Quantum Information & Computation
ISSN 1533-7146 Volume 25 (2025) pp.716–738
©Cerebration Science Publishing

Article

Realization of Deterministic Quantum Circuits for Non-Deterministic or Incompletely Specified Quantum State Machines

Manjith Kumar * and Marek Perkowski

Department of Electrical and Computer Engineering, Portland State University, 1900 SW 4th Avenue, Portland, OR 97201, USA; h8mp@pdx.edu
* Corresponding author: manjith@pdx.edu

Received date: 6 March 2025; Accepted date: 14 October 2025; Published online: 22 December 2025

Abstract: In classical logic design, there are machine learning methods based on converting a set of input-output traces to non-deterministic automata that are then converted to deterministic automata and synthesized using logic gates. This approach has not yet been extended to quantum automata. In this paper, we present a method to convert a set of input-output traces to a non-deterministic automaton, which is then converted to an incompletely specified multi-output Boolean function. The existing logic synthesis approaches for designing quantum circuits are insufficient to handle incompletely specified functions. So, we present a novel algorithm to synthesize logic functions with don't cares using permutative quantum gates. The original MMD (D.M.Miller, D. Maslov, and G.W.Dueck) algorithm minimizes only completely specified reversible functions using cascades of reversible gates. In this paper, this algorithm is modified to allow for the inclusion of don't cares within the given function's truth table (reversible or irreversible). The distinguishing property of our presented algorithm QAS, is that it does not add any ancilla qubits if it is not necessary. This algorithm solves the problem of synthesizing deterministic and non-deterministic quantum state machines, both completely specified and incompletely specified.

Keywords: deterministic quantum state machines; non-deterministic quantum state machines; incompletely specified boolean functions; reversible logic; quantum gates; quantum logic synthesis; ancilla qubit minimization; quantum cost; quantum automata

1. Introduction

Quantum Computing is a rapidly developing technology field that attempts to utilize quantum mechanical phenomena to build computers that are vastly more powerful than classical computers [1,2]. The building blocks of a quantum computer are quantum logic circuits that can manipulate information stored in qubits [3]. Quantum state transformations (other than measurement) are reversible in nature [4], i.e., given an output state, we can apply the inverse transformation to compute the input state [5]. The synthesis of quantum and reversible logic circuits is an important problem that needs to be solved to advance the field of quantum computing.

Many computational problems can be modeled using automata or "state machines". It may be useful to represent some of the computational problems in quantum computing using the quantum equivalent of classical automata. Several definitions for these automata exist under names like Quantum Automata (QA) [6], Quantum Finite Automata (QFA) [7], etc. The two well-known definitions of Quantum Automata are the Kondacs-Watrous Quantum Finite Automaton (KWQFA) [7] and the Moore-Crutchfield Quantum Finite Automaton (MCQFA) [6]. It has been demonstrated that QFA can solve certain problems more efficiently than its classical counterpart [8,9]. In classical logic, the abstract models of automata laid the foundations of computing and eventually evolved into practical state machines that can be implemented using logic circuits. Quantum computing can expect a similar evolution as we discover more useful algorithms and applications. It is important to develop practical state machine architectures and logic realizations using quantum gates. A variant of Quantum Automata called Quantum State Machines (QSMs) was introduced in [10,11]. It applies a sequence of quantum logic operations to realize the output and state transition functions of a state machine. The logic of quantum circuits is described by unitary matrices. If the matrices are permutative, then

DOI: 10.2478/qic-2025-0038

the circuit is deterministic [12]. We define a permutative quantum gate or permutative circuit as a quantum circuit whose unitary operator performs a permutation of the computational basis states, i.e., its unitary matrix is permutative. Such circuits use gates like CNOT, CCNOT, and NOT. If the matrices are general unitary, then the circuit is built from Hadamard, Chrestenson, and rotation gates in addition to Boolean or multi-valued logic gates (permutative gates) and measurement gates [13].

The survey of literature indicates that the problem of synthesizing non-deterministic quantum finite state machines has not been solved. The conversion of a non-deterministic FSM to a deterministic FSM is well known in classical logic. But this conversion generates don't cares, which are not handled by most of the logic synthesis frameworks in quantum computing, including Qiskit [14] and Quipper [15]. However, the problem of synthesizing incompletely specified reversible functions is of high importance in several areas listed below:

1. The oracles used in Grover [16], Quantum Walk [17] and similar algorithms like BHT QAOA [18] are normally permutative circuits, i.e., circuits built entirely from permutative gates. In some applications, the initial oracle is an incompletely specified logic function that should be converted to a completely specified Boolean or multi-valued permutative function. This has applications in Machine Learning.
2. Design of blocks included in larger oracles or spectral transforms, such as adders or other arithmetic circuits or code converters, where don't cares occur, similar to classical logic network design.
3. Circuit-level synthesis of reversible sequential circuits [19] and deterministic or non-deterministic QSM. This is especially useful in Machine Learning applications where the machine is created from input-output traces [20,21].
4. When a completely specified, multi-output Boolean function is synthesized, one of the methods is the so-called Miller Method [22] that transforms a multi-output (completely specified or incompletely specified) Boolean function to a strongly unspecified single-output function.
5. Another cause of the occurrence of don't cares in state machines is the encoding of states. Assume that the state machine is represented by a table where columns correspond to input states and rows correspond to internal states. For example, five symbolic input states encoded in minimum length code using three input variables (qubits) create three columns of don't cares in the transition table. Similarly, five internal states encoded in three memory elements (qubits) create three rows of don't cares.
6. Don't cares can occur in combinational logic when the number of output lines is smaller than the number of input lines. In such a situation, ancilla qubits are added to the output side to maintain the reversible nature. This ancilla qubit addition will manifest itself as a column of don't cares added to the output part of the truth table.

Many of these problems are known from logic synthesis of classical automata and circuits, but have never been solved for quantum automata. Clearly, there is a strong need to develop quantum logic synthesis tools that can handle don't cares in the output values.

The main contributions of this paper are as follows.

1. We propose a technique to realize non-deterministic state machines using quantum logic circuits.
2. We apply this technique to learning, where an automaton is derived from input-output traces and built using quantum logic circuits.
3. We present an algorithm, which will be referred to henceforth as "Quantum Automata Synthesizer" or QAS, that can transform an incompletely specified, or irreversible function, into a completely specified reversible function that can be synthesized using existing quantum logic synthesis tools.

To the best of our knowledge, this is the first paper to address the problem of realizing non-deterministic state machines using quantum circuits. The automata realization method presented in this paper is devoted mainly to QSM, but it can also be applied to other types of quantum automata, such as QA, QFA, two-way QFA [23], etc., and sequential reversible circuits. Moreover, not much is published on the efficient design of QSM. The current paper will partially fill this gap.

Our synthesis method makes use of the "transformation-based" reversible circuit synthesis software developed by D.M. Miller, D. Maslov, and G. W. Dueck (MMD) [24]. Our method has the following steps:

1. A non-deterministic symbolic automaton obtained from incomplete input-output traces is first transformed to an incompletely specified, encoded non-deterministic automaton.
2. A state transition table for the encoded non-deterministic automaton is created.
3. The QAS algorithm converts this state transition and output table to a fully specified reversible function.

4. The MMD algorithm is applied to generate a permutative quantum circuit that realizes the transition and output functions.
5. The complete state machine is implemented as a quantum circuit using one of the proposed QSM models.

The procedure of QAS can be briefly outlined in three steps:

- Step 1: Assign values to the don't cares outputs, and map the outputs according to the assigned input value. The result must be a n×n reversible function (where n is the number of input and output qubits) , as the algorithm assumes that each single-output Boolean function is a balanced function [25] and no ancilla qubits are added.
- Step 2: Use the MMD algorithm to synthesize the quantum circuit. Thus the result of QAS is both the truth table of the completely specified multi-output n×n function and the sequence of quantum reversible gates such as NOT, CNOT and generalized TOFFOLI.
- Step 3: Compare the cost in terms of the number of gates and keep track of the don't cares assignments with the minimal cost. Backtrack to find K solutions or until no further backtracking is possible. Thus, the algorithm creates several candidate circuits that may be evaluated for cost, quantum layout, etc.

QAS starts with the assumption that a reversible function can be built from an initial, incomplete description without adding ancilla qubits. In case no reversible function can be created for the initial specification, a single ancilla qubit is added, and the algorithm is repeated. This process continues until a fully specified reversible function is created. Therefore, our method also solves the problem of realizing a reversible circuit with the minimum number of qubits for an arbitrary irreversible initial function specification. Several reversible functions can be created for the same initial specification, and the lowest-cost function is picked.

The paper is organized as follows: In Section 2, we provide a brief background on classical and quantum automata, learning from input-output traces, etc. We also provide the basics of the MMD algorithm used by our QAS software. Section 3 reviews the existing logic synthesis approaches for quantum circuits. In Section 4, we present an example of converting a non-deterministic quantum automaton to a deterministic quantum automaton. Then, in Section 5, we present the QAS algorithm in detail. We will also present analytical results and an evaluation of the algorithm's complexity. In Section 6, we illustrate an example where the synthesis with non-minimal number of qubits is explored. Finally, we present our conclusions in Section 7.

2. Background

2.1. Classical Automata

In classical logic, a Deterministic Finite Automaton (DFA) [26] is a computing machine that has a tape head that reads a finite tape with cells that hold symbols from an input alphabet. The tape head is allowed to move only from left to right, changing the state of the machine as it moves right. The machine accepts or rejects a string of symbols that it reads from the tape. The machine is called deterministic because, given its current state and an input symbol, there is exactly one next state that it is allowed to enter, as specified by a state transition table. A Non-Deterministic Finite Automaton (NFA), on the other hand, has a set of states that it is allowed to enter, for a given current state and input symbol. In an NFA, there are non-deterministic transitions in which, for a given input value X_a, the automaton can transit from some internal present state S_i to any of the next states S_j, S_k, S_l, ..., etc. In addition, in the case of a non-deterministic Mealy machine (such as those discussed in our example), the output is also non-deterministic. Non-determinism can occur naturally in many real-life functions. For example, consider a modulo-3 counter which goes from $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$. Let us denote the states by S_0, S_1, and S_2. The binary encoding of this 3-state function will require 2 bits. When we create a state transition table as shown in Table 1, it can be observed that 2-bit encoding creates a fourth state S_3. If S_3 is the initial state of the counter, it should transition to any of the valid states S_0, S_1 or S_2, but not to S_3 itself. During hardware design, the designer, or the design automation tool, may pick one of the possible next states to simplify the combinational circuit of the automaton's realization.

In the case of Machine Learning, the non-determinism of a machine can be treated as a generalization of a don't care (don't know) symbol that is used in logic-based machine learning methods [27–29].

It should be noted that in the literature, there are two ways of understanding the behavior of a non-deterministic automaton. In the first variant, which we follow in this paper, the initial non-deterministic automaton can be converted to a final deterministic automaton that is realized using logic circuits. In the other variant, the final circuit includes probabilistic gates that realize the non-deterministic behavior in hardware. In the case of quantum circuits, this can be done, for instance, using Hadamard gates and measurements [13]. Although, our method in this paper can be extended to this second variant of non-deterministic automata, this topic is not focused on in our paper.

Table 1. State transition table for a simple non-deterministic machine.

Present State (PS)	Next State (NS)
S0	S1
S1	S2
S2	S0
S3	S0, S1, S2

2.2. Learning Using Input-Output Traces

One of the fundamental ideas in machine learning is to make sense of large amounts of data. In automata theory, this is similar to "grammatical inference" [30], where the learning algorithm is given some structured data and is asked to come up with a grammar that explains the data. Once the grammar is inferred using the training data, it can predict responses to, or classify unseen data. One approach is for the learning algorithm to derive a DFA that will provide outputs for the corresponding inputs, consistent with the training data. The data set may take the form of input-output traces, where the trace is a sequence of inputs, $i_1, i_2, i_3, ..., i_n$ and the corresponding sequence of outputs, $o_1, o_2, o_3, ..., o_n$ that the machine must produce when fed with the inputs [20,31]. This method of learning from traces has vast applications in data analysis, pattern recognition, robotics, speech processing, and many other areas.

We will illustrate this learning from input-output traces using a simple example.

Consider a set of input-output traces as shown in Table 2. We can derive a Mealy FSM [32] which produces the output response for the inputs provided by this table, as shown in Figure 1. Here, each arrow indicates a transition from the current state to the next state, and the label indicates the input/output pair for that transition.

Table 2. Example Input-Output Traces.

Trace #	Input	Output
1	000000	000000
2	000001	000001
3	000101	000100
4	010011	010001
5	100010	100000
6	111000	101000
7	110101	100100
8	111110	101010

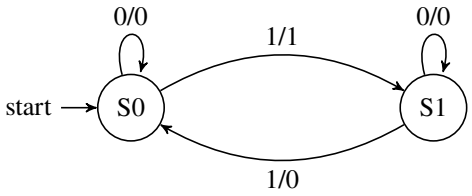


Figure 1. DFA which provides the input-output response shown in Table 2.

Realistic data sets may contain thousands of traces with more than 10 inputs. They will also contain a large amount of "don't cares", which may indicate unknown values or ambiguous values. Deriving state machines for large data sets is a complex task. The FSM may contain hundreds of possible states. State minimization techniques can be applied to reduce the size of the machine before synthesis [32].

One technique to reduce the complexity of deriving the initial specification from traces is to transform it into an NFA. It is well known that NFAs can offer exponential savings in the number of states compared to DFAs [33–35]. We illustrate this using a state machine that can take binary strings of length L as inputs and detect a string that has the k-th from last symbol as a 1. For example, if we assume L=9 and k=3, such a state machine will output a 1 for strings ($i0, i2, i3..i8$) such as 000001000, 010011101, 100001001, etc., and a 0 for strings such as 111110111, 010100110,

etc. A partial input-output trace for this problem is shown in Table 3. Note that, in this example, the traces are of variable length. By analyzing thousands of such traces, it is possible to derive a state machine that produces the desired output pattern.

Table 3. Input-Output Trace for a machine that detects a 1 in the third from last symbol of a string.

Trace #	Input	Output
1	0000	0000
2	1000	0001
3	000101	000000
4	011011	000001
5	011100	000001
6	1110001	0000000
7	1111001	0000001
8	111110111	000000000
9	000001000	000000001
..	.....	.....

The DFA for this problem is shown as a Mealy machine in Figure 2 and the NFA is shown in Figure 3. The dotted arcs in the NFA indicate non-deterministic transitions. It can be seen that the DFA has many more states than the NFA. In the general case, the NFA will need k+2 states and the DFA will need 2^k states [34]. For large traces, it is easier to derive an NFA and then convert it into a DFA. This conversion will result in don't cares being introduced into the output function and state transition function. The traces in our example were fully specified. But in large datasets, don't cares are very likely to show up. For example, a medical dataset may contain some patients whose ages are unknown. Hence, the ability to handle don't cares is an important feature for any synthesis approach.

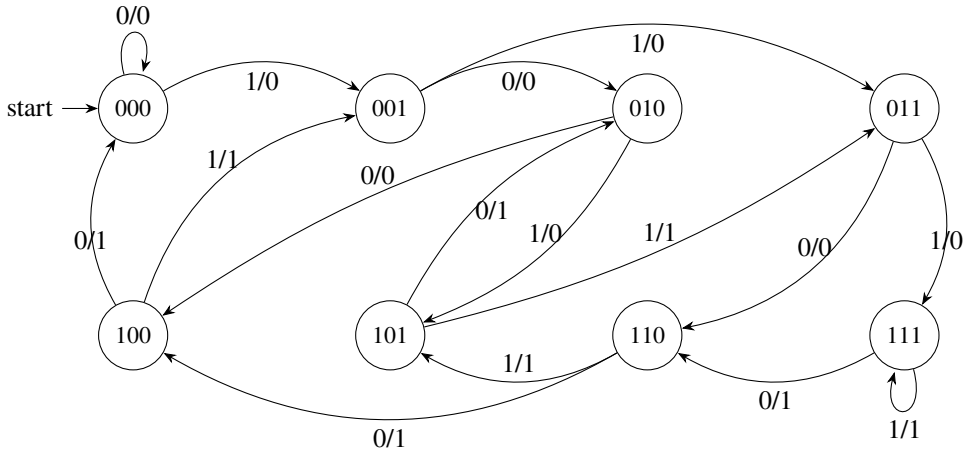


Figure 2. DFA for detecting a 1 in the third from last position of an input string of length L. Each state encodes the last three input bits. On input x, the output is the oldest bit of the window (the 3rd-from-last).

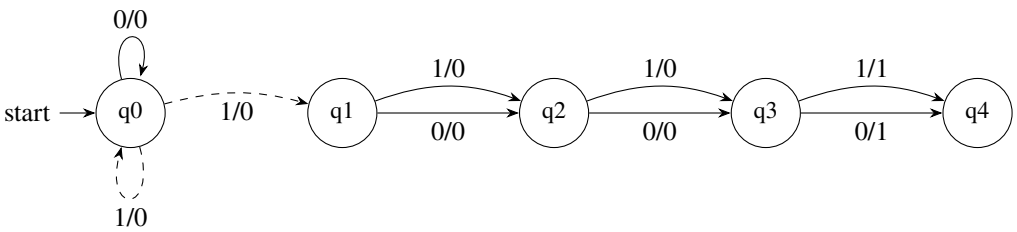


Figure 3. NFA for detecting a 1 in the third from last position of an input string of length L. The dotted arcs indicate non-deterministic transitions.

2.3. Review of Quantum Automata and QSM Models

Quantum Finite Automata (QFA) are the quantum analogs of classical finite automata. The one-way quantum finite automata (1QFA) has a tape head that moves one cell only to the right at each step. On the other hand, two-way quantum finite automata (2QFA) allow the tape head to move right or left, or remain stationary. There are two types of 1QFA: measure-once 1QFA (MO-1QFA) described by Moore and Crutchfield [6], where a single measurement is made after reading the last symbol, and measure-many 1QFA (MM-1QFA) introduced by Kondacs and Watrous [7], where measurement is made after reading each symbol.

Practical, quantum gate and circuit based implementations of Quantum Automata have been presented under the name Quantum State Machines (QSMs) [10,11]. Two models of QSMs are of interest to us.

1. Quantum State Machines with State Retention (QSM-SR)
2. Quantum State Machines with Classical Memory (QSM-CM)

Figure 4 illustrates a QSM-SR. Qubits are prepared to be in an initial state based on a stream of bits received from a classical computer. Some of the qubits represent the state of the system, while some of them are designated as input qubits. The qubits undergo a series of transformations that correspond to the state transition function. The input qubits may be re-initialized before the next pass through the quantum array. The state qubits retain their values and do not get re-initialized. This initialization process is dependent on the underlying quantum technology. It is assumed that the initializer is capable of re-initializing individual qubits without requiring all qubits to be re-initialized. The quantum array represents a series of unitary operations on the qubits. In this model, measurement is performed only once, after many passes through the quantum array.

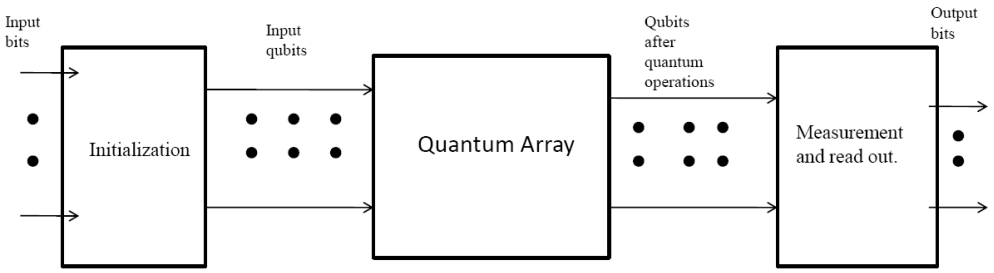


Figure 4. Quantum State Machines with State Retention (QSM-SR). Inputs are classical bits that are used to initialize some qubits. Quantum operations are performed on these qubits by the array. After each pass through the array, the states are retained by the qubits. Thus the state qubits do not get re-initialized after the first pass through the array, while some inputs may get re-initialized before every pass through the array.

Figure 5 illustrates a QSM-CM. After each pass through the array, the state qubits are measured, and their binary values are stored in a classical memory. When inputs change, the input qubits are reinitialized based on the new values. The state qubits are also reinitialized using the bits retrieved from classical memory. A QSM-CM needs more frequent measurements than a QSM-SR, but they are conceptually closer to classical logic and may be easier to realize using current quantum technologies.

The main differences between Quantum Automata described in Section 2.1 and our QSMs are the following:

1. There are two types of Quantum Automata discussed in the literature: Not unrolled (standard) and unrolled. The unrolled Quantum Automata unroll a standard automaton to a certain depth. So they are essentially quantum circuits of a certain depth. These automata accept only strings of limited length. Another variant of these automata has a controller that converts a string of theoretically arbitrary length to a sequence of control pulses for the quantum automaton. Both these types of automata do not work in real-time, they are only language acceptors.
2. Our QSM are not unrolled, they are automata with quantum circuits (binary or multi-valued) for transition and output logic. One variant of QSM (QSM with State Retaining qubits or QSM-SR) uses stateful logic. Another variant (QSM with Classical Memory or QSM-CM) uses classical memory elements like flip-flops (binary or multi-valued) to keep the internal state. In this variant, every output or next state qudit is measured before it is stored in classical memory. After keeping the internal state for some time, the new initialization of the quantum combinational circuits is done, using classical input signals and feedback signals from the classical

memory. From the outside, these machines behave like deterministic or probabilistic classical automata. The QSM-CM model is very similar to classical automata.

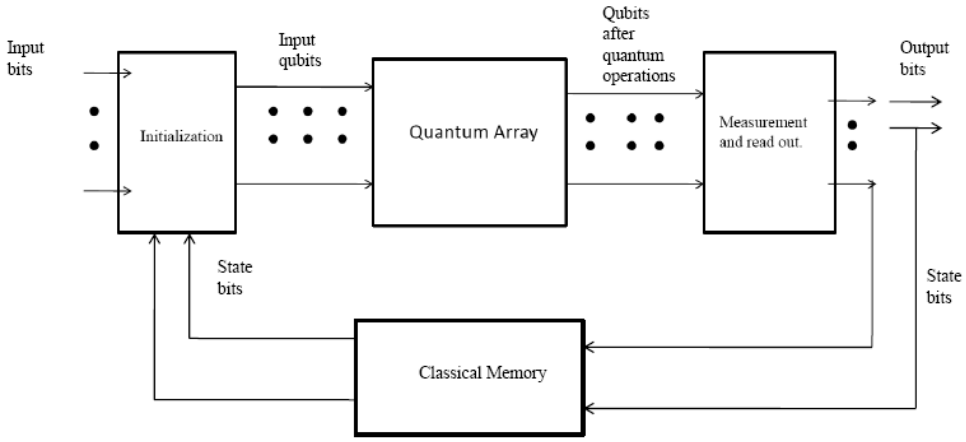


Figure 5. Quantum State Machines with Classical Memory (QSM-CM). Inputs are classical bits that are used to initialize some qubits. Quantum operations are performed on these qubits by the array. After each pass through the array, the states are read out and stored in a classical memory. These state bits are used to re-initialize some qubits before the next pass through the array.

2.4. MMD Algorithm

MMD is an algorithm to synthesize quantum circuits by finding a sequence of Toffoli, Feynman, and NOT gates to transform a given reversible function to the identity function. It does not add ancilla qubits. For simplicity, the term Toffoli gate is used to broadly refer to the standard generalized Toffoli gate, including variants with inverted control inputs, as well as to the Feynman (CNOT) gate, which represents a special case of the Toffoli gate. In Algorithm 1, the basic MMD algorithm, which applies Toffoli and other gates only on the specification's output side, is presented. It takes as its input a reversible function specified as a mapping over 0, 1, ..., $2^n - 1$, where n is the number of qubits.

Algorithm 1 Basic MMD Algorithm

Step 1:
if $f(0) \neq 0$ **then**
    Invert all output bits corresponding to the 1s in $f(0)$.
    Each inversion uses a single Toffoli gate with one control input (TOF1).
    After these inversions, the transformed function f^+ satisfies $f^+(0) = 0$.
end if
Step 2: Iterative Transformation
for $i = 1$ to $2^n - 1$ **do**
    Let f^+ denote the current reversible function.
    if $f^+(i) = i$ **then**
        No transformation (and hence no Toffoli gate) is needed.
    else
        Apply a sequence of gates to obtain a new function f^{++} such that $f^{++}(i) = i$.
    Let p be a bitstring with 1s at positions where the binary expansion of i has a 1 but $f^+(i)$ has a 0. These bits represent positions that must be set to 1 during the transformation $f^+(i) \rightarrow i$.
    Let q be a bitstring with 1s at positions where i has a 0 but $f^+(i)$ has a 1. These bits correspond to positions that must be reset to 0 during the transformation $f^+(i) \rightarrow i$.
    For each bit position j where $p_j = 1$, apply a Toffoli gate with control lines being all the outputs where the binary expansion of i has 1s and the target line being the output in position j.
    For each bit position k where $q_k = 1$, apply a Toffoli gate with control lines being all outputs where the binary expansion of $f^+(i)$ has 1s and the target line being the output in position k.
    end if
end for

In this procedure, once a row of the specification is transformed to the correct value by applying Toffoli gates, it will remain at that value regardless of the transforms required for later rows. The algorithm has a complexity of $n \times 2^n$. After applying the basic algorithm, the MMD package applies template matching to reduce the circuit size. A template consists of a sequence of gates to be matched and a sequence of gates to be substituted, when a match is found.

The algorithm constructs the circuit by applying Toffoli gates on the output side of the reversible specification. Because the specification itself is reversible, it is also possible to construct its inverse, and then select the smaller of the two implementations. An even more effective strategy, however, is to apply the synthesis procedure in both directions simultaneously, allowing gates to be added on either the input or output side as needed.

3. Related Work

The standard approach to hardware synthesis of classical DFA involves the following stages [32]:

1. Description of the automaton as a table, graph, flow chart, or other notation.
2. Minimization of the number of internal states and/or input symbols of the automaton.
3. Encoding (state assignment) for internal states and/or input symbols of the automaton.
4. Realization of the structure and logic of the automaton.

The same design stages as for classical automata have been proposed in a few published papers about quantum automata and particularly about QSM.

1. Description of quantum automata can be found in [8,10,36] and has been summarized in Section 2.3.
2. Minimization of quantum automata can be found in [37]. Two automata over the same input alphabet are considered to be equivalent if they have the same accepting probability for each input. The minimization problem is then reduced to finding the equivalent automaton with the smallest number of states that satisfy the constraints on the given initial automaton with respect to the initial state, acceptance probabilities for each input, etc.
3. Encoding of quantum automata can be found in [38], which applied one-hot encoding to quantum automata, and [10], which used the SIS package [39] from UC Berkeley to study the impact of multiple state encoding schemes like NOVA and Jedi for quantum automata realized with reversible circuits.
4. Realization of the structure and logic of quantum automata can be found in [11,13,40–42]. Traditionally, research on automata has either a mathematical or an engineering approach. The mathematical approach is related to grammars, languages, decisions, and mathematical properties. There are very few papers with an engineering approach related to the structure and practical hardware realization of quantum automata. In [11] and [40], the concept of a Classically Controlled Quantum Computer (CCQC) is introduced, where quantum state machines are implemented using combinational quantum circuits and a classical controller initiates the initialization, quantum transformations, and measurement of qubit state. Genetic algorithms are used to synthesize QFSMs as sequence detectors in [42]. The Chrestenson family of ternary quantum gates is used to realize QFA in the method introduced by [13]. Such a QFA can be combined with a DFA built from quantum reversible circuits to create a more powerful machine, capable of more complex language and pattern recognition.

None of these works discusses the synthesis of non-deterministic quantum automata. The application of quantum automata to machine learning has also not been researched in detail.

The implementation of state transition functions and output functions requires efficient logic synthesis techniques. Since 1994, many synthesis methods have been proposed for reversible and quantum combinational circuits. Some of the reported methods are: transformation-based synthesis [24,43]; using Toffoli and Maitra-like gates to implement an EXOR sum of products (ESOP) and wave cascade [44]; exhaustive enumeration [45]; search using Reed-Muller representation [46]; heuristic methods that iteratively make the function simpler (simplicity is measured in terms of Hamming distance) [47]; spectral methods [48]; template-based approaches [49]; bottom-up construction of circuit implementations of unitary matrices [50–52]; unitary synthesis using machine learning [53]; applying reinforcement learning for automated discovery and application of circuit transformation rules [54,55]; applying simulated annealing for circuit synthesis [56], etc. In terms of applications of these methods to particular types of automata, one notable approach was the method in [57] where Positive Polarity Reed Muller (PPRM) expansion of Boolean functions was used to synthesize synchronous counters with reversible quantum gates.

Most of the reported synthesis methods for quantum circuits are devoted only to the synthesis of completely specified reversible logic functions. A few notable exceptions include heuristic methods to handle don't cares [58]

and the application of Muller Transform to ESOP and similar circuits [59,60]. As of yet, don't cares cannot be handled efficiently by these methods. The algorithm from [59,60] is efficient for single-output functions, but uses several ancilla qubits. If this algorithm is used for multi-output functions, then the number of ancilla qubits can significantly increase. Another notable effort was in [61], which used repeated addition of ancilla qubits to make a function reversible. This method also produces a large number of ancilla qubits. Our approach in QAS targets an efficient circuit design for single-output and multi-output functions, and assumes that the minimum number of ancilla qubits is added. In the best case, our algorithm adds no ancilla qubits at all. In [41], an algorithm for the synthesis of incompletely specified functions using quantum logic gates was introduced, but it was not extended to the concept of quantum automata.

4. Illustration of Converting a Non-deterministic Automaton to a Deterministic Quantum Automaton

In this section, we provide an example for converting a non-deterministic automaton to a deterministic quantum automaton realized using a quantum circuit with no ancilla qubits.

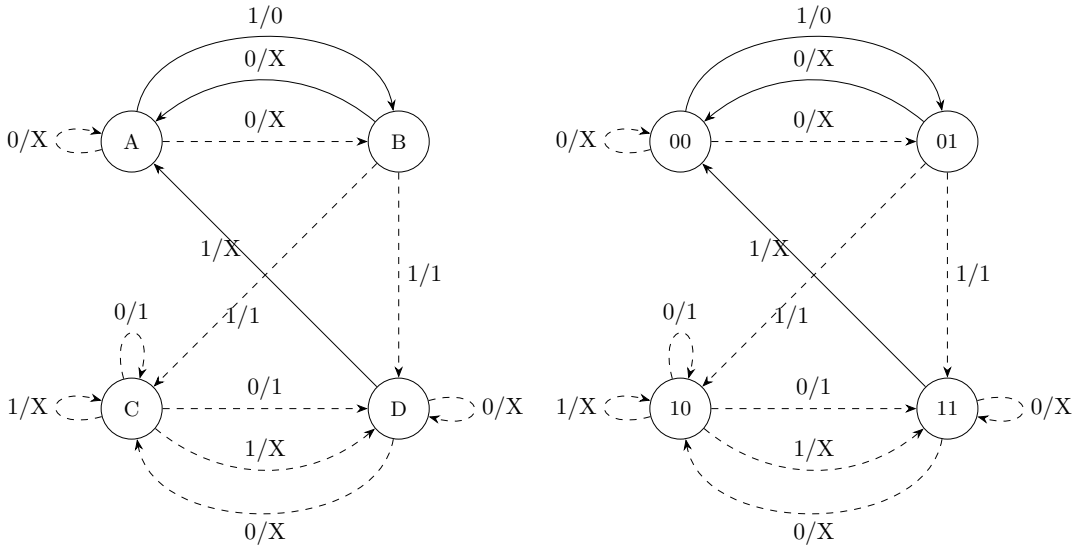
A classical non-deterministic symbolic automaton is presented in Figure 6a. In this example, the automaton is incompletely specified. Symbol X denotes a "don't care" also called "don't know" in Machine Learning, when this graph is created from examples of input-output traces. Notation A/B near the arrow from arbitrary state M to arbitrary state N means that when the machine is in state M and input A is received, the machine generates output B and transitions to state N. Please observe that the dotted arc transitions are non-deterministic because there are multiple outputs or state transitions possible for the same input. For instance, there are two arrows leaving state A with input symbol 0 and corresponding output symbol X. This graph represents a Mealy non-deterministic automaton. By encoding the internal states A, B, C, and D, the automaton is converted to the form shown in Figure 6b. There are many methods to find the minimum encoding, some of them presented in [10,38]. Note that the encoding still preserves the non-deterministic behavior of the automaton.

The standard encoded diagram of the non-deterministic machine from Figure 6b is transformed to the form shown in Figure 7. Here, we have added new states encoded as 0X and 1X, where 0X represents 00 or 01, and 1X represents 10 or 11. These states represent the end states of non-deterministic transitions shown as dotted arcs in Figure 6b. For instance, in Figure 6b, there are two non-deterministic transitions from state 00: 0/X to state 00 and 0/X to state 01. They are combined to form transition 0/X from state 00 to state 0X.

The automaton from Figure 7 can be described using the state transition table from Table 4. This conversion is a standard transformation in circuit design, from a graph of a Mealy machine to a truth table of the transition and output functions. The LHS of the table has a 3-qubit encoding (Q_1Q_2In) consisting of the 2-qubit "present state" (Q_1Q_2) and 1-qubit input (In). The RHS of the table also has a 3-qubit encoding ($Q'_1Q'_2Out$) consisting of the 2-qubit "next state" ($Q'_1Q'_2$) and 1-qubit output (Out). For future use, and consistency with other examples, we use the generic ABC to represent the input qubits and PQR to represent the output qubits. Observe the presence of Xs (don't cares) in the RHS, indicating that the function $(ABC) \xrightarrow{f} (PQR)$ is an incompletely specified function. The realization of a quantum circuit to implement this function will require a synthesis technique that can handle don't cares. Since the MMD algorithm cannot handle don't cares, we first transform the function to a fully specified reversible function using our QAS algorithm.

Starting from the incompletely specified multi-output function in Table 4, QAS creates the truth table of the completely specified reversible function (Table 5), as well as the sequence of reversible gates to realize the transition function that transforms the primary inputs $Q1$, $Q2$, In to the outputs $Q1$, $Q2$, Out. This completely specified reversible function does not require any ancilla qubits and is consistent with the initial incompletely specified Boolean function from Figure 6. For illustration, we also present in Figure 8, the graph of the quantum state machine corresponding to the truth table from Table 5. It can be verified that this machine is deterministic.

The advantage of using a non-deterministic automaton as the first step is that it allows us to reduce the number of states. An exact method to convert an NFA to a DFA using powerset construction exists [26], but the space and time complexity of this conversion makes it prohibitive for large machine learning data sets. The method we have illustrated here may result in the machine being unable to represent some of the traces that it is learning from. This is an acceptable trade-off for machine learning. Our method provides a statistically accurate machine that may not perfectly match every trace in a data set, but can represent the vast majority of the traces.



(a) NFA before state encoding.

(b) NFA after state encoding.

Figure 6. Non-deterministic finite automaton with some outputs as don't cares.

Table 4. State transition table that represents the operation of the non-deterministic automaton in Figure 7 as an incompletely specified 3×3 function.

Present State Inputs Q_1Q_2In (ABC)	Next State Outputs $Q_1'Q_2'Out$ (PQR)
000	0XX
001	010
010	00X
011	1X1
100	1X1
101	1XX
110	1XX
111	00X

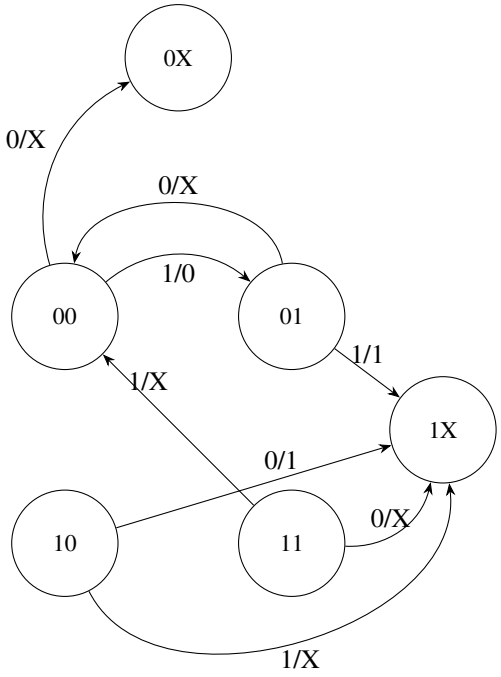


Figure 7. Non-deterministic finite automaton with some outputs and states as don't cares.

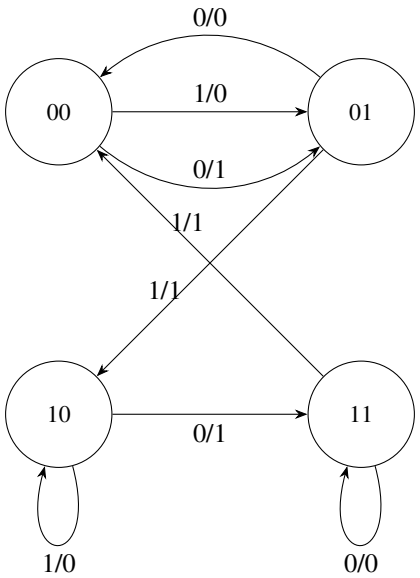


Figure 8. Deterministic automaton with state transition table that is consistent with the non-deterministic automaton in Figure 6.

Table 5. State Transition Table that represents the operation of the automaton in Figure 8 as a completely specified 3×3 function.

Present State Inputs Q_1Q_2In (ABC)	Next State Outputs $Q_1'Q_2'Out$ (PQR)
000	011
001	010
010	000
011	101
100	111
101	100
110	110
111	001

In the next section, we present in detail the steps involved in the QAS algorithm and apply them to the truth table in Table 4.

5. The QAS Algorithm for Multi-output Reversible Function Realization

In the MMD algorithm, the function is represented as a binary truth table. However, in QAS, there is an additional representation for don't cares. The don't cares are represented by the character (-) or (X) as a placeholder for future assignments that take place in the QAS process. This don't cares symbol does not initially have any binary value assigned to it. QAS uses a branching/backtracking approach to assign a binary value to the don't cares in each row of the truth table. The process is continued until all rows have only binary values. The inputs continue to be on the left-hand side and the outputs remain on the right-hand side of the truth table. The function is evaluated from the output to the input since the reversible circuit can be constructed from either end [24]. The MMD algorithm uses only reversible gates like NOT, CNOT, Fredkin, Toffoli, etc. to realize the function. All the gates used are self-inverses. QAS is designed as a pre-processor module that is completely separate from the MMD code. The pre-processor generates an output in the standard binary format used by MMD and passes it as input data to the MMD code (see Figure 9).

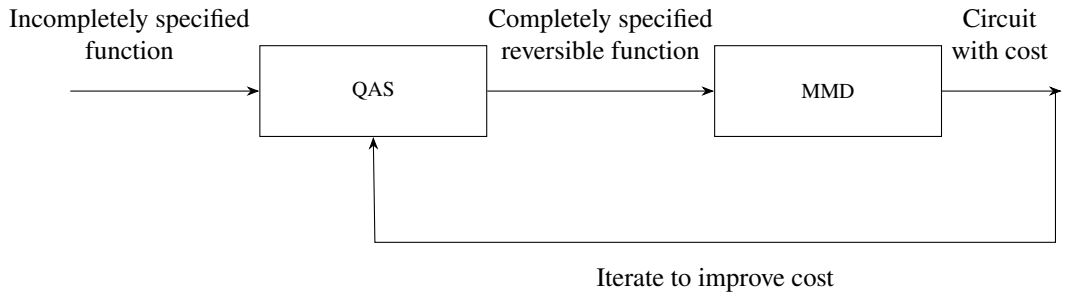


Figure 9. Black box model of the QAS system. The input and output of QAS are in the form of truth tables, incompletely specified and completely specified, respectively.

The original MMD algorithm works by taking a given binary truth table and for each row in the truth table, adding NOT, CNOT, Toffoli and Fredkin gates one at a time such that, finally, the output equals the input for that row. For each truth table row, MMD makes sure that the added gates do not affect the previous rows' outputs. This procedure continues until it has iterated through all rows in the truth table.

5.1. QAS Method for Constructing a Reversible Function

In this section, we provide a step-by-step procedure to convert an irreversible or incompletely specified function into a reversible function. It is also summarized in Algorithm 2.

- Step 1. For a function f with n input variables, write all the outputs as a binary truth table in the natural order of the 2^n input minterms (see Table 4).
- Step 2. If the function has the same completely specified output for more than one minterm (eg: if $f(011)$ = $f(001)$ = 001), then add a column of don't cares to the most significant bit position of the outputs.
- Step 3. Going from top to bottom in the truth table, assign values to the don't care bits in the first incompletely specified minterm in the truth table. The number of possible values for 'm' don't care bits in an 'n' bit minterm is 2^m, ranging from all zeros to all ones. Initially, all zeros are assigned to the don't care bits. When needed, increment the value of the don't care bits by one.
- Step 4. Once a value is assigned to all the don't care bits of a minterm, go back and check all previous minterms to ensure that the assigned don't care combination is unique (see definition of unique assignment below).
- Step 5. If unique, go to the next minterm and check for uniqueness. Wherever don't care bits are encountered, they are first assigned certain values before the uniqueness is checked.
- Step 6. If not unique, increment the value of the don't care bits by 1. For example, initially, if all the don't cares were assigned zeros, the least significant don't care bit is now set to 1 and the others continue to be zeros. If it still does not give a unique value, then increment it again. Keep incrementing until a unique value is found.
- Step 7. If none of the 2^m bit combinations in a 'n' bit value with 'm' number of don't cares gives a unique value, then add a column of don't cares to the most significant bit position of the outputs and try again from Step 3.
- Step 8. If at any point, it was found that a previously assigned don't care combination conflicts with a "no-other-choice" combination (see definition of "no-other-choice" assignment below) that comes down the truth table at a later instance, backtrack to the conflicting assignment and re-assign the don't cares with another combination. Start the process again from Step 5.
- Step 9. When all the minterms have been assigned values such that each value is unique, and in adherence with the original incompletely specified function, stop the process.

Definition 1. *Uniqueness of don't care assignment: A unique don't care assignment is any assignment of output don't cares for an incompletely specified function $f : X \rightarrow Y$ that maps each element $x \in X$ to a unique output $y \in Y$.*

Definition 2. *"No-other-choice" assignment: If the 'm' don't cares in an 'n' bit output value are such that only one of the 2^m bit combinations will give a unique, non-repeated value to the n bit combination (i.e., all other $2^m - 1$ combinations give an existing completely specified output), it is considered a "no-other-choice" assignment.*

Example 1. *Consider the transition function $f(ABC) = PQR$. If $f(110) = 001$ and $f(100) = 00X$, the only possible value that can be assigned to the don't care X is 0. Otherwise, the function f will not be reversible, since the input $\rightarrow$ output mapping is not unique. Note that if $f(110) = 00X$ and $f(100) = 00X$, then the assignment is not "no-other-choice" because $f(100)$ could also take the value 000 or 001 depending on whether $f(110)$ is 000 or 001.*

Algorithm 2 Conversion of an irreversible or incompletely specified function to a reversible function

Input: Truth table of a function f with n input variables. The function may be incompletely specified or irreversible.
Output: Fully specified reversible function f'

1: Write all outputs of f as a binary truth table.
2: If f has identical, completely specified outputs for multiple minterms, add a column of don't cares to the most significant bit position of the outputs.
3: For each minterm i ($0 \leq i \leq 2^n - 1$, assign values to the don't cares (if there are m don't care bits in the output, there are 2^m possible combinations). Initially, assign all zeros.
4: Check all previous minterms to ensure uniqueness.
if unique **then**
    Proceed to the next minterm.
else
    Increment the currently assigned value of the don't care bits and recheck.
end if
5: If none of the 2^m combinations yield uniqueness, add another column of don't cares and restart from Step 3.
6: If a conflict arises with a later "no-other-choice" assignment, backtrack and reassign.
7: Stop when all minterms have unique values consistent with the original function.

5.2. Formalisms and examples for QAS operation

Let $f(A, B, C)$ be defined by Table 4.

The task of the QAS algorithm is to convert outputs PQR to completely specified binary values so that the function f becomes reversible, i.e. there is a one-to-one mapping between inputs and outputs. Recall that these properties result from our assumption of not adding any ancilla qubits to the realization of QSM, unless necessary.

The steps of our algorithm are illustrated in the Figures 10–17. In each step (shown as a column S* in the figures), binary values are assigned to the don't cares in one row of the truth table. A black-colored square indicates a newly assigned bit combination that is invalid i.e., the bit combination appears previously in the truth table and will break the uniqueness of the outputs. Gray represents an assignment that is temporarily valid (i.e. it leads to an output value that is unique till now, but could turn out later to be invalid as QAS moves further down the table). Observe that similar to the MMD algorithm, the output vectors in Figure 10 become fixed from top to bottom, and a once-fixed output value is never modified. In each S* column, every row above the gray-colored square is fixed.

S1	S2	S3	S4	S5
0XX	000	000	000	000
010	010	010	010	010
00X	00X	000	001	001
1X1	1X1	1X1	1X1	101
1X1	1X1	1X1	1X1	1X1
1XX	1XX	1XX	1XX	1XX
1XX	1XX	1XX	1XX	1XX
00X	00X	00X	00X	00X

Figure 10. QAS outputs, steps S1–S5.

Column S1 is just the outputs (PQR) column from Table 4. For the first pass of column S1, QAS finds the first output that contains don't cares (row 0 in this case) and assigns binary values to the two don't care symbols, as detailed in Section 5.1. It checks this new assignment and finds it temporarily valid, as illustrated by the gray square in column S2 of Figure 10. QAS then continues down the truth table outputs until another assignment needs to be made. When it finds the next output that has don't cares (row 2), it applies the same steps that it did for row 0. Column S3 in Figure 10 has a black-colored square, which indicates that the bit combination 000 has already been used. So, it changes the assignment to 001 in S4. It proceeds with the same approach for the next rows.

S6	S7	S8	S9	S10
000	000	000	000	000
010	010	010	010	010
001	001	001	001	001
101	101	101	101	101
101	111	111	111	111
1XX	1XX	100	100	100
1XX	1XX	1XX	100	101
00X	00X	00X	00X	00X

Figure 11. QAS outputs, steps S6–S10.

S11	S12	S13	S14	S15
000	000	000	000	001
010	010	010	010	010
001	001	001	000	00X
101	101	101	1X1	1X1
111	111	111	1X1	1X1
100	100	100	1XX	1XX
110	110	110	1XX	1XX
00X	000	001	00X	00X

Figure 12. QAS outputs, steps S11-S15.

Partial assignment S13 in Figure 12 proves that one of the earlier assignments was invalid because there are no possible assignments to 00X in row 7 that could lead to unique Input→Output mapping. Thus, the first instance of backtracking occurs at S14. The backtracking to the possible invalid assignment (row 2) occurs and the re-assignment is performed as shown in S14. When QAS backtracks, it references the data structure for the original output values before they had been assigned binary values. It then restores the original output values to the last output that had don't cares. QAS then tries a new combination. If that new combination also leads to an invalid assignment, QAS backtracks to an earlier point in the process. In S14, it exhausts all possible bit combinations for row 2 and backtracks to an earlier possible invalid assignment as shown in S15 (Figure 12).

In S26 (Figure 15), QAS backtracks again and re-assigns a value to the minterm in row 0.

S16	S17	S18	S19	S20
001	001	001	001	001
010	010	010	010	010
000	000	000	000	000
1X1	101	101	101	101
1X1	1X1	101	111	111
1XX	1XX	1XX	1XX	100
1XX	1XX	1XX	1XX	1XX
00X	00X	00X	00X	00X

Figure 13. QAS outputs, steps S16-S20.

S21	S22	S23	S24
001	001	001	001
010	010	010	010
000	000	000	000
101	101	101	101
111	111	111	111
100	100	100	100
100	101	110	110
00X	00X	00X	000

Figure 14. QAS outputs, steps S21-S24.

S25	S26	S27	S28	S29
001	010	010	011	011
010	010	010	010	010
000	00X	00X	00X	000
101	1X1	1X1	1X1	1X1
111	1X1	1X1	1X1	1X1
100	1XX	1XX	1XX	1XX
110	1XX	1XX	1XX	1XX
001	00X	00X	00X	00X

Figure 15. QAS outputs, steps S25-S29.

S30	S31	S32	S33	S34
011	011	011	011	011
010	010	010	010	010
000	000	000	000	000
101	101	101	101	101
1X1	101	111	111	111
1XX	1XX	1XX	100	100
1XX	1XX	1XX	1XX	100
00X	00X	00X	00X	00X

Figure 16. QAS outputs, steps S30-S34.

S35	S36	S37	S38
011	011	011	011
010	010	010	010
000	000	000	000
101	101	101	101
111	111	111	111
100	100	100	100
101	110	110	110
00X	00X	000	001

Figure 17. QAS outputs, steps S35-S38.

At the end of Stage 38 (Figure 17), a completely specified reversible function $f(A,B,C)$ is found. This function is rewritten as a truth table in Table 6. This table is then provided as the input to the MMD algorithm to synthesize the network shown in Figure 18.

Table 6. Original function versus final function $f(A,B,C)$ after QAS application.

Inputs (ABC)	Original Outputs (PQR)	Final Outputs (PQR)
000	0XX	011
001	010	010
010	00X	000
011	1X1	101
100	1X1	111
101	1XX	100
110	1XX	110
111	00X	001

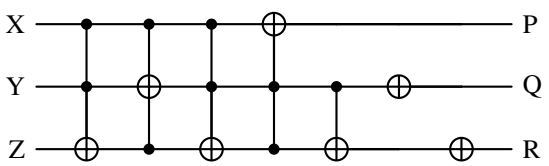


Figure 18. Final Circuit for realization of transition and output functions for automaton from Figure 6, assuming no ancilla qubits.

The full system for realizing the automaton as a QSM with State Retaining qubits (QSM-SR) was shown in Figure 4. The initialization step involves accepting a bit stream from a classical computer and preparing qubits in the desired initial state. The quantum array in Figure 4 represents a series of permutative operations on the qubits, as shown in Figure 18. In this model, the qubits that represent the state of the system (i.e., $Q1$ and $Q2$) undergo a sequence of transformations and retain their final values (i.e., $Q1'$ and $Q2'$) after completing a pass through the array. When any new inputs (In) are to be applied, the initializer preserves the values of the state qubits from the previous pass.

The same automaton can also be implemented as a QSM with classical memory (QSM-CM) as shown in Figure 5. The initialization step is the same as described in the QSM-SR model. After each pass through the quantum array, the state qubits (i.e., $Q1$ and $Q2$) are measured and their binary values are stored in a classical memory (latches, flip flops, etc.). When a new input (In) arrives, the initializer uses the input bit stream to initialize the corresponding qubit. The state bits stored in the classical memory are used to initialize the state qubits. Classical memory can be clocked (like a synchronous register), making the whole model synchronous.

5.3. Experimental Results of QAS

For testing, QAS was applied to 6-bit and 9-bit incompletely specified Boolean functions created from reversible (one-to-one) functions. The published results for the original MMD algorithm were for a maximum of 9 input variables. So, we picked 6 and 9 bits as representative sizes for the max and mid-sized functions that the software can support. The original functions used for MMD were reversible, while QAS is broader and should be tested with incompletely specified or irreversible functions. Some of the new benchmark functions for testing were obtained from Dmitri Maslov's Reversible Logic Synthesis Benchmarks Webpage [62]. The other incompletely specified functions were generated using an "Incompletely Specified Function Generator" program developed by our group. This program creates incompletely specified functions of any number of variables using random number generators. Multiple versions of these functions were tested. The cases included 20%, 40%, 60%, and 80% don't cares in the function outputs. When generating a function with a higher percentage of don't cares, the already existing don't cares were preserved. The successive increase in the number of don't cares in the given function was expected to allow QAS to find solutions with smaller costs. This expectation was tested by analyzing the MMD cost that was produced from each of the runs.

For each benchmark function and each percentage of don't cares, QAS produced up to a maximum of 25 completely specified functions. Out of those 25 completely specified functions, the completely specified function with the lowest MMD cost was selected. Figure 19 and Figure 20 show the trends found in this testing for 6-qubit and 9-qubit functions, respectively. The graphs of different functions are indicated by different colors as well as line patterns in the figures. The cost of synthesizing a function decreases as the percentage of don't cares in the function increases.

The goal was to see if QAS could allow MMD to handle incompletely specified functions. Also, these tests were an exploration into whether the MMD cost of the incompletely specified function could be improved upon through multiple iterations of QAS. We were successful in both of these goals. The current implementation of QAS doesn't guarantee a solution with the absolute minimum cost. There are several reasons for this:

1. We make an attempt to minimize the number of ancilla qubits. This is based on the current technological landscape, which imposes restrictions on how many qubits are available. In some functions, addition of more ancilla qubits can help reduce the cost of the quantum circuit. We provide an example in Section 6.
2. We limit the number of iterations performed by QAS to a configurable limit N. This is to reduce the computation time. The optimal solution may require all possible don't care assignments to be evaluated.
3. For a given completely specified function, MMD does not guarantee a circuit with the absolute minimal cost. This places limits on QAS's ability to find the lowest cost circuit.

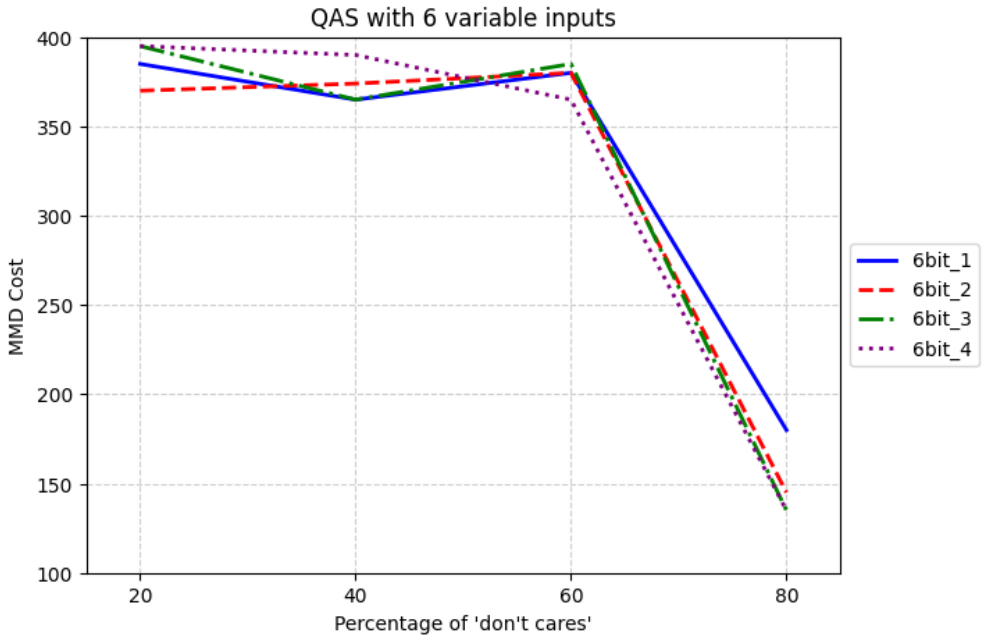


Figure 19. QAS trend of MMD cost versus percentage of don't cares for four 6-qubit benchmark functions.

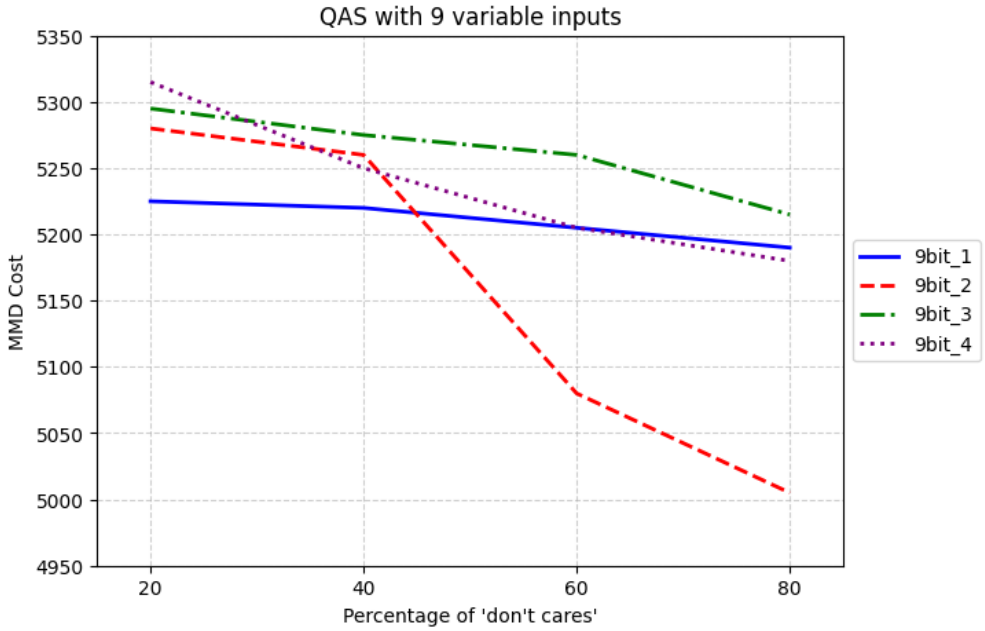


Figure 20. QAS trend of MMD cost versus percentage of don't cares for four 9-qubit benchmark functions.

The cost of a solution generated by QAS was expected to decrease with an increase in the percentage of don't cares in the input. This expectation was based on the greater flexibility in bit assignments associated with more don't cares. However, the graphs do not always show a monotonically decreasing trend because in many cases, the solution with lower cost was beyond reach since we have set limits on the number of iterations. In principle, QAS can operate not only with MMD but with any minimizer of reversible functions.

The algorithms from [60] and [59] require a large number of ancilla qubits for multi-output functions. In contrast, our approach tries to add the minimum number of qubits required to make the function reversible. In the best case, our algorithm adds no ancilla qubits at all. Considering the limited number of qubits available in state-of-the-art quantum computing systems today [63], this is a significant advantage.

5.4. Complexity of QAS

For the worst possible circumstances, the complexity of the QAS algorithm can run into O(2^M), where M is the total number of don't cares in the truth table of the function (It is assumed that $M >> n$, where n is the number of qubits). QAS will run much faster as the number of unassigned don't cares decreases over each iteration.

5.5. Performance limitations of QAS

The current version of QAS has been tested for a maximum of 9 qubits. Functions with more input qubits will be slower to evaluate. This is not a severe limitation since larger functions can be easily decomposed into smaller functions that can be realized with QAS. The original MMD algorithm was also limited to 8 or 9 inputs and outputs.

In the worst case, finding even a non-optimal solution may take a huge amount of iterations, because of the possibility of a large number of collisions and a proportionate increase in backtracking. QAS opts for a trade-off by adding a column of don't cares if the number of iterations exceeds a configurable maximum limit. In other words, if for n-qubit inputs/outputs, the program cannot find a solution even with many back-trackings, it will add a single ancilla qubit and repeat the program for n+1 qubit inputs/outputs. This addition of an ancilla bit guarantees a quick solution.

Since MMD uses truth tables to represent functions, the software's performance is slowed down by the size of data that the search method encounters. Further work needs to be done to speed up the code execution by better representation of data. The scope of this project did not address this optimization problem.

6. Synthesis with Non-Minimal Number of Qubits

It can be observed that when the assumption of the minimal number of qubits (i.e., no ancilla added) is not made, the circuit that realizes the function from Table 4 can be realized as shown in Figure 22 using the excitation functions derived in Figure 21. This implementation requires only 3 gates and is cheaper than the implementation in Figure 18, which requires 8 gates. This would suggest that the assumption of having a minimal number of qubits is not always optimal. However, there are applications for which, taking into consideration the layout of the quantum device, the assumption of no ancilla qubits is the better choice for the design of the entire circuit composed of many reversible and quantum blocks. Besides, the problem of minimizing the number of ancilla qubits is not yet solved in the literature and only heuristic approaches are known. Let us assume that our quantum computer has 20 qubits. QAS may be able to synthesize the circuit with 20 qubits (analogous to the example in Figure 18), which can be mapped to this particular quantum computer. In contrast, if one would use the method with non-minimal number of qubits, we would not be able to map the circuit (analogous to the example in Figure 22) to this quantum computer.

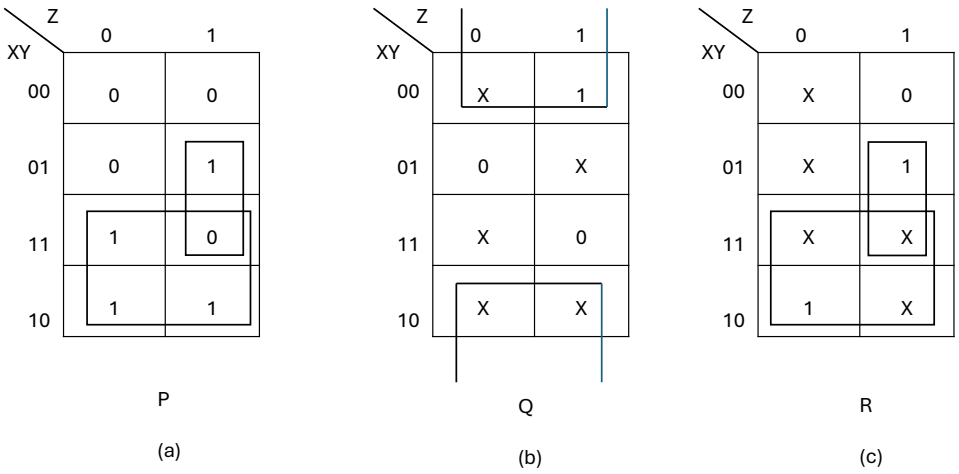


Figure 21. Karnaugh maps that represent the transition and output functions for the state machine in Table 4. (a) $P = X \oplus YZ$ (b) $Q = Y'$ and (c) $R = X \oplus YZ$

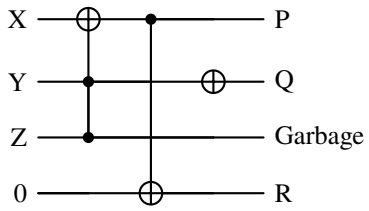


Figure 22. Circuit to realize transition and output functions for automaton from Figure 6 assuming one ancilla qubit.

7. Conclusions

Creating deterministic or non-deterministic state machines from input-output traces is an approach used in both machine learning and classical circuit design. This paper, for the first time, presents a complete methodology to build such quantum state machines. We also solve the problem of implementing incompletely specified boolean functions using quantum logic circuits. Such functions are very common in machine learning and circuit design. Although we outline the entire methodology in this paper, we focus on the conversion of traces to an incomplete truth table of the transition and output functions of a quantum state machine. The stages of state minimization and state encoding, as well as other methods to convert non-deterministic state machines to deterministic state machines, are not presented in this paper. Due to current technological constraints, we begin by assuming a machine with no ancilla qubits. If such a machine cannot be realized, we allow for the use of one ancilla qubit. If that too is not feasible, the method incrementally increases the number of ancilla qubits, adding them one at a time. Tests have confirmed the expectation of improved MMD costs when the percentage of don't cares included in a given function increases for the 9-qubit and 6-qubit functions.

Author Contributions
Manjith Kumar: Conceptualization, Software, Data Collection and Analysis, Writing—Original Draft. Marek Perkowski: Supervision, Guidance, Writing—Review and Editing. Both authors have read and agreed to the published version of the manuscript.

Funding
This research received no external funding.

Conflicts of Interest Statement
The authors declare no conflicts of interest.

Data Availability Statement
All data supporting reported results can be found in this paper.

References

1. Richard P Feynman (1986). Quantum mechanical computers. *Found. Phys.*, 16(6):507–532.
2. Eleanor Rieffel and Wolfgang Polak (2000). An introduction to quantum computing for non-physicists. *ACM Computing Surveys (CSUR)*, 32(3):300–335.
3. Michael A Nielsen and Isaac L Chuang (2001). *Quantum computation and quantum information*, volume 2. Cambridge university press Cambridge.
4. Asher Peres (1985). Reversible logic and quantum computers. *Physical review A*, 32(6):3266.
5. Charles H Bennett (1973). Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532.
6. Cristopher Moore and James P Crutchfield (2000). Quantum automata and quantum grammars. *Theoretical Computer Science*, 237(1-2):275–306.
7. Attila Kondacs and John Watrous (1997). On the power of quantum finite state automata. In *Proceedings 38th annual symposium on foundations of computer science*, pages 66–75. IEEE.
8. AC Cem Say and Abuzer Yakarɣlmaz (2014). Quantum finite automata: A modern introduction. In *Computing with New Resources: Essays Dedicated to Jozef Gruska on the Occasion of His 80th Birthday*, pages 208–222. Springer.
9. Jozef Gruska, Daowen Qiu, and Shenggen Zheng (2015). Potential of quantum finite automata with exact acceptance. *International Journal of Foundations of Computer Science*, 26(03):381–398.

10. Manjith Kumar, Samy Boshra-riad, Yasodha Nachimuthu, and Marek A Perkowski (2010). Comparison of state assignment methods for "quantum circuit" model of permutative quantum state machines. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE. 2010.
11. Martin Lukac and Marek Perkowski (2009). Quantum finite state machines as sequential quantum circuits. In *2009 39th International Symposium on Multiple-Valued Logic*, pages 92–97. IEEE.
12. Sebastian Horvat, Xiaoqin Gao, and Borivoje Dakić (2022). Universal quantum computation via quantum controlled classical operations. *Journal of Physics A: Mathematical and Theoretical*, 55(7):075301.
13. Yuchen Huang, Marek Perkowski, Xiaoyu Song, and John M Acken (2025). Quantum finite automaton using ternary rotation quantum gates and chrestenson family quantum gates. *Quantum Information and Computation*, 25(1):57.
14. Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J Wood, Jake Lishman, Julien Gacon, *et al.* (2024). Quantum computing with qiskit. *arXiv preprint arXiv:2405.08810*.
15. Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron (2013). Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342.
16. Lov K Grover (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219.
17. Edward Farhi and Sam Gutmann (1998). Quantum computation and decision trees. *Physical Review A*, 58(2):915.
18. Ali Al-Bayaty and Marek Perkowski (2024). Bht-qaoa: The generalization of quantum approximate optimization algorithm to solve arbitrary boolean problems as hamiltonians. *Entropy*, 26(10):843.
19. Himanshu Thapliyal and Nagarajan Ranganathan (2010). Design of reversible sequential circuits optimizing quantum cost, delay, and garbage outputs. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 6(4):1–31.
20. Georgios Giantamidis, Stavros Tripakis, and Stylianos Basagiannis (2021). Learning moore machines from input–output traces. *International Journal on Software Tools for Technology Transfer*, 23(1):1–29.
21. Martin Lukac and Marek Perkowski (2007). Inductive learning of quantum behaviors. *Facta universitatis-series: Electronics and Energetics*, 20(3):561–586.
22. Raymond Edward Miller (1965). Switching theory. volume 1- combinational circuits(book on switching theory covering circuit synthesis and analysis, boolean algebra, functional theory, relay type network, etc). *John Wiley and Sons, Inc., New York, 351 P*.
23. Andris Ambainis and John Watrous (2002). Two-way finite automata with quantum and classical states. *Theoretical Computer Science*, 287(1):299–311.
24. D Michael Miller, Dmitri Maslov, and Gerhard W Dueck (2003). A transformation based algorithm for reversible logic synthesis. In *Proceedings of the 40th annual Design Automation Conference*, pages 318–323.
25. K Chakrabarty and JP Hayes. Balanced boolean functions (1998). *IEE Proceedings-Computers and Digital Techniques*, 145(1):52–62.
26. JE Hopcroft (2001). Introduction to automata theory, languages, and computation.
27. Ryszard S Michalski and Kenneth A Kaufman (2001). The aq19 system for machine learning and pattern discovery: A general description and user's guide. Technical report.
28. Marek Perkowski, Tim Ross, Dave Gadd, Jeffrey A Goldman, and Ning Song (1995). Application of esop minimization in machine learning and knowledge discovery. In *Proc. Reed Muller*, volume 95.
29. Dong Wang, Yiwei Li, Edison Tsai, Xiaoyu Song, Marek Perkowski, and Han Li (2020). Boolean function decomposition based on grover algorithm and its simulation using quantum language quipper. In *International Conference on Artificial Intelligence and Security*, pages 582–592. Springer.
30. Colin De La Higuera (2005). A bibliographical study of grammatical inference. *Pattern recognition*, 38(9):1332–1348.
31. Dana Angluin (1987). Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106.
32. John F Wakerly (2008). *Digital Design: Principles and Practices, 4/E*. Pearson Education India.
33. Oleg B Lupanov (1963). A comparison of two types of finite automata. *Problemy kibernetiki*, 9:321–326.
34. Frank R Moore (1971). On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Transactions on computers*, 100(10):1211–1214.

35. Albert R Meyer and Michael J Fischer (1971). Economy of description by automata, grammars, and formal systems. In *12th annual symposium on switching and automata theory (swat 1971)*, pages 188–191. IEEE Computer Society.
36. Amandeep Singh Bhatia and Ajay Kumar (2019). Quantum finite automata: survey, status and research directions. *arXiv preprint arXiv:1901.07992*.
37. Paulo Mateus, Daowen Qiu, and Lvzhou Li (2012). On the complexity of minimizing probabilistic and quantum automata. *Information and Computation*, 218:36–53.
38. Yuchen Huang and Marek Perkowski (2023). One hot encoding synthesis of quantum automata from flowcharts. *Journal of Quantum Information Science*, 13(3):156–176.
39. E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha *et al.* (1992). Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41.
40. Martin Lukac, Michitaka Kameyama, and Marek A Perkowski (2013). Quantum finite state machines-a circuit based approach. *Int. J. Unconv. Comput.*, 9(3-4):267–301.
41. Manjith Kumar, Bala Iyer, Natalie Metzger, Ying Wang, and Marek Perkowski (2007). Realization of incompletely specified functions in minimized reversible cascades. *Proceedings of Reed-Muller 2007*, pages 59–65.
42. Martin Lukac and Marek Perkowski (2010). Evolutionary logic synthesis of quantum finite state machines for sequence detection. *New Achievements in Evolutionary Computation*, page 77.
43. Kazuo Iwama, Yahiko Kambayashi, and Shigeru Yamashita (2002). Transformation rules for designing cnot-based quantum circuits. In *Proceedings of the 39th annual Design Automation Conference*, pages 419–424.
44. Alan Mishchenko and Marek Perkowski (2002). Logic synthesis of reversible wave cascades.
45. Vivek V Shende, Aditya K Prasad, Igor L Markov, and John P Hayes (2002). Reversible logic circuit synthesis. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 353–360.
46. Abhinav Agrawal and Niraj K Jha (2004). Synthesis of reversible logic. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 1384–1385. IEEE.
47. D Maslov and GW Dueck (2003). Reversible function synthesis with minimum garbage outputs. In *International Symposium on Representations and Methodology of Future Computing Technologies (RM2003), Trier, Germany*.
48. D Michael Miller and Gerhard W Dueck (2003). Spectral techniques for reversible logic synthesis. In *6th International Symposium on Representations and Methodology of Future Computing Technologies*, pages 56–62.
49. Dmitri Maslov, Christina Young, D Michael Miller, and Gerhard W Dueck (2005). Quantum circuit simplification using templates. In *Design, Automation and Test in Europe*, pages 1208–1213. IEEE.
50. Marc Grau Davis, Ethan Smith, Ana Tudor, Koushik Sen, Irfan Siddiqi, and Costin Iancu (2019). Heuristics for quantum compiling with a continuous gate set. *arXiv preprint arXiv:1912.02727*.
51. Ethan Smith, Marc Grau Davis, Jeffrey Larson, Ed Younis, Lindsay Bassman Oftelie, Wim Lavrijsen, and Costin Iancu (2003). Leap: Scaling numerical optimization based synthesis using an incremental approach. *ACM Transactions on Quantum Computing*, 4(1):1–23.
52. Ed Younis, Koushik Sen, Katherine Yelick, and Costin Iancu (2020). Qfast: Quantum synthesis using a hierarchical continuous circuit space. *arXiv preprint arXiv:2003.04462*.
53. Mathias Weiden, Ed Younis, Justin Kalloor, John Kubiatowicz, and Costin Iancu (2023). Improving quantum circuit synthesis with machine learning. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 1–11. IEEE.
54. Thomas Fösel, Murphy Yuezhen Niu, Florian Marquardt, and Li Li (2021). Quantum circuit optimization with deep reinforcement learning. *arXiv preprint arXiv:2103.07585*.
55. David Kremer, Victor Villar, Hanhee Paik, Ivan Duran, Ismael Faro, and Juan Cruz-Benito (2024). Practical and efficient quantum circuit synthesis and transpiling with reinforcement learning. *arXiv preprint arXiv:2405.13196*.
56. Anouk Paradis, Jasper Dekoninck, Benjamin Bichsel, and Martin Vechev (2024). Synthetiq: Fast and versatile quantum circuit synthesis. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):55–82.
57. Mozammel HA Khan and Marek Perkowski (2011). Synthesis of reversible synchronous counters. In *2011 41st IEEE International Symposium on Multiple-Valued Logic*, pages 242–247. IEEE.
58. Majid Mohammadi and Mohammad Eshghi (2008). Heuristic methods to use don't cares in automated design of reversible and quantum logic circuits. *Quantum Information Processing*, 7:175–192.

59. Marek Perkowski, Robert Fiszer, Paweł Kerntopf, and Martin Lukac (2012). Synthesis of reversible circuits with pse gates. In *21st International Workshop on Post-Binary ULSI Systems*. University of Victoria.
60. Robert Adrian Fiszer (2014). Synthesis of irreversible incompletely specified multi-output functions to reversible eosops circuits with pse gates. Master's thesis, Portland State University.
61. Mathias Soeken, Martin Roetteler, Nathan Wiebe, and Giovanni De Micheli (2017). Logic synthesis for quantum computing. *arXiv preprint arXiv:1706.02721*.
62. Dmitri Maslov (2005). Reversible logic synthesis benchmarks page. *https://reversiblebenchmarks.github.io/*.
63. Barry C Sanders (2025). Superconducting quantum computing beyond 100 qubits. *Physics*, 18:45.