An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits

YU-FANG CHEN, Academia Sinica, Taiwan
KAI-MIN CHUNG, Academia Sinica, Taiwan
ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic
JYUN-AO LIN, Academia Sinica, Taiwan
WEI-LUN TSAI, Academia Sinica, Taiwan and National Taiwan University, Taiwan
DI-DE YEN, Max Planck Institute for Software Systems, Germany

We introduce a new paradigm for analysing and finding bugs in quantum circuits. In our approach, the problem is given by a triple $\{P\}\, C\, \{Q\}$ and the question is whether, given a set P of quantum states on the input of a circuit C, the set of quantum states on the output is equal to (or included in) a set Q. While this is not suitable to specify, e.g., functional correctness of a quantum circuit, it is sufficient to detect many bugs in quantum circuits. We propose a technique based on *tree automata* to compactly represent sets of quantum states and develop transformers to implement the semantics of quantum gates over this representation. Our technique computes with an algebraic representation of quantum states, avoiding the inaccuracy of working with floating-point numbers. We implemented the proposed approach in a prototype tool and evaluated its performance against various benchmarks from the literature. The evaluation shows that our approach is quite scalable, e.g., we managed to verify a large circuit with 40 qubits and 141,527 gates, or catch bugs injected into a circuit with 320 qubits and 1,758 gates, where all tools we compared with failed. In addition, our work establishes a connection between quantum program verification and automata, opening new possibilities to exploit the richness of automata theory and automata-based verification in the world of quantum computing.

CCS Concepts: • **Hardware → Quantum computation**; • **Theory of computation → Tree languages**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: quantum circuits, tree automata, verification

ACM Reference Format:
Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023. An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits. *Proc. ACM Program. Lang.* 7, PLDI, Article 156 (June 2023), 26 pages. https://doi.org/10.1145/3591270

1 INTRODUCTION

The concept of *quantum computing* appeared around 1980 with the promise to solve many problems challenging for classical computers. Quantum algorithms for such problems started appearing later, such as Shor's factoring algorithm [Shor 1994], a solution to the hidden subgroup problem by Ettinger *et al.* [Ettinger et al. 2004], Bernstein-Vazirani's algorithm [Bernstein and Vazirani 1993],

Authors' addresses: Yu-Fang Chen, Academia Sinica, Institute of Information Science, Taiwan, yfc@iis.sinica.edu.tw; Kai-Min Chung, Academia Sinica, Institute of Information Science, Taiwan, kmchung@iis.sinica.edu.tw; Ondřej Lengál, Faculty of Information Technology, Brno University of Technology, Czech Republic, lengal@fit.vutbr.cz; Jyun-Ao Lin, Academia Sinica, Taiwan, jyalin@gmail.com; Wei-Lun Tsai, Academia Sinica, Institute of Information Science, Taiwan, alan23273850@gmail.com and National Taiwan University, Graduate Institute of Electronics Engineering, Taiwan; Di-De Yen, Max Planck Institute for Software Systems, Germany, bottlebottle13@gmail.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/6-ART156
https://doi.org/10.1145/3591270

or Grover's search [Grover 1996]. For a long time, no practical implementation of these algorithms has been available due to the missing hardware. Recent years have, however, seen the advent of quantum chips claiming to achieve *quantum supremacy* [Arute et al. 2019], i.e., the ability to solve a problem that a state-of-the-art supercomputer would take thousands of years to solve. As it seems that quantum computers will occupy a prominent role in the future, systems and languages for their programming are in active development (e.g., [Altenkirch and Grattage 2005; Green et al. 2013; Wille et al. 2019]), and efficient quantum algorithms for solutions of real-world problems, such as machine learning [Biamonte et al. 2017; Ciliberto et al. 2018], optimization [Moll et al. 2018], or quantum chemistry [Cao et al. 2019], have started appearing.

The exponential size of the underlying computational space and the probabilistic nature makes it, however, extremely challenging to reason about quantum programs—both for human users and automated analysis tools. Currently, existing automated analysis approaches are mostly unable to handle large-scale circuits [Feng et al. 2017, 2015; Ying 2021; Ying and Feng 2021; Ying et al. 2014], inflexible in checking user-specified properties [Amy 2018; Burgholzer and Wille 2020; Coecke and Duncan 2011; Fagan and Duncan 2019; Green et al. 2013; Niemann et al. 2016; Pednault et al. 2017; Samoladas 2008; Tsai et al. 2021; Viamontes et al. 2009; Wecker and Svore 2014; Zulehner et al. 2019; Zulehner and Wille 2019], or imprecise and unable to catch bugs [Perdrix 2008; Yu and Palsberg 2021]. Scalable and flexible automated analysis tools for quantum circuits are indeed missing.

In this paper, we propose a new paradigm for analysing and finding bugs in quantum circuits. In our approach, the problem is given by a triple $\{P\}\, C\, \{Q\}$, where C is a quantum circuit and P and Q are sets of quantum states. The verification question that we address is whether the set of output quantum states obtained by running C on all states from P is equal to (or included in) the set Q. While this kind of property is not suitable to specify, e.g., functional correctness of a quantum circuit, it is sufficient to obtain a lot of useful information about a quantum circuit, such as finding constants (will a circuit evaluate to the same quantum state for all inputs in P) or detecting bugs.

We create a framework for analysing the considered class of properties based on *(finite) tree automata* (TAs) [Comon et al. 2008]. Languages of TAs are set of trees; in our case, we consider TAs whose languages contain full binary trees with the height being the number of qubits in the circuit. Each branch (a path from a root to a leaf) in such a tree corresponds to one *computational basis state* (e.g., $|0000\rangle$ or $|0101\rangle$ for a four-qubit circuit), and the corresponding leaf represents the *complex amplitude* of the state (we use an algebraic encoding of complex numbers by tuples of integers to have a precise representation and avoid possible inaccuracies when dealing with floating-point numbers[1]; this encoding is sufficient for a wide variety of quantum gates, including the Clifford+T universal set [Boykin et al. 2000]). Sets of such trees can be in many cases encoded compactly using TAs, e.g., storing the output of Bernstein-Vazirani's algorithm [Bernstein and Vazirani 1993] over n qubits requires a vector of 2^n complex numbers, but can be encoded by a linear-sized TA. For each quantum gate, we construct a transformation that converts the input states TA to a TA representing the gate's output states, in a similar way as classical program transformations are represented in [D'Antoni et al. 2015]. Testing equivalence and inclusion between the TA representing the set of outputs of a circuit and the postcondition Q (from $\{P\}\, C\, \{Q\}$) can then be done by standard TA language inclusion/equivalence testing algorithms [Abdulla et al. 2008, 2007; Comon et al. 2008; Lengál et al. 2012]. If the test fails, the framework generates a witness for diagnosis.

One application of our framework is as a quick *underapproximation of a quantum circuit non-equivalence test*. Our approach can switch to a lightweight specification when equivalence checkers fail due to insufficient resources and still find bugs in the design. Quantum circuit (non-)equivalence testing is an essential part of the quantum computing toolkit. Its prominent use is in verifying

[1]Integer numbers of an arbitrary precision can be handled, e.g., by the popular `GMP` [GMP 2022] package.

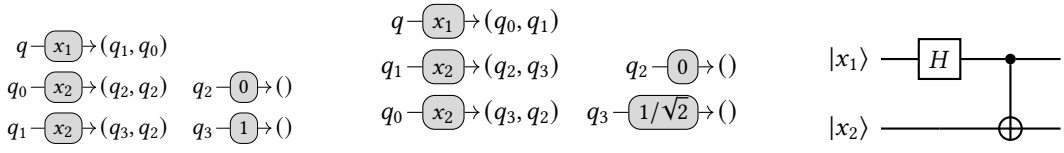
results of circuit optimization, which is a necessary part of quantum circuit compilation in order to achieve the expected fidelity of quantum algorithms running on real-world quantum computers, which are heavily affected by noise and decoherence [Amy 2019; Hattori and Yamashita 2018; Hietala et al. 2019; Itoko et al. 2020; Moll et al. 2018; Nam et al. 2018; Peham et al. 2022; Soeken et al. 2010; Xu et al. 2022b]. Already in the world of classical programs, optimizer bugs are being found on a regular basis in compilers used daily by tens of thousands of programmers (see, e.g., [Livinskii et al. 2020]). In the world of quantum, optimization is much harder than in the classical setting, with many opportunities to introduce subtle and hard-to-discover bugs into the optimized circuits. It is therefore essential to be able to check that an output of an optimizer is functionally equivalent to its input. Moreover, global optimization techniques, such as genetic algorithms [Massey et al. 2005; Spector 2006], may use (somehow quantified) circuit (non-)equivalence as the fitness function.

Testing quantum circuit (non-)equivalence is, however, a challenging task (QMA-complete [Janzing et al. 2005]). Due to its complexity, approaches that can quickly establish circuit non-equivalence are highly desirable to be used, e.g., as a preliminary check before a more heavy-weight procedure, such as [Burgholzer and Wille 2020; Peham et al. 2022; Viamontes et al. 2007; Wei et al. 2022; Yamashita and Markov 2010], is used. One currently employed fast non-equivalence check is to use random stimuli generation [Burgholzer et al. 2021]. Finding subtle bugs by random testing is, however, challenging with no guarantees due to the immense (in general uncountable) underlying state space.

Our approach can be used as follows: we start with a TA encoding the set of possible input states (created by the user or automatically) and run our analysis of the circuit over it, obtaining a TA $\mathcal{A}$ representing the set of all outputs. Then, we take the optimized circuit, run it over the same TA with inputs and obtain a TA $\mathcal{A}'$. Finally, we check whether $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. If the equality does not hold, we can conclude that the circuits are not functionally equivalent (if the equality holds, there can, however, still be some bug that does not manifest in the set of output states).

We implemented our technique in a prototype called AutoQ and evaluated it over a wide range of quantum circuits, including some prominent quantum algorithms, randomly generated circuits, reversible circuits from RevLib [Wille et al. 2008], and benchmarks from the tool Feynman [Amy 2018]. The results show that our approach is quite scalable. We did not find any tool with the same functionality with ours and hence pick the closest state-of-the art tools: a circuit simulator SliQSim [Tsai et al. 2021] and circuit equivalence checkers Feynman [Amy 2018] (based on path-sum) and Qcec [Burgholzer and Wille 2020] (combining ZX-calculus, decision diagrams, and random stimuli generation), as the baseline tools to compare with. In the first experiments, we evaluated AutoQ's capability in verification against pre- and post-conditions. We managed to verify the functional correctness (w.r.t. one input state) of a circuit implementing Grover's search algorithm with 40 qubits and 141,527 gates. We then evaluated AutoQ on circuits with injected bugs. The results confirm our claim—AutoQ was able to find injected bugs in various huge-scale circuits, including one with 320 qubits and 1,758 gates, which the other tools failed to find.

In addition to the practical utility, our work also bridges the gap between quantum and classical verification, particularly automata-based approaches such as *regular (tree) model checking* [Bouajjani et al. 2012, 2000; Neider and Jansen 2013] or string manipulation verification [Yu et al. 2008, 2011]. As far as we know, our approach to verification of quantum circuits is the first based on automata. The enabling techniques and concepts involved in this work are, e.g., the use of TAs to represent sets of quantum states and express the pre- and post-conditions, the compactness of the TA structure enabling efficient gate operations, and our TA transformation algorithms enabling the execution of quantum gates over TAs. We believe that the connection of automata theory with the quantum world we establish can start new fruitful collaborations between the two rich fields.



(a) The TA of $|00\rangle$.

(b) The TA of $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

(c) The EPR circuit

Fig. 1. Constructing the Bell state

Overview: We use a concrete example to demonstrate how to use our approach. Assume that we want to design a circuit constructing the Bell state, i.e., a 2-qubit circuit converting a basis state $|00\rangle$ to a maximally entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. We first prepare TAs corresponding to the precondition (Fig. 1a) and postcondition (Fig. 1b). Both TAs use q as the root state and accept only one tree. One can see the correspondence between quantum states and TAs by traversing their structure. The precise definition will be given in Section 2 and Section 3. Our approach will then use the transformers from Sections 4 to 6 to construct a TA $\mathcal{A}$ recognizing the quantum states after executing the EPR circuit (Fig. 1c) from the precondition TA (Fig. 1a). We will then use TA language inclusion/equivalence tool VATA [Lengál et al. 2012] to check $\mathcal{A}$ against the postcondition TA. If the circuit is buggy, our approach will return a witness quantum state that is reachable from the precondition, but not allowed by the postcondition. From our experience of preparing benchmark examples, in many cases, this approach helps us finding out bugs from incorrect designs.

2 PRELIMINARIES

We assume basic knowledge of linear algebra and quantum circuits. Below, we only give a short overview and fix notation; see, e.g., the textbook [Nielsen and Chuang 2011] for more details.

By default, we work with vectors and matrices over complex numbers $\mathbb{C}$. In particular, we use $\mathbb{C}^{m\times n}$ to denote the set of all $m \times n$ complex matrices. Given a $k \times \ell$ matrix (a_{xy}), its *transpose* is the $\ell \times k$ matrix $(a_{xy})^T = (a_{yx})$ obtained by flipping (a_{xy}) over its diagonal. A $1 \times k$ matrix is called a *row vector* and a $k \times 1$ matrix is called a column vector. To save vertical space, we often write a column vector v using its row transpose v^T. We use I to denote the *identity* matrix of any dimension (which should be clear from the context). The *conjugate* of a complex number $a+bi$ is the complex number $a-bi$, and the *conjugate transpose* of a matrix $A = (a_{xy})$ is the matrix $A^\dagger = (c_{yx})$ where c_{yx} is the conjugate of a_{yx}. For instance, $\begin{pmatrix} 1+i & 2-2i & 3 \\ 4-7i & 0 & 0 \end{pmatrix}^\dagger = \begin{pmatrix} 1-i & 4+7i \\ 2+2i & 0 \\ 3 & 0 \end{pmatrix}$. The *inverse* of a matrix A is denoted as A^{-1}. A square matrix A is *unitary* if $A^\dagger = A^{-1}$. The *Kronecker product* of $A = (a_{xy}) \in \mathbb{C}^{k\times\ell}$ and $B \in \mathbb{C}^{m\times n}$ is the $km \times \ell n$ matrix $A \otimes B = (a_{xy}B)$, for instance,

$$\begin{pmatrix} 1+i & 3 \\ 4-7i & 0 \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} = \begin{pmatrix} (1+i)\cdot\begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} & 3\cdot\begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} \\ (4-7i)\cdot\begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} & 0\cdot\begin{pmatrix} \frac{1}{2} & 1 \\ -\frac{1}{2} & 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \frac{1}{2}+\frac{1}{2}i & 1+i & \frac{3}{2} & 3 \\ -\frac{1}{2}-\frac{1}{2}i & 0 & -\frac{3}{2} & 0 \\ 2-\frac{7}{2}i & 4-7i & 0 & 0 \\ -2+\frac{7}{2}i & 0 & 0 & 0 \end{pmatrix}. \quad (1)$$

2.1 Quantum Circuits

Quantum states. In a quantum system with n qubits, the qubits can be entangled, and its *quantum state* can be a quantum superposition of *computational basis states* $\{|j\rangle \mid j \in \{0,1\}^n\}$. For instance, given a system with three qubits x_1, x_2, and x_3, the computational basis state $|011\rangle$ denotes a state where qubit x_1 is set to 0 and qubits x_2 and x_3 are set to 1. The superposition is then denoted in the

Dirac notation as a formal sum $\sum_{j\in\{0,1\}^n} a_j \cdot |j\rangle$, where $a_0, a_1, \ldots, a_{2^n-1} \in \mathbb{C}$ are *complex amplitudes*[2] satisfying the property that $\sum_{j\in\{0,1\}^n} |a_j|^2 = 1$. Intuitively, $|a_j|^2$ is the probability that when we measure the state in the computational basis, we obtain the state $|j\rangle$; these probabilities need to sum up to 1 for all computational basis states. We note that the quantum state can alternatively be represented by a 2^n-dimensional column vector[3] $(a_0, \ldots, a_{2^n-1})^T$ or by a function $T\colon \{0,1\}^n \to \mathbb{C}$, where $T(j) = a_j$ for all $j \in \{0,1\}^n$. In the following, we will work mainly with the function representation, which we will see as a mapping from the domain of assignments to Boolean variables (corresponding to qubits) to $\mathbb{C}$. For instance, the quantum state $\frac{1}{\sqrt{2}} \cdot |00\rangle + \frac{1}{\sqrt{2}} \cdot |01\rangle$ can be represented by the vector $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0, 0)^T$ or the function $T = \{00 \mapsto \frac{1}{\sqrt{2}}, 01 \mapsto \frac{1}{\sqrt{2}}, 10 \mapsto 0, 11 \mapsto 0\}$.

Quantum gates. Operations in quantum circuits are represented using quantum gates. A k-qubit *quantum gate* (i.e., a quantum gate with k inputs and k outputs) can be described using a $2^k \times 2^k$ unitary matrix. When computing the effect of a k-qubit quantum gate U on the qubits $x_\ell, \ldots, x_{\ell+k-1}$ of an n-qubit quantum state represented using a 2^n-dimensional vector v, we proceed as follows. First, we compute an auxiliary matrix $U' = I_{n-(\ell+k-1)} \otimes U \otimes I_{\ell-1}$ where I_j denotes the 2^j-dimensional identity matrix. Note that if U is unitary, then U' is also unitary. Then, the new quantum state is computed as $v' = U' \times v$. For instance, let $n = 2$ and U be the Pauli-X gate applied to the qubit x_1.

$$X' = X \otimes I = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \quad v' = X' \times v = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{pmatrix} = \begin{pmatrix} c_{10} \\ c_{11} \\ c_{00} \\ c_{01} \end{pmatrix} \tag{2}$$

Representation of complex numbers. In order to achieve accuracy with no loss of precision, in this paper, when working with $\mathbb{C}$, we consider only a subset of complex numbers that can be expressed by the following algebraic encoding proposed in [Zulehner and Wille 2019] (and also used in [Tsai et al. 2021]):

$$\left(\frac{1}{\sqrt{2}}\right)^k (a + b\omega + c\omega^2 + d\omega^3), \tag{3}$$

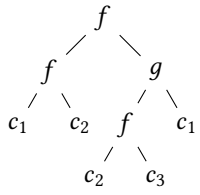
where $a, b, c, d, k \in \mathbb{Z}$ and $\omega = e^{\frac{i\pi}{4}}$, the unit vector that makes an angle of 45° with the positive real axis in the complex plane). A complex number is then represented by a five-tuple (a, b, c, d, k). Although the considered set of complex numbers is only a small subset of $\mathbb{C}$ (it is countable, while the set $\mathbb{C}$ is uncountable), the subset is already sufficient to describe a set of quantum gates that can implement universal quantum computation (cf. Section 4 for more details). The algebraic representation also allows efficient encoding of some operations. For example, because $\omega^4 = -1$, the multiplication of (a, b, c, d, k) by ω can be carried out by a simple right circular shift of the first four entries and then taking the opposite number for the first entry, namely $(-d, a, b, c, k)$, which represents the complex number $\left(\frac{1}{\sqrt{2}}\right)^k (-d + a\omega + b\omega^2 + c\omega^3)$. In the rest of the paper, we use $\mathbf{0}$ and $\mathbf{1}$ to denote the tuples for zero and one, i.e., $(0, 0, 0, 0, 0)$ and $(1, 0, 0, 0, 0)$, respectively. Using such an encoding, we represent quantum states by functions of the form $T\colon \{0,1\}^n \to \mathbb{Z}^5$.

Qubit Measurement. After executing a quantum circuit, one can measure the final quantum state in the computational basis. The probability that the qubit x_j of a quantum state $\sum_{i\in\{0,1\}^n} a_i \cdot |i\rangle$ is measured as the basis state $|0\rangle$ can be computed from the amplitude: $Prob[x_j = |0\rangle] = \sum_{i\in\{0,1\}^{n-j}\times\{0\}\times\{0,1\}^{j-1}} |a_i|^2$. When x_j collapses to $|0\rangle$ after the measurement, amplitudes of states with $x_j = |1\rangle$ become 0 and amplitudes of states with $x_j = |0\rangle$ are normalized using $\frac{1}{\sqrt{Prob[x_j=|0\rangle]}}$.

[2]We abuse notation and sometimes identify a binary string with its (unsigned) integer value in the *most significant bit first* (MSBF) encoding, e.g., the string 0101 with the number 5.

[3]Observe that in order to satisfy the requirement for the amplitudes of quantum states, it must be a *unit* vector.

2.2 Tree Automata



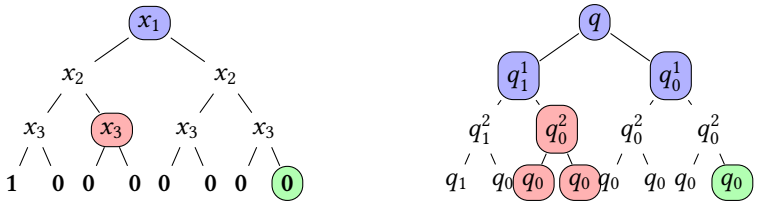
Binary Trees. We use a ranked alphabet Σ with binary symbols $f, g, \ldots$ and constant symbols $c_1, c_2, \ldots$. A *binary tree* is a *ground term* over Σ. For instance, $T = f(f(c_1, c_2), g(f(c_2, c_3), c_1))$, shown in the right, represents a binary tree. The set of *nodes* of a binary tree T, denoted as N_T, is defined inductively as a set of words over $\{0, 1\}$ such that for every constant symbol c, we define $N_c = \{\epsilon\}$, and for every binary symbol f, we define $N_{f(T_0,T_1)} = \{\epsilon\} \cup \{a.w \mid a \in \{0, 1\} \wedge w \in N_{T_a}\}$, where ϵ is the empty word and '.' is concatenation. Each binary tree T is associated with a labeling function $L_T : \{0, 1\}^* \to \Sigma$, which maps a node in T to its label in Σ. A tree is *single-valued* if it contains only one constant symbol.

Tree Automata. We focus on tree automata on binary trees and refer the interested reader to [Comon et al. 2008] for a general definition. A *(nondeterministic finite) tree automaton* (TA) is a tuple $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R} \rangle$ where Q is a finite set of *states*, Σ is a ranked alphabet, $\mathcal{R} \subseteq Q$ is the set of *root states*, and $\Delta = \Delta_i \cup \Delta_l$ is a set of tree transitions consisting of the set Δ_i of *internal transitions* of the form $q \xrightarrow{f} (q_0, q_1)$ (for a binary symbol f) and the set Δ_l of *leaf transitions* of the form $q \xrightarrow{c} ()$ (for a constant symbol c), for $q, q_0, q_1 \in Q$. W.l.o.g., to simplify our correctness proof, we assume every leaf transition of TAs has a unique parent state, namely, for any two leaf transitions $q \xrightarrow{c} (), q' \xrightarrow{c'} () \in \Delta$, it holds that $c \neq c' \implies q \neq q'$. We can conveniently describe TAs by providing only the set of root states $\mathcal{R}$ and the set of transitions Δ. The alphabet and states are implicitly defined as those that appear in Δ. For example, $\Delta = \{q \xrightarrow{x_1} (q_1, q_0), q \xrightarrow{x_1} (q_0, q_1), q_0 \xrightarrow{\mathbf{0}} (), q_1 \xrightarrow{\mathbf{1}} ()\}$ implies that $\Sigma = \{x_1, \mathbf{0}, \mathbf{1}\}$ and $Q = \{q, q_0, q_1\}$.

Run and Language. A *run* of $\mathcal{A}$ on a tree T is another tree ρ labeled with Q such that (i) T and ρ have the same set of nodes, i.e., $N_T = N_\rho$, (ii) for all leaf nodes $u \in N_T$, we have $L_\rho(u) \xrightarrow{L_T(u)} () \in \Delta$, and (iii) for all non-leaf nodes $v \in N_T$, we have $L_\rho(u) \xrightarrow{L_T(u)} (L_\rho(0.u), L_\rho(1.u)) \in \Delta$. The run ρ is *accepting* if $L_\rho(\epsilon) \in \mathcal{R}$. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of trees accepted by $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{A}) = \{T \mid \text{there exists an accepting run of } \mathcal{A} \text{ over } T\}$. A TA is (top-down) *deterministic* if it has at most one root state and for any of its transitions $q \xrightarrow{x} (q_l, q_r)$ and $q \xrightarrow{x} (q'_l, q'_r)$ it holds that $q_l = q'_l$ and $q_r = q'_r$. Any tree from the language of a deterministic TA has a unique run in the TA.

Example 2.1 (Accepted tree and its run). Assume a TA $\mathcal{A}_3$ with q as its single root state and the following transitions:

$q \xrightarrow{x_1} (q_0^1, q_1^1)$ $\quad$ $q \xrightarrow{x_1} (q_1^1, q_0^1)$ $\quad$ $q_1^1 \xrightarrow{x_2} (q_0^2, q_1^2)$ $\quad$ $q_1^1 \xrightarrow{x_2} (q_1^2, q_0^2)$ $\quad$ $q_0^1 \xrightarrow{x_2} (q_0^2, q_0^2)$ $\quad$ $q_1^2 \xrightarrow{x_3} (q_0, q_1)$ $\quad$ $q_1^2 \xrightarrow{x_3} (q_1, q_0)$ $\quad$ $q_0^2 \xrightarrow{x_3} (q_0, q_0)$ $\quad$ $q_0 \xrightarrow{\mathbf{0}} ()$ $\quad$ $q_1 \xrightarrow{\mathbf{1}} ()$



Among others, $\mathcal{A}_3$ accepts the above tree (in the left) with the run (in the right). Observe that all tree nodes satisfy the requirement of a valid run. E.g., the node 111 corresponds to the transition $q_0 \xrightarrow{\mathbf{0}} ()$, 01 to $q_0^2 \xrightarrow{x_3} (q_0, q_0)$, and ϵ to $q \xrightarrow{x_1} (q_1^1, q_0^1)$, etc.

In $\mathcal{A}_3$, we use states named q_0^n to denote only subtrees with all zeros (**0**) in leaves that can be generated from here, and states named q_1^n to denote only subtrees with a single **1** in the leaves that can be generated from it. Intuitively, the TA accepts all trees of the height three with exactly one **1** leaf and all other leaves **0** (in our encoding of quantum states, this might correspond to saying that $\mathcal{A}_3$ encodes an arbitrary computational basis state of a three-qubit system). □

3 ENCODING SETS OF QUANTUM STATES WITH TREE AUTOMATA

Observe that we can use (full) binary trees to encode functions $\{0,1\}^n \to \mathbb{Z}^5$, i.e., the function representation of quantum states. For instance, the tree

$$x_1(x_2(x_3(\mathbf{1},\mathbf{0}),x_3(\mathbf{0},\mathbf{0})),x_2(x_3(\mathbf{0},\mathbf{0}),x_3(\mathbf{0},\mathbf{0}))) \tag{4}$$

encodes the function T where $T(000) = \mathbf{1}$ and $T(i) = \mathbf{0}$ for all $i \in \{0,1\}^3 \setminus \{000\}$. Since TAs can concisely represent sets of binary trees, they can be used to encode sets of quantum states.

Example 3.1 (Concise representation of sets of quantum states by TAs). Here we consider the set of n-qubit quantum states $Q_n = \{|i\rangle \mid i \in \{0,1\}^n\}$, i.e., the set of all basis states. Note that $|Q_n| = 2^n$, which is exponential. Representing all possible basis states naively would require storing 2^{2^n} complex numbers. TAs can, however, represent such a set much more efficiently.

For the case when $n = 3$, the set Q_3 can be represented by the TA $\mathcal{A}_3$ from Example 2.1 with $3n+1$ transitions (i.e., linear-sized). The TA $\mathcal{A}_3$ can be generalized to encode the set of all n-qubit states $Q_n = \{|i\rangle \mid i \in \{0,1\}^n\}$ for each $n \in \mathbb{N}$ by setting the transitions to

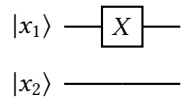
$$\begin{array}{llll}
q \xrightarrow{x_1} (q_0^1, q_1^1) & q_1^1 \xrightarrow{x_2} (q_0^2, q_1^2) & \dots & q_1^{n-1} \xrightarrow{x_n} (q_0, q_1) \quad q_0 \xrightarrow{\mathbf{0}} () \\
q \xrightarrow{x_1} (q_1^1, q_0^1) & q_1^1 \xrightarrow{x_2} (q_1^2, q_0^2) & \dots & q_1^{n-1} \xrightarrow{x_n} (q_1, q_0) \quad q_1 \xrightarrow{\mathbf{1}} () \\
 & q_0^1 \xrightarrow{x_2} (q_0^2, q_0^2) & \dots & q_0^{n-1} \xrightarrow{x_n} (q_0, q_0)
\end{array}$$

We denote the resulting TA by $\mathcal{A}_n$. Notice that although Q_n has 2^n quantum states, $\mathcal{A}_n$ has only $2n+1$ states and $3n+1$ transitions. □

Formally a TA $\mathcal{A}$ recognizing a set of quantum states is a tuple $\langle Q, \Sigma, \Delta, \mathcal{R}\rangle$, whose alphabet Σ can be partitioned into two classes of symbols: binary symbols $x_1, \dots, x_n$ and a finite set of leaf symbols $\Sigma_c \subseteq \mathbb{Z}^5$ representing all possible amplitudes of quantum states in terms of computational bases. By slightly abusing the notation, for a full binary tree $T \in \mathcal{L}(\mathcal{A})$, we also use T to denote the function $\{0,1\}^n \to \mathbb{Z}^5$ that maps a computational basis to the corresponding amplitude of T's quantum state. The two meanings of T are used interchangeably throughout the paper.

Remark. Note that TAs allow representation of *infinite* languages, yet we only use them for *finite* sets, which might seem like the model is overly expressive. We, however, stick to TAs for the following two reasons: (i) there is an existing rich toolbox for TA manipulation and minimization, e.g., [Abdulla et al. 2008, 2007; Comon et al. 2008; Lengál et al. 2012], and (ii) we want to have a robust formal model for extending our framework to parameterized verification, i.e., proving that an n-qubit algorithm is correct for any n, which will require us to deal with infinite languages (c.f., the framework of *regular tree model checking* [Abdulla et al. 2002; Bouajjani et al. 2012]).

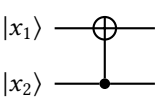
Moreover, we chose *full* binary trees as the representation of quantum states. We thought about using a more compact structure, e.g., allowing jump over a transition with common left and right children (similar to ROBDD's elimination of a node with isomorphic subtrees [Bryant 1986]). We decided against that because TAs already allow an efficient representation of common children via a transition to the same left and right states, e.g., $q \xrightarrow{x} (q', q')$. The benefit of using a more compact tree representation is thus limited. Using a more efficient data structure would also make



(a) X gate applied to qubit x_1

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

(b) Matrix of the X gate



(c) $CNOT_1^2$ gate with target qubit x_1 and control qubit x_2

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(d) Matrix of the $CNOT$ gate

Fig. 2. Applications of X and $CNOT$ gates and their matrices

the algorithms in the following sections harder to understand. We therefore leave the investigation of designing a more efficient data structure to our future work.

4 SYMBOLIC REPRESENTATION OF QUANTUM GATES

With TAs used to concisely represent sets of quantum states, the next task is to capture the effects of applying quantum gates on this representation. When quantum states are represented as vectors, gates are represented as matrices and gate operations are matrix multiplications. When states are represented as binary trees, we need a new representation for quantum gates and their operations. Inspired by the work of [Tsai et al. 2021], we introduce *symbolic update formulae*, which are formulae that describe how a gate transforms a tree representing a quantum state. Later, we will lift the tree update operation to a set of trees encoded in a TA.

We use the algebraic representation of quantum states from Eq. (3) also for their symbolic handling. For instance, consider a system with qubits x_1, x_2 and its state

$$T = c_{00} \cdot |00\rangle + c_{01} \cdot |01\rangle + c_{10} \cdot |10\rangle + c_{11} \cdot |11\rangle \tag{5}$$

for $c_{00}, c_{01}, c_{10}, c_{11} \in \mathbb{Z}^5$, four complex numbers represented in the algebraic way. The result of applying the X gate (the quantum version of the *NOT* gate) on qubit x_1 (cf. Fig. 2a) is $(c_{10}, c_{11}, c_{00}, c_{01})^T$ (cf. Eq. (2)). Intuitively, we observe that the effect of the gate is a permutation of the computational basis states that swaps the amplitudes of states where the x_1's value is 1 with states where the x_1's value is 0 (and the values of qubits other than x_1 stay the same). Concretely, it swaps the amplitudes of the pairs $(|\mathbf{0}0\rangle, |\mathbf{1}0\rangle)$ and $(|\mathbf{0}1\rangle, |\mathbf{1}1\rangle)$ to obtain the quantum state

$$X(T) = c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle + c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle . \tag{6}$$

Instead of executing the quantum gate by performing a matrix-vector multiplication, we will capture its semantics *symbolically* by directly manipulating the tree function $T\colon \{0,1\}^n \to \mathbb{Z}^5$. For this, we will use the following operators on T, parameterized by a qubit x_t (t for "target"):

$$T_{x_t}(b_n \dots b_t \dots b_1) = T(b_n \dots 1 \dots b_1) \qquad B_{x_t}(b_n \dots b_t \dots b_1) = b_t$$
$$T_{\overline{x_t}}(b_n \dots b_t \dots b_1) = T(b_n \dots 0 \dots b_1) \qquad B_{\overline{x_t}}(b_n \dots b_t \dots b_1) = \overline{b_t}.$$

(Projection) (Restriction)

In the previous, $\overline{b_t}$ denotes the complement of the bit b_t (i.e., $\overline{0} = 1$ and $\overline{1} = 0$). Intuitively, T_{x_t} and $T_{\overline{x_t}}$ fix the value of qubit x_t to be 1 and 0 respectively. On the other hand, B_{x_t} and $B_{\overline{x_t}}$ just take the value of qubit x_t (or its negation) in the computational basis state.

Equipped with the operators, we can now proceed to express the semantics of X symbolically. Let us first look at the first two summands on the right-hand side of Eq. (6): $T^0 = c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle$. These summands can be obtained by manipulating the input function T in the following way:

$$T^0 = B_{\overline{x_1}} \cdot T_{x_1}. \tag{7}$$

Table 1. Symbolic update formulae for the considered quantum gates; x_c and x'_c denote control bits (if they exist), and x_t denotes the target bit.

Gate	Update
X_t	$B_{x_t} \cdot T_{\overline{x_t}} + B_{\overline{x_t}} \cdot T_{x_t}$
Y_t	$\omega^2 \cdot (B_{x_t} \cdot T_{\overline{x_t}} - B_{\overline{x_t}} \cdot T_{x_t})$
Z_t	$B_{\overline{x_t}} \cdot T - B_{x_t} \cdot T$
H_t	$(T_{\overline{x_t}} + B_{\overline{x_t}} \cdot T_{x_t} - B_{x_t} \cdot T)/\sqrt{2}$
S_t	$B_{\overline{x_t}} \cdot T + \omega^2 \cdot B_{x_t} \cdot T$
T_t	$B_{\overline{x_t}} \cdot T + \omega \cdot B_{x_t} \cdot T$
$Rx(\frac{\pi}{2})_t$	$(T - \omega^2 \cdot (B_{x_t} \cdot T_{\overline{x_t}} + B_{\overline{x_t}} \cdot T_{x_t}))/\sqrt{2}$
$Ry(\frac{\pi}{2})_t$	$(T_{\overline{x_t}} + B_{x_t} \cdot T - B_{\overline{x_t}} \cdot T_{x_t})/\sqrt{2}$
$CNOT_t^c$	$B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}})$
CZ_t^c	$B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_t}} \cdot T - B_{x_t} \cdot T)$
$Toffoli_t^{c,c'}$	$B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_{c'}}} \cdot T + B_{x_{c'}} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}}))$

Here, $T^0 = B_{\overline{x_1}} \cdot T_{x_1}$ is a shorthand for $T^0(b_1 \dots b_n) = B_{\overline{x_1}}(b_1 \dots b_n) \cdot T_{x_1}(b_1 \dots b_n)$. When we view T as a tree, the operation T_{x_1} essentially copies the right subtree of every x_1-node to its left subtree, and $B_{\overline{x_1}} \cdot T_{x_1}$ makes all leaves in every right subtree of T_{x_1}'s x_1-node zero. This would give us

$$T^0 \quad = \quad c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle + 0 \cdot |10\rangle + 0 \cdot |11\rangle \quad = \quad c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle \,. \tag{8}$$

On the other hand, the last two summands in the right-hand side of Eq. (6), i.e., $T^1 = c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle$, could be obtained by manipulating T as follows:

$$T^1 = B_{x_1} \cdot T_{\overline{x_1}}. \tag{9}$$

The tree view of $B_{x_1} \cdot T_{\overline{x_1}}$ is symmetric to $B_{\overline{x_1}} \cdot T_{x_1}$, which would give us the following state:

$$T^1 \quad = \quad 0 \cdot |00\rangle + 0 \cdot |01\rangle + c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle \quad = \quad c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle \,. \tag{10}$$

Finally, by summing T^0 and T^1, we obtain Eq. (6): $T^0 + T^1 = c_{10} \cdot |00\rangle + c_{11} \cdot |01\rangle + c_{00} \cdot |10\rangle + c_{01} \cdot |11\rangle$. That is, the semantics of the X gate could be expressed using the following symbolic formula:

$$X_1(T) = B_{\overline{x_1}} \cdot T_{x_1} + B_{x_1} \cdot T_{\overline{x_1}}. \tag{11}$$

Observe that the sum effectively swaps the left and right subtrees of each x_1-node.

For multi-qubit gates, the update formulae get more complicated, since they involve more than one qubit. Consider, e.g., the "controlled-NOT" gate $CNOT_t^c$ (see Fig. 2c for the graphical representation and Fig. 2d for its semantics). The $CNOT_t^c$ gate uses x_t and x_c as the target and control qubit respectively. Intuitively, it "flips" the target qubit's value when the control qubit's value is 1 and keeps the original value if it is 0. Similarly, as for the X gate, we can deduce a symbolic formula for the update done by a $CNOT$ gate:

$$CNOT_t^c(T) = B_{\overline{x_c}} \cdot T + B_{x_c} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}}). \tag{12}$$

The sum consists of the following two summands:

- The summand $B_{\overline{x_c}} \cdot T$ says that when the control qubit is 0, x_t and x_c stay the same.
- The summand $B_{x_c} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}})$ handles the case when x_c is 1. In such a case, we apply the X gate on x_t (observe that the inner term is the update formula of X_t in Eq. (11)).

One can obtain symbolic update formulae for other quantum gates in a similar way. In Table 1 we give the formulae for the gates supported by our framework (see [Chen et al. 2023b] for their usual definition using matrices).

For a gate G, we use the superscripts c and c' to denote that x_c and x'_c are the gate's control qubits (if they exist) and the subscript t to denote that x_t is the target bit (e.g., $G_t^{c,c'}$). We note that the supported set of gates is much larger than is required to achieve (approximate) universal

quantum computation (for which it suffices to have, e.g., (i) Clifford gates (H, S, and $CNOT$) and T (see [Boykin et al. 2000]) or (ii) Toffoli and H (see [Aharonov 2003])).

Theorem 4.1. *The symbolic update formulae in Table 1 are correct (w.r.t. the standard semantics of quantum gates, cf. [Nielsen and Chuang 2011]).*

A note on expressivity. The expressivity of our framework is affected by the following factors:

(1) *Algebraic complex number representation* (a, b, c, d, k): This representation can arbitrarily closely approximate any complex number: First, note that $\omega = \cos 45^\circ + i \sin 45^\circ = \frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}}$ and when $b = d = 0$, we have $(a, 0, c, 0, k) = \frac{1}{\sqrt{2}^k}(a + c\omega^2) = \frac{a}{\sqrt{2}^k} + \frac{ci}{\sqrt{2}^k}$. Then any complex number can be approximated arbitrarily closely by picking suitable a, c, and k.
(2) *Supported quantum gates*: We covered all standard quantum gates supported in modern quantum computers except parameterized rotation gate. From Solovay-Kitaev theorem [Dawson and Nielsen 2006], gates performing rotations by $\frac{\pi}{2^k}$ can be approximated with an error rate ϵ with $O(\log^{3.97}(\frac{1}{\epsilon}))$-many gates that we support.
(3) *Tree automata structure*: We use non-deterministic transitions of tree automata to represent a set of trees compactly. Nevertheless, we can currently encode only a finite set of states, so encoding, e.g., all quantum states that satisfy $|\,|10\rangle\,| = |\,|01\rangle\,|$ is future work.

In the next two sections, we discuss how to lift the tree update operation to a set of trees encoded in a TA. Our framework allows different instantiations. We will introduce two in this paper, namely the (i) *permutation-based* (Section 5) and (ii) *composition-based* (Section 6) approach. The former is simple, efficient, and works for all but the H_t, $\mathrm{Rx}(\frac{\pi}{2})_t$, and $\mathrm{Ry}(\frac{\pi}{2})_t$ gates from Table 1 (those whose effect is a permutation of tree leaves, i.e., for gates whose matrix contains only one non-zero element in each row, potentially with a constant scaling of amplitude), while the latter supports all gates in the table but is less efficient. The two approaches are compatible with each other, so one can, e.g., choose to use the permutation-based approach by default and for unsupported gates fall back on the composition-based approach.

5 PERMUTATION-BASED ENCODING OF QUANTUM GATES

Let us first look at the simplest gate $\mathrm{X}_t(T) = B_{x_t} \cdot T_{\overline{x_t}} + B_{\overline{x_t}} \cdot T_{x_t}$. Recall that in Section 4, we showed that the formula essentially swaps the left and right subtrees of each x_t-labeled node. For a TA $\mathcal{A}$, we can capture the effect of applying X_t to all states in $\mathcal{L}(\mathcal{A})$ by swapping the left and the right children of all x_t-labeled transitions $q \to (q_0, q_1)$, i.e., update them to $q \to (q_1, q_0)$. We use $\mathrm{X}_t(\mathcal{A})$ to denote the TA constructed following this procedure.

Theorem 5.1. $\mathcal{L}(\mathrm{X}_t(\mathcal{A})) = \{\mathrm{X}_t(T) \mid T \in \mathcal{L}(\mathcal{A})\}$.

The update formulae of gates Z_t, S_t, and T_t are all in the form $a_1 \cdot B_{x_t} \cdot T + a_0 \cdot B_{\overline{x_t}} \cdot T$ for $a_1, a_0 \in \mathbb{C}$. Intuitively, the formulae scale the left and right subtrees of T with scalars a_0 and a_1, respectively. Their construction (Algorithm 1) can be done by (1) making one primed copy of $\mathcal{A}$ whose leaf labels are multiplied with a_1 (Line 3), (2) multiplying all leaf labels of $\mathcal{A}$ with a_0 (Line 4), and (3) updating all x_t-labeled transitions $q \to (q_0, q_1)$ to $q \to (q_0, q_1')$, i.e., for the right child, jump to the primed version (Line 4). In the algorithms, we define $Q' = \{q' \mid q \in Q\}$ for any set of state Q and $\Delta' = \{q' \to (q_l', q_r') \mid q \to (q_l, q_r) \in \Delta\}$ for any set of transitions Δ. The case of Y_t is similar, but we need both *constant scaling* (Lines 1-4) and *swapping* (Lines 7-9) (the left-hand side and right-hand side scalars being ω^2 and $-\omega^2$, respectively).

Theorem 5.2. $\mathcal{L}(\mathrm{U}(\mathcal{A})) = \{\mathrm{U}(T) \mid T \in \mathcal{L}(\mathcal{A})\}$, *for* $\mathrm{U} \in \{\mathrm{Y}_t, \mathrm{Z}_t, \mathrm{S}_t, \mathrm{T}_t\}$.

Algorithm 1: Algorithm for constructing $\mathrm{U}(\mathcal{A})$, for $\mathrm{U} \in \{X_t, Y_t, Z_t, S_t, T_t\}$

Input: A TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R}\rangle$ and a gate U
Output: The TA $\mathrm{U}(\mathcal{A})$

1 **if** $\mathrm{U} \in \{Y_t, Z_t, S_t, T_t\}$ **then** // need constant scaling
2 Let a_1 and a_0 be the left and right scalar in $\mathrm{U}(T) = a_1 \cdot B_{x_t} \cdot T_1 + a_0 \cdot B_{\overline{x_t}} \cdot T_0$;
3 $\mathcal{A}_1 := \langle Q', \Sigma, \Delta_1, \mathcal{R}'\rangle$, where $\Delta_1 = \Delta_i' \cup \{q' \xrightarrow{a_1 \cdot c} () \mid q \xrightarrow{c} () \in \Delta_l\}$;
4 $\mathcal{A}^R := \langle Q \cup Q', \Sigma, \Delta^R \cup \Delta_1, \mathcal{R}\rangle$, where

$$\Delta^R = \{q \xrightarrow{a_0 \cdot c} () \mid q \xrightarrow{c} () \in \Delta_l\} \cup \{q \xrightarrow{x_k} (q_0, q_1) \mid q \xrightarrow{x_k} (q_0, q_1) \in \Delta_i \wedge k \neq t\} \cup \{q \xrightarrow{x_k} (q_0, q_1') \mid q \xrightarrow{x_k} (q_0, q_1) \in \Delta_i \wedge k = t\}$$

5 **else**
6 $\mathcal{A}^R := \mathcal{A}$; // when $\mathrm{U} = \mathrm{X}_t$
7 **if** $\mathrm{U} \in \{X_t, Y_t\}$ **then** // need swapping
8 Assume $\mathcal{A}^R = \langle Q^R, \Sigma, \Delta^R, \mathcal{R}\rangle$;
9 $\mathcal{A}^R := \langle Q^R, \Sigma, \Delta_1^R, \mathcal{R}\rangle$, where

$$\Delta_1^R = \{q \xrightarrow{x_k} (q_0, q_1) \mid q \xrightarrow{x_k} (q_0, q_1) \in \Delta_i^R \wedge k \neq t\} \cup \{q \xrightarrow{x_k} (q_1, q_0) \mid q \xrightarrow{x_k} (q_0, q_1) \in \Delta_i^R \wedge k = t\} \cup \{t \mid t \in \Delta_l^R\}$$

10 **return** $\mathcal{A}^R$;

The cases of multi-qubit gates CNOT_t^c, CZ_t^c, and $\mathrm{Toffoli}_t^{c,c'}$ can be handled when t is the lowest of the three qubits, i.e., $c < t \wedge c' < t$. We can assume w.l.o.g. that $c < c'$. Output of these gates can be constructed recursively following Algorithm 2. Let us look at the corresponding update formulae:

$$\mathrm{CNOT}_t^c(T) = B_{\overline{x_c}} \cdot T + B_{x_c} \cdot \boxed{(B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}})}$$

$$\mathrm{CZ}_t^c(T) = B_{\overline{x_c}} \cdot T + B_{x_c} \cdot \boxed{(B_{\overline{x_t}} \cdot T - B_{x_t} \cdot T)}$$

$$\mathrm{Toffoli}_t^{c,c'}(T) = B_{\overline{x_c}} \cdot T + B_{x_c} \cdot \boxed{(B_{\overline{x_{c'}}} \cdot T + B_{x_{c'}} \cdot (B_{\overline{x_t}} \cdot T_{x_t} + B_{x_t} \cdot T_{\overline{x_t}}))}$$

We first construct the TA of the inner term, the shaded area, which are TAs for X^t, Z^t, or $\mathrm{CNOT}_t^{c'}$. We call it the primed version here (cf. $\mathcal{A}_1'$ at Line 4). We then update all x_c-labeled transitions $q \xrightarrow{x_c} (q_0, q_1)$ to $q \xrightarrow{x_c} (q_0, q_1')$, i.e., jump to the primed version in the right subtree.

Theorem 5.3. $\mathcal{L}(\mathrm{U}(\mathcal{A})) = \{\mathrm{U}(T) \mid T \in \mathcal{L}(\mathcal{A})\}$, *for* $\mathrm{U} \in \{\mathrm{CNOT}_t^c, \mathrm{CZ}_t^c, \mathrm{Toffoli}_t^{c,c'}\}$.

6 COMPOSITION-BASED ENCODING OF QUANTUM GATES

We introduce the composition-based approach in this section. The task is to develop TA operations that handle the update formulae in Table 1 compositionally. The idea is to lift the basic tree operations, such as projection T_{x_k}, restriction $B \cdot T$, and binary operation $\pm$ to operations over TAs and then compose them to have the desired gate semantics. The update formulae in Table 1 are always in the form of $\mathrm{term}_1 \pm \mathrm{term}_2$. For example, for the X_t gate, $\mathrm{term}_1 = B_{x_t} \cdot T_{\overline{x_t}}$ and $\mathrm{term}_2 = B_{\overline{x_t}} \cdot T_{x_t}$. Our idea is to first construct TAs $\mathcal{A}_{\mathrm{term}_1}$ and $\mathcal{A}_{\mathrm{term}_2}$, recognizing quantum states of term_1 and term_2, and then combine them using binary operation $\pm$ to produce a TA recognizing

Algorithm 2: Algorithm for constructing $\mathrm{U}(\mathcal{A})$, for $\mathrm{U} \in \{\mathrm{CNOT}_t^c, \mathrm{CZ}_t^c, \mathrm{Toffoli}_t^{c,c'}\}$

Input: A TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R}\rangle$ and a gate U
Output: The TA $\mathrm{U}(\mathcal{A})$

1 **if** $\mathrm{U} = \mathrm{CNOT}_t^c$ **then** $\mathcal{A}_1 := \mathrm{X}_t(\mathcal{A})$;
2 **if** $\mathrm{U} = \mathrm{CZ}_t^c$ **then** $\mathcal{A}_1 := \mathrm{Z}_t(\mathcal{A})$;
3 **if** $\mathrm{U} = \mathrm{Toffoli}_t^{c,c'}$ **then** $\mathcal{A}_1 := \mathrm{CNOT}_t^{c'}(\mathcal{A})$;
4 Let $\mathcal{A}_1' = \langle Q_1', \Sigma, \Delta_1', \mathcal{R}'\rangle$ be obtained from $\mathcal{A}_1$ by priming all occurrences of states;
5 $\mathcal{A}^R := \langle Q \cup Q_1', \Sigma, \Delta^R \cup \Delta_1', \mathcal{R}\rangle$, where

$$\Delta^R = \{q \xrightarrow{x_k} (q_0, q_1) \mid q \xrightarrow{x_k} (q_0, q_1) \in \Delta_i \wedge k \neq c\} \cup$$
$$\{q \xrightarrow{x_k} (q_0, q_1') \mid q \xrightarrow{x_k} (q_0, q_1) \in \Delta_i \wedge k = c\} \cup \{t \mid t \in \Delta_l\}$$

return $\mathcal{A}^R$;

the quantum states of $\mathsf{term}_1 \pm \mathsf{term}_2$. The TAs $\mathcal{A}_{\mathsf{term}_1}$, $\mathcal{A}_{\mathsf{term}_2}$ would be constructed using TA versions of basic operations introduced later in this section.

For a TA accepting the trees $\{T_1, T_2\}$, a correct construction would produce a TA with the language $\{T_1' \pm T_1'', T_2' \pm T_2''\}$, for $T_i' = \mathsf{term}_1[T \mapsto T_i]$ and $T_i'' = \mathsf{term}_2[T \mapsto T_i]$, where $[T \mapsto T_i]$ is a substitution defined in the standard way. Obtaining this result is, however, not straightforward. If we just performed the $\pm$ operation pairwise between all elements of T_i' and T_i'', we would obtain the language $\{T_1' \pm T_1'', T_2' \pm T_2'', T_1' \pm T_2'', T_2' \pm T_1''\}$, which is wrong, since we are losing the information that T_1' and T_1'' are related (and so are T_2' and T_2'').

In the rest of the section, we will describe implementation of the necessary operations for the composition-based approach.

6.1 Tree Tag

We introduce the concept of *tree tags* to keep track of the origins of trees. For any tree T, its tag $\mathrm{Tag}(T)$ is the tree obtained from T by replacing all leaf symbols with a special symbol $\square$. E.g., for the tree $T_1 = x_1(x_2(\mathbf{1}, \mathbf{0}), x_2(\mathbf{0}, \mathbf{0}))$, its tag is $\mathrm{Tag}(T_1) = x_1(x_2(\square, \square), x_2(\square, \square))$. Our construction needs to maintain the following invariants: (1) each tree in a TA has a unique tag, (2) all derived trees should have the same tag, and (3) binary operations over two sets of trees represented by TAs only combine trees with the same tag. When we say T' is *derived from* T, it means T' is obtained by applying basic tree operations on T. E.g., the tree $B_{\overline{x_1}} \cdot T_{x_1}$ is derived from T.

Example 6.1. Let $\mathcal{A}$ be a TA with root states $\mathcal{R} = \{q\}$ and transitions

$$q \xrightarrow{x_1} (q_l, q_r) \qquad q_l \xrightarrow{x_2} (q_1, q_0) \qquad q_0 \xrightarrow{\mathbf{0}} () \qquad q_l \xrightarrow{x_2} (q_0, q_1) \qquad q_1 \xrightarrow{\mathbf{1}} () \qquad q_r \xrightarrow{x_2} (q_0, q_0)$$

Observe that $\mathcal{L}(\mathcal{A}) = \{x_1(x_2(\mathbf{1}, \mathbf{0}), x_2(\mathbf{0}, \mathbf{0})), x_1(x_2(\mathbf{0}, \mathbf{1}), x_2(\mathbf{0}, \mathbf{0}))\}$. In Dirac notation, this is the set $\{|00\rangle, |01\rangle\}$. The tag of both trees is $x_1(x_2(\square, \square), x_2(\square, \square))$, which violates invariant (1) above. □

In general, invariant (1) does not hold, as we can see from Example 6.1. Our solution to this is introducing the *tagging* procedure (cf. Algorithm 3). The idea of tagging is simple: for each transition, we assign to its function symbol a unique number. After tagging a TA, every transition has a different symbol. Let $\mathrm{Untag}(T)$ be a function that removes the number j (added by the tagging procedure) from each symbol x_k^j in T's labels.

Algorithm 3: The tagging procedure $\mathrm{Tag}(\mathcal{A})$.

Input: A TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R} \rangle$
Output: A tagged TA $\langle Q, \Sigma', \Delta', \mathcal{R} \rangle$
1 $\Delta_1 := \{q \xrightarrow{c} () \mid q \xrightarrow{c} () \in \Delta\}$;
2 $\Delta_2 := \{q \xrightarrow{x_k^j} (q_1, q_2) \mid \delta = (q \xrightarrow{x_k} (q_1, q_2)) \in \Delta, \mathit{ord}(\delta) = j\}$, where $\mathit{ord}\colon \Delta \to \mathbb{N}$ is an arbitrary injection (e.g., an ordering of the transitions);
3 $\Delta' := \Delta_1 \cup \Delta_2$;
4 Σ' is the set of all symbols appearing in Δ';
5 **return** $\langle Q, \Sigma', \Delta', \mathcal{R} \rangle$;

Example 6.2. After tagging $\mathcal{A}$ from Example 6.1, we obtain the TA $\mathcal{A}_{\mathrm{Tag}}$ with the root state q and the following transitions:

$$q \xrightarrow{x_1^1} (q_l, q_r) \qquad q_l \xrightarrow{x_2^2} (q_1, q_0) \qquad q_0 \xrightarrow{\mathbf{0}} ()$$
$$q_l \xrightarrow{x_2^3} (q_0, q_1) \qquad q_1 \xrightarrow{\mathbf{1}} ()$$
$$q_r \xrightarrow{x_2^4} (q_0, q_0)$$

Here $\mathcal{L}(\mathcal{A}_{\mathrm{Tag}}) = \{T_1, T_2\}$, where $T_1 = x_1^1(x_2^2(\mathbf{1}, \mathbf{0}), x_2^4(\mathbf{0}, \mathbf{0}))$ and $T_2 = x_1^1(x_2^3(\mathbf{0}, \mathbf{1}), x_2^4(\mathbf{0}, \mathbf{0}))$. The two trees T_1 and T_2 have different tags now. □

Lemma 6.3. *All non-single-valued trees in a tagged TA have different tags.*

Definition 6.4 (Tag preservation). Given a tagged TA $\mathcal{A}_{\mathrm{Tag}}$ and an operation U over binary trees, a TA construction procedure O transforming $\mathcal{A}_{\mathrm{Tag}}$ to $O(\mathcal{A}_{\mathrm{Tag}})$ is called *tag-preserving* if there is a bijection $S\colon \mathcal{L}(\mathcal{A}_{\mathrm{Tag}}) \to \mathcal{L}(O(\mathcal{A}_{\mathrm{Tag}}))$ such that $\mathrm{Tag}(T) = \mathrm{Tag}(S(T))$ for all $T \in \mathcal{L}(\mathcal{A}_{\mathrm{Tag}})$. In such a case, we write $\mathcal{A}_{\mathrm{Tag}} \simeq_{\mathrm{Tag}} O(\mathcal{A}_{\mathrm{Tag}})$. Further, if the above correspondence satisfies $U(\mathrm{Untag}(T)) = \mathrm{Untag}(S(T))$ for each T, we say that the TA construction procedure O is *tag-preserving over* U.

6.2 The Complete Picture of the Quantum Gate Application Procedure

Tagging a TA is the first step in applying a quantum gate. In the second step, for each term in the update formulae (cf. Table 1), we make a copy of the tagged TA and apply the operations that we are going to introduce (projection, restriction, and multiplication) to construct the corresponding TA. Notice that the operations are *tag-preserving*, i.e., they will keep the tag of all accepted trees. Then we use the binary operation to merge trees with the same tag and complete the update formula compositionally. In the end, we remove the TA's tag to finish the quantum gate application[4].

[4]This is a design choice. Another possibility is to keep the tag until finishing all gate operations. Untagging after finishing a gate has the advantage that it allows a more aggressive state space reduction.

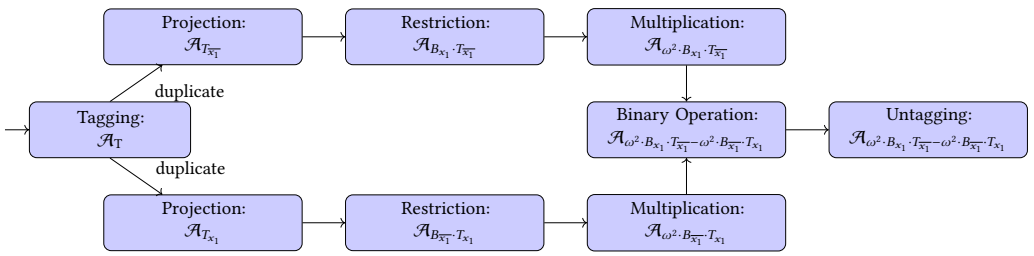


Fig. 3. Constructions performed when applying the gate Y_1 to $\mathcal{A}_{\mathrm{Tag}}$

Algorithm 4: Restriction operation on x_t, $\mathrm{Res}(\mathcal{A}, x_t, b)$

Input: A tagged TA $\mathcal{A} = \{Q, \Sigma, \Delta, \mathcal{R}\}$,
Output: A tagged TA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \{b \ ? \ B_{x_t} \cdot T \ : \ B_{\overline{x_t}} \cdot T \mid T \in \mathcal{L}(\mathcal{A})\}$

1 $\Delta'_i := \{q'_0 \xrightarrow{x_j^i} (q'_1, q'_2) \mid q_0 \xrightarrow{x_j^i} (q_1, q_2) \in \Delta\}$;
2 $\Delta'_l := \{q'_0 \xrightarrow{\mathbf{0}} () \mid q_0 \xrightarrow{(a,b,c,d,k)} () \in \Delta\}$;
3 $\Delta' := \Delta'_i \cup \Delta'_l$;
4 $\Delta_{\mathsf{add}} := \Delta_{\mathsf{rm}} := \emptyset$;
5 **foreach** $q \xrightarrow{x_t^i} (q_l, q_r) \in \Delta$ **do**
6 **if** b **then** $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{x_t^i} (q'_l, q_r)\}$ **else** $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{x_t^i} (q_l, q'_r)\}$;
7 $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{x_t^i} (q_l, q_r)\}$;
8 **return** $\{Q \cup Q', \Sigma \cup \{\mathbf{0}\}, ((\Delta \cup \Delta') \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R}\}$;

Example 6.5. From Table 1, we have

$$Y_1(T) = \omega^2 \cdot B_{x_1} \cdot T_{\overline{x_1}} - \omega^2 \cdot B_{\overline{x_1}} \cdot T_{x_1}.$$

For applying the gate Y_1 to a tagged TA $\mathcal{A}_{\mathrm{T}}$, we perform the constructions shown in Fig. 3. □

6.2.1 Restriction Operation: Constructing $\mathcal{A}_{B_{x_t} \cdot T}$ and $\mathcal{A}_{B_{\overline{x_t}} \cdot T}$ from $\mathcal{A}_T$. Observe that the tree $B_{x_t} \cdot T$ can be obtained by changing all leaf labels of the $\overline{x_t}$-subtrees in T to $(0, 0, 0, 0, 0)$. In Algorithm 4 we show the procedure for constructing the restriction operation based on this observation. Here $b \ ? \ s_1 \ : \ s_2$ is a shorthand for "if b is true then s_1 else s_2." Intuitively, when encountering a transition with variants of x_t as its label, in case $b =$ **true**, we reconnect its zero (left) child to the primed version (Line 6 of Algorithm 4), so the leaves of this subtree would be all zero. The case when $b =$ **false** is symmetric. Note that the structure of the original and the primed versions are identical, so this modification will not change the tags of accepted trees.

Theorem 6.6. *Let $\mathcal{A}$ be a tagged TA. Then it holds that* $\mathrm{Res}(\mathcal{A}, x_t, b) \simeq_{\mathrm{Tag}} \mathcal{A}$ *and, moreover,* $\mathcal{L}(\mathrm{Res}(\mathcal{A}, x_t, b)) = \{b \ ? \ B_{x_t} \cdot T \ : \ B_{\overline{x_t}} \cdot T \mid T \in \mathcal{L}(\mathcal{A})\}$.

6.2.2 Multiplication Operation: Constructing $\mathcal{A}_{v \cdot T}$ from $\mathcal{A}_T$. Algorithm 5 gives the multiplication operation that works on both tagged and non-tagged version.

Algorithm 5: Multiplication operation, $\mathsf{Mult}(\mathcal{A}, v)$

Input: A tagged TA $\mathcal{A} = \{Q, \Sigma, \Delta, \mathcal{R}\}$ and a constant value v (either ω or $\frac{1}{\sqrt{2}}$)
Output: A tagged TA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \{v \cdot T \mid T \in \mathcal{L}(\mathcal{A})\}$

1 $\Delta_{\mathsf{add}} := \Delta_{\mathsf{rm}} := \emptyset$;
2 **foreach** $q \xrightarrow{(a,b,c,d,k)} () \in \Delta$ **do**
3 **if** $v = \omega$ **then**
4 $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{(-d,a,b,c,k)} ()\}$;
5 **else** // $v = \frac{1}{\sqrt{2}}$
6 $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{(a,b,c,d,k+1)} ()\}$;
7 $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{(a,b,c,d,k)} ()\}$;
8 **return** $\{Q, \Sigma, (\Delta \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R}\}$;

Algorithm 6: Subtree copying procedure on x_t, $\mathsf{s.copy}(\mathcal{A}, x_t, b)$.

Input: A tagged TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R}\rangle$, variable x_t to copy, and a Boolean value b to indicate which branch to copy

Output: The tagged TA $\langle Q, \Sigma, \Delta', \mathcal{R}\rangle$

$\Delta_{\mathsf{rm}} := \Delta_{\mathsf{add}} := \emptyset$;

foreach $q \xrightarrow{x_t^i} (q_l, q_r) \in \Delta$ **do**

  if b **then** $q_c := q_r$ **else** $q_c := q_l$;

  $\Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{x_t^i} (q_c, q_c)\}$;

  $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{x_t^i} (q_l, q_r)\}$;

return $\langle Q, \Sigma, (\Delta \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R}\rangle$;

Theorem 6.7. *Let $\mathcal{A}$ be a tagged TA. Then it holds that* $\mathsf{Mult}(\mathcal{A}, v) \simeq_{\mathrm{Tag}} \mathcal{A}$ *and, moreover,* $\mathcal{L}(\mathsf{Mult}(\mathcal{A}, v)) = \{v \cdot T \mid T \in \mathcal{L}(\mathcal{A})\}$.

6.2.3 Projection Operation: Constructing $\mathcal{A}_{T_{x_t}}$ and $\mathcal{A}_{T_{\overline{x_t}}}$ from $\mathcal{A}_T$. Recall that T_{x_t} is obtained from T by fixing the t-th input bit to be 1, i.e., $T_{x_t}(b_1 \dots b_t \dots b_n) = T(b_1 \dots 1 \dots b_n)$. Intuitively, the construction of $\mathcal{A}_{T_{x_t}}$ from $\mathcal{A}_T$ can be done by copying all right subtrees of x_t^i (i.e., corresponding to $x_t^i = 1$) to replace its left ($x_t^i = 0$) subtrees. A seemingly correct construction can be found in Algorithm 6. For short, we use $\mathsf{s.copy_t}(\mathcal{A})$ to denote $\mathsf{s.copy}(\mathcal{A}, x_t, \mathbf{true})$ and $\mathsf{s.copy_{\bar{t}}}(\mathcal{A})$ to denote $\mathsf{s.copy}(\mathcal{A}, x_t, \mathbf{false})$.

However, this construction has two issues (1) it would change the tag of accepting trees and (2) when there are more than one possible subtrees below q_r (or q_l), say, for example, T_1 and T_2, it might happen that the resulting TA accepts a tree such that one subtree below the symbol x_t^i is T_1 while another subtree is T_2, i.e., they are still not equal and hence not the result after copying.

Although the procedure is incorrect in general, it is correct when $t = n$, i.e., the layer just above the leaf. Notice that constant symbols are irrelevant to a tree's tag (all constant symbols will be replaced with $\square$ in a tag). So copying one subtree to the other will not affect the tag at the leaf transition. Moreover, recall that from TA's definition, all leaf transitions have unique starting states. So it will not encounter the issue (2) mentioned above.

Lemma 6.8. *Subtree copying* $\mathsf{s.copy_t}$ *is tag-preserving over the tree projection operation $T \to T_{x_t}$ and* $\mathsf{s.copy_{\bar{t}}}$ *is tag-preserving over $T \to T_{\overline{x_t}}$ when $t = n$.*

From the lemma above, we get the hint that the copy subtree procedure works only at the layer directly above leaf transitions, i.e., when applied to x_n. However, if we can reorder the variable without changing the set of quantum states encoded in a TA, then the projection procedure can be applied to any qubit. Below we will demonstrate a procedure for variable reordering (it is similar to a BDD variable reordering procedure [Felt et al. 1993]), but with an additional effort to preserve tree tags.

Example 6.9. Consider the following tree with the variable order $x_1 > x_2$

$$x_1(x_2(c_{00}, \mathbf{c_{01}}), x_2(\mathbf{c_{10}}, c_{11})),$$

here c_{ij} is the amplitude of $|ij\rangle$, which intuitively means x_1 takes value i and x_2 takes j. If we swap the variable order of the two variables, one can construct the tree below to capture the same quantum state

$$x_2(x_1(c_{00}, \mathbf{c_{10}}), x_1(\mathbf{c_{01}}, c_{11})).$$

Algorithm 7: Forward variable order swapping procedure on x_t, $\mathsf{f.swap}_t(\mathcal{A})$

Input: A tagged TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R}\rangle$
Output: The tagged TA $\langle Q', \Sigma', \Delta', \mathcal{R}\rangle$

1 $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{add}} := \emptyset, Q' := Q, \Sigma' := \Sigma$;
2 **foreach** $q \xrightarrow{x_t^h} (q_0, q_1), q_0 \xrightarrow{x_l^i} (q_{00}, q_{01}), q_1 \xrightarrow{x_l^j} (q_{10}, q_{11}) \in \Delta$ **do**
3 $\quad \Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{x_l^{i,j}} (q_0', q_1'), q_0' \xrightarrow{x_t^h} (q_{00}, \mathbf{q_{10}}), q_1' \xrightarrow{x_t^h} (\mathbf{q_{01}}, q_{11})\}$;
4 $\quad \Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{x_t^h} (q_0, q_1), q_0 \xrightarrow{x_l^i} (q_{00}, \mathbf{q_{01}}), q_1 \xrightarrow{x_l^j} (\mathbf{q_{10}}, q_{11})\}$;
5 $\quad Q' := Q' \cup \{q_0', q_1'\}$;
6 $\quad \Sigma' := \Sigma' \cup \{x_l^{i,j}\}$;
7 **return** $\langle Q', \Sigma', (\Delta \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R}\rangle$;

Notice the main difference of the two trees is that the two leaf labels c_{10} and c_{01} are swapped. This is because the second tree first picks the value of x_2 and then x_1, so the 01 node should be labeled c_{10}, which denotes x_1 takes value 1 and x_2 takes value 0. □

Inspired by the example, we can swap the order of two consecutive variables by modifying the transitions of a TA. One difficulty is that we want to keep trees' tags, so we introduce two procedures *forward variable order swapping* (Algorithm 7) and *backward variable order swapping* (Algorithm 8) to modify a variable's order while maintaining the trees' tag.

Algorithm 7 swaps the variable order of x_t and its succeeding symbol x_l, assuming the variable order is $\ldots > x_t > x_l > \ldots$. We assume that before running forward variable swapping, all symbols corresponding to qubits x_t and x_l are assigned unique numbers by the tagging procedure. After running the forward swapping procedure, we remember the unique numbers of both succeeding symbols x_l^i and x_l^j at the new upper layer's symbol $x_l^{i,j}$ (Line 3). So the trees' tag can be recovered in the backward variable order swapping procedure (Line 3 of Algorithm 8).

Then, the projection is computed as follows:

$$\mathsf{Prj}(\mathcal{A}, x_t, b) = \mathsf{b.swap}_t^{n-t}(\mathsf{s.copy}(\mathsf{f.swap}_t^{n-t}(\mathcal{A}), x_t, b)), \tag{13}$$

where a superscript i denotes repetition of the procedure i times. Each time when the forward swapping procedure is triggered, we move x_t^h one layer lower in all trees accepted by $\mathcal{A}$. We can move x_t^h to the layer above the leaf by repeatedly applying the forward swapping procedure, which fulfills the requirement for executing the subtree copying procedure. Then we use the backward swap procedure to return the variables to the original order. This procedure is potentially

Algorithm 8: Backward variable order swapping procedure on x_t, $\mathsf{b.swap}_t(\mathcal{A})$

Input: A tagged TA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R}\rangle$
Output: The tagged TA $\langle Q', \Sigma', \Delta', \mathcal{R}\rangle$

1 $\Delta_{\mathsf{rm}} := \Delta_{\mathsf{add}} := \emptyset, Q' := Q, \Sigma' := \Sigma$;
2 **foreach** $q \xrightarrow{x_l^{i,j}} (q_0', q_1'), q_0' \xrightarrow{x_t^h} (q_{00}, q_{10}), q_1' \xrightarrow{x_t^h} (q_{01}, q_{11}) \in \Delta$ **do**
3 $\quad \Delta_{\mathsf{add}} := \Delta_{\mathsf{add}} \cup \{q \xrightarrow{x_t^h} (q_0'', q_1''), q_0'' \xrightarrow{x_l^i} (q_{00}, \mathbf{q_{01}}), q_1'' \xrightarrow{x_l^j} (\mathbf{q_{10}}, q_{11})\}$;
4 $\quad \Delta_{\mathsf{rm}} := \Delta_{\mathsf{rm}} \cup \{q \xrightarrow{x_l^{i,j}} (q_0', q_1'), q_0' \xrightarrow{x_t^h} (q_{00}, \mathbf{q_{10}}), q_1' \xrightarrow{x_t^h} (\mathbf{q_{01}}, q_{11})\}$;
5 $\quad Q' := Q' \cup \{q_0'', q_1''\}$;
6 $\quad \Sigma' := \Sigma' \setminus \{x_l^{i,j}\}$;
7 **return** $\langle Q', \Sigma', (\Delta \setminus \Delta_{\mathsf{rm}}) \cup \Delta_{\mathsf{add}}, \mathcal{R}\rangle$;

expensive, but TA minimization algorithms [Abdulla et al. 2008, 2007; Comon et al. 2008] can help to significantly reduce the cost.

Example 6.10. Here we demonstrate how the projection operations works with a concrete example. We assume that $\mathcal{A}$ is a tagged TA with the root state q and the following transitions:

$$q \xrightarrow{x_1^1} (q_l, q_r) \qquad \begin{array}{l} q_l \xrightarrow{x_2^2} (q_1, q_0) \\ q_l \xrightarrow{x_2^3} (q_0, q_1) \\ q_r \xrightarrow{x_2^4} (q_0, q_0) \end{array} \qquad \begin{array}{l} q_0 \xrightarrow{\mathbf{0}} () \\ q_1 \xrightarrow{\mathbf{1}} () \end{array}$$

Observe that $\mathcal{L}(\mathcal{A}) = \{T_1, T_2\}$, where

$$T_1 = x_1^1(x_2^2(\mathbf{1},\mathbf{0}), x_2^4(\mathbf{0},\mathbf{0})) \quad \text{and} \quad T_2 = x_1^1(x_2^3(\mathbf{0},\mathbf{1}), x_2^4(\mathbf{0},\mathbf{0})).$$

Then $\mathsf{f.swap}_k(\mathcal{A})$ produces a TA with a single root state q and the following transitions

$$\begin{array}{llll} q \xrightarrow{x_2^{2,4}} (q_2, q_3) & q_2 \xrightarrow{x_1^1} (q_1, q_0) & q_4 \xrightarrow{x_1^1} (q_0, q_0) & q_0 \xrightarrow{\mathbf{0}} () \\ q \xrightarrow{x_2^{3,4}} (q_4, q_5) & q_3 \xrightarrow{x_1^1} (q_0, q_0) & q_5 \xrightarrow{x_1^1} (q_1, q_0) & q_1 \xrightarrow{\mathbf{1}} () \end{array}$$

The language $\mathcal{L}(\mathsf{f.swap}_k(\mathcal{A}))$ is $\{T_1', T_2'\}$, where

$$T_1' = x_2^{2,4}(x_1^1(\mathbf{1},\mathbf{0}), x_1^1(\mathbf{0},\mathbf{0})) \quad \text{and} \quad T_2' = x_2^{3,4}(x_1^1(\mathbf{0},\mathbf{0}), x_1^1(\mathbf{1},\mathbf{0})).$$

Note that T_1' and T_2' represent the same quantum states as T_1 and T_2 above. Then $\mathsf{s.copy}_1(\mathsf{f.swap}_1(\mathcal{A}))$ produces the following TA with the root state q:

$$\begin{array}{llll} q \xrightarrow{x_2^{2,4}} (q_2, q_3) & q_2 \xrightarrow{x_1^1} (q_0, q_0) & q_4 \xrightarrow{x_1^1} (q_0, q_0) & q_0 \xrightarrow{\mathbf{0}} () \\ q \xrightarrow{x_2^{3,4}} (q_4, q_5) & q_3 \xrightarrow{x_1^1} (q_0, q_0) & q_5 \xrightarrow{x_1^1} (q_0, q_0) & q_1 \xrightarrow{\mathbf{1}} () \end{array}$$

Next we apply the backward swapping procedure to obtain $\mathcal{A}_{T_{x_1}}$, the final result of applying projection on $\mathcal{A}$. More concretely, $\mathcal{A}_{T_{x_1}} = \mathsf{b.swap}_1(\mathsf{s.copy}_1(\mathsf{f.swap}_1(\mathcal{A}))$ produces a TA with the root state q and the following transitions:

$$\begin{array}{llll} q \xrightarrow{x_1^1} (q_2', q_3') & q_2' \xrightarrow{x_2^2} (q_0, q_0) & q_4' \xrightarrow{x_2^3} (q_0, q_0) & q_0 \xrightarrow{\mathbf{0}} () \\ q \xrightarrow{x_1^1} (q_4', q_5') & q_3' \xrightarrow{x_2^4} (q_0, q_0) & q_5' \xrightarrow{x_2^4} (q_0, q_0) & q_1 \xrightarrow{\mathbf{1}} () \end{array}$$

Observe that the language after projection is

$$\mathcal{L}(\mathcal{A}_{T_{x_1}}) = \{x_1(x_2^2(\mathbf{0},\mathbf{0}), x_2^4(\mathbf{0},\mathbf{0})), x_1(x_2^3(\mathbf{0},\mathbf{0}), x_2^4(\mathbf{0},\mathbf{0}))\},$$

which is the expected result. □

Theorem 6.11. *Let $\mathcal{A}$ be a tagged TA. Then it holds that* $\mathsf{Prj}(\mathcal{A}, x_t, b) \simeq_{\mathrm{Tag}} \mathcal{A}$ *and, moreover,* $\mathcal{L}(\mathsf{Prj}(\mathcal{A}, x_t, b)) = \{b \ ? \ T_{x_t} \ : \ T_{\overline{x_t}} \mid T \in \mathcal{L}(\mathcal{A})\}$.

6.2.4 Binary Operation: $\mathcal{A}_{T_1 \pm T_2}$. Binary operation can be done by a modified product construction (cf. Algorithm 9). Notice that since we apply binary operations only over TAs derived from the same source TA, i.e., initially they have the same k at the leaf transitions, and the only possibility of changing the k part of a leaf symbol is the multiplication with $\frac{1}{\sqrt{2}}$, which is done only after all binary operations in Table 1, we can safely assume without loss of generality that $k_1 = k_2$.

Theorem 6.12. *Let $\mathcal{A}_{T_1}$ and $\mathcal{A}_{T_2}$ be two tagged TAs. Then it holds that* $\mathcal{L}(\mathsf{Bin}(\mathcal{A}_1, \mathcal{A}_2, \pm)) = \{T_1 \pm T_2 \mid T_1 \in \mathcal{L}(\mathcal{A}_{T_1}) \wedge T_2 \in \mathcal{L}(\mathcal{A}_{T_2}) \wedge \mathrm{Tag}(T_1) = \mathrm{Tag}(T_2)\}$.

Corollary 6.13. *The composition-based encoding of quantum gate operations is correct.*

Proof. Follows by Theorems 6.6, 6.7, 6.11 and 6.12. □

Algorithm 9: Binary operation, $\mathsf{Bin}(\mathcal{A}_1, \mathcal{A}_2, \pm)$

Input: Two tagged TAs $\mathcal{A}_1 = \langle Q_1, \Sigma, \Delta_1, \{q_1\}\rangle$ and $\mathcal{A}_2 = \langle Q_2, \Sigma, \Delta_2, \{q_2\}\rangle$.
Output: The tagged TA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \langle T_1 \pm T_2 \mid T_1 \in \mathcal{L}(\mathcal{A}_1) \wedge T_2 \in \mathcal{L}(\mathcal{A}_2)\rangle$

1 $\Delta'_i := \{(q^1, q^2) \xrightarrow{x_j^i} ((q_l^1, q_l^2), (q_r^1, q_r^2)) \mid q^1 \xrightarrow{x_j^i} (q_l^1, q_r^1) \in \Delta_1 \wedge q^2 \xrightarrow{x_j^i} (q_l^2, q_r^2) \in \Delta_2\}$;

2 $\Delta'_l := \{(q^1, q^2) \xrightarrow{(a_1 \pm a_2, b_1 \pm b_2, c_1 \pm c_2, d_1 \pm d_2, k_1)} () \mid q^1 \xrightarrow{(a_1, b_1, c_1, d_1, k_1)} () \in \Delta_1 \wedge q^2 \xrightarrow{(a_2, b_2, c_2, d_2, k_2)} () \in \Delta_2\}$;

3 **return** $\langle Q_1 \times Q_2, \Sigma', \Delta', \{(q_1, q_2)\}\rangle$;

7 EXPERIMENTAL EVALUATION

We implemented the proposed TA-based algorithm as a prototype tool named AutoQ in C++. We provide two settings: Hybrid, which uses the permutation-based approach (Section 5) to handle supported gates and switches to the composition-based approach for the other gates, and Composition, which handles all gates using the composition-based approach (Section 6). For checking language equivalence between the TA representing the set of reachable configurations and the TA for the post-condition, we use the Vata library [Lengál et al. 2012]. We use a lightweight simulation-based reduction [Bustan and Grumberg 2003] after finishing the Y, Z, S, T, CNOT, CZ, and Tofolli gate operations to keep the obtained TAs small.[5] All experiments were conducted on a server with an AMD EPYC 7742 64-core processor (1.5 GHz), 1,152 GiB of RAM (24 GiB for each process), and a 1 TB SSD running Ubuntu 20.04.4 LTS. Further details (pre- and post-conditions, circuits, etc.) can be found in [Chen et al. 2023b].

Data sets. We use the following set of benchmarks with quantum circuits:

- BV: Bernstein-Vazirani's algorithm with one hidden string of length n [Bernstein and Vazirani 1993],
- MCToffoli: circuits implementing multi-controlled Toffoli gates of size n using a variation of Nielsen and Chuang's decomposition [Nielsen and Chuang 2011] with standard Toffoli gates,
- Grover-Sing and Grover-All: implementation of Grover's search [Grover 1996] for a single oracle and for all possible oracles of length n (we encode the oracle's answer to be taken from the input; cf. [Chen et al. 2023b] for more details),
- FeynmanBench: 45 benchmarks from the tool suite Feynman [Amy 2018],
- RevLib: 80 benchmarks of reversible and quantum circuits [Wille et al. 2008], and
- Random: 20 randomly generated quantum circuits (10 circuits with 35 qubits and 105 gates and 10 circuits with 70 qubits and 210 gates).

We note that the benchmarks did not contain any unsupported gates.

Other tools. Since no existing work follows the same approach as we do, we compared AutoQ with representatives of the following approaches:

- *Quantum circuit simulators*: These compute the output of a quantum circuit for a given input quantum state. As a representative, we selected SliQSim [Tsai et al. 2021], a state-of-the-art quantum circuit simulator based on decision diagrams, which also works with a precise algebraic representation of complex numbers. We also tried the simulator from Qiskit [ANIS

[5] Our technique computes a non-maximum simulation by only checking whether states have the same successors. The results are in many cases the same as if the maximum simulation were computed, but the performance is much better. Further evaluation of this optimization of simulation is a future work.

Table 2. Verification of quantum algorithm. Here, n denotes the parameter value for the circuit, **#q** denotes the number of qubits, **#G** denotes the number of gates in the circuit. For AutoQ, the columns **before** and **after** have the format "states (transitions)" denoting the number of states and transitions in TA in the pre-condition and the output of our analysis respectively. The column **analysis** contains the time it took AutoQ to derive the TA for the output states and = denotes the time it took Vata to test equivalence. The timeout was 12 min. We use colours to distinguish the best result in each row and timeouts.

				AutoQ-Hybrid						AutoQ-Composition						SliQSim	Feynman	
	n	#q	#G	before		after		analysis	=	before		after		analysis	=	time	verdict	time
BV	95	96	241	193	(193)	193	(193)	6.0s	0.0s	193	(193)	193	(193)	7.1s	0.0s	0.0s	equal	0.5s
	96	97	243	195	(195)	195	(195)	5.9s	0.0s	195	(195)	195	(195)	7.1s	0.0s	0.0s	equal	0.5s
	97	98	246	197	(197)	197	(197)	6.3s	0.0s	197	(197)	197	(197)	7.4s	0.0s	0.0s	equal	0.6s
	98	99	248	199	(199)	199	(199)	6.5s	0.0s	199	(199)	199	(199)	7.7s	0.0s	0.0s	equal	0.6s
	99	100	251	201	(201)	201	(201)	6.7s	0.0s	201	(201)	201	(201)	7.8s	0.0s	0.0s	equal	0.6s
Grover-Sing	12	24	5,215	49	(49)	71	(71)	11s	0.0s	49	(49)	71	(71)	49s	0.0s	2.8s	timeout	
	14	28	12,217	57	(57)	83	(83)	31s	0.0s	57	(57)	83	(83)	2m26s	0.0s	18s	timeout	
	16	32	28,159	65	(65)	95	(95)	1m29s	0.0s	65	(65)	95	(95)	6m59s	0.0s	1m41s	timeout	
	18	36	63,537	73	(73)	107	(107)	4m1s	0.0s	timeout						9m27s	timeout	
	20	40	141,527	81	(81)	119	(119)	10m56s	0.0s	timeout						timeout	timeout	
MCToffoli	8	16	15	33	(42)	104	(149)	0.0s	0.0s	33	(42)	404	(915)	2.8s	0.0s	1.6s	equal	0.0s
	10	20	19	41	(52)	150	(216)	0.0s	0.0s	41	(52)	1,560	(3,607)	27s	0.0s	6.1s	equal	0.1s
	12	24	23	49	(62)	204	(295)	0.0s	0.0s	49	(62)	6,172	(14,363)	6m48s	0.1s	25s	equal	0.1s
	14	28	27	57	(72)	266	(386)	0.1s	0.0s	timeout						1m40s	equal	0.1s
	16	32	31	65	(82)	336	(489)	0.2s	0.0s	timeout						timeout	equal	0.2s
Grover-All	6	18	357	37	(43)	252	(315)	3.3s	0.0s	37	(43)	510	(573)	12s	0.0s	1.7s	timeout	
	7	21	552	43	(50)	481	(608)	10s	0.0s	43	(50)	1,123	(1,250)	42s	0.0s	5.4s	timeout	
	8	24	939	49	(57)	934	(1,189)	39s	0.1s	49	(57)	2,472	(2,727)	2m40s	0.0s	26s	timeout	
	9	27	1,492	55	(64)	1,835	(2,346)	2m17s	0.4s	55	(64)	5,421	(5,932)	10m13s	0.1s	2m5s	timeout	
	10	30	2,433	61	(71)	3,632	(4,655)	9m48s	2.1s	timeout						11m31s	timeout	

et al. 2021] (which does not provide a precise representation of numbers), but it was slower than SliQSim so we do not include it in the results.

- *Quantum circuit equivalence checkers*: We selected the following equivalence checkers: the verifier from the Feynman[6] tool suite [Amy 2018] (based on the path sum) and Qcec[7] [Burgholzer and Wille 2020] (combining decision diagrams, the ZX-calculus [Coecke and Duncan 2011], and random stimuli generation [Burgholzer et al. 2021]).

We evaluated AutoQ in two use cases, described in detail below.

7.1 Verification Against Pre- and Post-Conditions

In the first experiment, we compared how fast AutoQ computes the set of output quantum states and checks whether the set satisfies a given post-condition. We compared against the simulator SliQSim in the setting when we ran it over all states encoded in the pre-condition of the quantum algorithm and accumulated the times. We note that we did not include the time for comparing the result of SliQSim against a post-condition specification due to the following limitation of the tool: it can produce the state after executing the circuit in the vector form, but this step is not optimized and is quite time-consuming. Since the step of accumulating the obtained states could possibly be done in a more efficient way, avoiding transforming them first into the vector form, we do not include it in the runtime to not give SliQSim an unfair disadvantage. The timeout was 12 min.

We also include the time taken by Feynman to check the equivalence of the circuits with themselves. Although checking equivalence of quantum circuits is a harder problem than what we are solving (so the results cannot be used for direct comparison with AutoQ), we include these results in order to give an idea about hardness of the circuits for path-sum-based approaches.

[6]Git commit 56e5b771
[7]Version 2.0.0

Table 3. Results for bug finding. The notation is the same as in Table 2. In addition, the column **bug?** indicates if the tool caught the injected bug: T denotes that the bug was found, F denotes that the tool gave an incorrect result, and — means unknown result (includes the tool reporting *unknown*, crash, or not enough resources). AutoQ finds all bugs within the time limit, and we provide the number of iterations needed to catch the bug (column **iter**). The timeout was 30 min.

				AutoQ		Feynman		Qcec						AutoQ		Feynman		Qcec	
	circuit	#q	#G	time	iter	time	bug?	time	bug?		circuit	#q	#G	time	iter	time	bug?	time	bug?
FeynmanBench	csum_mux_9	30	141	0.8s	1	6.5s	—	44.0s	F		hwb10	16	31,765	1m42s	1	timeout		30.2s	T
	gf2^10_mult	30	348	2.0s	1	0.6s	—	42.7s	F		hwb11	15	87,790	4m23s	1	timeout		35.9s	T
	gf2^16_mult	48	876	11s	1	4.8s	—	58.5s	T		hwb12	20	171,483	13m43s	1	timeout		1m3s	T
	gf2^32_mult	96	3,323	2m4s	1	48.1s	—	1m58s	T		hwb8	12	6,447	15s	1	timeout		23.4s	T
	ham15-high	20	1,799	8.0s	1	3m51s	—	30.2s	T		qcla_adder_10	36	182	2.8s	1	1.3s	—	46.6s	F
	mod_adder_1024	28	1,436	10s	1	9.2s	—	31.9s	T		qcla_mod_7	26	295	2.6s	1	1m24s	—	38.4s	F
Random	35a	35	106	3.2s	1	0.2s	—	45.7s	F		70a	70	211	16s	1	1.1s	—	1m18s	T
	35b	35	106	1.4s	1	0.2s	T	47.8s	F		70b	70	211	14s	1	0.8s	T	1m11s	T
	35c	35	106	1.3s	1	0.2s	T	47.5s	T		70c	70	211	12s	1	0.9s	—	1m24s	T
	35d	35	106	1.3s	1	0.2s	T	48.2s	T		70d	70	211	29m29s	36	1.2s	T	1m26s	T
	35e	35	106	1.3s	1	0.1s	—	50.6s	T		70e	70	211	17s	1	1.0s	—	1m30s	T
	35f	35	106	2.4s	1	0.3s	T	49.7s	F		70f	70	211	33s	1	0.9s	T	1m26s	F
	35g	35	106	4.0s	3	0.2s	—	55.3s	T		70g	70	211	14m42s	44	1.2s	—	1m35s	T
	35h	35	106	1.0s	1	0.2s	—	0.6s	—		70h	70	211	13s	1	1.2s	—	1m36s	T
	35i	35	106	1.3s	1	0.2s	T	54.8s	T		70i	70	211	23s	1	1.2s	—	1m36s	T
	35j	35	106	1.8s	1	0.2s	—	51.4s	F		70j	70	211	2m5s	1	1.4s	—	1m34s	T
RevLib	add16_174	49	65	2.6s	1	timeout		1m8s	T		urf1_149	9	11,555	30s	1	timeout		35.8s	T
	add32_183	97	129	17s	1	timeout		2m4s	T		urf2_152	8	5,031	11s	1	21m33s	T	32.5s	T
	add64_184	193	257	1m55s	1	timeout		0.6s	—		urf3_155	10	26,469	1m19s	1	timeout		33.0s	T
	avg8_325	320	1,758	21m18s	1	timeout		0.5s	—		urf4_187	11	32,005	1m57s	1	timeout		31.4s	T
	bw_291	87	308	10s	1	11.7s	T	1m55s	T		urf5_158	9	10,277	27s	1	timeout		26.6s	T
	cycle10_293	39	79	0.5s	1	0.4s	T	1m7s	T		urf6_160	15	10,741	1m6s	1	timeout		36.2s	T
	e64-bdd_295	195	388	36s	1	timeout		0.5s	—		hwb6_301	46	160	2.0s	1	2.7s	T	1m7s	T
	ex5p_296	206	648	1m52s	1	1m29s	T	0.4s	—		hwb7_302	73	282	8.3s	1	10.9s	T	1m38s	T
	ham15_298	45	154	0.6s	1	0.6s	T	1m14s	T		hwb8_303	112	450	27s	1	37.9s	T	2m22s	T
	mod5adder_306	32	97	0.5s	1	0.7s	T	1m1s	T		hwb9_304	170	700	1m33s	1	2m20s	T	0.6s	—
	rd84_313	34	105	0.5s	1	1.1s	T	1m2s	T										

We ran this experiment on the benchmarks where the semantics was known to us so that we could construct TAs with pre- and post-conditions. These were the following: BV, MCToffoli, Grover-Sing, and Grover-All. We give the results in Table 2. Both BV and Grover-Sing work with only one input state, which should be most favourable for simulators. Surprisingly, for the case of Grover-Sing, AutoQ outperforms SliQSim on large cases (out of curiosity, we tried to run SliQSim on Grover-Sing (n=20) without a timeout; the running time was 51m43s). We attribute the good performance of AutoQ to the compactness of the TA representation of Grover's state space. On the other hand, both MCToffoli and Grover-All consider 2^n input states and we can observe the exponential factor emerging; hence AutoQ outperforms SliQSim in large cases. All tools perform pretty well on BV, even cases with 100 qubits can be easily handled. We can also see that Hybrid is consistently faster than Composition.

7.2 Finding Bugs

In the following experiment, we compared AutoQ with the equivalence checkers Feynman and Qcec and evaluated the ability of the tools to determine that two quantum circuits are non-equivalent (this is to simulate the use case of verifying the output of an optimizer). We took circuits from the benchmarks FeynmanBench, Random, and RevLib, and for each circuit, we created a copy and injected an artificial bug (one additional randomly selected gate at a random location). Then we ran the tools and let them check circuit equivalence; for AutoQ, we let it compute two TAs representing sets of output states for both circuits for the given set of input states and then checked their language equivalence with Vata.

Our strategy for finding bugs with AutoQ (we used the Hybrid setting) was the following: We started with a TA representing a single basis state, i.e., a TA with no top-down nondeterminism, and gradually added more non-deterministic transitions (in each iteration one randomly chosen

transition) into the TA, making it represent a larger set of states, running the analysis for each of the TAs, until we found the bug. This proved to be a successful strategy, since running the analysis with an input TA representing, e.g., all possible basis states, might be too challenging (generally speaking, the larger is the TA representing the set of states, the slower is the analysis).

We present the results in Table 3. We exclude trivial cases (all tools can finish within 5 s) and difficult cases that no tool can handle within the timeout period (30 min). We can see that many of the cases were so tricky that equivalence checkers failed to conclude anything, while AutoQ succeeded in finding the bug with just the first few TAs. For two instances from Random (70d and 70g), we found the bug after trying 36 TAs after 29m29s and 44 TAs after 14m42s, respectively. For a few cases (e.g., csum_mux_9), Qcec did not find the bug and reported that the circuits were equivalent (F)[8], while AutoQ reported it (T). For these cases, we fed the witness produced by AutoQ to SliQSim and confirmed the two circuits are different.

The results show that our approach to hunting for bugs in quantum circuits is beneficial, particularly for larger circuits where equivalence checkers do not scale. For such cases, AutoQ can still find bugs using a weaker specification. For instance, AutoQ was able to find bugs in some large-scale instances from RevLib with hundreds of qubits, e.g., add64_184 and avg_8_325, while both Feynman and Qcec fail.

We note that the area of quantum circuit equivalence checking is rapidly advancing. When preparing the final version, we became aware of SliQEC [Chen et al. 2022; Wei et al. 2022], a recent tool that outperforms the other equivalence checkers that we tried on this benchmark.

8 RELATED WORK

Circuit equivalence checkers are often very efficient but less flexible in specifying the desired property (only equivalence). Our approach can switch to a lightweight specification when verification fails due to insufficient resources and still find bugs in the design. Often equivalence checking is done by a reduction to normal form using a set of rewriting rules. *Path-sum* is a recent approach proposed in [Amy 2018], whose rewrite rules can solve the equivalence problem of Clifford group circuits in polynomial time. The *ZX-calculus* [Coecke and Duncan 2011] is a graphical language that is particularly useful in circuit optimization and proving equivalence. The works of [Hietala et al. 2019] ensures correctness of the rewrite rules with a theorem prover. Quartz [Xu et al. 2022b] is a circuit optimization framework consisting of an equivalence checker based on some precomputed equivalence sets. We pick Feynman [Amy 2018], a state-of-the-art equivalence checker based on path-sum, and Qcec [Burgholzer and Wille 2020], based on decision diagrams and ZX-calculus, as the baseline tools for comparison. *Quantum circuit simulators*, e.g. SliQSim [Tsai et al. 2021], can be used as equivalence checkers for a finite number of inputs by trying all basis states.

Quantum abstract interpretation [Perdrix 2008; Yu and Palsberg 2021] is particularly efficient in processing large-scale circuits, but it over-approximates state space and cannot conclude anything when verification fails. For instance, the work in [Yu and Palsberg 2021] can only distinguish quantum states with zero and non-zero probability (and cannot derive exact boundary probabilities). In contrast, our approach precisely represents reachable states and can reveal bugs. One can consider our approach to be an instantiation of classical abstract interpretation [Cousot and Cousot 1977] that is precise, and our approach to non-equivalence testing as comparing output abstract contexts of two programs. *Quantum model checking* supports a rich specification language (flavors of temporal logic [Feng et al. 2013; Mateus et al. 2009; Xu et al. 2022a]). It can be seen as an extension of probabilistic model checking [Feng et al. 2017, 2015, 2013; Xu et al. 2022a; Ying 2021; Ying and Feng 2021; Ying et al. 2014] and is more suitable for verifying high-level protocols due to the

[8]This bug has been confirmed by the Qcec team and fixed later, cf. [QCE 2022].

limited scalability [Anticoli et al. 2016]. Techniques based on *quantum simulation* [Green et al. 2013; Niemann et al. 2016; Pednault et al. 2017; Samoladas 2008; Tsai et al. 2021; Viamontes et al. 2009; Wecker and Svore 2014; Zulehner et al. 2019; Zulehner and Wille 2019] allow only one input quantum state and thus have limited analyzing power.

Quantum Hoare logic [Feng and Ying 2021; Liu et al. 2019; Unruh 2019; Ying 2012; Zhou et al. 2019]) allows verification against complex correctness properties and rich program constructs such as branches and loops, but requires significant manual work. On the other hand, *quantum incorrectness logic* [Yan et al. 2022] is a dual of quantum Hoare logic that allows showing the existence of a bug, but cannot prove its absence. The Qbricks [Chareton et al. 2021] approach alleviates the difficulty of proof search by combining state-of-the-art theorem provers with decision procedures, but, according to their experiments, still requires a significant amount of human intervention. For instance, their experiments show that it requires 125 times intervention during verification of Grover's search w.r.t. an arbitrary number of qubits.

9 CONCLUDING REMARKS

We have introduced a new paradigm for quantum circuit analysis that is exciting from both practical and theoretical lenses. We demonstrated one of its potential applications—circuit non-equivalence checking, but we believe there could be much more. In our own experience of using the method to prepare the benchmarks, its role is similar to a static assertion checker (like *software model checkers* for classical programs [Chen et al. 2016; Heizmann et al. 2018]); it helped us greatly to find several problems while composing the circuits. The connection to automata-based verification is also quite exciting. A series of approaches from the classical world should also be helpful in the quantum case. For instance, the idea of regular tree model checking could be leveraged to verify parameterized quantum circuits (w.r.t. an arbitrary number of qubits) [Abdulla et al. 2002; Bouajjani et al. 2012]. For this, one would need to deal with TAs with loops, where tagging cannot be done anymore to impose relations among trees (one would need to use an unbounded number of tags)—new ideas are needed. Automata-learning can be used for automatic loop invariant inference [Chen et al. 2017a]. Symbolic automata [D'Antoni and Veanes 2017] and register automata [Chen et al. 2017b] would allow using variables to describe amplitude (instead of a fixed alphabet as we use now). We believe there are many other techniques from the automata world that could be used to extend our framework and be applied in the area of analysing quantum circuits.

ACKNOWLEDGMENTS

We thank the reviewers for their in-depth remarks that helped us improve the quality of the paper and the artifact committee members for their helpful suggestions about the artifact. This material is based on a work supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme; the Czech Science Foundation project GA23-07565S; the FIT BUT internal project FIT-S-23-8151; and the NSTC QC project under Grant no. NSTC 111-2119-M-001-004-.

DATA AVAILABILITY STATEMENT

An environment with the tools and data used for the experimental evaluation in the current study is available at [Chen et al. 2023a].

REFERENCES

2022. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/

2022. The Qcec repository: Issue #200 (ZX-Checker produces invalid result). https://github.com/cda-tum/qcec/issues/200

Parosh Aziz Abdulla, Ahmed Bouajjani, Lukás Holík, Lisa Kaati, and Tomás Vojnar. 2008. Computing Simulations over Tree Automata. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*

2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (LNCS, Vol. 4963), C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 93–108. https://doi.org/10.1007/978-3-540-78800-3_8

Parosh Aziz Abdulla, Johanna Högberg, and Lisa Kaati. 2007. Bisimulation Minimization of Tree Automata. *Int. J. Found. Comput. Sci.* 18, 4 (2007), 699–713. https://doi.org/10.1142/S0129054107004929

Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. 2002. Regular Tree Model Checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings (LNCS, Vol. 2404)*. Springer, 555–568. https://doi.org/10.1007/3-540-45657-0_47

Dorit Aharonov. 2003. A Simple Proof that Toffoli and Hadamard are Quantum Universal. https://doi.org/10.48550/arxiv.quant-ph/0301040

Thorsten Altenkirch and Jonathan Grattage. 2005. A Functional Quantum Programming Language. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 249–258. https://doi.org/10.1109/LICS.2005.1

Matthew Amy. 2018. Towards Large-scale Functional Verification of Universal Quantum Circuits. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018 (EPTCS, Vol. 287)*, Peter Selinger and Giulio Chiribella (Eds.). 1–21. https://doi.org/10.4204/EPTCS.287.1

Matthew Amy. 2019. *Formal Methods in Quantum Circuit Design.* Ph. D. Dissertation. University of Waterloo.

MD SAJID ANIS, Abby-Mitchell, Héctor Abraham, et al. 2021. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

Linda Anticoli, Carla Piazza, Leonardo Taglialegne, and Paolo Zuliani. 2016. Towards Quantum Programs Verification: From Quipper Circuits to QPMC. In *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings (LNCS, Vol. 9720)*, Simon J. Devitt and Ivan Lanese (Eds.). Springer, 213–219. https://doi.org/10.1007/978-3-319-40578-0_16

Frank Arute et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (Oct. 2019), 505–510. https://doi.org/10.1038/s41586-019-1666-5 Number: 7779 Publisher: Nature Publishing Group.

Ethan Bernstein and Umesh V. Vazirani. 1993. Quantum complexity theory. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal (Eds.). ACM, 11–20. https://doi.org/10.1145/167088.167097

Jacob D. Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549, 7671 (2017), 195–202. https://doi.org/10.1038/nature23474

Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. 2012. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer* 14, 2 (2012), 167–191. https://doi.org/10.1007/s10009-011-0205-y

Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. 2000. Regular Model Checking. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings (LNCS, Vol. 1855)*, E. Allen Emerson and A. Prasad Sistla (Eds.). Springer, 403–418. https://doi.org/10.1007/10722167_31

P. Oscar Boykin, Tal Mor, Matthew Pulver, Vwani P. Roychowdhury, and Farrokh Vatan. 2000. A new universal and fault-tolerant quantum basis. *Inf. Process. Lett.* 75, 3 (2000), 101–107. https://doi.org/10.1016/S0020-0190(00)00084-3

Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

Lukas Burgholzer, Richard Kueng, and Robert Wille. 2021. Random Stimuli Generation for the Verification of Quantum Circuits. In *ASPDAC '21: 26th Asia and South Pacific Design Automation Conference, Tokyo, Japan, January 18-21, 2021*. ACM, 767–772. https://doi.org/10.1145/3394885.3431590

Lukas Burgholzer and Robert Wille. 2020. Advanced equivalence checking for quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 9 (2020), 1810–1824. https://doi.org/10.1109/TCAD.2020.3032630

Doron Bustan and Orna Grumberg. 2003. Simulation-based minimazation. *ACM Trans. Comput. Log.* 4, 2 (2003), 181–206. https://doi.org/10.1145/635499.635502

Yudong Cao, Jonathan Romero, Jonathan P. Olson, Matthias Degroote, Peter D. Johnson, Mária Kieferová, Ian D. Kivlichan, Tim Menke, Borja Peropadre, Nicolas P. D. Sawaya, Sukin Sim, Libor Veis, and Alán Aspuru-Guzik. 2019. Quantum Chemistry in the Age of Quantum Computing. *Chemical Reviews* 119, 19 (2019), 10856–10915. https://doi.org/10.1021/acs.chemrev.8b00803 arXiv:https://doi.org/10.1021/acs.chemrev.8b00803 PMID: 31469277.

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-Building Quantum Programs. In *ESOP (LNCS, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 148–177. https://doi.org/10.1007/978-3-030-72019-3_6

Tian-Fu Chen, Jie-Hong R. Jiang, and Min-Hsiu Hsieh. 2022. Partial Equivalence Checking of Quantum Circuits. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 594–604. https://doi.org/10.1109/QCE53715.2022.00082

Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023a. *An Automata-based Framework for Verification and Bug Hunting in Quantum Circuits*. https://doi.org/10.5281/zenodo.7811406

Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023b. An Automata-based Framework for Verification and Bug Hunting in Quantum Circuits (Technical Report). (2023). arXiv:2301.07747 [cs.LO]

Yu-Fang Chen, Chih-Duo Hong, Anthony W Lin, and Philipp Rümmer. 2017a. Learning to prove safety over parameterised concurrent systems. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 76–83. https://doi.org/10.23919/FMCAD.2017.8102244

Yu-Fang Chen, Chiao Hsieh, Ondřej Lengál, Tsung-Ju Lii, Ming-Hsien Tsai, Bow-Yaw Wang, and Farn Wang. 2016. PAC learning-based verification and model synthesis. In *Proceedings of the 38th International Conference on Software Engineering*. 714–724. https://doi.org/10.1145/2884781.2884860

Yu-Fang Chen, Ondřej Lengál, Tony Tan, and Zhilin Wu. 2017b. Register automata with linear arithmetic. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–12. https://doi.org/10.1109/LICS.2017.8005111

Carlo Ciliberto, Mark Herbster, Alessandro Davide Ialongo, Massimiliano Pontil, Andrea Rocchetto, Simone Severini, and Leonard Wossnig. 2018. Quantum Machine Learning: A Classical Perspective. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 474, 2209 (January 2018). https://doi.org/10.1098/rspa.2017.0551

Bob Coecke and Ross Duncan. 2011. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics* 13, 4 (apr 2011), 043016. https://doi.org/10.1088/1367-2630/13/4/043016

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. Tree automata techniques and applications.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Loris D'Antoni, Margus Veanes, Benjamin Livshits, and David Molnar. 2015. Fast: A Transducer-Based Language for Tree Manipulation. *ACM Trans. Program. Lang. Syst.* 38, 1 (2015), 1:1–1:32. https://doi.org/10.1145/2791292

Christopher M. Dawson and Michael A. Nielsen. 2006. The Solovay-Kitaev algorithm. *Quantum Inf. Comput.* 6, 1 (2006), 81–95. https://doi.org/10.26421/QIC6.1-6

Loris D'Antoni and Margus Veanes. 2017. The power of symbolic automata and transducers. In *International Conference on Computer Aided Verification*. Springer, 47–67. https://doi.org/10.1007/978-3-319-63387-9_3

Mark Ettinger, Peter Høyer, and Emanuel Knill. 2004. The quantum query complexity of the hidden subgroup problem is polynomial. *Inf. Process. Lett.* 91, 1 (2004), 43–48. https://doi.org/10.1016/j.ipl.2004.01.024

Andrew Fagan and Ross Duncan. 2019. Optimising Clifford Circuits with Quantomatic. *Electronic Proceedings in Theoretical Computer Science* 287 (jan 2019), 85–105. https://doi.org/10.4204/eptcs.287.5

Eric Felt, Gary York, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. 1993. Dynamic variable reordering for BDD minimization. In *Proceedings of the European Design Automation Conference 1993, EURO-DAC '93 with EURO-VHDL'93, Hamburg, Germany, September 20-24, 1993*. IEEE Computer Society, 130–135. https://doi.org/10.1109/EURDAC.1993.410627

Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Shenggang Ying. 2017. Model checking omega-regular properties for quantum Markov chains. In *28th International Conference on Concurrency Theory (CONCUR 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.CONCUR.2017.35

Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Lijun Zhang. 2015. QPMC: A Model Checker for Quantum Programs and Protocols. In *International Symposium on Formal Methods*, Nikolaj Bjørner and Frank de Boer (Eds.). Springer International Publishing, 265–272. https://doi.org/10.1007/978-3-319-19249-9_17

Yuan Feng and Mingsheng Ying. 2021. Quantum Hoare logic with classical variables. *ACM Transactions on Quantum Computing* 2, 4 (2021), 1–43. https://doi.org/10.1145/3456877

Yuan Feng, Nengkun Yu, and Mingsheng Ying. 2013. Model checking quantum Markov chains. *J. Comput. Syst. Sci.* 79, 7 (2013), 1181–1198. https://doi.org/10.1016/j.jcss.2013.04.002

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 333–342. https://doi.org/10.1145/2491956.2462177

Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, Gary L. Miller (Ed.). ACM, 212–219. https://doi.org/10.1145/237814.237866

Wakaki Hattori and Shigeru Yamashita. 2018. Quantum Circuit Optimization by Changing the Gate Order for 2D Nearest Neighbor Architectures. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings (LNCS, Vol. 11106)*, Jarkko Kari and Irek Ulidowski (Eds.). Springer, 228–243. https://doi.org/10.1007/978-3-319-99498-7_16

Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, et al. 2018. Ultimate Automizer and the search for perfect interpolants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 447–451. https://doi.org/10.1007/978-3-319-89963-3_30

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2019. Verified optimization in a quantum intermediate representation. *arXiv preprint arXiv:1904.06319* (2019).

Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. 2020. Optimization of quantum circuit mapping using gate transformation and commutation. *Integr.* 70 (2020), 43–50. https://doi.org/10.1016/j.vlsi.2019.10.004

Dominik Janzing, Pawel Wocjan, and Thomas Beth. 2005. "Non-Identity-Check" Is QMA-complete. *International Journal of Quantum Information* 03, 03 (2005), 463–473. https://doi.org/10.1142/S0219749905001067

Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. 2012. VATA: A library for efficient manipulation of non-deterministic tree automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 79–94. https://doi.org/10.1007/978-3-642-28756-5_7

Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal verification of quantum algorithms using quantum Hoare logic. In *International conference on computer aided verification*. Springer, 187–207. https://doi.org/10.1007/978-3-030-25543-5_12

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

Paul Massey, John A. Clark, and Susan Stepney. 2005. Evolution of a human-competitive quantum fourier transform algorithm using genetic programming. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, Hans-Georg Beyer and Una-May O'Reilly (Eds.). ACM, 1657–1663. https://doi.org/10.1145/1068009.1068288

Paulo Mateus, Jaime Ramos, Amílcar Sernadas, and Cristina Sernadas. 2009. *Temporal Logics for Reasoning about Quantum Systems*. Cambridge University Press, 389–413. https://doi.org/10.1017/CBO9781139193313.011

Nikolaj Moll, Panagiotis Barkoutsos, Lev S Bishop, Jerry M Chow, Andrew Cross, Daniel J Egger, Stefan Filipp, Andreas Fuhrer, Jay M Gambetta, Marc Ganzhorn, Abhinav Kandala, Antonio Mezzacapo, Peter Müller, Walter Riess, Gian Salis, John Smolin, Ivano Tavernelli, and Kristan Temme. 2018. Quantum optimization using variational algorithms on near-term quantum devices. *Quantum Science and Technology* 3, 3 (jun 2018), 030503. https://doi.org/10.1088/2058-9565/aab822

Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4 (2018). Issue 23. https://doi.org/10.1038/s41534-018-0072-4

Daniel Neider and Nils Jansen. 2013. Regular Model Checking Using Solver Technologies and Automata Learning. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings (LNCS, Vol. 7871)*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer, 16–31. https://doi.org/10.1007/978-3-642-38088-4_2

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

Philipp Niemann, Robert Wille, D. Michael Miller, Mitchell A. Thornton, and Rolf Drechsler. 2016. QMDDs: Efficient Quantum Function Representation and Manipulation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 35, 1 (2016), 86–99. https://doi.org/10.1109/TCAD.2015.2459034

Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, Erik W. Draeger, Eric T. Holland, and Robert Wisnieff. 2017. Pareto-Efficient Quantum Circuit Simulation Using Tensor Contraction Deferral. *CoRR* abs/1710.05867 (2017). http://arxiv.org/abs/1710.05867

Tom Peham, Lukas Burgholzer, and Robert Wille. 2022. Equivalence checking paradigms in quantum circuit design: a case study. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, Rob Oshana (Ed.). ACM, 517–522. https://doi.org/10.1145/3489517.3530480

Simon Perdrix. 2008. Quantum entanglement analysis based on abstract interpretation. In *International Static Analysis Symposium*. Springer, 270–282. https://doi.org/10.1007/978-3-540-69166-2_18

Vasilis Samoladas. 2008. Improved BDD Algorithms for the Simulation of Quantum Circuits. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings (LNCS, Vol. 5193)*, Dan Halperin and Kurt Mehlhorn (Eds.). Springer, 720–731. https://doi.org/10.1007/978-3-540-87744-8_60

Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 124–134. https://doi.org/10.1109/SFCS.1994.365700

Mathias Soeken, Robert Wille, Gerhard W. Dueck, and Rolf Drechsler. 2010. Window optimization of reversible and quantum circuits. In *13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2010, Vienna, Austria, April 14-16, 2010*. IEEE Computer Society, 341–345. https://doi.org/10.1109/DDECS.2010.5491754

Lee Spector. 2006. Automatic Quantum Computer Programming: A Genetic Programming Approach. (2006).

Yuan-Hung Tsai, Jie-Hong R. Jiang, and Chiao-Shan Jhang. 2021. Bit-Slicing the Hilbert Space: Scaling Up Accurate Quantum Circuit Simulation. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 439–444. https://doi.org/10.1109/DAC18074.2021.9586191

Dominique Unruh. 2019. Quantum Hoare logic with ghost variables. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785779

George F. Viamontes, Igor L. Markov, and John P. Hayes. 2007. Checking equivalence of quantum circuits and states. In *2007 International Conference on Computer-Aided Design, ICCAD 2007, San Jose, CA, USA, November 5-8, 2007*, Georges G. E. Gielen (Ed.). IEEE Computer Society, 69–74. https://doi.org/10.1109/ICCAD.2007.4397246

George F. Viamontes, Igor L. Markov, and John P. Hayes. 2009. *Quantum Circuit Simulation*. Springer. https://doi.org/10.1007/978-90-481-3065-8

Dave Wecker and Krysta M. Svore. 2014. LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing. *CoRR* abs/1402.4467 (2014). arXiv:1402.4467 http://arxiv.org/abs/1402.4467

Chun-Yu Wei, Yuan-Hung Tsai, Chiao-Shan Jhang, and Jie-Hong R. Jiang. 2022. Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, Rob Oshana (Ed.). ACM, 523–528. https://doi.org/10.1145/3489517.3530481

R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *Int'l Symp. on Multi-Valued Logic*. 220–225. https://doi.org/10.1109/ISMVL.2008.43 RevLib is available at http://www.revlib.org.

Robert Wille, Rod Van Meter, and Yehuda Naveh. 2019. IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, Jürgen Teich and Franco Fummi (Eds.). IEEE, 1234–1240. https://doi.org/10.23919/DATE.2019.8715261

Ming Xu, Jianling Fu, Jingyi Mei, and Yuxin Deng. 2022a. Model checking QCTL plus on quantum Markov chains. *Theor. Comput. Sci.* 913 (2022), 43–72. https://doi.org/10.1016/j.tcs.2022.01.044

Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. 2022b. Quartz: superoptimization of Quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 625–640. https://doi.org/10.1145/3519939.3523433

Shigeru Yamashita and Igor L. Markov. 2010. Fast equivalence-checking for quantum circuits. *Quantum Inf. Comput.* 10, 9&10 (2010), 721–734. https://doi.org/10.26421/QIC10.9-10-1

Peng Yan, Hanru Jiang, and Nengkun Yu. 2022. On incorrectness logic for Quantum programs. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–28. https://doi.org/10.1145/3527316

Mingsheng Ying. 2012. Floyd-Hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 6 (2012), 1–49. https://doi.org/10.1145/2049706.2049708

Mingsheng Ying. 2021. Model Checking for Verification of Quantum Circuits. In *International Symposium on Formal Methods*. Springer, 23–39. https://doi.org/10.1007/978-3-030-90870-6_2

Mingsheng Ying and Yuan Feng. 2021. *Model Checking Quantum Systems: Principles and Algorithms*. Cambridge University Press.

Mingsheng Ying, Yangjia Li, Nengkun Yu, and Yuan Feng. 2014. Model-checking linear-time properties of quantum systems. *ACM Transactions on Computational Logic (TOCL)* 15, 3 (2014), 1–31. https://doi.org/10.1145/2629680

Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H Ibarra. 2008. Symbolic string verification: An automata-based approach. In *International SPIN Workshop on Model Checking of Software*. Springer, 306–324. https://doi.org/10.1007/978-3-540-85114-1_21

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. 2011. Relational String Verification Using Multi-Track Automata. *Int. J. Found. Comput. Sci.* 22, 8 (2011), 1909–1924. https://doi.org/10.1142/S0129054111009112

Nengkun Yu and Jens Palsberg. 2021. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 542–558. https://doi.org/10.1145/3453483.3454061

Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. An applied quantum Hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1149–1162. https://doi.org/10.1145/3314221.3314584

Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, David Z. Pan (Ed.). ACM, 1–7. https://doi.org/10.1109/ICCAD45719.2019.8942057

Alwin Zulehner and Robert Wille. 2019. Advanced Simulation of Quantum Computations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 38, 5 (2019), 848–859. https://doi.org/10.1109/TCAD.2018.2834427

Received 2022-11-10; accepted 2023-03-31