_update_gate` method in a loop to swap along a path, the complexity is On^2, where n is the number of gates in the circuit. While using the `swap` method in a loop to achieve the same is of complexity On. We mitigate this problem by verifying the efficient implementation and providing it as a primitive move. For example, in CertiQ we verify the above implementation with the `swap` and provide it as the `swap_along_path` method.

Extensibility. The contract-based design provides a powerful abstraction for complicated optimization algorithms, thus we believe CertiQ is extensible to future transpiler implementations. On the other hand, since CeritQ already provides abstract specifications and contracts for important data structures in Qiskit, we expect CertiQ to be very extensible for other components of the Qiskit toolchain that rely on the same infrastructure.

7 RELATED WORKS

Quantum programming environments with a verifier. Several quantum programming environments support the verification of quantum programs running on it. For example, in the QWire quantum language [35, 37], programmers can use the embedded verifier based on the Coq proof assistant [7] to perform mechanized proof for their programs. The $Q|SI\rangle$ programming environment [25] allows users to reason about their programs written in the quantum **while**-language with quantum Floyd-Hoare logic [47]. In contrast to CertiQ, these environments require expertise both in quantum computing and formal verification to construct proofs and these proofs verify at the program level, not the compilation level.

Verified quantum-related compilers. Previous studies on compiler verification for reversible circuits [3], ancillae uncomputation [35, 36] and compiler optimizations [18] utilize interactive theorem provers such as F* [28] and Coq [7] to conduct manual proofs, which do not provide an extensible interface for developers to verify future extensions with low proof burden. In contrast, CertiQ verification framework allows developers to plugin their implementations that can be verified in a mostly-automated manner.

Algorithms to perform efficient quantum circuits equivalence checking have been discussed from the view of quantum algorithms [45], quantum communication protocols [5], and verification of compilation [2]. However, while powerful, these checking algorithms are too complicated to automated verification like we use with CeritQ.

Model checking in quantum computation verification. The early adoption of Model checking in quantum computation focused on verifying quantum communication protocols [10, 16, 43, 48]. Recently, model checking techniques have been applied to more areas, including quantum Markov chain analysis [12–14], checking physical systems, program analysis [21, 24, 49]. However no automated verification tool based on model checking like CertiQ exists for quantum computing until now.

8 CONCLUSION AND FUTURE WORK

We describe CertiQ, a mostly-automated verification framework for the Qiskit compiler (called Terra) that addresses the key challenges of verifying a real-world quantum compiler. CeritQ gives small-step semantics to quantum circuits for efficient verification of compiler transformations and adapts a combination of proof techniques to reduce verification complexity from the underlying quantum computation as well as from the large user code space. These proof techniques include SMT reasoning, contract-based design, invariant contracts and contract continuation. With extensive case studies, we demonstrate that CertiQ can detect critical bugs, some of which are unique to quantum software.

To our knowledge, our work with CertiQ is the first effort to apply formal verification and SMT reasoning to a real-world commercial quantum compiler. The approach we establish with CertiQ paves the way for end-to-end verification of a complete quantum software toolchain, an important step towards practical near-term quantum computing. Going forward, we are working to use our contract-based approach to incorporate verification of both higher and lower components of Qiskit. These include Qiskit Aqua, the high-level quantum algorithm library and the OpenPulse interface [27], which implements quantum operations through microwave control pulses.

9 ACKNOWLEDGMENT

Y.S is funded in part by the QISE-NET fellowship, under NSF grant DMR-1747426. This work is funded in part by EPiQC, an NSF Expedition in Computing, under grant CCF-1730449, and in part by the STAQ project, under NSF grant Phy-1818914. This work is also supported in part by NSF grant 1918400 and a research gift from Qtum Foundation.

REFERENCES

[1] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. Qiskit: An open-source framework for quantum computing, 2019.

[2] Matthew Amy. Towards Large-scale Functional Verification of Universal Quantum Circuits. *Electronic Proceedings in Theoretical Computer Science*, 287:1–21, jan 2019.

[3] Matthew Amy, Martin Roetteler, and Krysta M. Svore. Verified Compilation of Space-Efficient Reversible Circuits. In *Computer Aided Verification*, pages 3–21. International Conference on Computer Aided Verification, 2017.

[4] F Bloch. Nuclear Induction. *Phys. Rev.*, 70(7-8):460–474, oct 1946.

[5] Juan Carlos Garcia-Escartin and Pedro Chamorro-Posada. Equivalent Quantum Circuits. Technical report, Universidad de Valladolid, Dpto. Teoria de la Senal e Ing., 2011.

[6] Cirq, 2016.

[7] The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at http://coq.inria.fr/doc.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified latice model. *POPL '77 Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[9] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. Validating quantum computers using randomized model circuits. *arXiv*, nov 2018.

[10] T. Davidson, S.J. Gay, H. Mlnarik, R. Nagarajan, and N. Papanikolaou. Model checking for communicating quantum processes. *arXiv*, 2012.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 LNCS, pages 337–340, 2008.

[12] Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Shenggang Ying. Model Checking Omega-regular Properties for Quantum Markov Chains. *DROPS-IDN/7781*, 2017.

[13] Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Lijun Zhang. QPMC: A model checker for quantum programs and protocols. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9109, pages 265–272. Springer Verlag, 2015.

[14] Yuan Feng, Nengkun Yu, and Mingsheng Ying. Model checking quantum Markov chains. In *Journal of Computer and System Sciences*, volume 79, pages 1181–1198, nov 2013.

[15] Cormac Flanagan, Shaz Qadeer, Cormac Flanagan, and Shaz Qadeer. Predicate abstraction for software verification. *ACM SIGPLAN Notices*, 37(1):191–202, 2002.

[16] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. QMC: A model checker for quantum systems - Tool paper. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5123 LNCS, pages 543–547, 2008.

[17] Philippe Gille and Tamas Szamuely. *Central Simple Algebras and Galois Cohomology*. Cambridge University Press, 2009.

[18] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. Verified Optimization in a Quantum Intermediate Representation. *arxiv*, apr 2019.

[19] IBM-Q Experience, 2016.

[20] D Janzing, P Wocjan, and T Beth. Identity check is QMA-complete. *eprint arXiv:quant-ph/0305050*, may 2003.

[21] Ali Javadi-Abhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 1:1—-1:10, New York, NY, USA, 2014. ACM.

[22] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Technical report, UIUC, 2003.

[23] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 2009.

[24] Yangjia Li and Mingsheng Ying. Algorithmic analysis of termination problems for quantum programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, dec 2017.

[25] Shusen Liu, Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan, and Mingsheng Ying. Q| SI?: A quantum programming environment. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11180 LNCS, pages 133–164. Springer Verlag, 2018.

[26] D Maslov, G W Dueck, D M Miller, and C Negrevergne. Quantum Circuit Simplification and Level Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, mar 2008.

[27] David C. McKay, Thomas Alexander, Luciano Bello, Michael J. Biercuk, Lev Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, and Jay M. Gambetta. Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments. *arXiv*, sep 2018.

[28] Microsoft. F*, 2016.

[29] John Preskill. Quantum Computing in the NISQ era and beyond. *ArXiv e-prints*, jan 2018.

[30] PyCQA. Astroid, 2013.

[31] Pyquil, 2016.

[32] Qiskit bug report, 2018.

[33] Qiskit bug report, 2018.

[34] Qutip, 2012.

[35] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. ReQWIRE: Reasoning about Reversible Quantum Circuits. *arXiv e-prints*, page arXiv:1901.10118, Jan 2019.

[36] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. ReQWIRE: Reasoning about Reversible Quantum Circuits. Technical report, University of Pennsylvania, 2019.

[37] Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. *Electronic Proceedings in Theoretical Computer Science*, 2018.

[38] M. Schuld, M. Fingerhuth, and F. Petruccione. Implementing a distance-based classifier with a quantum interference circuit. *EPL (Europhysics Letters)*, 2017.

[39] Sarah Sheldon, Jay M. Gambetta, David C. McKay, Christopher J. Wood, and Jerry M. Chow. Efficient Z gates for quantum computing . *Physical Review A*, 96(2):1–8, 2017.

[40] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffmann, and Frederic T. Chong. Optimized Compilation of Aggregated Instructions for Realistic Quantum Computers. In *ASPLOS*, Providence, Rhode Island, 2019.

[41] Scott E. Smart, David I. Schuster, and David A. Mazziotti. Experimental data from a quantum computer verifies the generalized Pauli exclusion principle. *Communications Physics*, 2019.

[42] Mathias Soeken and Michael Kirkedal Thomsen. White Dots do Matter: Rewriting Reversible Logic Circuits. In Gerhard W Dueck and D Michael Miller, editors, *Reversible Computation*, pages 196–208, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[43] Amir M. Tavala, Soroosh Nazem, and Ali A. Babaei-Brojeny. Verification of quantum protocols with a probabilistic model-checker. In *Electronic Notes in Theoretical Computer Science*, volume 270, pages 175–182, feb 2011.

[44] Qiskit Terra Github issue page, 2018.

[45] George F. Viamontes, Igor L. Markov, and John P. Hayes. Checking equivalence of quantum circuits and states. In Georges G E Gielen, editor, *ICCAD*. IEEE, 2007.

[46] Christophe Vuillot. Is error detection helpful on IBM 5Q chips ? *arXiv*, 2017.

[47] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems*, 33(6):1–49, dec 2011.

[48] Mingsheng Ying and Yuan Feng. Model Checking Quantum Systems — A Survey. *arXiv*, jul 2018.

[49] Mingsheng Ying, Yangjia Li, Nengkun Yu, and Yuan Feng. Model-Checking Linear-Time Properties of Quantum Systems. *ACM Transactions on Computational Logic*, 15(3):1–31, aug 2014.

[50] Zhikuan Zhao, Alejandro Pozas-Kerstjens, Patrick Rebentrost, and Peter Wittek. Bayesian deep learning on a quantum computer. *Quantum Machine Intelligence*, 2019.