


QSeqSim: A Symbolic Simulator for Qiskit While Loops Using Sequential Quantum Circuits (Long Tool Paper)

Zihao Li[1,2], Ji Guan[1(✉)], and Mingsheng Ying[3]

[1] Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{lizh,guanj}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing 101408, China
[3] Centre for Quantum Software and Information, University of Technology Sydney, Ultimo, NSW 2007, Australia
Mingsheng.Ying@uts.edu.au

Abstract. We present a tool *QSeqSim*, a Qiskit-integrated symbolic backend that fills the current gap of having no Qiskit-native support for simulating `while`-loop quantum programs and their induced sequential quantum circuits. *QSeqSim* takes Qiskit `QuantumCircuit` objects, translates them into OpenQASM 3 code, and organises the resulting program into a combination of combinational, dynamic, and sequential circuits, thereby assigning `while`-loops a precise sequential circuit semantics with explicit internal and external qubits. Building on this semantics, *QSeqSim* adopts a Binary Decision Diagram (BDD)-based symbolic representation and integrates weighted model counting to compute measurement probabilities efficiently by exploiting sharing in structured and sparse BDDs. On top of this Boolean backbone, it introduces dedicated symbolic operators for quantum state composition and state retention, thereby enabling efficient symbolic execution of sequential quantum circuits. Our experiments demonstrate that *QSeqSim* scales to substantial `while`-induced sequential circuits; in particular, in the quantum random walk benchmark we successfully simulate circuits with over 1000 qubits for more than 10 loop iterations.

QSeqSim is available at https://github.com/Veri-Q/QSeqSim.

Keywords: Quantum circuit simulation · Symbolic simulation · Sequential quantum circuits · Boolean operations · Formal methods

1 Introduction

The formal methods community has devoted substantial effort to the analysis and verification of quantum circuits and programs, both at the foundational level and in the form of automated tools. There is now a considerable body of work on quantum Hoare logic [26,40,44,48], as well as on other quantum program logics, deductive verification and semantics for languages with measurements and classical control [34,46]. On the circuit side, equivalence checking

A. Sampaio and M. Stoelinga (Eds.): FM 2026, LNCS 16556, pp. 578–598, 2026.
https://doi.org/10.1007/978-3-032-26204-2_30

and optimisation techniques based on decision diagrams [6,8,39,42,43,49], ZX-calculus [25,32], tree automata [9–11], and model counting [27,28,33] have been developed, and model checking and abstract-interpretation approaches for quantum and probabilistic systems have also been investigated [15,16,31,45].

In parallel with these developments, Qiskit [22], one of the most widely used quantum programming frameworks, has recently enriched the traditional circuit model with classical control, allowing users to express quantum algorithms with high-level constructs such as conditional branches (`if`/`else`, `switch`) and loops (`for`, `while`). In particular, the availability of `while`-loops makes it possible to encode measurement-controlled iteration patterns such as repeat-until-success (RUS) schemes [5,30], weakly measured variants of Grover's search [3], and quantum random walks with feedback [24]. At the level of abstract syntax, this brings mainstream quantum programming closer to the control-flow constructs long studied in quantum programming languages [34] and quantum Hoare logics [44].

On the execution side, however, the situation is markedly less mature:

At present, there is no Qiskit-native tool that can simulate Qiskit `while`*-loop programs.*

Qiskit-Aer, the default simulator in the Qiskit ecosystem, currently *does not* support the `while`-loop construct at all. In particular, attempting to run the official Qiskit API examples that use `QuantumCircuit.while_loop` through Aer results in parsing or compilation errors, indicating that such control flow cannot be lowered to a circuit that Aer is able to simulate. A similar limitation holds for the Qiskit cloud backends: when submitting circuits that contain `while`-loop constructs, the service explicitly reports that the available real-device backends *do not* support `while`-loop constructs. In other words, although Qiskit and OpenQASM 3 [13] expose `while`-loops at the programming and IR levels, neither the standard Qiskit simulator nor the current hardware backends can execute these constructs, and there is, at present, no Qiskit-native execution engine that realises the intended iterative semantics of non-trivial `while`-loop programs.

To bridge this gap, we present *QSeqSim*, a symbolic simulation backend for Qiskit programs with classical control, with a particular emphasis on `while`-loop–induced sequential behaviour. On the semantic side, we interpret Qiskit `while`-loops as *sequential quantum circuits (SQCs)* with an explicit partition into external and internal qubits, thereby making precise the notion of quantum state feedback across loop iterations. On the algorithmic side, we build on a BDD-based representation of quantum circuits [39] and extend it from purely combinational circuits to programs with structured control flow: `if`/`switch` constructs are treated as dynamic circuits with mid-circuit measurements, `for` loops with static bounds as repeated combinational circuits, and `while` loops as sequential circuits. At the Boolean level, we introduce new symbolic operators for *state composition* and *state retention*, which model how external inputs are combined with the current internal state and how the post-measurement internal state is fed back into the next iteration. These operators are integrated with existing Boolean gate update rules and a Boolean mid-circuit measurement operator

into a unified BDD-based engine that executes mixed combinational, dynamic, and sequential fragments according to a small-step operational semantics, while systematically exploiting weighted model counting [7] techniques to carry out probability computation efficiently on structured and sparse BDDs.

Contributions. This paper makes the following main contributions.

1. *First Qiskit backend with direct* `while`*-loop support.* We present *QSeqSim*, to the best of our knowledge, the first backend that directly executes Qiskit programs containing `while`-loops, by lowering them to OpenQASM 3 and giving an executable small-step semantics for the resulting control flow.
2. *From combinational to sequential Boolean models. QSeqSim* extends a BDD-based encoding of combinational quantum circuits [39] with symbolic operators for state composition and retention, lifting the Boolean model from purely combinational to sequential circuits so as to support `while`-loops.
3. *Efficient and scalable symbolic simulation of* `while`*-loop programs (SQCs).* Leveraging mature BDD backends and model-counting techniques for efficient probability evaluation, *QSeqSim* symbolically simulates quantum circuits induced by `while`-loops, scaling beyond existing tools [9,41].

1.1 Related Work

Quantum Programming Backends. A rich ecosystem of quantum programming frameworks and simulators has emerged in recent years, including general-purpose stacks such as Qiskit [22], Cirq [12], ProjectQ [36], CUDA-Q [38], and Amazon Braket [17], as well as specialised high-performance simulators like QuEST [23], Qulacs [37], and Intel-QS (IQS) [20]. However, there remains a clear gap between the expressiveness of modern quantum programming models and the capabilities of current execution backends. On the language side, Qiskit has recently enriched its model with high-level classical control constructs such as `if_else`, `for_loop`, and in particular `while_loop`, enabling natural encodings of measurement-controlled iteration patterns. On the backend side, however, mainstream backends effectively lack native support for `while`-loops: Qiskit-Aer cannot currently execute `QuantumCircuit.while_loop`, and Qiskit's cloud hardware backends likewise report `while`-loops as unsupported. *QSeqSim* is designed to fill precisely this gap. Starting from `QuantumCircuit` objects, it translates circuits to OpenQASM 3, assigns the resulting programs a sequential-circuit semantics, and uses a BDD-based engine to execute `while`-loop programs at a scale that makes it suitable as an efficient backend.

Simulation Techniques for Quantum Circuits. There is also extensive work in the formal methods community on the simulation of quantum circuits [8,10,11,25,27,39,42,49]. However, these approaches remain essentially confined to combinational and very limited dynamic settings; they do not extend their models to *sequential quantum circuits* with explicit state feedback, and thus cannot support the Qiskit `while`-loop construct. *QSeqSim* builds directly on the Boolean framework of [39], and further integrates weighted model counting

for probability queries, while also extending the framework with new symbolic operators for state composition and state retention. This yields a unified BDD-based engine that can simulate, within a single Qiskit program, combinational, dynamic, and sequential fragments and thereby natively support `while`-loop induced sequential behaviour.

2 Background

This section provides the necessary background on Qiskit, OpenQASM 3, and SQCs that underpins our semantics and implementation.

Figure 1 structures Qiskit `while`-loop programs into a three-stage pipeline that encompasses the programming language layer, intermediate representation layer, and circuit layer. It also maps each quantum stage to a well-known classical counterpart, allowing the quantum components to be understood through direct comparison with their classical equivalents.

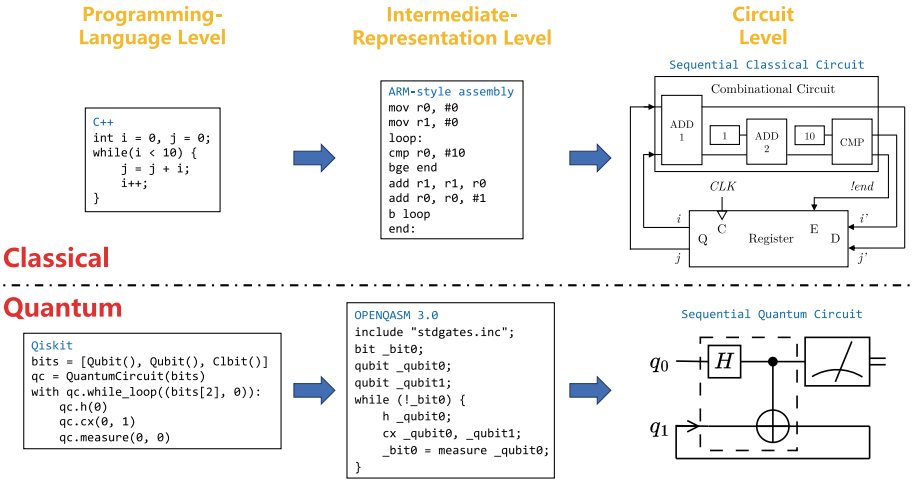

Fig. 1. Classical–quantum analogy from programs to IRs to circuits.

Programming-Language Level. In the top left of Fig. 1, a C++ `while` loop over integers (i, j) is the programmer-facing description of an iterative computation. In the bottom left, the Qiskit code shows a concrete quantum analogue: a RUS implementation of the X gate. Here, the `QuantumCircuit.while_loop` construct repeatedly invokes a small quantum subroutine and measures qubit q_0 to obtain a classical outcome; the loop terminates only when this measurement result is 1, at which point the target qubit q_1 has effectively been subjected to an X operation. This example illustrates how Qiskit `while`-loops are used in practice to express measurement-controlled iteration patterns.

Intermediate-Representation (IR) Level. The C++ `while` loop is then compiled to an ARM-style assembly fragment (top middle of Fig. 1). This assembly is a control-flow oriented IR that is easier to analyse and further compile than the original C++. Similarly, Qiskit compiles the Python-level program to an *internal* IR represented by a `QuantumCircuit` object. This object records the gates, measurements, and classical control structure in a uniform form. From there, Qiskit can export an *external* IR in the form of OpenQASM 3 code [13,29] (bottom middle of Fig. 1): a quantum assembly language with a formally defined syntax and semantics. In this work we treat OpenQASM 3, in the restricted fragment generated by Qiskit, as the semantic interface for `QuantumCircuit`.

Circuit Level. On the classical side (top right of Fig. 1), the ARM-style assembly fragment is implemented as a *sequential classical circuit*: a combinational logic block composed of standard components such as adders (ADD) and a comparator (CMP), together with a register and feedback wires. On the quantum side (bottom right of Fig. 1), we obtain the analogous notion of a *sequential quantum circuit*. The dashed box is the loop body: in this example it is a fixed subcircuit consisting of a Hadamard on q_0 followed by a controlled-X from q_0 to q_1, and one pass through this box corresponds to a single loop iteration. After the body, q_0 is measured and the outcome is used as the loop guard: a result of 0 triggers another iteration, whereas a result of 1 terminates the loop, exactly mirroring the role of the comparator in the sequential classical circuit. The wire for q_1 passes through the body and feeds back to its input, indicating that the quantum state on q_1 is preserved and updated across iterations, in direct analogy with the classical register that stores i and j.

3 Implementation Challenges

Filling the current gap of having no Qiskit-native tool that can simulate `while`-loop programs raises three main specific challenges.

Challenge 1: Matching Qiskit `while`-*loops with sequential semantics.* From the circuit point of view, a `while`-loop induces sequential behaviour: quantum state is fed back from one iteration to the next, and the guard is evaluated from intermediate measurement outcomes. The sequential quantum circuit (SQC) model of Wang et al. [41] captures this by iterating a fixed block over *external* and *internal* qubits, with only the external ones measured each round. Realistic Qiskit `while`-loops, however, both exceed and restrict this abstraction: their loop bodies may contain mid-circuit measurements and nested `if`/`else` branches not covered by the original SQC definition, yet they arise from concrete `QuantumCircuit` programs and do not exploit arbitrary per-iteration external inputs.

Our Approach. We adopt a Qiskit-oriented generalisation of SQCs with an explicit split between internal and external qubits and treat each `while`-loop body as a fixed dynamic subcircuit with mid-circuit measurements and classical control, executed unchanged in every iteration. This generalised SQC view is realised concretely by the CQC/DQC/SQC decomposition of the *Parser* and

the small-step operational semantics implemented by the *Simulator* (Sect. 4, the *Parser* and Algorithm 1).

Challenge 2: *Symbolic simulation of sequential quantum circuits.* Even with a suitable semantics and IR, simulating `while`-loops with internal measurements and branches remains difficult. Sequential circuits involve feedback across iterations, entangled internal state, and dynamic bodies with nested `if`/`else` depending on mid-circuit outcomes. Naive statevector simulation scales poorly in both qubits and iterations, and existing decision-diagram-based techniques mostly target combinational or, at best, non-feedback dynamic circuits; they do not extend their models to general sequential behaviour with explicit state feedback.

Our Approach. Building on the BDD-based Boolean encoding of [39], we extend the symbolic model from purely combinational circuits to general SQCs by introducing operators for state composition and state retention, and by supporting external inputs and internal state feedback across iterations. The resulting SQC engine is implemented in the *Kernel* component `BDDSeqSim`, whose structure is detailed in Sect. 4 (Kernel and Algorithm 2).

Challenge 3: Probability computation for large-qubit, loop-heavy programs. Symbolic simulation of `while`-loop programs requires repeated evaluation of measurement probabilities, especially for mid-circuit measurements and loop guards, often over large yet structured decision diagrams. In existing BDD-based simulation techniques [39], such probabilities are obtained via a hyper-function construction over the decision diagrams; however, for circuits with large numbers of qubits and deep structure this approach becomes a major bottleneck and can dominate the overall runtime, even when the underlying BDDs remain compact. This raises a specific scalability challenge: how to obtain exact measurement probabilities while preserving and exploiting the sharing and sparsity structure of the BDDs generated by large quantum circuits.

Our Approach. We incorporate model counting techniques [7,27,28] from formal methods and adopt weighted model counting on structured and sparse BDDs as the core mechanism for evaluating measurement probabilities. This counting-based engine implements the GetProb interface used by both the *Simulator* and the SQC *Kernel*, as detailed in Sect. 4 (Kernel, weighted model counting for probability evaluation).

4 Tool Architecture

This section describes the architecture of *QSeqSim* and explains how its components jointly address the challenges identified in Sect. 3.

Figure 2 presents the overall architecture of *QSeqSim*. Starting from a Qiskit program written in Python, *QSeqSim* operates over Qiskit's internal intermediate representation (IR), namely the `QuantumCircuit` object, and processes it through three core modules: a *Parser*, a *Simulator*, and a Boolean *Kernel*. At a high level, the *Parser* realises the compilation phase, whereas the Simulator and Kernel jointly implement the symbolic execution back-end. Together,

the *Parser* and Simulator instantiate the generalised SQC semantics for Qiskit `while`-loops (Challenge 1), while the *Kernel* provides the symbolic machinery for SQCs (Challenge 2) and scalable probability computation (Challenge 3).

The end-to-end workflow, which corresponds to Fig. 2, is as follows.

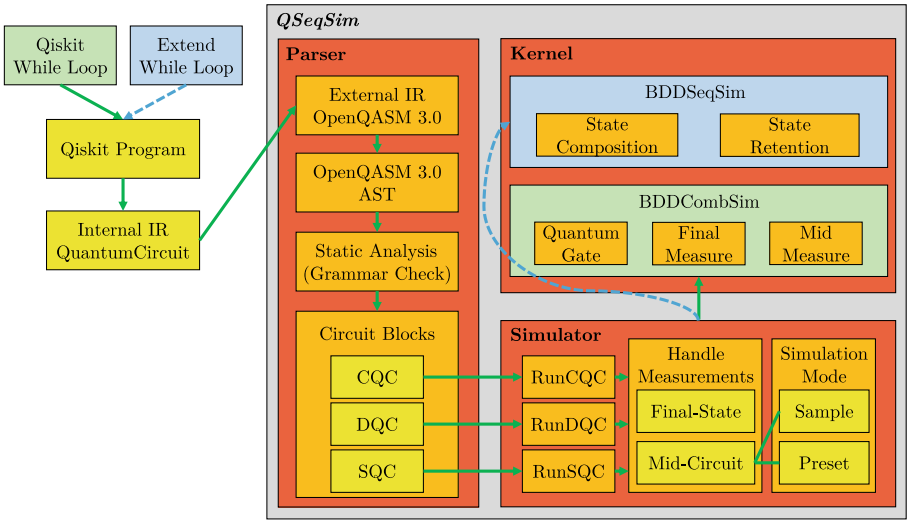

Fig. 2. *QSeqSim* workflow.

From Qiskit Program to Internal IR. The user specifies a circuit via the Qiskit Python API. Qiskit lowers it to a `QuantumCircuit` object, which is both its internal IR and the entry point to *QSeqSim*.

Parser (compile phase). Given a `QuantumCircuit`, the *Parser* performs the following steps:

1. *OpenQASM 3 export.* We invoke Qiskit's `qasm3.dumps` routine to obtain an *external* IR in the form of an OpenQASM 3 program.
2. *AST construction.* The resulting OpenQASM 3 code is parsed by the official OpenQASM 3 parser into an abstract syntax tree (AST).
3. *Static analysis.* On this AST, *QSeqSim* performs the following analyses: (i) it checks that all operations belong to the OpenQASM 3 fragment supported by the tool; (ii) it verifies that every unitary can be decomposed into the Clifford+T gate set implemented by the *Kernel*; (iii) it classifies each measurement as *mid-circuit* or *final*, depending on whether its outcome is used in any subsequent control condition; and (iv) it partitions the program into an ordered sequence of circuit blocks, each labelled as CQC, DQC or SQC. In particular, although we present the result as an ordered block sequence for simplicity, the *Parser* preserves the underlying *hierarchical* block structure induced by nested `if`/`switch`/`while` constructs.

The resulting block sequence preserves the original program order of the Qiskit circuit and constitutes the sole input to the symbolic backend. By explicitly identifying CQC, DQC, and in particular SQC blocks for loop bodies, the *Parser* provides exactly the structural interface required for the sequential semantics of Qiskit `while`-loops (Challenge 1).

Simulator (operational semantics). The *Simulator* provides an operational interpretation of the block sequence produced by the *Parser*. CQC blocks are treated as straight-line gate segments; DQC blocks implement branching controlled by measurement outcomes; and SQC blocks capture `while`-loops, which are unfolded by repeatedly executing their bodies until the loop guard, read from classical bits, evaluates to false. Algorithm 1 formalises this behaviour in high-level pseudo-code and instantiates the small-step semantics for mixed combinational, dynamic, and sequential fragments, completing the realisation of Challenge 1 on top of the *Parser*'s CQC/DQC/SQC decomposition.

Interfaces Used by the Simulator. Algorithm 1 relies on a small set of primitive operations provided by the BDD-based Kernel and the *Parser*:

- BDDCombSim(n, r): create an empty BDD-based symbolic simulator for n qubits with precision parameter r, i.e., the number of bit slices used to represent amplitudes in the encoding of [39].
- IsFinalMeas(G): return *true* iff measurement G is marked as *final* by the OpenQASM 3 static analysis in the *Parser*.
- GetProb(s, q): for a BDD-encoded state s, return the unnormalised probabilities $(p_0^{\text{joint}}, p_1^{\text{joint}})$ of outcomes 0 and 1 on qubit q (computed via weighted model counting in the *Kernel*; Challenge 3).
- MidMeasure(s, q, b): perform a symbolic mid-circuit measurement of qubit q with outcome $b \in \{0, 1\}$ on state s.

Kernel (Boolean engine). All quantum-state manipulation is delegated to the *Kernel*, which is implemented on top of BDDs. The *Kernel* provides the low-level symbolic operators used by the *Simulator* and is organised into two components, addressing Challenges 2 and 3.

- `BDDCombSim`, which implements Boolean gate-update rules for combinational circuits together with a mid-circuit measurement operator, following the BDD techniques of Tsai et al. [39]. Combined with CQC/DQC decomposition, this suffices to handle all combinational and dynamic fragments, as well as the mid-circuit and final measurements that arise in Qiskit programs.
- `BDDSeqSim`, which extends the *Kernel* to full sequential quantum circuits in the sense of Wang et al. [41]. Beyond Qiskit's current `while_loop` semantics, `BDDSeqSim` also supports an *extended* `while`-loop style, where each iteration injects a fresh external input state into a fixed SQC body, e.g.:

Algorithm 1. SIMULATOR($\mathcal{B}, n, r, m$)

Input: Parsed block sequence $\mathcal{B} = [B_1, \ldots, B_K]$; number of qubits n; BDD precision parameter r; measurement mode $m \in \{\mathit{sample}, \mathit{preset}\}$.
Output: BDD-encoded final quantum state s; path probability p_{global}.

$s \leftarrow$ BDDCOMBSIM(n, r) ▷ initialise BDD kernel
initialise s to $|0 \cdots 0\rangle$
$p_{\text{global}} \leftarrow 1$
clbits $\leftarrow$ empty map ▷ classical store
EXECUTEBLOCKS($\mathcal{B}$)
return $[s, p_{\text{global}}]$

procedure EXECUTEBLOCKS($\mathcal{B}'$)
  for each block B in $\mathcal{B}'$ in program order **do**
    if B is a CQC block **then**
      for each operation G in B in program order **do**
        if G is a measurement $q \to c$ **then**
          HANDLEMEASURE$(q, c,$ ISFINALMEAS$(G))$
        else
          apply the Boolean gate update for G on s
        end if
      end for
    else if B is a DQC block **then**
      read guard bits from clbits and compute switch value v
      select branch $\mathcal{B}_{\text{sub}}$ of B according to v
      EXECUTEBLOCKS($\mathcal{B}_{\text{sub}}$)
    else if B is an SQC block **then** ▷ while loops
      while loop condition of B evaluated from clbits holds **do**
        let $\mathcal{B}_{\text{body}}$ be the body-block sequence of B
        EXECUTEBLOCKS($\mathcal{B}_{\text{body}}$)
      end while
    end if
  end for
end procedure

procedure HANDLEMEASURE($q, c,$ *isFinal*)
  if *isFinal* **then**
    record $(q \to c)$ as a deferred final measurement
  else
    $(p_0^{\text{joint}}, p_1^{\text{joint}}) \leftarrow$ GETPROB(s, q) ▷ query prob. of q
    $p_0 \leftarrow p_0^{\text{joint}} / (p_0^{\text{joint}} + p_1^{\text{joint}})$; $p_1 \leftarrow 1 - p_0$
    if $m = \mathit{sample}$ **then**
      sample $b \in \{0, 1\}$ with $\Pr[b = 0] = p_0$ and $\Pr[b = 1] = p_1$
    else if $m = \mathit{preset}$ **then**
      $b \leftarrow$ next preset value for classical bit c
    end if
    $p_{\text{global}} \leftarrow p_{\text{global}} \times (b = 0 \ ? \ p_0 : p_1)$
    $s \leftarrow$ MIDMEASURE(s, q, b) ▷ symbolic collapse on q
    clbits$[c] \leftarrow b$
  end if
end procedure

```
q_internal = ... # internal qubits, persist across iterations
c = 0 # loop guard (classical bit)
while c == 0:
    p = new_qubit() # fresh external input with arbitrary state
    U(p, q_internal, ...) # body unitary using p and internal qubits
    c = measure(p) # update loop flag
```

To simulate such general SQCs, `BDDSeqSim` introduces two dedicated operators: *state composition*, which tensors the external input state with the current internal state, and *state retention*, which discards the measured external qubits while preserving the updated internal state (Challenge 2). Their pseudo-code is given in Algorithm 2.

Remark (internal vs. external qubits). For an SQC induced by a `while`-loop, *internal* qubits are the iteration-to-iteration state that is retained and fed back, whereas *external* qubits are iteration-scoped qubits that are measured and then discarded.

Algorithm 2. BDDSeqSim($|\psi_{\text{in}}\rangle, \mathcal{C}, \mathcal{I}, \mathcal{M}$)

Input: Initial internal quantum state $|\psi_{\text{in}}\rangle = (k, \{F_x^i\})$ (notation as in [39]); an SQC $\mathcal{C}$ whose body is a DQC $\mathcal{C}'$; a pair of sequences (or iterators) $\mathcal{I}$ and $\mathcal{M}$ providing, for each iteration, the external input state and the corresponding external measurement outcomes (given in *preset* mode or generated on the fly in *sample* mode).
Output: Final internal state $(\hat{k}, \{\hat{F}_x^i\})$; path probability p_{global} of $\mathcal{M}$.

1: $p_{\text{global}} \leftarrow 1$ ▷ accumulated probability of observing $\mathcal{M}$
2: $(k_{\text{in}}, \{F^i_{x,\text{in}}\}) \leftarrow (k, \{F_x^i\})$ ▷ initialise internal state
3: **for** each external input state $(0, \{F^i_{x,\text{ex}}\}) \in \mathcal{I}$ **do**
4: $(k_{\text{total}}, \{F^i_{x,\text{total}}\}) \leftarrow (k_{\text{in}}, \{F^0_{d,\text{ex}} \wedge F^i_{x,\text{in}}\})$ ▷ state composition
5: apply $\mathcal{C}'$ to $(k_{\text{total}}, \{F^i_{x,\text{total}}\})$ using Boolean gate transformations [39] and the HandleMeasure procedure of Algorithm 1
6: $M[q_0], \ldots, M[q_{m-1}] \leftarrow \mathcal{M}$ ▷ get the corresponding measurement outcomes
7: Construct assignment $\widetilde{q_0} \ldots \widetilde{q}_{m-1}$ based on $M[q_0], \ldots, M[q_{m-1}]$.
8: $(k_{\text{in}}, \{F^i_{x,\text{in}}\}) \leftarrow \left(k_{\text{total}}, \left\{ F^i_{x,\text{total}} \middle|_{\widetilde{q_0} \ldots \widetilde{q}_{m-1}} \right\}\right)$ ▷ state retention
9: compute probability p of observing $M[q_0], \ldots, M[q_{m-1}]$ via weighted model counting (see *Weighted model counting for probability evaluation*)
10: $p_{\text{global}} \leftarrow p_{\text{global}} \cdot p$
11: **end for**
12: $(\hat{k}, \{\hat{F}_x^i\}) \leftarrow (k_{\text{in}}, \{F^i_{x,\text{in}}\})$
13: **return** $\left[(\hat{k}, \{\hat{F}_x^i\}), p_{\text{global}}\right]$

Remark (iterators for while loops). The `for`-loop in Algorithm 2 is conceptual: in the implementation, the SQC is unfolded by repeatedly consuming one element from $\mathcal{M}$ (and, if needed, $\mathcal{I}$) *while the loop guard holds*. In *sample* mode, $\mathcal{M}$ is an

on-demand iterator whose next outcome is sampled when the guard is evaluated, and it stops producing values once the guard becomes false. In *preset* mode, the user provides a finite prefix of $\mathcal{M}$ (and $\mathcal{I}$), which fixes a bounded unrolling of the `while`-loop.

Weighted Model Counting for Probability Evaluation (Challenge 3). A central task of the *Kernel* is to evaluate measurement probabilities on BDD-encoded quantum states. Instead of using the hyper-function construction of [39], *QSeqSim* adopts a model counting view: probabilities are obtained via weighted model counting over structured and sparse BDDs representing the amplitude polynomials (see [7]). Because BDDs compactly encode large families of basis states with shared structure, weighted model counting can aggregate their contributions in time proportional to the size of the BDD representation (up to caching and arithmetic costs), avoiding explicit enumeration of exponentially many assignments. This does not imply polynomial-time counting in the worst case with respect to the original circuit; the gain comes from a compact BDD structure. This probability computation is tightly integrated into GetProb and into the SQC engine (Algorithm 2), and forms the main algorithmic ingredient in addressing the scalability issues highlighted in Challenge 3.

5 Functionalities

We summarise here the main functionalities offered to users by *QSeqSim*.

5.1 Simulation of Qiskit While Loop and Its Extensions

QSeqSim can be used as a backend to simulate a broad class of Qiskit `while`-loops, including patterns that are not yet natively supported by current Qiskit.

- **Qiskit `while_loop` programs.** Users can directly run Qiskit circuits containing `while_loop` constructs (possibly mixed with `if`/`else` and `for`). *QSeqSim* executes the loop semantics and returns the resulting quantum state together with the probability of a chosen measurement pattern.
- **Extended while loops.** Beyond standard Qiskit `while`-loop programs, users can also specify SQCs that admit an arbitrary external input state at each iteration. *QSeqSim* simulates such extended `while`-loops via `BDDSeqSim` and allows inspection of the state after any iteration.
- **General Qiskit programs with arbitrary control flow.** Qiskit programs with arbitrary combinations and nesting of `if`/`else`, `switch`, `for`, and `while` constructs are supported.

In all cases, users may either let *QSeqSim* randomly sample measurement outcomes (*sample* mode), or specify a concrete outcome pattern (*preset* mode) to fix a particular execution path.

5.2 Analysis of Quantum State Reachability

In formal methods, state reachability, namely deciding whether a given configuration can be reached and with what probability, is a central problem, from model checking of Markov chains and other probabilistic systems to the verification of (quantum) while loops [1,2,4,9,15,16,19,45].

QSeqSim supports such (bounded) *quantum state reachability* queries for while loops. A program point is fixed (for example, "after the k-th loop iteration") together with a target quantum state or basis configuration on some qubits. For a quantum random walk implemented with a `while`-loop, a reachability question of common interest is:

> "After the k-th iteration, is it possible for the walker to be at position i? If so, what is the probability of the execution paths that realise this?"

To answer such queries, candidate paths are encoded via the *preset* mode (fixing the sequence of measurement outcomes across iterations). *QSeqSim* then computes, for each path, the final quantum state and its probability, which allows users to check reachability and quantify the likelihood of reaching the target.

5.3 Analysis of Measurement Outcome Reachability

QSeqSim also supports queries about whether particular *measurement outcome patterns* can occur and with what probability, a central concern in the analysis of quantum protocols with repeated measurements and feedback, and in studying loop termination behaviour [14,41,47]. Given a sequence of measurement outcomes across iterations for a while loop, specified via the *preset* measurement mode, *QSeqSim* returns the probability of the corresponding execution. For a RUS circuit expressed as a `while`-loop that repeats until a measurement returns 0, a typical measurement outcome reachability question is:

> "What is the probability that the sequence of outcomes is `1111110` (that is, six times 1 followed by 0), and the loop terminates at that point?"

By varying such queries over families of patterns (for example, "eventually a 0 occurs"), one can empirically study whether a RUS or feedback-based protocol converges or may fail to terminate with non-zero probability.

6 Evaluation

We evaluate *QSeqSim* along three dimensions matching its intended uses: (i) simulation of Qiskit `while`-loops (Sect. 6.1), (ii) quantum state reachability (Sect. 6.2), and (iii) measurement outcome reachability (Sect. 6.3). All experiments were run on a MacBook Air 2023 (Apple M2, 16 GB unified memory) with Python 3.12.

6.1 Simulation of Qiskit While Loops

We consider three structured benchmarks (RUS [30], QRW [35], Grover [3,18]) and, separately, randomly generated `while`-loop programs. Unless stated otherwise, we use *preset* mode: a concrete sequence of mid-circuit measurement outcomes is fixed in advance, so that *QSeqSim* symbolically executes a single predetermined path.

For the RUS and random `while`-loop benchmarks, circuits are generated via the Qiskit Python API, exported to OpenQASM 3, and then passed to *QSeqSim* as in Sect. 3. In contrast, QRW and Grover rely in Qiskit on custom Python functions that repeatedly instantiate large static subcircuits (shift, oracle, diffusion), incurring significant OpenQASM parsing and construction overhead. To isolate the cost of symbolic simulation, we therefore bypass Qiskit for QRW and Grover and directly invoke the `BDDSeqSim` kernel on hand-specified sequential circuits, using the same BDD-based engine.

Repeat-Until-Success (RUS). We use the six two-qubit RUS circuits from Paetznick and Svore [30, Figures 7–10], where a common RUS pattern is instantiated with different single-qubit unitaries U and corresponding ancilla measurement rules. We implement these six RUS variants (RUS-1 to RUS-6) in the Qiskit API and invoke *QSeqSim* in *preset* mode. In each case the ancilla is measured every iteration and the loop terminates on a designated "success" outcome: for RUS-1–RUS-4 success is 0, while for RUS-5 and RUS-6 success is 1. Accordingly, we fix the ancilla outcome pattern to $0^{k-1}1$ for RUS-1–RUS-4 and to $1^{k-1}0$ for RUS-5/RUS-6, enforcing exactly k iterations.

Table 1 reports a representative configuration for six different RUS implementations U. The number of qubits and per-iteration gate count remain tiny (two qubits, 7–37 gates), so runtimes mainly reflect the symbolic cost of unfolding the loop. For short loops (10 iterations), all variants complete in under 1s.

Table 1. Performance of *QSeqSim* on Qiskit RUS `while`-loops (*preset* mode).

Benchmark	Implementation U	#Qubits	#Gates	#Iterations	Time (s)
RUS-1	$\frac{2X+\sqrt{2}Y+Z}{\sqrt{7}}$	2	7	10	0.300
				100	0.960
RUS-2	$\frac{I+2i\sqrt{2}X}{\sqrt{3}}$	2	13	10	0.673
				100	1.427
RUS-3	$\frac{I+2iZ}{\sqrt{5}}$	2	12	10	0.656
				100	11.081
RUS-4	$\frac{3I+2iZ}{\sqrt{13}}$	2	20	10	0.760
				100	20.807
RUS-5	$\frac{4I+iZ}{\sqrt{17}}$	2	37	10	0.963
				100	90.027
RUS-6	$\frac{5I+2iZ}{\sqrt{29}}$	2	33	10	0.994
				100	108.148

For 100 iterations, however, runtimes span nearly three orders of magnitude: from below 1.5s for RUS-1/RUS-2 to around 90–108s for RUS-5/RUS-6. This variability shows that *QSeqSim* is sensitive not only to loop depth but also to the specific unitary U, which affects BDD structure and sharing.

Quantum Random Walk (QRW). For QRW, we work directly with `BDDSeqSim`, following the experimental setup of Wang et al. [41, Figure 3]. Each iteration applies a Hadamard on the coin, a coin-controlled shift on an ℓ-qubit position register, and a multi-controlled X on a single flag qubit that flips when a fixed target basis state is reached, namely the internal all-ones configuration of the coin and the position register.

Table 2a shows runtimes as we scale the position register from $N = 16$ to $N = 1024$ sites and increase the number of iterations. For $N = 16$ (16 qubits, 31 gates per iteration), *QSeqSim* comfortably reaches hundreds of iterations, but the cost grows rapidly: going from 100 to 785 iterations increases the time from 5.3s to about 1800s. Larger walks are limited by qubit and gate counts rather than loop depth: for $N = 1024$ (1024 qubits, 2047 gates per iteration), only up to 13 iterations are feasible within a 30-minute timeout.

Grover's Search. The Grover benchmark follows the same pattern: a fixed number of data qubits plus a flag qubit, with each iteration consisting of an oracle phase flip on a single marked basis state followed by a diffusion operator. Again

Table 2. Performance of *QSeqSim* on different `while`-loops (`BDDSeqSim`).

(a) QRW `while`-loops

#Qubits	#Gates	#Iterations	Time (s)
16	31	3	0.041
		10	0.111
		100	5.303
		200	30.375
		785	1797.676
128	255	3	0.698
		10	2.861
		100	239.485
		200	1118.501
		244	1792.934
1024	2047	3	235.568
		5	438.156
		8	867.763
		10	1124.445
		13	1622.304

(b) Grover `while`-loops

#Qubits	#Gates	#Iterations	Time (s)
16	81	3	0.177
		10	2.354
		100	340.669
		150	866.182
		203	1781.141
128	641	2	3.115
		3	16.293
		6	135.108
		10	622.699
		15	1700.021
256	1281	2	256.593
		3	527.699
		4	840.786
		5	1156.052
		6	Timeout

we emulate a `while`-loop by enforcing a specific termination iteration via the mid-circuit measurement pattern.

Table 2b summarises the results. For 16 work qubits (81 gates per iteration), runtimes scale from 0.18s at 3 iterations to around 1780 s near 200 iterations. For 128 qubits, even 15 iterations already saturate our timeout. At 256 qubits and 1281 gates per iteration, *QSeqSim* handles up to 5 iterations (about 19 min) before timeout (30 min) at 6 iterations. These trends are consistent with QRW: BDD-based symbolic simulation can handle substantial gate counts and deep loops, but very large multi-qubit operators combine unfavourably with repeated iteration.

Random Quantum While Loops. To evaluate robustness under unstructured control flow, we generated random quantum circuits using the Qiskit API, which contain a `while`-loop with additional quantum gates placed outside the loop. We fixed the system size at 100 qubits and varied two key experimental parameters: computational density (defined as the total number of gates in the circuit, denoted G) and control complexity (defined as the total number of mid-circuit measurements, denoted M). These experiments were executed in *sample* mode: for each mid-circuit measurement performed within the loop, *QSeqSim* computes the exact outcome distribution over the current symbolic state, then samples the next execution branch, and proceeds until the loop terminates or reaches a predefined iteration bound of 1000.

As in *sample* mode the loop termination time is a random variable determined by the sampled measurement outcomes, the resulting runtime distribution is typically right-skewed; we therefore repeat each configuration 50 times and report the median and interquartile range (IQR $[Q_1, Q_3]$)[1]. Table 3 shows that *QSeqSim* remains tractable on 100-qubit circuits: across all tested settings, median runtimes stay within a minute. Runtime generally grows with circuit size, while

Table 3. Performance of *QSeqSim* on Qiskit random `while`-loops (*sample* mode).

Benchmark	Qubits	Gates	Mid-Meas	Total Time (s)
rqc_q100_g50_m5	100	50	5	0.218 [0.215, 0.915]
rqc_q100_g50_m10	100	50	10	0.273 [0.194, 1.158]
rqc_q100_g50_m20	100	50	20	0.214 [0.211, 0.367]
rqc_q100_g100_m5	100	100	5	0.971 [0.945, 3.053]
rqc_q100_g100_m10	100	100	10	0.949 [0.942, 2.789]
rqc_q100_g100_m20	100	100	20	1.149 [1.008, 6.092]
rqc_q100_g200_m5	100	200	5	2.235 [2.151, 5.910]
rqc_q100_g200_m10	100	200	10	15.978 [3.056, 22.853]
rqc_q100_g200_m20	100	200	20	2.116 [1.767, 10.228]

[1] The median is the 0.5 quantile of the sample distribution, and the interquartile range $[Q_1, Q_3]$ is the interval between the 0.25 and 0.75 quantiles; see [21].

mid-circuit measurements have a non-monotone, control-flow-dependent effect (e.g., G=200, M=10 vs. 20). As random `while`-loops may not terminate, our generator enforces almost termination; these results thus characterize robustness under terminating control flow.

6.2 Analysis of Quantum State Reachability

We next use the QRW benchmark to study quantum state reachability for a `while`-loop semantics. Each QRW step consists of a coin flip on a single qubit, a controlled shift on an ℓ-qubit position register, and a multi-controlled X that flips an external flag qubit q_0 precisely when the coin and position jointly equal a fixed target basis state (here, the all-ones internal configuration).

Reachability formulation. A reachability question of common interest is:

> *"After k QRW iterations, can the internal configuration (coin + position) reach the all-ones basis state, and if so, with what exact probability?"*

This reduces to the marginal state of the flag qubit q_0 after k steps: the all-ones state is reachable at iteration k iff q_0 can be in $|1\rangle$. Under *preset* mode, we fix the measurement outcomes controlling the loop guard to $0^{k-1}1$, so that the loop continues for $k-1$ iterations and terminates at step k. The branch probability of this classical pattern is then exactly the reachability probability of the all-ones internal configuration at step k.

Results. Table 4 reports results for a fixed system size of 256 qubits and iteration counts $k = 1, \ldots, 10$. For each k, `BDDSeqSim` symbolically executes the constrained path $0^{k-1}1$, checks whether measuring qubit q_0 in the computational basis can yield outcome 1, and returns the exact branch probability together with the runtime.

Table 4. Quantum state reachability of QRW `while`-loop.

#Qubits	k	Reachable?	Probability	Time (s)
256	1	Yes	0.5	1.264
	2	No	0	2.489
	3	Yes	0.125	3.808
	4	No	0	5.053
	5	No	0	6.399
	6	No	0	7.992
	7	Yes	0.0078125	9.729
	8	No	0	11.409
	9	No	0	13.453
	10	No	0	15.563

The non-zero rows reveal that the all-ones configuration is only reachable at specific iteration counts (here, $k = 1, 3, 7$), and the associated branch probabilities decay quickly. *QSeqSim* computes these probabilities exactly, without sampling noise, and the runtimes grow approximately linearly with k in this setting.

6.3 Analysis of Measurement Outcome Reachability

We finally evaluate how easily users can pose measurement-outcome reachability queries on the RUS-1 `while`-loop from Sect. 6.1. Each loop iteration applies the fixed two-qubit RUS-1 subcircuit and measures an ancilla; the loop continues while the ancilla is 1 and terminates on the first 0.

A typical measurement outcome reachability query is:

> *"What is the probability that the ancilla outcomes are* 1^k0*, so that the loop terminates exactly at iteration* $k + 1$*?"*

In *preset* mode, the user specifies the concrete pattern 1^k0; *QSeqSim* then symbolically follows this unique branch and returns: (i) the probability of that pattern, $p_{\text{path}}(k)$, and (ii) the probability that the loop has terminated by iteration $k + 1$, $p_{\text{term.}}(k)$. Here $p_{\text{term.}}(k)$ is obtained as the cumulative probability of all termination patterns up to iteration $k + 1$, i.e. $p_{\text{term.}}(k) = \sum_{i=0}^{k} p_{\text{path}}(i)$.

Table 5. Measurement-outcome reachability for the RUS-1 `while`-loop.

k	0	1	2	3	4	5	6	7	8
p_{path}	0.75000	0.18750	0.04688	0.01172	0.00293	0.00073	0.00018	0.00005	0.00001
$p_{\text{term.}}$	0.75000	0.93750	0.98438	0.99609	0.99902	0.99976	0.99994	0.99998	1.00000
Time (s)	0.02607	0.05073	0.04982	0.04946	0.07638	0.11201	0.15824	0.21086	0.29526

Table 5 shows the outputs for $k = 0, \ldots, 8$. The per-pattern probabilities p_{path} quickly become very small as k increases, while termination is already almost certain after a few iterations. All queries complete in well under one second, allowing interactive exploration of rare measurement patterns.

7 Conclusion

We have presented *QSeqSim*, a Qiskit-integrated symbolic backend that directly executes `while`-loop programs by interpreting them as sequential quantum circuits with explicit state feedback across iterations. Building on a BDD-based Boolean representation of quantum states and operations, *QSeqSim* provides symbolic operators tailored to this sequential setting and thus enables efficient

exploration of loop-induced behaviours. Our experiments on RUS schemes, quantum random walks, and Grover's search on commodity hardware show that *QSeqSim* scales to substantial `while`-induced sequential circuits; in particular, in the quantum random walk benchmark we successfully simulate circuits with over 1000 qubits for more than 10 loop iterations, and we support both quantum state reachability and measurement outcome reachability queries for Qiskit `while`-loop programs. All experiments in this work are conducted on a PC. As future work, we plan to deploy *QSeqSim* on server-class machines and exploit parallelism to support larger-scale simulations and better serve subsequent formal verification of quantum programs with `while`-loops.

Acknowledgments. We sincerely thank the anonymous reviewers for their valuable comments and constructive suggestions. This work was partially supported by the Innovation Program for Quantum Science and Technology (Grant No. 2024ZD0300500), the Youth Innovation Promotion Association, Chinese Academy of Sciences (Grant No. 2023116), the National Natural Science Foundation of China (Grant No. 62402485), the Young Elite Scientists Sponsorship Program, China Association for Science and Technology, the CCF-QuantumCtek Superconducting Quantum Computing Special Cooperation Program (Grant No. CCF-QC2025007), and the International Partnership Program of the Chinese Academy of Sciences (Grant No. 096GJHZ2025013FN).

References

1. Agrawal, M., Akshay, S., Genest, B., Thiagarajan, P.: Approximate verification of the symbolic dynamics of Markov chains. J. ACM (JACM) **62**(1), 1–34 (2015)
2. Akshay, S., Antonopoulos, T., Ouaknine, J., Worrell, J.: Reachability problems for Markov chains. Inf. Process. Lett. **115**(2), 155–158 (2015)
3. Andrés-Martínez, P., Heunen, C.: Weakly measured while loops: peeking at quantum states. Quant. Sci. Technol. **7**(2), 025007 (2022)
4. Beauquier, D., Rabinovich, A., Slissenko, A.: A Logic of Probability with Decidable Model-Checking. In: Bradfield, J. (ed.) CSL 2002. LNCS, vol. 2471, pp. 306–321. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45793-3_21
5. Bocharov, A., Roetteler, M., Svore, K.M.: Efficient synthesis of universal repeat-until-success quantum circuits. Phys. Rev. Lett. **114**(8), 080502 (2015)
6. Burgholzer, L., Wille, R.: Advanced equivalence checking for quantum circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **40**(9), 1810–1824 (2020)
7. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. **172**(6–7), 772–799 (2008)
8. Chen, T.F., Jiang, J.H.R.: SliQSim: A quantum circuit simulator and solver for probability and statistics queries. In: Gurfinkel, A., Heule, M. (eds.) TACAS 2025. LNCS, vol. 15698, pp. 129–138. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-90660-2_7
9. Chen, Y.F.: AutoQ 2.0: from verification of quantum circuits to verification of quantum programs. In: Gurfinkel, A., Heule, M. (eds.) TACAS 2025. LNCS, vol. 15698, pp. 87–108. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-90660-2_5

10. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L.: AutoQ: an automata-based quantum circuit verifier. In: Enea, C., Lal, A. (eds.) CAV 2023. LNCS, vol. 13966, pp. 139–153. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37709-9_7
11. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L., Yen, D.D.: An automata-based framework for verification and bug hunting in quantum circuits. Proc. ACM Program. Lang. 7(PLDI), 1218–1243 (2023)
12. Cirq Developers: Cirq (v1.6.1) (2025). doi: 10.5281/zenodo.16867504
13. Cross, A., et al.: OpenQASM 3: a broader and deeper quantum assembly language. ACM Trans. Quant. Comput. **3**(3), 1–50 (2022)
14. Eisert, J., Müller, M.P., Gogolin, C.: Quantum measurement occurrence is undecidable. Phys. Rev. Lett. **108**(26), 260501 (2012)
15. Feng, Y., Yu, N., Ying, M.: Model checking quantum Markov chains. J. Comput. Syst. Sci. **79**(7), 1181–1198 (2013)
16. Gay, S., Nagarajan, R., Papanikolaou, N.: Probabilistic model-checking of quantum protocols, (2005). arXiv preprint
17. Gonzalez, C.: Cloud based QC with Amazon Braket. Digitale Welt **5**(2), 14–17 (2021)
18. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, pp. 212–219. (1996)
19. Guan, J., Feng, Y., Turrini, A., Ying, M.: Measurement-based verification of quantum Markov chains. In: Gurfinkel, A., Ganesh, V. (eds.) CAV 2024. LNCS, vol. 14683, pp. 533–554. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-65633-0_24
20. Guerreschi, G.G., Hogaboam, J., Baruffa, F., Sawaya, N.P.: Intel Quantum Simulator: a cloud-ready high-performance simulator of quantum circuits. Quant. Sci. Technol. **5**(3), 034007 (2020)
21. Hoaglin, D.C., Mosteller, F., Tukey, J.W.: Understanding Robust and Exploratory Data Analysis, Wiley (2000)
22. Javadi-Abhari, A.: Quantum computing with Qiskit, (2024). arXiv preprint
23. Jones, T., Brown, A., Bush, I., Benjamin, S.C.: QuEST and high performance simulation of quantum computers. Sci. Rep. **9**(1), 10736 (2019)
24. Kendon, V., Tregenna, B.: Decoherence can be useful in quantum walks. Phys. Rev. A **67**(4), 042315 (2003)
25. Kissinger, A., Van De Wetering, J.: PyZX: large scale automated diagrammatic reasoning, (2019). arXiv preprint
26. Liu, J., Zhan, B., Wang, S., Ying, S., Liu, T., Li, Y., Ying, M., Zhan, N.: Formal Verification of Quantum Algorithms Using Quantum Hoare Logic. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 187–207. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_12
27. Mei, J., Bonsangue, M., Laarman, A.: Simulating quantum circuits by model counting. In: Gurfinkel, A., Ganesh, V. (eds.) CAV 2024. LNCS, vol. 14683, pp. 555–578. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-65633-0_25
28. Mei, J., Coopmans, T., Bonsangue, M., Laarman, A.: Equivalence checking of quantum circuits by model counting. In: Benzmüller, C., Heule, M.J., Schmidt, R.A. (eds.) IJCAR 2024. LNCS, vol. 14740, pp. 401–421. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-63501-4_21
29. OpenQASM Contributors: OpenQASM Live Specification (2025). https://openqasm.com/index.html. Accessed 02 Mar 2026

30. Paetznick, A., Svore, K.M.: Repeat-until-success: non-deterministic decomposition of single-qubit unitaries, (2013). arXiv preprint
31. Papanikolaou, N.K.: Model checking quantum protocols, University of Warwick (2009). Ph.D. thesis
32. Peham, T., Burgholzer, L., Wille, R.: Equivalence checking of quantum circuits with the ZX-calculus. IEEE J. Emerg. Sel. Top. Circuits Syst **12**(3), 662–675 (2022)
33. Quist, A.J., Mei, J., Coopmans, T., Laarman, A.: Advancing quantum computing with formal methods. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) FM 2024. LNCS, vol. 14934, pp. 420–446. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-71177-0_25
34. Selinger, P.: Towards a quantum programming language. Math. Struct. Comput. Sci. **14**(4), 527–586 (2004)
35. Shenvi, N., Kempe, J., Whaley, K.B.: Quantum random-walk search algorithm. Phys. Rev. A **67**(5), 052307 (2003)
36. Steiger, D.S., Häner, T., Troyer, M.: ProjectQ: an open source software framework for quantum computing. Quantum **2**, 49 (2018)
37. Suzuki, Y., et al.: Qulacs: a fast and versatile quantum circuit simulator for research purpose. Quantum **5**, 559 (2021)
38. The CUDA Quantum development team: CUDA Quantum (0.11.0) (2025). doi: 10.5281/zenodo.15407754
39. Tsai, Y.H., Jiang, J.H.R., Jhang, C.S.: Bit-slicing the Hilbert space: scaling up accurate quantum circuit simulation. In: 2021 58th ACM/IEEE Design Automation Conference (DAC), pp. 439–444. IEEE (2021)
40. Unruh, D.: Quantum relational Hoare logic. Proc. ACM Program. Lang. 3(POPL), 1–31 (2019)
41. Wang, Q., Li, R., Ying, M.: Equivalence checking of sequential quantum circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(9), 3143–3156 (2021)
42. Wang, Z., Cheng, B., Yuan, L., Ji, Z.: FeynmanDD: quantum circuit analysis with classical decision diagrams. In: Piskac, R., Rakamarić, Z. (eds.) CAV 2025. LNCS, vol. 15934, pp. 28–52. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-98685-7_2
43. Wei, C.Y., Tsai, Y.H., Jhang, C.S., Jiang, J.H.R.: Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp. 523–528. (2022)
44. Ying, M.: Floyd-Hoare logic for quantum programs. ACM Trans. Program. Lang. Syst. (TOPLAS) **33**(6), 1–49 (2012)
45. Ying, M.: Model Checking for Verification of Quantum Circuits. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 23–39. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_2
46. Ying, M.: Foundations of Quantum Programming, Elsevier (2024)
47. Ying, M., Feng, Y.: Quantum loop programs. Acta Informatica **47**(4), 221–250 (2010)
48. Zhou, L., Yu, N., Ying, M.: An applied quantum Hoare logic. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1149–1162. (2019)
49. Zulehner, A., Wille, R.: Advanced simulation of quantum computations. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **38**(5), 848–859 (2018)