Validating Quantum State Preparation Programs

ANONYMOUS AUTHOR(S)

One of the key steps in quantum algorithms is to prepare an initial quantum superposition state with different kinds of features. These so-called *state preparation* algorithms are essential to the behavior of quantum algorithms, and complicated state preparation algorithms are difficult to program correctly and effectively. This paper presents QSV: a high-assurance framework implemented with the Rocq proof assistant, permitting the development of quantum state preparation programs and validating them to correctly reflect quantum program behaviors. The key in the framework is to reduce the program correctness assurance of a program containing a quantum superposition state to the program correctness assurance for the program state without superposition. The reduction allows the development of *an effective framework for validating quantum state preparation algorithm implementations on a classical computer* — considered a hard problem with no clear solution until this point. We utilize the QuickChick property-based testing framework to validate state preparation programs. We evaluated the effectiveness of our approach over 5 case studies implemented using QSV; such cases are not even simulatable in the current quantum simulators.

1 Introduction

Despite recent advances [Fortunato et al. 2022; Long and Zhao 2024; Paltenghi and Pradel 2022; Wang et al. 2021, 2022; Xia et al. 2024], quantum program developers still lack tools to quickly validate program correctness—testing a program with many test inputs in a short time—as well as other properties when writing comprehensive quantum programs [Gill et al. 2024; Swayne 2023]. Testing quantum programs directly on quantum hardware is problematic because running actual quantum computers is expensive, and the probabilistic nature of quantum computing means repeated trials may be necessary to validate correctness, driving up costs further. Ideally, we should ensure that a program satisfies user specifications before running it on quantum hardware. Unfortunately, such a framework might not exist for validating (testing) a quantum program for arbitrary properties since quantum programs are not classically simulatable without an exponential number of classical bits relative to qubits.

A quantum validation framework needs to satisfy three key design goals.

- Programmers can develop a quantum program based on a proper abstraction, with respect to high-level program properties, without worrying too much about low-level gates.
- The framework contains a scalable and effective validator to quickly judge the correctness of a user-defined program, as well as other properties, based on certain types of quantum program patterns.
- The validated program can be compiled into the quantum circuit for execution.

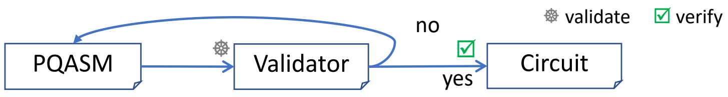
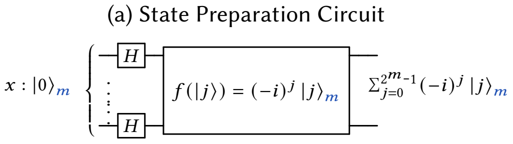


Fig. 1. The QSV Flow

We propose the Quantum State preparation program Validation framework (QSV), which has the flow in Figure 1, permitting effective validation of state preparation programs (The program limitation is discussed in Section 8). It includes three components. Users can develop their programs in a PQASM language, specifically to allow them to write state preparation programs in a high-level abstraction. Such programs can be validated by our validator, based on QuickChick [Paraskevopoulou et al. 2015] (a Rocq-based property-based testing facility), and we show several quantum program patterns that can be effectively validated via our framework. After a program is


(a) State Preparation Circuit
$x : |0\rangle_m$
H
H
$f(|j\rangle) = (-i)^j |j\rangle_m$
$\sum_{j=0}^{2^m-1}(-i)^j |j\rangle_m$

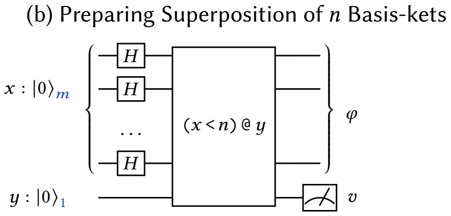


Fig. 2. x has m qubits, and y register has 1 qubit. The right figure is one step in the repeat-until-success program to prepare the superposition state. $\varphi = \frac{1}{\sqrt{n}} \sum_{j=0}^{n} |j\rangle_m$ if $v = 1$; otherwise, $\varphi = \frac{1}{\sqrt{2^m - n}} \sum_{j=n}^{2^m} |j\rangle_m$.

adequately developed in our framework, users can use our certified circuit compiler to compile the program to a quantum circuit, executable in quantum hardware.

1.1 Motivating Examples

We begin by discussing a simple state preparation subroutine, preparing a superposition of n distinct basis-ket states, appearing in many algorithms [Alagic et al. 2007; Nielsen and Chuang 2011], having the following program transition property predicate (a pre-state is transitioned to a post-state connected by $\longrightarrow$) with program input of a length m qubit array, initialized as $|0\rangle_m$, and output a superposition of n different basis-ket states, each with basis-vector $|k\rangle_m$.

$$|0\rangle_m \longrightarrow \sum_{j=0}^{n-1} \frac{1}{\sqrt{n}} |j\rangle_m$$

Generally speaking, *a state preparation program can be defined as the starting component of a quantum algorithm.* A quantum algorithm typically starts with a length m qubit array, each qubit initialized as zero ($|0\rangle_m$) state, and prepares a superposition state $\sum_j \alpha_j |c_j\rangle_m$ — a linear sum of pairs (basis-kets) of complex amplitude α_j and bitstring (basis-vector) c_j such that $\sum_j |\alpha_j|^2 = 1$ — via a series of quantum operations.

Superposition is a key feature of quantum states, and quantum computers can execute programs with superposition states to query all possible inputs simultaneously, as discussed in Section 2. Many quantum algorithms require a comprehensive design of state preparation components with different superposition structures. For example, in Shor's algorithm, we need to prepare a superposition of pairs of a number (x) and the modular multiplication result of the number ($c^x \% n$) (Figure 13).

A difficulty in developing state preparation programs is that quantum program operations might affect every basis-ket in a superposition that might contain exponentially many basis-ket states, unlike the classical programs, where only a single basis-ket might be affected. For example, in Figure 2a, a function f is applied to every basis-ket in the quantum superposition state after all Hadamard operations were applied. Moreover, many quantum languages are circuit gate-based [Aleksandrowicz et al. 2019; Cambridge Quantum Computing Ltd 2019; Cross et al. 2022; Cross et al. 2017]. These hinder the quantum program development as writing programs becomes unintuitive. For example, given the above simple n basis-ket program property, it is not straightforward to develop the program.

Figure 2b shows a one-step procedure of the repeat-until-success program implementation of the n basis-ket program. The procedure starts with a series of Hadamard operations to prepare a uniform superposition of 2^m basis-kets as $\frac{1}{\sqrt{2^m}} \sum_{j=0}^{2^m} |j\rangle_m$, and then compares each basis-ket with the natural number n and stores the comparison result in the extra y qubit. After measuring the y qubit and if the result is 1, all the basis-kets ($\alpha_j |j\rangle_m$), having bitstring numbers $j \geq n$, disappear, while those having bitstrings $j < n$ will stay in x's quantum state; thus, the correct state is prepared. Since the measurement result 1 happens probabilistically, the repeat-until-success program requires repeating the one-step procedure many times to prepare the target state probabilistically. In writing the program, a key component is the comparator $(x < n) @ y$, comparing every basis-vector of a quantum array x with number n and storing the result in qubit y. Such arithmetic operations have effective implementations [Li et al. 2022] and many works discuss circuit-level optimizations [Hietala et al. 2023; Xu et al. 2022]. Therefore, QSV abstracts all these quantum arithmetic operations and relieves the pain of writing quantum programs. The QSV compiler compiles and optimizes the arithmetic operations to quantum circuits. Moreover, we also provide types to classify different program patterns for users to write state preparation programs.

Even if we permit high-level abstractions in QSV, validating a program might still be challenging because of the exponential basis-kets in a quantum state, e.g., the output state $\sum_{j=0}^{n-1} \frac{1}{\sqrt{n}} |j\rangle_m$ might contain exponentially many basis-kets, checking them individually might be challenging. In developing our validator, we have two observations on quantum algorithms. First, almost all quantum algorithms start with m different Hadamard operations to prepare a uniform superposition having 2^m basis-kets. These beginning Hadamard operations are simple enough and do not need to be validated, but they provide the source of superposition state for later program operations to carve on. Second, even though quantum operations are probabilistic, one can "determinize" their behaviors. If we consider the superposition of basis-kets as an array of basis-kets, quantum operations, except measurement, behave similarly to higher-order map functions applying to the quantum state. Measurement behaves similarly to a set selection, selecting a basis-ket element in the array, and the probability of such selection can be computed based on the amplitude value associated with the basis-ket.

In QSV, we classify beginning Hadamard operations as a special `Had` type, to indicate that they are the source of the superposition state. When performing testing, instead of faithfully representing their operational behavior, QSV adopts a random pick of an individual basis-ket state as a representative, and validates programs based on transitions of the basis-ket. A measurement operation is then determinized and its probability is simply calculated via the amplitude value in the basis-ket.

For example, in dealing with the n basis-ket program above, after applying the Hadamard operations, the program property is turned as the one on the left below, where $\sum_{j=0}^{2^m-1} \frac{1}{\sqrt{2^m}} |j\rangle_m$ being the result of applying m Hadamard operations. The QSV process of determinizing the basis-kets is to select a particular j, which turns the program property to be the right one.

$$\sum_{j=0}^{2^m-1} \frac{1}{\sqrt{2^m}} |j\rangle_m \to \sum_{j=0}^{n-1} \frac{1}{\sqrt{n}} |j\rangle_m \qquad\qquad \forall j \in [0, 2^m) \, , \, \frac{1}{\sqrt{2^m}} |j\rangle_m \to (\frac{1}{\sqrt{n}} |j\rangle_m \wedge j \in [0, n))$$

Essentially, the validation process based on the right property is to assume a single basis-ket $\frac{1}{\sqrt{2^m}} |j\rangle_m$ with a bitstring $j \in [0, 2^m)$, then to validate to see if the output is the bitstring j within range of $[0, n)$ and the associated amplitude being $\frac{1}{\sqrt{n}}$. Via a property-based testing facility, such as QuickChick, by testing the program with enough candidate basis-kets, it will be highly likely that our validator can capture a bug if there is any.

After validation, we compile the program to a quantum circuit via our certified compiler.

1.2 Contributions and Roadmap

We present QSV, a framework that enables programmers to develop state preparation programs. Our contributions are as follows, with all Rocq proofs and experiment results available. We discuss QSV and PQASM limitations in Section 8.

- We present the syntax, semantics, and type system of PQASM, allowing users to define state preparation programs with the proof of type soundness in Rocq (Section 3).

- We develop a property-based testing (PBT) framework for validating programs written in PQASM (Section 4) by showing a general flow of constructing such PBT frameworks for validating quantum programs.
- We certify a compiler from PQASM to SQIR [Hietala et al. 2023] (Section 4.2) to ensure that our PQASM tool correctly reflect quantum program behaviors.
- We evaluate PQASM via a selection of state preparation programs and demonstrate that QSV is capable of validating the programs (Sections 5 and 6). These programs were previously considered to be hard or impossible to simulate on classical machines, and some of them (amplitude amplification, hamming weight, and element distinctness state preparation programs) were never verified or validated. We attempted to run these programs in an industrial quantum simulator, Qiskit and DDSim [Burgholzer et al. 2021] (a key in the Munich Quantum Toolkit [Wille et al. 2024]). None of the state preparation programs are simulatable in the simulators (Section 6) if the input qubit length is a normal one (60 qubits per register and up to 361 qubits as the total input qubit size). In contrast, QSV can effectively run 10,000 test cases to validate these large qubit length programs via QuickChick.

2 Background

Here, we provide background information on Quantum Computing.

Quantum Data. A quantum datum consists of one or more quantum bits (*qubits*), which can be expressed as a two-dimensional vector $\binom{\alpha}{\beta}$ where the *amplitudes* α and β are complex numbers and $|\alpha|^2 + |\beta|^2 = 1$. We frequently write the qubit vector as $\alpha\,|0\rangle_1 + \beta\,|1\rangle_1$ (the Dirac notation [Dirac 1939]), where $|0\rangle_1 = \binom{1}{0}$ and $|1\rangle_1 = \binom{0}{1}$ are *computational basis-vectors* and $\alpha\,|0\rangle_1$ and $\beta\,|1\rangle_1$ are basis-kets. The subscripts indicate the number of qubits for the basis-ket. When no necessity of mentioning qubit numbers, we denote the basis-ket as $|0\rangle$ or $|1\rangle$ by omitting them. When both α and β are non-zero, we can think of the qubit being "both 0 and 1 at once," a.k.a. in a *superposition* [Nielsen and Chuang 2011], e.g., $\frac{1}{\sqrt{2}}(|0\rangle_1 + |1\rangle_1)$ represents a superposition of $|0\rangle_1$ and $|1\rangle_1$. Larger quantum data can be formed by composing smaller ones with the *tensor product* ($\otimes$) from linear algebra, e.g., the two-qubit datum $|0\rangle_1 \otimes |1\rangle_1$ (also written as $|01\rangle_2$) corresponds to vector $[\ 0\ 1\ 0\ 0\ ]^T$. However, many multi-qubit data cannot be *separated* and expressed as the tensor product of smaller data; such inseparable datum states are called *entangled*, e.g. $\frac{1}{\sqrt{2}}(|00\rangle_2 + |11\rangle_2)$, known as a *Bell pair*, which can be rewritten to $\Sigma_{d=0}^{1} \frac{1}{\sqrt{2}}|dd\rangle_2$, where dd is a bit string consisting of two bits, both being the same (i.e., $d = 0$ or $d = 1$). Each term $\frac{1}{\sqrt{2}}\,|dd\rangle_2$ is named a *basis-ket*, consisting an amplitude $\frac{1}{\sqrt{2}}$ and a basis vector $|dd\rangle_2$.

Quantum Computation and Measurement. Computation on a quantum datum consists of a series of *quantum operations*, each acting on a subset of qubits in the quantum datum. In the standard form, quantum computations are expressed as *circuits*, as in Figure 3, which depicts a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [Greenberger et al. 1989] — an n-qubit entangled datum of the form: $|\text{GHZ}^n\rangle = \frac{1}{\sqrt{2}}(\bigotimes^n |0\rangle + \bigotimes^n |1\rangle)$. In these circuits, each horizontal wire represents a qubit, and boxes on these wires indicate quantum operations, or *gates*. The circuit in Figure 3 uses n qubits and applies n gates: a *Hadamard* (H) gate and $n-1$ *controlled-not* (CNOT) gates. Applying a gate to a quantum datum *evolves* it. Its traditional semantics is expressed by multiplying the datum's vector form by the gate's corresponding matrix representation: n-qubit gates are 2^n-by-2^n matrices. Except for measurement gates, a gate's matrix must be *unitary* and thus preserve appropriate invariants of quantum data's amplitudes. A *measurement* operation extracts classical information from a quantum datum. It collapses the datum to a basis-ket with a probability

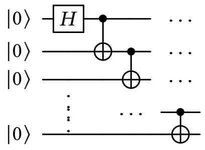


Fig. 3. GHZ Circuit

related to the datum's amplitudes (*measurement probability*), e.g., measuring $\frac{1}{\sqrt{2}}(|0\rangle_1+|1\rangle_1)$ collapses the datum to $|0\rangle_1$ or $|1\rangle_1$, each with probability $\frac{1}{2}$. The ket values correspond to classical values 0 or 1, respectively. A more general form of quantum measurement is *partial measurement*, which measures a subset of qubits in a qubit array; such operations often have simultaneity effects due to entanglement, *i.e.*, in a Bell pair $\frac{1}{\sqrt{2}}(|00\rangle_2 + |11\rangle_2)$, measuring one qubit guarantees the same outcome for the other — if the first bit is measured as 0, the second bit will be measured as 0.

Quantum Oracles. Quantum algorithms manipulate input information encoded in "oracles", which are callable black-box circuits. Quantum oracles are usually quantum-reversible implementations of classical operations, especially arithmetic operations. Their behavior is defined in terms of transitions between single basis-kets. We can infer the global state behavior based on the single basis-ket behavior through the quantum summation formula below. This resembles an array map operation in Figure 2a. OQASM in VQO [Li et al. 2022] is a language that permits the definitions of quantum oracles with efficient verification and testing facilities using the following summation formula:

$$\frac{\forall j \,.\, x_j \longrightarrow f(x_j)}{\Sigma_j \alpha_j \, |x_j\rangle \longrightarrow \Sigma_j \alpha_j f(|x_j\rangle)}$$

Repeat-Until-Success Quantum Programs. A repeat-until-success program utilizes the probabilistic feature of partial measurement operations. We first set up a one-step repeat-until-success by linking the desired quantum state with the success measurement of a certain classical value. If such a value is observed after measurement, we know that the desired state is successfully prepared; otherwise, we repeat the one-step procedure. One example of a one-step repeat-until-success procedure is in Figure 2b to repeat the n basis-ket superposition state. If we measure out $v = 1$, the desired state φ is prepared; otherwise, we repeat the procedure.

No Cloning. The *no-cloning theorem* indicates no general way of exactly copying a quantum datum. In quantum circuits, this is related to ensuring the reversible property of unitary gate applications. For example, the controlled node and controlled body of a quantum control gate cannot refer to the same qubits, e.g., `CU` $q\ \iota$ violates the property if q is mentioned in ι. PQASM enforces no cloning through our type system.

3 PQASM: An Assembly Language for Quantum State Preparations

We designed PQASM to express quantum state preparation programs in a high-level abstraction, facilitating the validation of these programs. PQASM operations leverage a quantum state design, with the type system to track the types of different qubits. Such types restrict the kinds of quantum states, facilitating effective validation and analysis of the PQASM program utilizing our quantum state representations. This section presents PQASM states and the language's syntax, semantics, typing, and soundness results.

As a running example, we program the n basis-ket state preparation in Figure 2b in PQASM below. The repeat-until-success program creates an qubit array $\overline{q}$ consisting of all zeroes and a new qubit q', applies a Hadamard gate to each qubit in $\overline{q}$, uses a comparison operator $(\overline{q}<n)\ @\ q'$ comparing $\overline{q}$ with n and storing the result in q', and measures the qubit q'. If the measurement result is 1, we stop the process, and $\overline{q}$ prepares the correct superposition state; otherwise, we repeat the process. $\mathtt{H}(\overline{q})$ is a syntactic sugar of sequence of Hadamard operations as $\mathtt{H}(\overline{q}[0])\ ;\ \ldots\ ;\ \mathtt{H}(\overline{q}[m-1])$.

Definition 3.1 (n Basis-ket State Preparation Program). Example PQASM program P to prepare n superposition state in $\overline{q}$, whose qubit array length is at least $\log(n) + 1$; q' is a single qubit.

$P \triangleq \mathtt{new}(\overline{q})\ ;\ \mathtt{new}(q')\ ;\ \mathtt{H}(\overline{q})\ ;\ (\overline{q}<n)\ @\ q'\ ;\ \mathtt{let}\ x = \mathcal{M}(q')\ \mathtt{in\ if}\ (x = 1)\ \{\}\ \mathtt{else}\ P$

Qubit Name q		Nat $n, m \in \mathbb{N}$		Real $r \in \mathbb{R}$	
Complex $z \in \mathbb{C}$		Bit $b \in \{0, 1\}$		Bitstring $c ::= \overline{b}$	
Qubit Basis State	v	$::=$	$\lvert b\rangle_1$		$\mid \lvert\Delta(r)\rangle$
Qubit Records	θ	$::=$	$(\overline{q}$	$, \overline{q}$	$, \overline{q})$
Type	τ	$::=$	$\texttt{Had}$	$\mid \texttt{Nor}$	$\mid \texttt{Rot}$
Basis Vector	η	$::=$	$\bigotimes_j v_j$		
Basis-Ket	ρ	$::=$	$z \cdot \eta$		
Quantum Data	φ	$::=$	ρ	$\mid \sum_{b=0}^{1} \varphi$	
Quantum State	Φ	$::=$	$\theta \longrightarrow \varphi$		

Fig. 4. PQASM state syntax. $\overline{S}$ denotes a sequence of S. $\lvert c\rangle_{n+1} \equiv \lvert c[0]\rangle_1 \otimes ... \otimes \lvert c[n]\rangle_1$, with $|c| = n+1$.

3.1 PQASM States

A PQASM program state is represented according to the grammar in Figure 4. A quantum state is managed in terms of qubit records, each of which is a collection of qubits possibly being entangled, while qubits in different records are guaranteed to have no entanglement. A state Φ maps from qubit records θ to a quantum datum φ. Our quantum data consists of a quantum entanglement state that can be analyzed as two portions: 1) a sequence of sum operators $\sum_{b_1=0}^{1} \cdots \sum_{b_n=0}^{1}$, and 2) a basis-ket ρ, a pair of a complex amplitude z and a tensor product of basis vector η, which is a tensor of single qubit basis states v. Each sum operator represents the creation of a superposition state via a Hadamard operation $\texttt{H}$, i.e., the number of sum operators in a state for a qubit record θ represents the number of Hadamard operations applied to qubits in the state so far. The variable $\rho = z \cdot \eta$ represents a basis-ket of a quantum state. To understand the relation between a basis-ket and a whole quantum superposition state connected with a sequence of sum operators, one can think of a superposition state as a collection of "quantum choices", and a basis-ket represents a possible choice, i.e., a measurement of a qubit record produces one possible choice, with the amplitude z related to the probability of the choice.

A qubit basis state v has one of two forms, $\lvert b\rangle_1$ and $\lvert\Delta(r)\rangle$. The former corresponds to the two types $\texttt{Had}$ and $\texttt{Nor}$, and the latter corresponds to the $\texttt{Rot}$ type. The three types of qubit basis states are represented as the three fields in a qubit record, i.e., $(\overline{q}_1, \overline{q}_2, \overline{q}_3)$ has three disjoint qubit sequences. $\overline{q}_1$ is always typed as $\texttt{Had}$, $\overline{q}_2$ has type $\texttt{Nor}$, and $\overline{q}_3$ has type $\texttt{Rot}$. The $\texttt{Had}$ and $\texttt{Nor}$ typed qubits are in the computational basis. The $\texttt{Rot}$ typed basis state is different from the other types in terms of *bases*, and it has the form $\lvert\Delta(r)\rangle = \texttt{cos}(r)\,\lvert 0\rangle_1 + \texttt{sin}(r)\,\lvert 1\rangle_1$, which is a basis state in the Hadamard basis with Y-axis rotations. Applying a $\texttt{Ry}$ with the Y-axis angle r to a $\lvert 0\rangle_1$ qubit results in $\lvert\Delta(r)\rangle = \texttt{cos}(r)\,\lvert 0\rangle + \texttt{sin}(r)\,\lvert 1\rangle$.

3.2 PQASM Syntax

Classical Variable x, y		Boolean Expressions B	
Parameters	α	$::=$	$\overline{q} \mid n$
OQASM Arithemtic Ops	μ	$::=$	$\texttt{add}(\alpha, \alpha) \mid (n * \alpha)\,\%\,m \mid (\alpha = \alpha)\,@\,q \mid (\alpha < \alpha)\,@\,q \mid ...$
Instruction	ι	$::=$	$\mu \mid \texttt{Ry}^r q \mid \texttt{CU}\ q\ \iota \mid \iota\ ;\ \iota$
Program	e	$::=$	$\iota \mid e\ ;\ e \mid \texttt{H}(q) \mid \texttt{new}(q) \mid \texttt{let}\ x = \mathcal{M}(\overline{q})\ \texttt{in}\ e \mid \texttt{if}\ (B)\ e\ \texttt{else}\ e$

Fig. 5. PQASM syntax.

Figure 5 presents PQASM's syntax. A PQASM program e is either an instruction ι, a sequence operation $e\ ;\ e$, applying a Hadamard operation $\texttt{H}(q)$ to a qubit q to create a superposition, creating ($\texttt{new}(q)$) a new blank qubit q, a $\texttt{let}$ binding that measures a sequence of qubits $\overline{q}$ and uses the result x in e, or classical conditional $\texttt{if}\ (B)\ e\ \texttt{else}\ e$ with classical Boolean guard B. Each $\texttt{let}$ binding

$$\begin{array}{lll}
[\![\mu]\!]\eta & = \eta[\overline{q} \mapsto [\![\mu]\!]\eta(\overline{q})] & \texttt{where } FV(\mu) = \overline{q} \\
[\![\texttt{Ry}^r q]\!]\eta & = \eta[q \mapsto |\Delta(r)\rangle] & \texttt{where } \eta(q) = |0\rangle\, 1 \\
[\![\texttt{Ry}^r q]\!]\eta & = \eta[q \mapsto |\Delta(\frac{3\pi}{2} - r)\rangle] & \texttt{where } \eta(q) = |1\rangle\, 1 \\
[\![\texttt{Ry}^r q]\!]\eta & = \eta[q \mapsto |\Delta(r + r')\rangle] & \texttt{where } \eta(q) = |\Delta(r')\rangle \\
[\![\texttt{CU}\ q\ \iota]\!]\eta & = \texttt{cu}(\eta(q), \iota, \eta) & \texttt{where } \texttt{cu}(|0\rangle\, 1, \iota, \eta) = \eta \quad \texttt{cu}(|1\rangle\, 1, \iota, \eta) = [\![\iota]\!]\eta \\
[\![\iota_1 ; \iota_2]\!]\eta & = [\![\iota_2]\!]([\![\iota_1]\!]\eta) &
\end{array}$$

$$\eta[\overline{q} \mapsto \eta'] = \eta[\forall q \in \overline{q}.\ q \mapsto \eta'(q)]$$

Fig. 6. Instruction level PQASM semantics; $\eta(\overline{q})$ restricts the qubit states $\overline{q}$ in η.

assigns the measurement result of a quantum function expression to a variable x, representing a binary sequence. In measurement operations ($\mathcal{M}$), we apply an operator to a *qubit* q or a sequence of quantum qubits $\overline{q}$. We assume $\texttt{H}(\overline{q})$ and $\texttt{new}(\overline{q})$ as syntactic sugars of applying a sequence of Hadamard and new-qubit operations.

The instructions ι correspond to unitary quantum circuit operations, including unitary oracle arithemtic operations (μ) implementable through OQASM operations [Li et al. 2022] on a qubit sequence $\overline{q}$ (detailed in Appendix, and it permits Z-axis rotation gates), a Y-axis rotation gate $\texttt{Ry}^r q$ that rotates an angle r, a quantum control instruction ($\texttt{CU}\ q\ \iota$), and a sequence operation (ι ; ι). Operation $\texttt{CU}\ q\ \iota$ applies instruction ι *controlled* on qubit q. In this paper, we provide several sample arithmetic oracle operations μ in Figure 5, such as addition ($\texttt{add}(\alpha, \alpha)$, adding the first to the second), modular multiplication ($(n * \alpha)\ \%\ m$), quantum equality ($(\alpha = \alpha)\ @\ q$), quantum comparison ($(\alpha < \alpha)\ @\ q$), etc. Each parameter α is either a group of qubits $\overline{q}$ or a number n. Recall that a basis-ket state of a qubit array $\overline{q}$ is essentially a bitstring with a complex amplitude. As in the summation formula in Section 2, a quantum arithmetic operation applies the classical version of the operation to each basis-ket in a quantum superposition state, e.g., $(\overline{q} = n)\ @\ q$ compares the bitstring representation of each basis-ket in $\overline{q}$ with the number n and stores the result in q.

In a PQASM program containing qubit array $\overline{q}$, x in a `let` binding binds a local classical value — we bind x's value with the computational basis measurement result ($\mathcal{M}$) on qubits $\overline{q}$. While the classical variable scope is local, the quantum qubits are immutable and globally scoped, i.e., quantum operations are applied to a global quantum state; each qubit in the state is referred to by quantum qubit names (q) in the program. In PQASM, we express a SKIP operation ($\{\}$) via a μ operation having empty qubits, as $\mu\ \equiv\ \{\}\ \texttt{when}\ FV(\mu) = \emptyset$ (FV collects free variables).

3.3 Semantics

PQASM has two levels of semantics: instruction and program levels. The *instruction* level semantics is a partial function $[\![-]\!]$ from an instruction ι and input basis vector state ρ to an output state η', written $[\![\iota]\!]\eta = \eta'$, shown in Figure 6. The *program* level semantics is a labelled transition system $(\Phi, e) \xrightarrow{r} (\Phi', e')$ in Figure 7, stating that the input configuration (Φ, e) is possibly evaluated to an output configuration (Φ', e') with the probability r. It essentially represents a Markov chain, where a program evaluation path represents a chain of probabilities, showing the probability path leads to a particular configuration from the initial configuration.

In the instruction level semantics, the semantic rule description assumes that one can locate a qubit state q in η as $\eta(q)$, where we can refer to $\eta[q \mapsto v]$ as updating the qubit state v for the qubit q in η. Recall that a length n basis vector state η is a tuple of n qubit values, modeling the tensor product $v_1 \otimes \cdots \otimes v_n$. The rules implicitly map each qubit variable q to a qubit value position in the state, e.g., $\eta(q)$ corresponds to some sub-state v_q, where v_q locates at the q's position in η. Many of the rules in Figure 6 update a *portion* of a state. We write $\eta[q \mapsto v_q]$ to update the qubit value of q in η with v_q, and $\eta[\overline{q} \mapsto \eta']$ to update a range of qubits $\overline{q}$ according to the vector state η', S-Ins

$$\frac{b=b_1,\dots,b_n \qquad \varphi=\sum_{b_1=0}^{1}\dots\sum_{b_n=0}^{1} z_b\cdot\eta_b \qquad [\![\iota]\!](\eta_b)=\eta'_b}{(\Phi[\theta\mapsto\varphi],\iota)\xrightarrow{1}(\Phi[\theta\mapsto\sum_{b_1=0}^{1}\dots\sum_{b_n=0}^{1} z_b\cdot\eta'_b],\{\})}$$

S-SeqC

$$\frac{(\Phi,e_1)\xrightarrow{r}(\Phi',e'_1)}{(\Phi,e_1\ ;\ e_2)\xrightarrow{r}(\Phi',e'_1\ ;\ e_2)}$$

S-SeqT

$$(\Phi,\{\}\ ;\ e_2)\xrightarrow{1}(\Phi,e_2)$$

S-New

$$(\Phi,\texttt{new}(q))\xrightarrow{1}(\Phi[(\emptyset,q,\emptyset)\mapsto|0\rangle_1],\{\})$$

S-IfT

$$(\Phi,\texttt{if (true)}\ e_1\ \texttt{else}\ e_2)\xrightarrow{1}(\Phi,e_1)$$

S-IfF

$$(\Phi,\texttt{if (false)}\ e_1\ \texttt{else}\ e_2)\xrightarrow{1}(\Phi,e_2)$$

S-Had

$$(\Phi[(\emptyset,q,\emptyset)\mapsto|b\rangle_1],\texttt{H}(q))\xrightarrow{1}(\Phi[(q,\emptyset,\emptyset)\mapsto\sum_{j=0}^{1}(-1)^{j\cdot b}\,|j\rangle_m],\{\})$$

S-Mea

$$\frac{\Phi=\Phi'\uplus\{\uparrow\overline{q}\mapsto\sum_j z_j\,|c\rangle_m\,|c_j\rangle_n+\phi\langle\overline{q}',c\neq\overline{q}'\rangle\} \qquad r=\sum_j|z_j|^2}{(\Phi,\texttt{let}\ x=\mathcal{M}(\overline{q})\ \texttt{in}\ e)\xrightarrow{r}(\Phi'\uplus\{(\uparrow\overline{q})\backslash\overline{q}:\sum_j\frac{z_j}{\sqrt{r}}|c_j\rangle_m\},e[c/x])}$$

$$\uparrow\overline{q} \quad\triangleq\quad \exists\overline{q}_1,\overline{q}_2,\overline{q}_3\,.\,\uparrow\overline{q}=(\overline{q}_1,\overline{q}_2,\overline{q}_3)\wedge\overline{q}\subseteq\overline{q}_1\uplus\overline{q}_2\uplus\overline{q}_3$$

$$(\Sigma_i\,z_i|c_i\rangle_m|c'_i\rangle_n+\varphi)\langle\overline{q},b\rangle \quad\triangleq\quad \Sigma_i\,z_i|c_i\rangle_m|c'_i\rangle_n \qquad \texttt{where}\quad\forall i.\,|c_i|=|\overline{q}'|=m\wedge[\![b[c_i/\overline{q}']]\!]=\texttt{true}$$

Fig. 7. Program Level PQASM semantics.

i.e., we update each $q\in\overline{q}$ with the qubit value $\eta'(q)$ and $|\eta'|=|\overline{q}|$. The function cu is a conditional operation depending on the Nor/Had typed qubit q.

Figure 7 shows the program level semantics. Φ is the quantum state mapping from qubit records to superposition state values. Since qubit records in Φ partition the qubit domain, we can think of Φ as a multiset of pairs of qubit records and state values, as in S-Mea, i.e., $\Phi[\theta\mapsto\varphi]\equiv\Phi\uplus\{\theta\mapsto\varphi\}$. Rule S-Ins connects the instruction level semantics with the program level by evaluating each basis vector state η through the instruction ι. Rule S-New creates a new blank ($|0\rangle_1$) qubit, which is stored in the record $(\emptyset,q,\emptyset)$, a qubit (q) being created are Nor typed. For a Nor typed qubit $(\emptyset,q,\emptyset)$, rule S-Had turns the qubit to be Had typed superposition, as $(q,\emptyset,\emptyset)$.

Rules S-IfT and S-IfF perform classical conditional evaluation. In PQASM, the classical variables are evaluated via a substitution-based approach, as in S-Mea. The measurement rule (S-Mea) produces a probability r label, and the value comes from the measurement result. We first rewrite the quantum state to be a linear sum of computational basis-kets $\Sigma_j\,r_j\,|c\rangle_m\,|c_j\rangle_n+\varphi\langle\overline{q}',c\neq\overline{q}'\rangle$, where every basis-vector $|c\rangle_m$ (or $|c_j\rangle_n$) is a bitstring, and all the sum operators are resolved as a single sum.

Any PQASM state can be written as a sum of computational basis-kets. As the equations shown below, the basis-ket state $|c\rangle_n\,|\Delta(r)\rangle$ can be rewritten to be a sum of two computational basis-kets as $\cos(r)\,|c\rangle_n\,|0\rangle_1+\sin(r)\,|c\rangle_n\,|1\rangle_1$, while the two sum operators can be replaced as a single sum operator over length-2 bitstring c, where we replace b_j with $c[0]$ (indexing 0 of c) and b_k with $c[1]$.

$$|c\rangle_n\,|\Delta(r)\rangle\equiv\cos(r)\,|c\rangle_n\,|0\rangle_1+\sin(r)\,|c\rangle_n\,|1\rangle_1 \qquad \Sigma_{b_j=0}^{1}\,\Sigma_{b_k=0}^{1}\,\eta\equiv\Sigma_{c\in\{0,1\}^2}\,\eta[c[0]/b_j][c[1]/b_k]$$

3.4 Typing

In PQASM, typing is with respect to a *type environment* Ω, a set of qubit records partitioning qubits into different disjoint union regions, and a *kind environment* Σ, a set tracking local variable scopes. Typing judgments are two leveled, and are written as $\Omega\vdash\iota\triangleright_g\Omega'$ and $\Sigma;\Omega\vdash e\triangleright\Omega'$, which state that instruction ι and program expression e are well-typed under environments Ω and Σ, and transforms variables' bases as characterized by type environment Ω'. Ω is populated via qubit creation operations (new), while Σ is populated via `let` binding. Typing rules are in Figure 8.

Eqv
$$\frac{\Omega \equiv \Omega' \qquad \Omega' \vdash_g e \triangleright \Omega''}{\Omega \vdash_g e \triangleright \Omega''}$$

RyN
$$\{(\emptyset,\{q\},\emptyset)\} \uplus \Omega \vdash_{\mathsf{C}} \mathsf{Ry}^r q \triangleright \{(\emptyset,\emptyset,\{q\})\} \uplus \Omega$$

RyH
$$\frac{\mathsf{Rot}(\theta) = \{q\} \uplus \overline{q}}{\{\theta\} \uplus \Omega \vdash_g \mathsf{Ry}^r q \triangleright \{\theta\} \uplus \Omega}$$

MuT
$$\frac{\overline{q} \subseteq \overline{q_1} \cup \overline{q_2}}{\{(\overline{q_1},\overline{q_2},\overline{q_3})\} \uplus \Omega \vdash_g \mu(\overline{q}) \triangleright \{(\overline{q_1},\overline{q_2},\overline{q_3})\} \uplus \Omega}$$

CuN
$$\frac{\{\mathsf{Nor}(\theta) \downarrow \overline{q}\} \uplus \Omega \vdash_{\mathsf{M}} \iota \triangleright \{\mathsf{Nor}(\theta) \downarrow \overline{q}\} \uplus \Omega}{\{\mathsf{Nor}(\theta) \downarrow \{q\} \uplus \overline{q}\} \uplus \Omega \vdash_g \mathsf{CU}\ q\ \iota \triangleright \{\mathsf{Nor}(\theta) \downarrow \{q\} \uplus \overline{q}\} \uplus \Omega}$$

CuH
$$\frac{\{\mathsf{Had}(\theta) \downarrow \overline{q}\} \uplus \Omega \vdash_{\mathsf{M}} \iota \triangleright \{\mathsf{Had}(\theta) \downarrow \overline{q}\} \uplus \Omega}{\{\mathsf{Had}(\theta) \downarrow \{q\} \uplus \overline{q}\} \uplus \Omega \vdash_g \mathsf{CU}\ q\ \iota \triangleright \{\mathsf{Had}(\theta) \downarrow \{q\} \uplus \overline{q}\} \uplus \Omega}$$

Seq
$$\frac{\Omega \vdash_g \iota_1 \triangleright \Omega' \qquad \Omega' \vdash_g \iota_2 \triangleright \Omega''}{\Omega \vdash_g \iota_1\ ;\ \iota_2 \triangleright \Omega''}$$

ESeq
$$\frac{\Sigma;\Omega \vdash e_1 \triangleright \Omega' \qquad \Sigma;\Omega' \vdash e_2 \triangleright \Omega''}{\Sigma;\Omega \vdash e_1\ ;\ e_2 \triangleright \Omega''}$$

New
$$\frac{q \notin \Omega}{\Sigma;\Omega \vdash \mathsf{new}(q) \triangleright \Omega \uplus \{(\emptyset,q,\emptyset)\}}$$

HT
$$\Sigma;\Omega \uplus \{(\emptyset,q,\emptyset)\} \vdash \mathsf{H}(q) \triangleright \Omega \uplus \{(q,\emptyset,\emptyset)\}$$

TUp
$$\frac{\Omega \vdash_{\mathsf{C}} \iota \triangleright \Omega'}{\Sigma;\Omega \vdash \iota \triangleright \Omega'}$$

Mea
$$\frac{\overline{q} \subseteq \theta \qquad \Sigma \cup \{x\};\Omega \uplus \{\theta \backslash \overline{q}\} \vdash e \triangleright \Omega'}{\Sigma;\Omega \uplus \{\theta\} \vdash \mathsf{let}\ x = \mathcal{M}(\overline{q})\ \mathsf{in}\ e \triangleright \Omega'}$$

TIf
$$\frac{\Sigma \vdash B \qquad \Sigma;\Omega \vdash e_1 \triangleright \Omega' \qquad \Sigma;\Omega \vdash e_2 \triangleright \Omega'}{\Sigma;\Omega \vdash \mathsf{if}\ (B)\ e_1\ \mathsf{else}\ e_2 \triangleright \Omega'}$$

Fig. 8. PQasm typing rules. $(\overline{q}_1,\overline{q}_2,\overline{q}_3)\backslash\overline{q} \triangleq (\overline{q}_1\backslash\overline{q},\overline{q}_2\backslash\overline{q},\overline{q}_3\backslash\overline{q})$, where $\overline{q}_1\backslash\overline{q}$ is set subtraction.

The instruction level type system is flow-sensitive, where g is the context flag and can be either M or C, indicating whether the current instruction is inside a controlled operation. The program level type system communicates with the instruction level by assuming a C mode context flag, shown in rule TUp. We explain the necessity of the context flag below. Each qubit record θ represents an entanglement group, i.e., qubits in the same record might or might not be entangled, while qubits in different records are ensured not to be entangled.

$$\begin{array}{ll} (\overline{q_1},\overline{q_2},\overline{q_3}) \uplus (\overline{q_4},\overline{q_5},\overline{q_6}) & \equiv (\overline{q_1} \uplus \overline{q_4},\overline{q_2} \uplus \overline{q_5},\overline{q_3} \uplus \overline{q_6}) \\ (\emptyset,\overline{q_1} \uplus \overline{q_2},\overline{q_3} \uplus \overline{q_4}) & \equiv (\emptyset,\overline{q_1},\overline{q_3}) \uplus (\emptyset,\overline{q_2},\overline{q_4}) \end{array}$$

In our type system, we permit ordered equational rewrites among quantum qubit states. Each type environment, mainly the operation $\uplus$, admits associativity, commutativity, and identity equational properties. The $\uplus$ operations in the three fields in a qubit record also admit the three properties. Other than the three equational properties, we admit the above partial order relations, where we permit the rewrites from left to right in our type system to permit qubit records merging and splitting. Record merging can always happen, i.e., two qubit entanglement groups can be merged into one. Qubit splitting cannot occur in Had typed qubits. A qubit record, including a Had typed qubit, represents a quantum entanglement with qubits not separable, while qubits being Nor and Rot typed can be split into different records. Rule Eqv imposes the equivalence relation to permit the rewrites, only allowing rewrites from left to right, of equivalent qubit records. Note that a type environment determines qubit record scopes in a quantum state φ (Figure 4), i.e., a quantum state should have the same qubit record domain as the type environment at a program point, so the equational rewrite of a type environment might affect the qubit state representation.

Other than the equational rewrites, the type system enforces three properties. First, it enforces that classical and quantum variables are properly scoped. Rule Mea includes the local variable x in Σ, while rule TIf ensures that any variables mentioned in B are properly scoped by the constraint $FV(B) \subseteq \Sigma$ [1], i.e., all free variables in B are bounded by Σ. Note that the two branches of TIf have the same output Σ', i.e., the qubit manipulations in the two branches need to have the same effect. If one branch has a measurement on a qubit, the other branch must have the same qubit measurement.

[1]The rule is parameterized by different B formalism.

Rule New creates a new record $(\emptyset, q, \emptyset)$ in the post-environment, provided that q does not appear in any record in Ω. In rule RyH, the premise $\texttt{Rot}(\theta) = \{q\} \uplus \overline{q}$ utilizes Rot to finds the Rot filed in the record θ and ensures that q is in the field. In rule CuH, the premise $\texttt{Had}(\theta) \downarrow \{q\} \uplus \overline{q}$ ensures that the controlled position q is in the Had field in θ. In rule MuT, we ensure that the qubits $\overline{q}$ being applied by the μ operation are Nor and Had typed, through the premise $\overline{q} \subseteq \overline{q_1} \cup \overline{q_2}$.

Second, we ensure that expressions and instructions are well-formed, i.e., any control qubit is distinct from the target(s), which enforces the quantum *no-cloning rule*. In rules CuH and CuN for control operations, when typing the target instruction ι (the upper level), we remove the control qubit q in the records to ensure that q cannot be mentioned in ι. In rule Mea, we also remove the measured qubits $\overline{q}$ from the record θ.

Third, the type system enforces that expressions and instructions leave affected qubits in a proper type (Nor, Had, and Rot), representing certain forms of qubit states, mentioned in Figure 4; therefore, one can utilize the procedure mentioned in Section 1 to analyze PQasm programs effectively. The key is to utilize the summation formula to reduce the analysis of a general quantum state to that of a quantum state without entanglement. Specifically, the $\texttt{Ry}^r q$ opeartion is permitted only if q is of Nor type, which is turned to Rot type and stays there; μ can be applied to Nor typed qubits $\overline{q}$ where $FV(\mu) = \overline{q}$; and a control qubit q in CU q ι can be applied to a Nor and Had typed qubit.

We ensure type restrictions for qubits via pre- and post-type environments. Rule HT permits the generation of Had type qubits, a.k.a. superposition qubits q, provided that q is of Nor type and not entangled with other qubits. Once a Hadamard operation is applied, we turn the qubit types to Had in the post-type environment, so one cannot apply Hadamard operations again to the qubits. This does not mean that users can only apply Hadamard operations once in PQasm, because combining X (our oracle operation) and Ry gates can produce a Hadamard gate. We utilize the type information to locate the first appearance of Hadamard operations, identify them as the source of superposition, and apply treatments for them in our validation testing framework; see Section 4.1.

In controlled operations (rules CuN and CuH), we ensure that the pre- and post-type environments are the same. In CU q ι, if ι contains a Ry operation, applying to a qubit q, q must already be Rot type. Figure 9 provides a programming prototype satisfying this type requirement, where programmers explicitly add a Ry gate before the controlled Ry operation to ensure the second qubit is in Rot type. The extra Ry operation can be a 0 rotation, equivalent to a SKIP operation, and can be removed by an optimizer when compiling to quantum circuits. The above qubit type restriction does not depend on the applications of controlled operations. We ensure this by associating the context flags with the instruction level type system. When applying a CU q ι operation, rules CuN and CuH turn the context flag to M, indicating that ι lives inside a controlled operation. Rule RyN requires a context flag C, meaning that the rule is valid only if the Ry operation lives outside any controlled operation. In contrast, rule RyH does not require a specific context flag, which indicates that a Ry operation inside a controlled node must apply to a qubit already in Rot type.

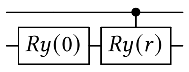


Fig. 9. Ensuring qubits inside a controlled Ry have the same type.

Soundness. We prove that well-typed PQasm programs are well defined; i.e., the type system is sound with respect to the semantics. The type soundness theorem relies on a well-formed definition of a program e, $FV(e) \subseteq \Sigma$, meaning that all free variables in e are bounded by Σ. We also need the definition of the well-formedness of an PQasm state as follows.

Definition 3.2 (Well-formed PQasm state). A state Φ is *well-formed*, written $\Omega \vdash \Phi$, iff:

- For every q such that $\Omega(q) = \texttt{Nor}$ or $\Omega(q) = \texttt{Had}$, $\Phi(q)$ has the form $|b\rangle_1$.
- For every q such that $\Omega(q) = \texttt{Rot}$, $\Phi(q)$ has the form $|\Delta(r)\rangle$.

Type soundness is stated as two theorems: type progress and preservation; the proof is by induction on ι and is mechanized in Rocq.

Theorem 3.3. [Pqasm Type Progress] If $\emptyset; \Omega \vdash e \triangleright \Omega'$, $FV(e) \subseteq \emptyset$, and $\Omega \vdash \Phi$, then either $e = \{\}$ or there exists r, e', and Φ', such that $(\Phi, e) \xrightarrow{r} (\Phi', e')$.

Proof. Fully mechanized proofs were done by induction on type rules using Rocq. □

Theorem 3.4. [Pqasm Type Preservation] If $\Sigma; \Omega \vdash e \triangleright \Omega'$, $FV(e) \subseteq \Sigma$, $\Omega \vdash \Phi$ and $(\Phi, e) \xrightarrow{r} (\Phi', e')$, then there exists Ω_a, such that $\Sigma; \Omega_a \vdash e' \triangleright \Omega'$ and $\Omega_a \vdash \Phi'$.

Proof. Fully mechanized proofs were done by induction on type rules using Rocq. □

4 QSV Applications

This section presents QSV applications, based on Pqasm programs, to validate and compile the programs. We start by discussing QSV's PBT framework for Pqasm programs. Next, we consider translation from Pqasm to SQIR and proof of its correctness.

4.1 PQASM Typing for Effectively Validating State Preparation Programs

The QSV validator is built on PBT to give assurance that a Pqasm program property is correct by attempting to falsify it using thousands of randomly generated instances with good coverage of the program's input space. We have used PBT to validate the correctness of a variety of state preparation programs, presented in Section 5. Below, we show our validator construction.

Pqasm's state representation and type system ensure that states can be represented effectively. We leverage this fact to implement a validation framework for Pqasm programs using QuickChick [Paraskevopoulou et al. 2015], a property-based testing (PBT) framework for Rocq in the style of Haskell's QuickCheck [Claessen and Hughes 2000]. We use this framework for two purposes: to validate the correctness properties of Pqasm programs and to experiment with effective implementations of correct state preparation programs. Pqasm contains measurement operations, which, due to the randomness inherent in quantum measurement, are difficult to test effectively, even with the assistance of program abstractions. To have an effective validation framework, we restrict the properties that can be questioned to solely focus on the properties related to program correctness.

Implementation. PBT randomly generates inputs using a hand-crafted *generator* and confirms that a property holds for these inputs. We develop a validation toolchain, based on the methodology in Section 1.1, using the symbolic state representation concept and carefully selecting the properties to validate for a program.

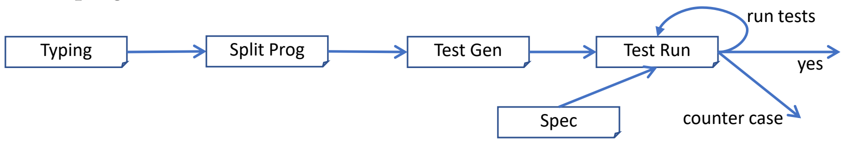


Fig. 10. The Flow of PBT for QSV

The flow of the QSV PBT framework is given in Figure 10. To validate a Pqasm program, we first utilize our Pqasm type system to generate a type environment Ω for qubits in a program e, i.e., $\emptyset; \emptyset \vdash e \triangleright \Omega$, typing with an empty kind and type environment. Essentially, Ω partitions all qubits used in e into three sets $(\overline{q_1}, \overline{q_2}, \overline{q_3})$, with $\overline{q_1}$ containing all qubits in Had type. In Pqasm, once a qubit is turned into Had, it stays in the type. We utilize the property to locate all the Had typed qubits in a program e and generate random testing data based on these qubits, i.e., we identify the set $\overline{q_1}$ in Ω as the set of Had typed qubits and generate random boolean values in $\{0, 1\}$ for variables in the set.

Second, we assume that a program is in the form of $e = \texttt{new}(\overline{q})\ ;\ \texttt{H}(\overline{q})\ ;\ e'$ [2], i.e., all the new and H operations appear in the front of the program and e' does not contain any such operations. We then split the program by taking the e' part and removing the new and H operations, and assuming that these operations have been applied. For some program patterns, such as repeat-until-success programs, we replace recursive process variables in e' with {} operations so that we only validate one step of a repeat-until-success, because every recursive step in these programs is interdependent. One example of programming splitting for Definition 3.1 is given below; it removes the part $\texttt{new}(\overline{q})\ ;\ \texttt{new}(q')\ ;\ \texttt{H}(\overline{q})$ and replacing P with {}.

$$P' = (\overline{q} < n)\,@\,q'\ ;\ \texttt{let}\ x = \mathcal{M}(q')\ \texttt{in}\ \texttt{if}\ (x = 1)\ \{\}\ \texttt{else}\ \{\}$$

The "Test Gen" step in our PBT framework (Figure 10) generates test cases to validate the key component P' above. In Definition 3.1, after applying the new and H operations, the post-state is (Φ, P'), with Φ mapping θ, entanglement groups, to superposition states φ. Each H operation generates a single qubit uniformly distributed superposition state. However, transitions over a superposition state make it hard to perform effective validation. To resolve this, we treat quantum program operations P' as higher-order map operations and validate the transition correctness based on basis-kets, instead of validating over the whole superposition state.

Generally, φ can be written in the Dirac notation of $\sum_j \rho_j$ with $\rho_j = z_j \cdot \eta_j$, i.e., given $\theta = (b_0, b_1, ..., b_m, \overline{q_2}, \overline{q_3})$, the superposition state φ could also be written as the follows.

$$\textstyle\sum_{b_0=0}^{1} \sum_{b_1=0}^{1} \cdots \sum_{b_m=0}^{1} z(b_0, b_1, ..., b_m) \cdot \eta(b_0, b_1, ..., b_m)$$

Here, b_0, b_1, ..., b_m are qubit variables assumed to be already manipulated by the initial H operations, e.g., P' above assumes that $\overline{q}$ were manipulated by H operations. Applying a H operation to a Nor typed qubit $|b_a\rangle_1$ creates a uniformly distributed superposition $\sum_{b=0}^{1} \frac{1}{\sqrt{2}}(-1)^{b \cdot b_a} |b\rangle_1$, so it results in the state form above, with $z(b_0, b_1, ..., b_m)$ being an amplitude formula and $\eta(b_0, b_1, ..., b_m)$ being a basis-vector formula. We can then select the symbolic basis-ket state $z(b_0, b_1, ..., b_m) \cdot \eta(b_0, b_1, ..., b_m)$ as the representative basis-ket and rewrite the φ state to be in the form $(\theta \longrightarrow z \cdot \eta, P')$ for validation, with b_0, b_1, ..., b_m acting as random variables. For each random variable b_j, we can randomly choose the value $b_j \in \{0, 1\}$ for a particular test instance. Thus, for m qubits, we have 2^m different test instances depending on the different value selection for the random variables.

For the P' program above, we view each element in the m-qubit array $\overline{q}$ as a random variable. Then, we generate an initial state $(\theta \longrightarrow z(\overline{q}[0], ..., \overline{q}[m-1]) \cdot \eta(\overline{q}[0], ..., \overline{q}[m-1]), P')$ with $\overline{q}[0], ..., \overline{q}[m-1]$ being random variables and $\theta = (\overline{q}, q, \emptyset)$. We can then generate test instances for choosing different variables for $\overline{q}[j]$ with $j \in [0, m)$.

Once we randomly generate test instances, we can then validate the program by running each test instance in our PQASM interpreter, developed based on our program semantics. The result of the interpreter is provided as input to a specification checker to validate if the specification is satisfied. If the checker makes all test instances answer `true`, we validate the program; otherwise, we report a fault in the program. Below, we show the construction of the specification checker to validate the program's correctness and other properties.

Validating Correctness. We first run a test instance in our interpreter with the initial state, as $(\theta \mapsto z \cdot \eta, P) \longrightarrow^* (\theta' \mapsto z' \cdot \eta', \{\})$, where $\theta \equiv \theta'$. For a user-specified property ψ, a satisfiability check is applied to $\psi(z \cdot \eta, z' \cdot \eta')$ by replacing variables with $z \cdot \eta$ and $z' \cdot \eta'$. Recall that we demonstrate a transformation of correctness property in Section 1 for the program in Definition 3.1, to conduct validation on individual basis-kets rather than the whole quantum state. Such property transformations can be summarized as the transformation from (1) to (3) as below:

$$(1)\ \textstyle\sum_j z_j \cdot \eta_j \rightarrow f(\sum_j z_j \cdot \eta_j) \qquad (2)\ \sum_j z_j \cdot \eta_j \rightarrow \sum_k g(z_k \cdot \eta_k) \wedge \phi(k) \qquad (3)\ \forall j. z_j \cdot \eta_j \rightarrow g'(z_j \cdot \eta_j) \wedge \phi(j)$$

[2] $\texttt{new}(\overline{q})$ and $\texttt{H}(\overline{q})$ are syntactic sugar for multiple $\texttt{new}(q)$ and $\texttt{H}(q)$ operations.

The correctness property should be written in the format of (3). The property (1) describes the program semantics, i.e., given a superposition state $\sum_j r_j \cdot \eta_j$, f represents the semantic function for a program e. The effects of such a function can always be in the form of a linear sum by moving the sum operator to the front, as in (2): one can always find g such that $f(\sum_j r_j \cdot \eta_j) = \sum_k g(r_k \cdot \eta_k)$. In some cases, we might need to insert the $\phi(k)$ predicate above, constraining index k in the sum operator. Here, both j and k are indices for two different sum operators. Often, the function g can be turned into an equivalent form g' based on the index j. Once we equate the two indices, we can then transform the formula to (3) without any sum operators, via the axiom of extensionality. Such transformation might come with the index restriction ϕ based on index j, as shown in (3), which refers to the fact that we start with a superposition state with a representative basis-ket state $r_j \cdot \eta_j$ and output a basis-ket state $g'(r_j \cdot \eta_j)$, with the post-state index restriction $\phi(j)$.

To validate P' above, we transform the correctness property from the left to the right one below ($\Rightarrow$ is logic implication).

$$x = 1 \Rightarrow \sum_j^{2^m} \frac{1}{\sqrt{2^m}} |j\rangle_m |0\rangle_1 \to \sum_j^n \frac{1}{\sqrt{n}} |j\rangle_m \qquad \forall j \in [0, 2^m) \,.\, x = 1 \Rightarrow |j\rangle_m |0\rangle_1 \to |j\rangle_m \wedge j < n$$

The right property states that, if the measurement results in 1 ($x = 1$), each basis-ket in the pre-state for $\overline{q}$ and q respectively have the basis vector forms $|j\rangle_m$ and $|0\rangle_1$, and results in $\overline{q}$ being the same $|j\rangle_m$ with the restriction $j < n$. The $\otimes m$ and $\otimes 1$ flags refer to the number of qubits in quantum array variables $\overline{q}$ and q. In implementing a validation property, the flag essentially indicates the length of a bitstring piece, which is cast into a natural number for comparison. To validate the correctness property against P', we create a length m bitstring for $|j\rangle_m$ and view $j[k]$, for $k \in [0, m)$, being the k-th bit in the bitstring. Recall that P' has an initial basis vector state pattern as $|\overline{q}[0]\rangle_1 \dots |\overline{q}[m-1]\rangle_1 |0\rangle_1$. Here, $|0\rangle_1$ is the state for qubit q and $\overline{q}[0], \dots, \overline{q}[m-1]$ are random variables for qubit array $\overline{q}$. To check the property, we bind each $j[k]$ with $\overline{q}[k]$ for $k \in [0, m)$, and see if the output basis-ket state results in the same $\overline{q}[0], \dots, \overline{q}[m-1]$. We also check if $\overline{q}$'s natural number representation is less than n, i.e., we turn $\overline{q}[0], \dots, \overline{q}[m-1]$ to a number and compare it with n.

Validating Other Properties. The above procedure is only useful in validating correctness. There might be other interesting properties, such as probability and effectiveness properties. For example, in validating Definition 3.1, we might want to ask how likely the qualified state can be prepared, which is hard to validate in general, but it can be effectively sampled out in some cases.

$$x = 1 \Rightarrow \sum_j^{2^m} \frac{1}{\sqrt{2^m}} |j\rangle_m |0\rangle_1 \to \sum_j^n \frac{1}{\sqrt{n}} |j\rangle_m$$

In the superposition state-based property for P' above, the number of qubits in $\overline{q}$ is m and $n \in [0, 2^m)$. Here, let's see how to validate the effectiveness of the program, i.e., the probability that the repeat-until-success program produces the correct state. Note that superposition states are always uniformly distributed without any Ry operations. The success rate of preparing a superposition state in the repeat-until-success scheme is the ratio between the number of possible basis-vector values less than n and the total number of possible basis-vector values, i.e., $\overline{q}[0], \dots, \overline{q}[m-1]$. We can validate the effectiveness by calculating the number of possible values $\overline{q}$ less than n, by interpreting $\overline{q}$ as a natural number, dividing the number of possible values in $\overline{q}$; that is, 2^m.

In general, assume that we have a basis-vector expression $e(\overline{q})$ for m qubits $\overline{q}$, a measurement statement $\mathcal{M}(e(\overline{q}))$ storing the result in v, and have a boolean check on v as $B(v)$ defining the good states. By assigning $\{0, 1\}^m$ for $\overline{q}$, the probability of having the good states is the division of number of possible basis-vector values with $B(e(\overline{q})) = \texttt{true}$ and the number of possible basis-vector values $e(\overline{q})$ for all possible assignments. In Figure 2b, the effectiveness can be described as $\frac{n}{2^m}$; such a property can be validated by sampling. In some complicated cases, the right property might be hard to validate, but one can always use Rocq to verify the effectiveness via the above scheme.

$$\textsc{CRy}\quad \Xi \vdash \texttt{Ry}^r q \gg (\gamma, \texttt{Ry}^r(\Xi(q)))$$

$$\textsc{CCU}\quad \frac{\Xi \vdash \iota \gg \epsilon \qquad \epsilon' = \texttt{ctrl}(\gamma(q), \epsilon)}{\Xi \vdash \texttt{CU}\ p\ \iota \gg \epsilon'}$$

$$\textsc{CNext}\quad \frac{\Xi \vdash \iota \gg \epsilon}{(n, \Xi, \iota) \gg (n, \Xi, \epsilon)}$$

$$\textsc{CHad}\quad (n, \Xi, \texttt{H}(q)) \gg (n, \Xi, \texttt{H}(\Xi(q)))$$

$$\textsc{CSeq}\quad \frac{(n, \Xi, e_1) \gg (n', \Xi', \chi_1) \qquad (n', \Xi', e_2) \gg (n'', \Xi'', \chi_2)}{(n, \Xi, e_1\ ;\ e_2) \longrightarrow (n'', \Xi'', \chi_1\ ;\ \chi_2)}$$

$$\textsc{CNew}\quad \frac{\Xi' = \Xi[\forall q \in \overline{q}\,.\, q \mapsto |\Xi| + \texttt{ind}(\overline{q}, q)]}{(n, \Xi, \texttt{new}(\overline{q})) \gg (n + |\overline{q}|, \Xi', \{\})}$$

$$\textsc{CMea}\quad \frac{(n, \Xi \backslash \overline{q}, e) \gg (n', \Xi', \chi)}{(n, \Xi, \texttt{let}\ x = \mathcal{M}(\overline{q})\ \texttt{in}\ e) \longrightarrow (n', \Xi', \mathcal{M}(\overline{q}); \chi)}$$

Fig. 11. Select PQASM to SQIR translation rules (SQIR circuits are marked blue). $|\Xi|$: the length of Ξ; $\texttt{ind}(\overline{q}, q)$: the index of q in array $\overline{q}$. $\mathcal{M}(\overline{q})$ repeats $|\overline{q}|$ times on measuring qubit array $\overline{q}$.

Performance Optimizations. We took several steps to improve validation performance, e.g., we streamlined the representation of states: per the semantics in Figure 6, in a state with n qubits, the amplitude associated with each qubit can be written as $\Delta(\frac{v}{2^n})$ for some natural number v. Qubit values in both bases are thus pairs of natural numbers: the global phase v (in range $[0, 2^n)$) and b (for $|b\rangle_1$) or y (for $|\Delta(\frac{y}{2^n})\rangle$). An PQASM state φ is a map from qubit positions p to qubit values q; in our proofs, this map is implemented as a partial function, but for validation, we use an AVL tree implementation (proved equivalent to the functional map). To avoid excessive stack use, we implemented the PQASM semantics function tail-recursively. To run the tests, QuickChick runs OCaml code that it *extracts* from the Rocq definitions; during extraction, we replace natural numbers and operations thereon with machine integers and operations. Performance results are in Section 5.

4.2 Translation from PQASM to SQIR

We translate PQASM to SQIR by mapping PQASM virtual qubits to SQIR concrete qubit indices and expanding PQASM instructions to sequences of SQIR gates. SQIR [Hietala et al. 2023] is a quantum circuit language based on Rocq, containing standard quantum gates, such as Hadamard, controlled, and Ry gates, sequential operations, and quantum measurement operations. To express the classical components of quantum algorithms, SQIR typically utilizes Rocq program constructs. To define our compiler, we utilize the SQIR one-step non-deterministic semantics, containing one-step operational semantics for simple Rocq constructs, such as conditionals and classical sequential operations.

The PQASM to SQIR translation is expressed as the two-level judgments $\Xi \vdash \iota \gg \epsilon$ and $(n, \Xi, e) \gg (n', \Xi', \chi)$, where ϵ is the output SQIR circuit, and Ξ and Ξ' map an PQASM qubit q to a SQIR concrete qubit index (i.e., offset into a global qubit register), χ is a hybrid program including SQIR quantum circuits and Rocq classical programs, and n and n' are the qubit sizes in the whole system.

Figure 11 depicts a selection of translation rules. Rules CRY and CCU are the instruction level translation rules, which translate a PQASM instruction to a SQIR unitary operation. $\texttt{Ry}^r q$ has a directly corresponding gate in SQIR. In the CU translation, the rule assumes that ι's translation does not affect the Ξ position map. This requirement is assured for well-typed programs per rule CU in Figure 8. `ctrl` generates the controlled version of an arbitrary SQIR program using standard decompositions [Nielsen and Chuang 2011, Chapter 4.3].

The other rules in Figure 11 are the program level rules, which translate a PQASM program to a hybrid Rocq program including SQIR circuits with possible measurement operations. Rule CNEXT connects the instruction and the program-level translations. Rule CHAD translates a Hadamard operation to a SQIR Hadamard gate, while rule CSEQ translates a sequencing operator.

Rule CNew translates a qubit creation operation in PQASM. In SQIR, there is no qubit creation, in the sense that every qubit is assumed to exist in the first place. The translation essentially translates the operation to a SKIP operation in SQIR and increments the qubit heap size in the generated SQIR program. Note that the qubit size in the translation is always incrementing, i.e., a quantum measurement does not remove qubits but just makes some qubits inaccessible. Rule CMea translates a PQASM measurement operation to SQIR measurements by repeatedly measuring out qubits in $\overline{q}$. The translation removes measured qubits from Ξ, but it does not modify the qubit size.

We have proved the PQASM-to-SQIR translation correct. The proof utilizes the SQIR nondeterministic semantics, where a qubit measurement produces two possible outcomes with different probabilities associated with the outcomes, i.e., this nondeterministic semantics is essentially SQIR's way of describing a Markov-chain procedure. To formally state the correctness property, we relate PQASM superposition states Φ to SQIR states, written as $[\![\Phi]\!]^{n'}$, which are vectors of $2^{n'}$ complex numbers. We can utilize Ξ to relate qubits in PQASM with qubit positions in SQIR.

THEOREM 4.1. [Translation Correctness] Suppose $\Sigma; \Omega \vdash e \triangleright \Omega'$ and $(n, \Xi, e) \gg (n', \Xi', \chi)$. Then for $\Omega \vdash \Phi$ and $(\Phi, e) \xrightarrow{r} (\Phi', e')$, we have $([\![\Phi]\!]^{n'}, \chi) \xrightarrow{r} ([\![\Phi']\!]^{n'}, \chi')$ and $(n', \Xi', e') \gg (n'', \Xi'', \chi')$.

PROOF. The proof of translation correctness is by induction on the PQASM program e. Most of the proof simply shows the correspondence of operations in e to their translated-to gates ϵ in SQIR, except for `new` and measurement operations, which update the Ξ map. □

5 Evaluation: Applicativity via Case Studies

Here, we present an experimental evaluation of QSV on many programs to judge how QSV can be used to effectively build and validate useful quantum state preparation programs that have different patterns. We compare QSV's efficiency and scalability with other frameworks in Section 6.

Implementation. We implement QSV in Roqc and utilize QuickChick to create a quantum program validation framework. We also provide a PQASM circuit compiler to translate PQASM programs to OpenQASM quantum circuits, via SQIR.

Experimental Setup. We perform our evaluation of QuickChick tests and Qiskit on an Ubuntu computer, which has 8-core 13-gen i9 Intel processors and 16 GiB DDR5 memory.

State Preparation Program	8B Gate #	8B Qubit #	60B Gate #	60B Qubit #
n basis-ket	766	9	5,732	61
Modular Exponentiation	277K	27	3.3M	183
Amplitude Amplification	65	9	481	61
Hamming Weight	9,088	16	170K	120
Distinct Element	16,036	49	120K	361

Fig. 12. Program statistics for single registers with 8 and 60 Qubits (8B/60B); $n = 5$ for Distinct element. 'K' means thousand, 'M' means million'.

We show several case studies here to demonstrate the power of QSV for constructing and validating state preparation programs. We classify *two different program patterns* below and examine their validation strategies. We list the qubit and gate counts for the case study programs in Figure 12. The data are collected by compiling our programs to SQIR based on the elementary gateset $\{\mathtt{X}, \mathtt{H}, \mathtt{CX}, \mathtt{Rz}\}$. To describe the programs, we define the following repeat operator for repeating a process n times. Here, P is a function that takes a natural number and outputs a quantum program.

$$Re(P, n) \triangleq \mathtt{if}\ (n = 0)\ \{\}\ \mathtt{else}\ Re(P, n - 1)\ ;\ P(n - 1)$$

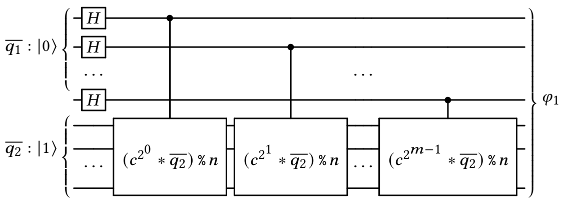

$\overline{q_1} : |0\rangle$
$\overline{q_2} : |1\rangle$
H
$(c^{2^0} * \overline{q_2})\,\%\,n$
$(c^{2^1} * \overline{q_2})\,\%\,n$
$(c^{2^{m-1}} * \overline{q_2})\,\%\,n$
φ_1

Fig. 13. Modular Exponentiation Circuits

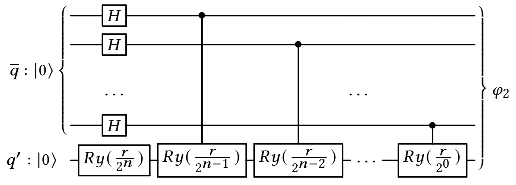


Fig. 14. Amplitude amplification state preparation.

5.0.1 *Quantum Loop Programs.*

We first examine the class of programs only involving quantum unitary gates without measurement. Such programs typically contain quantum loops, a repetition of subroutines formed via unitary gates. These programs usually act as a large part of some quantum algorithms. For example, the modular exponentiation state preparation program is the major part of Shor's algorithm, while the amplitude amplification state preparation program is the major part of an upgraded amplitude estimation algorithm [Suzuki et al. 2020].

Modular Exponentiation State Preparation. In Shor's algorithm [Shor 1994], we prepare the the superposition state of modular exponentiation, as $\varphi_1 = \frac{1}{\sqrt{2^m}} \sum_j^{2^m} |j\rangle_m |c^j\,\%\,n\rangle_m$, based on two natural numbers c and n with $\gcd(c, n) = 1$.

$$
\begin{array}{ll}
Q(\overline{q_1}, \overline{q_2})(k) & \triangleq \texttt{CU}\ (\overline{q_1}[k])\ (c^{2^k} * \overline{q_2})\,\%\,n \\
P(m) & \triangleq \texttt{new}(\overline{q_1})\ ;\ \texttt{new}(\overline{q_2})\ ;\ \texttt{H}(\overline{q_1})\ ;\ Re(Q(\overline{q_1}, \overline{q_2}), m)
\end{array}
$$

We show the program to prepare the modular exponentiation state below, with the circuit diagram in Figure 13. The program starts with two new length m qubit arrays $\overline{q_1}$ and $\overline{q_2}$, and turns $\overline{q_1}$ to a uniform superposition by applying m Hadamard gates. We then repeat m times a controlled modular multiplication ($\texttt{CU}\ (\overline{q_1}[k])\ (c^{2^k} * \overline{q_2})\,\%\,n$) application, controlling on the qubit $\overline{q_1}[k]$ and applying modular multiplications to the qubit array $\overline{q_2}$.

As we mentioned in Section 4.1, to conduct the validation of the program correctness, we first cut off the first three operations (new and H operations) in $P(m)$, resulting in a program piece $Re(Q(\overline{q_1}, \overline{q_2}), m)$, where $Q(\overline{q_1}, \overline{q_2})$ is a function taking in a number k and outputting a program $\texttt{CU}\ (\overline{q_1}[k])\ (c^{2^k} * \overline{q_2})\,\%\,n$. The validation correctness specification is also transformed as the one without having the superposition state description below.

$$\forall j \in [0, 2^m)\ .\ |j\rangle_m |0\rangle_m \rightarrow |j\rangle_m |c^j\,\%\,n\rangle_m$$

Validating the m-step loop program $Re(Q(\overline{q_1}, \overline{q_2}), m)$ essentially executes the Q program m times. In executing the k-th loop step, we have the following loop invariant.

$$|j\rangle_k |c^j\,\%\,n\rangle_m \rightarrow |j\rangle_{k+1} |c^j\,\%\,n\rangle_m$$

In the pre-state, $|j\rangle_k$ is a length k bitstring while the post-state has $|j\rangle_{k+1}$ being length k+1. To understand the behavior, notice that we have the most significant bit on the right. A k+1 length bitstring $|j\rangle_{k+1}$ can be expressed as a composition over a length k bitstring as $|j\rangle_k |0\rangle_1$ or $|j\rangle_k |1\rangle_1$, with the most sigificant bit being 0 or 1. For the former case, applying the controlled modular multiplication results in $|j\rangle_k |0\rangle_1 |c^j\,\%\,n\rangle_m$, i.e., the $\overline{q_2}$ part of the bitstring remains the same. For the latter case ($|j\rangle_k |1\rangle_1$), the controlled modular multipliation results in the state $|j\rangle_k |1\rangle_1 |(c^{2^k} * c^j\,\%\,n)\,\%\,n\rangle_m = |j\rangle_k |1\rangle_1 |c^{j+2^k}\,\%\,n\rangle_m$. Both cases can be rewritten to the post-state in the loop invariant above.

To validate the program piece $Re(Q(\overline{q_1}, \overline{q_2}), m)$ against the above specification, we are given a length $2m$ bitstring with $\overline{q_1}$ and $\overline{q_2}$ both being length m, and view $\overline{q_1}$ as a length m array of random variables, and prepare a initial state $|\overline{q_1}[0], ..., \overline{q_1}[m-1]\rangle_m |0\rangle_m$, with random generation of length m

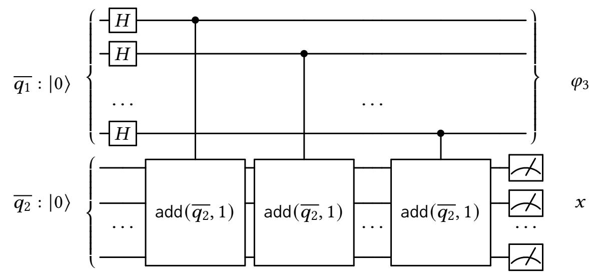


Fig. 15. One step Hamming weight state preparation (repeat-until-success).

binary bitstrings, as test instances, for the variables $\overline{q_1}[0], ..., \overline{q_1}[m-1]$. We then use the mechanism in Section 4.1 to validate the program piece, performing a sort of unit testing. As we mentioned in Section 1, after we test enough samples of different input basis-ket states with different values for random variables, we have a high assurance that the modular exponentiation state preparation program correctly prepares a superposition state.

Note that the program is a deterministic quantum circuit program, so the probability of preparing the superposition state is 100%.

Amplitude Amplification State Preparation Through Ry Gates. In the amplitude amplification algorithm, one needs to prepare a special superposition state [Suzuki et al. 2020], with the circuits shown in Figure 14. The prepared state is $\varphi_2 = \frac{1}{\sqrt{2^n}} \sum_j^{2^n} |j\rangle_n |\Delta(\frac{(2j+1)r}{2^n})\rangle$. Then, the amplitude amplification algorithm utilizes the last qubits ($|\Delta(\frac{(2j+1)r}{2^n})\rangle$) to amplify the amplitudes of the basis-kets having a particular property with respect to some j. The r value is the upper limit of the possible amplitude value, i.e., we want to carefully select r to ensure $\frac{(2j+1)r}{2^n} \in [0, \frac{\pi}{2})$.

$$
\begin{aligned}
Q(\overline{q}, q')(j) &\triangleq \mathsf{CU}\ (\overline{q}[j])\ \mathsf{Ry}^{\frac{r}{2^{n-j}}}\ q' \\
P(n) &\triangleq \mathsf{new}(\overline{q})\ ;\ \mathsf{new}(q')\ ;\ \mathsf{H}(\overline{q})\ ;\ \mathsf{Ry}^{\frac{r}{2^n}}\ q'\ ;\ Re(Q(\overline{q}, q'), n)
\end{aligned}
$$

We implement the program P in PQASM with the input of a qubit array $\overline{q}$ and a single qubit q'. We then apply Hadamard operations on $\overline{q}$ and a Y-axis rotation on q'. Eventually, we a series of controlled Y-axis rotation operations — controlling on the $\overline{q}[j]$ qubit, for $j \in [0, n)$ and applying Ry on q'; each single controlled Y axis roation is handled by the Q process.

Since there is no measurement in the above program, the success rate of preparing the amplitude amplification state is theoretically 100%. We mainly test the correctness here.

$$
\mathsf{Ry}^{\frac{r}{2^n}}\ q'\ ;\ Re(Q(\overline{q}, q'), n) \qquad \forall j \in [0, 2^n)\ .\ |j\rangle_n |0\rangle_1 \to |j\rangle_n |\Delta(\frac{(2j+1)r}{2^n})\rangle
$$

In doing so, we can adapt the same strategy in the modular exponentiation state preparation, where we assume the correctness of the portion, $\mathsf{new}(\overline{q})$; $\mathsf{new}(q')$; $\mathsf{H}(\overline{q})$, can be easily judged, so we should mainly focus on validating the portion shown above on the left. The correctness specification is described above on the right. In validating this portion, we notice that the qubit array $\overline{q}$ is in Had type, and we generate random variables, with binary values 0 or 1, for every qubit in the array. Through the validation procedure in validating the modular exponentiation program above, we can assure that the result of the program produces the superposition state φ_2.

5.0.2 Repeat-until-success Programs. We then examine the class of repeat-until-success programs, using the strategy exactly demonstrated in Section 4.1. We show the Hamming weight and the distinct element state preparation program validation below.

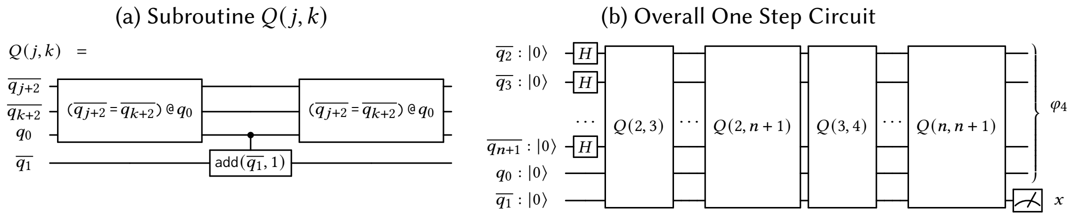

(a) Subroutine $Q(j,k)$
$Q(j,k)$ =
$\overline{q_{j+2}}$
$\overline{q_{k+2}}$
q_0
$\overline{q_1}$
$(\overline{q_{j+2}} = \overline{q_{k+2}})$ @ q_0
add($\overline{q_1}$, 1)
(b) Overall One Step Circuit
$\overline{q_2}$: $|0\rangle$
$\overline{q_3}$: $|0\rangle$
$\overline{q_{n+1}}$: $|0\rangle$
q_0 : $|0\rangle$
$\overline{q_1}$: $|0\rangle$
H
$Q(2,3)$
$Q(2,n+1)$
$Q(3,4)$
$Q(n,n+1)$
φ_4
x

Fig. 16. One step of the distinct element state preparation program.

Hamming Weights State Preparation. The quantum clique finding algorithm [Childs et al. 2002] requires the state preparation of a k-th Hamming weight superposition state, i.e., we prepare a state $\varphi_3 = \frac{1}{\sqrt{N}} \sum_j^N |c_j\rangle_n$, with the number of 1's bit in c_j is k. Assuming that φ_3 is a length n qubit state, φ_3 has N different basis-kets, with $N = \binom{n}{k}$.

$$
\begin{array}{ll}
Q(\overline{q_1}, \overline{q_2})(j) & \triangleq \texttt{CU}\ (\overline{q_1}[j])\ \texttt{add}(\overline{q_2}, 1) \\
P(n,k) & \triangleq \texttt{new}(\overline{q_1})\ ;\ \texttt{new}(\overline{q_2})\ ;\ \texttt{H}(\overline{q_1})\ ;\ Re(Q(\overline{q_1}, \overline{q_2}), n)\ ;\ \texttt{let}\ x = \mathcal{M}(\overline{q_2})\ \texttt{in}\ \texttt{if}\ (x = k)\ \{\}\ \texttt{else}\ P(n,k)
\end{array}
$$

The above is a repeat-until-success program of the Hamming weight program above, with the circuit in Figure 15 showing a single quantum step in $P(n,k)$. The program starts with two new length n qubit arrays $\overline{q_1}$ and $\overline{q_2}$, and turns $\overline{q_1}$ to a uniform superposition by applying n Hadamard gates. We then repeat n times of a controlled addition (CU $(\overline{q_1}[j])$ add$(\overline{q_2}, 1)$) applications — controlling on the qubit $\overline{q_1}[j]$ and applying additions to the qubit array $\overline{q_2}$. The controlled additions count the number of 1's bits in $\overline{q_1}$ and store the result in $\overline{q_2}$. If the measurement on the qubit array $\overline{q_2}$ results in k (assigning to x), it means that the φ_2 state of the qubit array $\overline{q_1}$ is a superposition of basis-ket states with the vector having k bits of 1. Otherwise, we repeat the process P with two new qubit arrays $\overline{q_1}$ and $\overline{q_2}$ until the measurement result k appears. Note that $Q(\overline{q_1}, \overline{q_2})$ in Re is a function taking in a natural number argument j and then performing a controlled addition.

$$
\begin{array}{c}
Re(Q(\overline{q_1}, \overline{q_2}), n)\ ;\ \texttt{let}\ x = \mathcal{M}(\overline{q_2})\ \texttt{in}\ \texttt{if}\ (x = k)\ \{\}\ \texttt{else}\ \{\} \\
\forall j \in [0, 2^n)\ .\ |j\rangle_n\, |0\rangle_n \longrightarrow |j\rangle_n \wedge \texttt{sum}(\texttt{n2b}(j)) = k
\end{array}
$$

To validate the correctness of the Hamming weight program, we shrink the program by removing the new and H operations. The program piece and the transformed correctness specification are listed above. We then utilize the procedure in Section 4.1 to perform the validation. Here, we assume that the Had typed qubits $\overline{q_1}$ are already prepared, and we randomly generate a length n bitstring for the random variables $\overline{q_1}[0], ..., \overline{q_1}[n-1]$. Each random variable, possibly being 0 or 1, represents the basis-bit of a single qubit superposition. We set up the PBT to randomly sample values for the random variables and exclusively test the correctness of the transition behavior of basis-ket states. The key correctness property ($\texttt{sum}(\texttt{n2b}(j)) = k$) for the Hamming weight state is that each output basis-ket of $\overline{q_1}$ should have exactly k bits of 1.

The judgment of the efficiency of the program in successfully preparing the superposition state can easily be done by counting the number of basis-kets in a superposition quantum state. Notice that every superposition state prepared by a simple Hadamard operation produces a uniform superposition, meaning that the likelihood of measuring out any basis-ket vector is equally likely. Thus, we only need to compare the ratio between the number of basis-kets after the Hadamard operations are applied and the basis-ket number in $\overline{q_1}$ after the measurement is applied. The former contains 2^n different basis-kets for n Hadamard operations, and the latter has $\binom{n}{k}$ basis-kets in a k-th Hamming weight state. So, the success rate of a single try in the program is $\binom{n}{k}/2^n$.

Distinct Element State Preparation. Another special superposition state is the one in the element distinctness algorithm. Here, we assume that we are given a graph with n different vertices, and the algorithm begins with a superposition of different combinations of vertices, as shown below.

$$\varphi_4 = \frac{1}{\sqrt{n!}} \sum_j \sigma_j(|x_1\rangle\,|x_2\rangle \dots |x_n\rangle)$$

Here, x_1, x_2, ..., x_n are different vertex keys in the graph, σ_j is a permutation of the key list $|x_1\rangle\,|x_2\rangle \dots |x_n\rangle$. There are $n!$ different kinds of permutations, so the uniform amplitude for each basis-ket is $\frac{1}{\sqrt{n!}}$. Essentially, the superposition of different vertex combinations means we are preparing a superposition state containing all the permutations of different vertex keys. Such superposition of permutation is widely used in many algorithms, such as the quantum fingerprinting algorithm [Buhrman et al. 2001].

$$\begin{aligned}
Q(k)(j) &\triangleq (\overline{q_{j+2}} = \overline{q_{k+2}})\,@\,q_0 \;;\; \mathtt{CU}\; q_0\; (\overline{q_1} + 1) \;;\; (\overline{q_{j+2}} = \overline{q_{k+2}})\,@\,q_0 \\
R(j)(n) &\triangleq Re(Q(j), n) \\
H(j) &\triangleq \mathtt{H}(\overline{q_{j+2}}) \\
T(j) &\triangleq \mathtt{new}(\overline{q_{j+1}}) \\
P(n) &\triangleq \mathtt{new}(q_0) \;;\; Re(T, n+1) \;;\; Re(H, n) \;;\; Re(R(n-1), n) \;;\; \mathtt{let}\; x = \mathcal{M}(\overline{q_1})\; \mathtt{in\; if}\; (x = 0)\; \{\}\; \mathtt{else}\; P(n)
\end{aligned}$$

For simplicity, we only implement the above program to prepare a superposition state of distinct elements, i.e., each basis-ket in the superposition state stores n distinct elements (vertex key), each key having a qubit size m. Note that if $n = 2^m$, i.e., we have 2^m different vertices having keys $u \in [0, 2^m)$, then the superposition state represents a superposition of all the permutations. We show a repeat-until-success program of preparing the distinct element superposition state above, with the circuit in Figure 16 showing a single quantum step in $P(n)$. We first initialize a single qubit q_0, and use $T(j)$ to initialize $n + 1$ different qubit arrays, $\overline{q_{j+1}}$, with $j \in [0, n + 1)$, and we assume that $\overline{q_{j+1}}$ is an m length qubit array. We then apply Hadamard operations to all qubit arrays $\overline{q_{j+2}}$, for $j \in [0, n)$. Here, q_0 and $\overline{q_1}$ are ancillary qubits.

Essentially, we can view $\overline{q_{j+2}}$, for $j \in [0, n)$, as an n-length array of qubit arrays. The program applies $O(n^2)$ times of Q processes, each of which applies an equivalent check on two elements in the n-length array, i.e., we compare the basis-ket data in q_{j+2} and q_{k+2} ($j, k \in [0, n)$), if $j + 2 \neq k + 2$ (same as $j \neq k$), and store the boolean result in q_0 bit, where 0 represents the two basis-ket data are not equal and 1 means they are equal. Then, we also add the result to $\overline{q_1}$ and apply the comparison circuit again to clean up the ancillary qubit q_0, meaning that we restore q_0's state to $|0\rangle$. This procedure describes the circuit in Figure 16a.

After we apply the Q function to any two different elements in the n-length array, we observe that q_0 is back to $|0\rangle$ state, and $\overline{q_1}$ stores the number of same basis-kets between any two distinct elements in the n-length array. We then measure $\overline{q_1}$ and see if the measurement result is 0. If so, a permutation superposition state is prepared because it means that in all the basis-kets in the prepared superposition, there are no two-qubit array elements $\overline{q_k}$ and $\overline{q_l}$ that have equal key. If not, we repeat the process, and the repeat-until-success program guarantees the creation of the permutation superposition state.

To validate the correctness, we utilize the procedure in Section 4.1. We create the validating program piece by shrinking out the new and H operations in the original program. The program piece and the transformed specification are listed below.

$$Re(R(n-1), n) \;;\; \mathtt{let}\; x = \mathcal{M}(\overline{q_1})\; \mathtt{in\; if}\; (x = 0)\; \{\}\; \mathtt{else}\; \{\}$$

$$x = 0 \Rightarrow \forall j, j' \in [0, 2^{n*m})\,.\; |0\rangle_1\,|0\rangle_n\,|j\rangle_{n*m} \rightarrow |0\rangle_1\,|j'\rangle_{n*m} \wedge \mathtt{dis}(j', m)$$

In the specification, j and j' represent the values for two arrays of qubit arrays, i.e., j represents the bitstring value for composing basis-ket values of all elements in the qubit array $\overline{q_{l+2}}$, with $l \in [0, n)$. The qubit array $\overline{q_{j+2}}$ has $n * m$ qubits, and we slice the basis-vector for the whole qubit

Program	QSV QCT 8B	QSV QCT 60B	Qiskit Sim 8B	Qiskit Sim 60B	DDSim Sim 60B
n basis-ket	< 1.5	2.4	< 1.5	No	No
Modular Exponentiation	< 1.5	19.7	No	No	No
Amplitude Amplification	< 1.5	2.1	<1.5	No	No
Hamming Weight	< 1.5	24.9	<1.5	No	No
Distinct Element	< 5	336	No	No	No

Fig. 17. Evaluation on different state preparation programs for 8/60 qubit single registers (8B/60B). "QCT" is the time (in seconds) for QuickChick to run 10,000 tests. "Sim" records the time (in seconds) or whether or not Qiskit/DDSim can execute a single test.

array into n different small segments for the qubit ranges $[l * m, l * (m + 1) - 1)$, each basis-vector segment representing a vertex key. Since we apply Hadamard operations to all of them, it creates a uniform superposition state containing 2^{n*m} different basis-vector states. In the post-state of the specification, the q_0 qubit is still $|0\rangle$. For the qubit arrays $\overline{q_2}, ..., \overline{q_{m+2}}$, if the measurement result is $x = 0$, we result in a superposition state of distinct elements, i.e., any two elements (each element l is a segment of $[l * m, l * (m + 1) - 1))$ in the qubit array $\overline{q_{j+2}}$ have distinct basis-vectors. We use the predicate $\texttt{dis}(j', m)$ to indicate that all length m segments in the bitstring $|j\rangle$ are pairwise distinct. Via our PBT framework, we have a high assurance that the distinct element state preparation program correctly prepares such a superposition state.

The judgment of the efficiency of the program in successfully preparing the superposition state can easily be done by counting the number of basis-ket states in a superposition quantum state. Notice that every superposition state prepared by a simple Hadamard operation produces a uniform superposition, meaning that the likelihood of measuring out any basis state vectors is equally likely. Thus, we only need to compare the ratio between the number of basis-kets right after the Hadamard operations are applied and the basis-ket number in $\overline{q_{j+2}}$. Here, we count the case for $n = 2^m$ where a permutation superposition state is prepared. For $j \in [0, n)$ with $n = 2^m$, after the measurement is applied. The former contains 2^{n*m} different basis-kets for $n * m$ Hadamard operations, and the latter has $n!$ basis-kets in a k-th permutated superposition state. So, the success rate of a single try in the program is $\frac{n!}{2^{n*m}}$.

6 Discussion: Efficiency, Scalability, and Utility, Compared to State-of-the-art

This section compares QSV's efficiency and scalability with the state-of-the-art platforms. We demonstrate that QSV can effectively validate program properties to provide confidence in program correctness. We also show that QSV *scales* to validate and realize state preparation programs regardless of the program size (in terms of number of qubits required). We thus justify the QSV's utility in successfully capturing bugs.

Efficiency. We discuss the program development procedure in QSV, compared to state-of-the-art systems such as Qiskit. Discussing the efficiency of a validation framework needs to be put in the context of human efforts for program development, as users mainly care about how to effectively use QSV to develop state preparation programs.

The general procedure for developing state preparation programs in QSV is the traditional test-driven program development. We are first given the program correctness properties, in the superposition state format, for different programs, such as the one in Sections 1 and 5. We then start implementing the program via a possible program pattern and see if we can write the correct program based on the pattern, where the PQASM high-level abstraction helps write programs. For example, in dealing with all the programs in Figure 12, we first try to see if we can write all these programs via the quantum loop program pattern, and rewrite the correctness properties based on the strategy presented in Section 5.0.1.

We then run our QSV validator to validate the implemented programs against the properties. After several rounds of corrections, one can typically judge if the program is implementable or not. For example, via our validator, we found that it might be hard to implement the Hamming weight and distinct element programs based on the quantum loop program pattern. We then switch to other program patterns to implement these programs, e.g., we utilize the repeat-until-success program pattern to implement the two programs by rewriting the correctness properties for the two programs to the ones in Section 5.0.2 and successfully find a solution. Our validator can validate a program effectively via our PBT framework, which generates 10, 000 test cases every time. As one can see in Figure 17, executing a validation step involving running 10, 000 randomly generated test cases for all our example programs costs us less than 5 seconds for small size programs (8 qubits) and less than 5.5 minutes for large size programs (60 qubits). This indicates that a program developer can quickly correct minor bugs when developing their programs.

On the other hand, developing state preparation programs in the start-of-the-art system might be painful, e.g., it is unlikely one can perform the test-driven development for implementing the programs in Figure 12, mainly because of a lack of proper validation facilities. As we can see in Figure 17, Qiskit might not execute a single test for some small-size (8 qubits) programs, such as distinct element programs. The modular exponentiation program is executed for some small number settings, but is not executable in general because the Qiskit simulator adopts some special optimizations for components in Shor's algorithm. Executing a large-size program (60 qubits) is completely impossible; either the program is too large for IBM's Aer simulator to run at all, and the simulator errors out, or the circuit construction may take hours and still not be done. This indicates the difficulty of developing large-scale programs by conducting small-scale testing in Qiskit, not to mention the need to validate large datasets and coverage. Another key issue is that the high-level abstraction support in the state-of-the-art systems is not well provided. In Qiskit, we can only find the quantum addition operations, while the other arithmetic and comparison operations are missing. In fact, we implement these operations in Qiskit based on our implementation in QSV.

Scalability. To evaluate the QSV scalability, we not only compare the execution of small and large sizes against Qiskit but also another state-of-the-art quantum simulator, DDSim. Here, we attempted to recreate (or find existing implementations of) the aforementioned programs on DDSim and Qiskit. We also performed PBT on our PQASM implementations on systems of various sizes and verified the rigidity of our tests by mutating either the properties or the states and verifying that the tests failed.

As shown in Figure 17, we have fully validated the five examples in these papers via our PBT framework. As far as we know, these constitute the first validated-correct implementations of the n basis-ket, Hamming weight, and distinct elements programs. All other operations in the figure were validated with QuickChick. To ensure these tests were efficacious, besides our program development procedures above, we also confirmed they could find hand-injected bugs; e.g., we changed the rotation angles in the Ry gate in the amplitude amplification state preparation and confirmed that our PBT could catch the inserted bugs. The tables in Figure 17 give the running times for our validator to validate programs—the times include the cost of extracting the Roqc code to OCaml, compiling it, and running it with 10, 000 randomly generated inputs via QuickChick. We validated these programs on small (8-qubit) and large size (60-qubit) inputs (the number relevant to the reported qubit and gate sizes in Figure 12), with all the validation happening within 2.5 minutes (most of them are finished within seconds). For comparison, we translated our programs to SQIR, converted the SQIR programs to OpenQASM 2.0 [Cross et al. 2017], and then attempted to simulate the resulting circuits on a *single test input* using DDSim [Burgholzer et al. 2021], a state-of-the-art quantum simulator, and list the result in the fifth column. Unsurprisingly, the simulation of the 60-bit versions did not complete when running overnight. The third and fourth columns in Figure 17 show the results for executing a *single program run* in Qiskit, and Qiskit executes a few small-size programs (Section 6) and none of the large-size programs. The experiment provided a good degree of assurance of the scalability of QSV.

There is a difference in the program execution between DDSim/Qiskit and QSV. The latter abstracts the arithmetic operations and assumes that the operations can be dealt with in the previous VQO [Li et al. 2022] framework, while DDSim executes the whole circuits generated from a QSV program. To compare the effects, we list the QuickChick testing time (running 10,000 tests) of the operations used in our state preparation programs in Figure 18; such running time data was given in VQO. The addition and comparison circuits do not greatly affect the execution of our PQASM programs. The validation of modular multiplication circuits might be costly, as our 60-bit modular exponentiation contains 60 modular multiplication operations. However, a typical validation scheme might only validate the correctness of a costly subcomponent once and use its semantic property in validating other programs utilizing the subcomponents. More importantly, the purpose of the experiment of DDSim/Qiskit executions is to show the state-of-the-art impossibility of executing quantum programs in a classical computer, while our QSV framework can validate quantum programs.

Operation	QCT
Addition	2
Comparison	5
Modular Multiplication	794

Fig. 18. Arith operation QC time (60B).

Utility. One of the utility of a program validation framework is to find bugs or faults in the existing algorithms. During the quantum state preparation program development, we found several issues in two original algorithms [Ambainis 2004; Buhrman et al. 2001] that utilize these special superposition states. The two algorithms both require the preparation of a superposition state of distinct elements (or the superposition state of permutations of distinct elements), but they do not specify how such a state can be effectively prepared. To the best of our knowledge, the state preparation program in Section 5.0.2 is the first program implementation of the state via the repeat-until-success scheme. As we can see in our probability analysis, the chance of preparing such a state is not very high. This fact might indicate that the quantum algorithms advantage arguments over classical algorithms in these works might not be solid because of the unclear preparation for the initial states in these papers. Without our implementations for these state preparation programs, it is impossible to discover these delicate potential faults in these algorithms.

Indeed, in the algorithm [Childs et al. 2002] that uses the initial Hamming weight superposition state, the authors realized the potential low probability of preparing the initial state via the repeat-until-success scheme and pointed out the uses of a specialized gate instead of Hadamard gates, to start their repeat-until-success state preparation program. The special gates created a simple superposition state with a different probability distribution, rather than the uniform distribution in the case of using Hadamard gates. The analysis of these specialized superposition gates with different probability distributions will be included in our future work.

Another utility of using QSV is to judge the implemented programs' correctness and find a more optimized implementation. In QSV, we can perform the tasks by using our PBT framework. In other frameworks, such as OpenQASM, nothing fundamentally stopped the OpenQASM developers from making the same choices. Still, we note they did not have the benefit of the PQASM type system and PBT framework.

7 Related Work

This section gives related work beyond the discussion in Section 4.

Quantum Circuit Languages. Prior research has developed circuit-level compilers to compile quantum circuit languages to quantum computers, such as Qiskit [Aleksandrowicz et al. 2019], t|ket⟩ [Cambridge Quantum Computing Ltd 2019], Staq [Amy and Gheorghiu 2020], PyZX [Kissinger and van de Wetering 2020], Nam *et al.* [Nam et al. 2018], quilc [Rigetti Computing 2019], Cirq [Google Quantum AI 2019], ScaffCC [Javadi-Abhari et al. 2015], and Project Q [Steiger et al. 2018]. In addition, many quantum programming languages have been developed in recent years. Many of these languages (e.g. Quil [Rigetti Computing 2019], OpenQASM [Cross et al. 2022; Cross et al. 2017], SQIR [Hietala et al. 2023]) describe low-level circuit programs. Higher-level languages may provide library functions for performing common oracle operations (e.g., Q# [Microsoft 2017], Scaffold [Abhari et al. 2012; Litteken et al. 2020]) or support compiling from classical programs to quantum circuits (e.g., Quipper [Green et al. 2013]), but still leave some important details (like deallocating extra intermediate qubits) to the programmer. There has been some work on type systems to enforce that deallocation happens correctly (e.g., Silq [Bichsel et al. 2020]) and on automated insertion of deallocation circuits (e.g., Quipper [Green et al. 2013], Unqomp [Paradis et al. 2021]), but while these approaches provide useful automation, they may also lead to inefficiencies in compiled circuits.

Quantum Software Testing and Validation. There have been many approaches developed for validating quantum programs [Fortunato et al. 2022; Long and Zhao 2024; Paltenghi and Pradel 2022; Wang et al. 2021, 2022; Xia et al. 2024] including the use differential [Wang et al. 2021] and metamorphic testing [Paltenghi and Pradel 2023], as well as mutation testing [Fortunato et al. 2022] and fuzzing [Xia et al. 2024]. Some key challenges exist for testing quantum programs. First, their input space explodes due to superposition. Second, their results are probabilistic (meaning we need to use statistical measures and/or other approaches to evaluate results). Last, the expected result may be difficult or even impossible to determine. To date, the testing approaches have focused on validating small circuit subroutines (in the sense of having limited input qubit size) rather than testing comprehensive quantum programs, and they are all limited to performing tests in Qiskit, which might not capture all of the machine limitations.

Methodologies Possibly Used for Validating Quantum Programs. SymQV [Bauer-Marquart et al. 2023] proposed a method of encoding quantum states and gates as SMT-solvable predicates to perform automated verification. Chen *et al.* [Chen et al. 2023] and Abdulla *et al.* [Abdulla et al. 2024] used tree automata to symbolize quantum gates, instead of quantum states, and utilized tree automata to construct a tree structure for easing automated verification. These works can handle some large programs, but these programs have simple program structures, such as QFT. Mei *et al.* [Mei et al. 2024] performed quantum stabilizer simulation based on the Gottesman–Knill theorem, which is a small subset of quantum programs and mainly used for error correction programs. Quasimodo [Sistla et al. 2023] is another symbolic execution based on a BDD-like structure to symbolize gates rather than states; their results are similar to the tree automata works [Abdulla et al. 2024; Chen et al. 2023]. Qafny [Li et al. 2024] transformed quantum program verification to Dafny for automated verification. These works tried to transform states and gates to perform automated verification, different from QSV, which tries to perform program testing and validation. The methodologies are also different from QSV where they try to symbolize quantum gates and states, while QSV only inserts special treatments in the standard quantum state representations. As a result, QSV can deal with large programs with comprehensive program structures.

Verified Quantum Compilers. Recent work has looked at verified optimization of quantum circuits (e.g., VOQC [Hietala et al. 2023], CertiQ [Shi et al. 2019]), but the problem of verified *compilation* from high-level languages to quantum circuits has received less attention. The only examples of verified compilers for quantum circuits are ReVerC [Amy et al. 2017] and ReQWIRE [Rand et al. 2018]. Both of these tools support verified translation from a low-level Boolean expression language to circuits consisting of `X`, `CNOT`, and `CCNOT` gates. VQO [Li et al. 2022] is a certified compilation framework for verifying and compiling quantum arithmetic operations. QSV utilizes VOQC [Hietala et al. 2023] and VQO [Li et al. 2022] to compile PQASM programs to quantum circuits.

8 Conclusion and Limitations

We presented QSV, a framework for expressing and automatically validating quantum state preparation programs. The core of QSV is a quantum language PQASM, which can express a restricted class of quantum programs that are efficiently testable for certain properties and are useful for implementing state preparation programs. We have verified the translator from PQASM to SQIR and have validated (or randomly tested) many programs written in PQASM. We have used PQASM to implement state preparation programs useful in quantum computation, such as the ones in Figure 12. We hope this work will be the basis for building a quantum validation framework for validating quantum programs on classical computers.

Our QSV is capable of defining most quantum program patterns with program validation. As mentioned in Section 1, QSV targets validating state preparation programs. For almost all quantum programs, QSV is able to validate the most significant part of the program. For example, the validated modular multiplication program in Section 5.0.1 is essentially 90% of Shor's algorithm, except for the final inverse QFT gate and measurement.

Our type system specifically locates Hadamard operations, but there are no actual restrictions on Hadamard operations, as users can easily define similar behaviors via our `Ry` and oracle operations. Via our type system, we identify the beginning Hadamard operations as a general quantum algorithm component to generate superposition sources so that QSV can locate the places to create random inputs for validating programs. We recognize the superposition state generation in many quantum algorithms as the major bottleneck for testing quantum programs; therefore, we utilize types to identify them, with special treatment to transform the superposition states to a simple and testable format.

References

Parosh Aziz Abdulla, Yo-Ga Chen, Yu-Fang Chen, Lukáš Holík, Ondřej Lengál, Jyun-Ao Lin, Fang-Yi Lo, and Wei-Lun Tsai. 2024. Verifying Quantum Circuits with Level-Synchronized Tree Automata (Technical Report). arXiv:2410.18540 [cs.LO] https://arxiv.org/abs/2410.18540

Ali Abhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, Frederic Chong, Margaret Martonosi, Martin Suchara, Ken Brown, Massoud Pedram, and Todd Brun. 2012. *Scaffold: Quantum Programming Language.* Technical Report. Princeton University.

Gorjan Alagic, Cristopher Moore, and Alexander Russell. 2007. Quantum Algorithms for Simon's Problem Over General Groups. arXiv:quant-ph/0603251 [quant-ph] https://arxiv.org/abs/quant-ph/0603251

Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. Qiskit: An open-source framework for quantum computing. doi:10.5281/zenodo.2562110

A. Ambainis. 2004. Quantum walk algorithm for element distinctness. In *45th Annual IEEE Symposium on Foundations of Computer Science*. 22–31. doi:10.1109/FOCS.2004.54

Matthew Amy and Vlad Gheorghiu. 2020. staq – A full-stack quantum processing toolkit. 5, 3, Article 034016 (June 2020), 21 pages. doi:10.1088/2058-9565/ab9359 arXiv:1912.06070

Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified Compilation of Space-Efficient Reversible Circuits. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 3–21.

Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. 2023. symQV: Automated Symbolic Verification of Quantum Programs. In *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings* (Lübeck, Germany). Springer-Verlag, Berlin, Heidelberg, 181–198. doi:10.1007/978-3-031-27481-7_12

Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 286–300. doi:10.1145/3385412.3386007

Harry Buhrman, Richard Cleve, John Watrous, and Ronald de Wolf. 2001. Quantum Fingerprinting. *Physical Review Letters* 87, 16 (Sept. 2001). doi:10.1103/physrevlett.87.167902

Lukas Burgholzer, Hartwig Bauer, and Robert Wille. 2021. Hybrid Schrödinger-Feynman Simulation of Quantum Circuits With Decision Diagrams . In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE Computer Society, Los Alamitos, CA, USA, 199–206. doi:10.1109/QCE52317.2021.00037

Cambridge Quantum Computing Ltd. 2019. pytket. https://cqcl.github.io/pytket/build/html/index.html

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

Andrew M. Childs, Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2002. Finding cliques by quantum adiabatic evolution. *Quantum Info. Comput.* 2, 3 (April 2002), 181–191.

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266

Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (sep 2022), 50 pages. doi:10.1145/3505636

Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open quantum assembly language. *arXiv e-prints* (Jul 2017). arXiv:1707.03429 [quant-ph]

Paul Adrien Maurice Dirac. 1939. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society* 35 (1939), 416 – 418.

Daniel Fortunato, José Campos, and Rui Abreu. 2022. Mutation testing of quantum programs written in QISKit. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 358–359. doi:10.1145/3510454.3528649

Sukhpal Singh Gill, Oktay Cetinkaya, Stefano Marrone, Daniel Claudino, David Haunschild, Leon Schlote, Huaming Wu, Carlo Ottaviani, Xiaoyuan Liu, Sree Pragna Machupalli, Kamalpreet Kaur, Priyansh Arora, Ji Liu, Ahmed Farouk, Houbing Herbert Song, Steve Uhlig, and Kotagiri Ramamohanarao. 2024. Quantum Computing: Vision and Challenges. arXiv:2403.02240 [cs.DC] https://arxiv.org/abs/2403.02240

Google Quantum AI. 2019. Cirq: An Open Source Framework for Programming Quantum Computers. https://quantumai.google/cirq

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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 333–342. doi:10.1145/2491956.2462177

Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 1989. *Going beyond Bell's Theorem.* Springer Netherlands, Dordrecht, 69–72. doi:10.1007/978-94-017-0849-4_10

Kesha Hietala, Robert Rand, Liyi Li, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2023. A Verified Optimizer for Quantum Circuits. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 18 (Sept. 2023), 35 pages. doi:10.1145/3604630

Ali Javadi-Abhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2015. ScaffCC: Scalable Compilation and Analysis of Quantum Programs. *Parallel Comput.* 45 (June 2015), 2–17. doi:10.1016/j.parco.2014.12.001 arXiv:1507.01902

Aleks Kissinger and John van de Wetering. 2020. PyZX: Large scale automated diagrammatic reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (04 2020), 230–242. doi:10.4204/EPTCS.318.14

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified compilation of Quantum oracles. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 146 (Oct. 2022), 27 pages. doi:10.1145/3563309

Liyi Li, Mingwei Zhu, Rance Cleaveland, Alexander Nicolellis, Yi Lee, Le Chang, and Xiaodi Wu. 2024. Qafny: A Quantum-Program Verifier. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 24:1–24:31. doi:10.4230/LIPIcs.ECOOP.2024.24

Andrew Litteken, Yung-Ching Fan, Devina Singh, Margaret Martonosi, and Frederic T Chong. 2020. An updated LLVM-based quantum research compiler with further OpenQASM support. *Quantum Science and Technology* 5, 3 (may 2020), 034013. doi:10.1088/2058-9565/ab8c2c

Peixun Long and Jianjun Zhao. 2024. Testing Multi-Subroutine Quantum Programs: From Unit Testing to Integration Testing. *ACM Trans. Softw. Eng. Methodol.* (apr 2024). doi:10.1145/3656339 Just Accepted.

Jingyi Mei, Marcello Bonsangue, and Alfons Laarman. 2024. Simulating Quantum Circuits by Model Counting. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 555–578.

Microsoft. 2017. *The Q# Programming Language.* https://docs.microsoft.com/

Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4, 1 (10 May 2018), 23. doi:10.1038/s41534-018-0072-4

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

Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 1–27. doi:10.1145/3527330

Matteo Paltenghi and Michael Pradel. 2023. MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2413–2424. doi:10.1109/ICSE48619.2023.00202

Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: Synthesizing Uncomputation in Quantum Circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 222?236. doi:10.1145/3453483.3454040

Zoe Paraskevopoulou, Cătălin HriŢcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 325–343. doi:10.1007/978-3-319-22102-1_22

Robert Rand, Jennifer Paykin, Dong-Ho Lee, and S. Zdancewic. 2018. ReQWIRE: Reasoning about Reversible Quantum Circuits. In *QPL*.

Rigetti Computing. 2019. The @rigetti optimizing Quil compiler. https://github.com/rigetti/quilc

Yunong Shi, Xupeng Li, Runzhou Tao, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2019. Contract-based verification of a realistic quantum compiler. *arXiv e-prints* (Aug 2019). arXiv:1908.08963 [quant-ph]

P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 124–134. doi:10.1109/SFCS.1994.365700

Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. 2023. Symbolic Quantum Simulation with Quasimodo. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 213–225. doi:10.1007/978-3-031-37709-9_11

Damian S. Steiger, Thomas Haner, and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum* 2 (Jan. 2018), 49. doi:10.22331/q-2018-01-31-49

Yohichi Suzuki, Shumpei Uno, Rudy Raymond, Tomoki Tanaka, Tamiya Onodera, and Naoki Yamamoto. 2020. Amplitude Estimation Without Phase Estimation. *Quantum Information Processing* 19, 2 (Jan. 2020), 17 pages. doi:10.1007/s11128-019-2565-2

Matt Swayne. 2023. *What Are The Remaining Challenges Of Quantum Computing?* https://thequantuminsider.com/2023/03/24/quantum-computing-challenges/

Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 692–704. doi:10.1109/ASE51524.2021.9678792

Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. Quito: a coverage-guided test generator for quantum programs. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 1237–1241. doi:10.1109/ASE51524.2021.9678798

Robert Wille, Lucas Berent, Tobias Forster, Jagatheesan Kunasaikaran, Kevin Mato, Tom Peham, Nils Quetschlich, Damian Rovara, Aaron Sander, Ludwig Schmid, Daniel Schoenberger, Yannick Stade, and Lukas Burgholzer. 2024. The MQT

Handbook: A Summary of Design Automation Tools and Software for Quantum Computing. In *IEEE International Conference on Quantum Software (QSW)* (2024). doi:10.1109/QSW62656.2024.00013 arXiv:2405.17543

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Article 126. doi:10.1145/3597503.3639121

Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: superoptimization of Quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 625–640. doi:10.1145/3519939.3523433