Automated Equivalence Checking of Concurrent Quantum Systems

EBRAHIM ARDESHIR-LARIJANI, School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Iran
SIMON J. GAY, School of Computing Science, University of Glasgow, UK
RAJAGOPAL NAGARAJAN, Department of Computer Science, Faculty of Science and Technology, Middlesex University, UK

The novel field of quantum computation and quantum information has gathered significant momentum in the last few years. It has the potential to radically impact the future of information technology and influence the development of modern society. The construction of practical, general purpose quantum computers has been challenging, but quantum cryptographic and communication devices have been available in the commercial marketplace for several years. Quantum networks have been built in various cities around the world and a dedicated satellite has been launched by China to provide secure quantum communication. Such new technologies demand rigorous analysis and verification before they can be trusted in safety- and security-critical applications. Experience with classical hardware and software systems has shown the difficulty of achieving robust and reliable implementations.

We present *CCS^q*, a concurrent language for describing quantum systems, and develop verification techniques for checking equivalence between *CCS^q* processes. *CCS^q* has well-defined operational and superoperator semantics for protocols that are *functional*, in the sense of computing a deterministic input-output relation for all interleavings arising from concurrency in the system. We have implemented QEC (Quantum Equivalence Checker), a tool which takes the specification and implementation of quantum protocols, described in *CCS^q*, and automatically checks their equivalence. QEC is the first fully automatic equivalence checking tool for concurrent quantum systems. For efficiency purposes, we restrict ourselves to Clifford operators in the stabilizer formalism, but we are able to verify protocols over all input states. We have specified and verified a collection of interesting and practical quantum protocols ranging from quantum communication and quantum cryptography to quantum error correction.

CCS Concepts: • **Theory of computation** → **Quantum computation theory**; **Process calculi**; **Program semantics**;

Additional Key Words and Phrases: Quantum information processing, process calculi, programming language semantics, concurrency, equivalence checking

ACM Reference format:
Ebrahim Ardeshir-Larijani, Simon J. Gay, and Rajagopal Nagarajan. 2018. Automated Equivalence Checking of Concurrent Quantum Systems. *ACM Trans. Comput. Logic* 1, 1, Article 1 (January 2018), 33 pages.
https://doi.org/10.1145/3231597

1 INTRODUCTION

Quantum technologies are evolving rapidly and the design of correct, secure hardware and software for hybrid quantum/classical systems is a major task necessary for this development. Simulating the quantum behaviour of such systems, with classical computers, is generally infeasible. On the other hand, exploiting features such as entanglement, which are in contrast to common intuition, imposes major challenges at the design level. Therefore developing formal verification tools for quantum information systems is essential in achieving viable and safe quantum technologies such as quantum cryptography and communication.

In classical computing, techniques from the area of *formal methods* have been used widely to model and analyse systems. In order to describe complex systems where a large number of components run and interact concurrently, high level languages are used to specify the systems rigorously. For such languages, verification techniques and tools have been developed and these have proved invaluable in the design and analysis of modern information systems. However, in the case of quantum systems, the description is usually informal, for example by textual explanation of sequences of actions. Another commonly used description is via quantum circuits, which are similar to digital circuits, where quantum operations and measurements are applied by using quantum gates. It can be argued that these methods are not sufficient for the analysis of more complicated quantum protocols with many interacting components, running concurrently. That is why in recent years many high level methods have been introduced for the modelling and analysis of quantum protocols. These include quantum programming languages such as QPL [41] and Quipper [26], as well as a categorical description of quantum mechanics [3]. These formalisms are designed for specifying sequential quantum programs. For concurrent protocols, languages such as *qCCS* [43] and *CQP* [22] have been introduced (for an introduction to quantum programming see [20, 44]). Associated tools include the simulation environment LIQ$Ui|\rangle$ [36] and verification frameworks for quantum circuits [42] and programs [6, 7, 23].

To address the problem of formal verification of quantum systems, we present CCS^q, a concurrent language for describing quantum systems, and develop verification techniques for checking equivalence between CCS^q processes. CCS^q has well-defined operational and superoperator semantics for protocols that are *functional*, in the sense of computing a deterministic input-output relation for all interleavings arising from concurrency in the system. We have implemented QEC (Quantum Equivalence Checker), a tool which takes the specification and implementation of quantum protocols, described in CCS^q, and automatically checks their equivalence. QEC is the first fully automatic equivalence checking tool for concurrent quantum systems. For efficiency purposes, we restrict ourselves to Clifford operators in the stabilizer formalism, but we are able to verify protocols over all input states. We have specified and verified a collection of interesting and practical quantum protocols ranging from quantum communication and quantum cryptography to quantum error correction.

CCS^q can express physical separation of agents, concurrency and classical/quantum communication explicitly, and the process of equivalence checking is automatic. We consider the description of a quantum protocol by an informal sequence of actions or by a quantum circuit as its *implementation*, and its expected or intended behaviour as its *specification*. The equivalence checking of a protocol involves showing that an implementation meets its specification. For example, consider the case of the well-known quantum teleportation protocol [11]. The implementation of this protocol is usually presented as a circuit (Figure 2). Our equivalence checking tool can be used to establish that for all (quantum) inputs, the implementation is equivalent to the *identity* process, which maps an input directly to the output.

We work within the *stabilizer* formalism and we restrict ourselves to certain operators called *Clifford operators* in order to build an efficient tool. Moreover, we only apply equivalence checking to protocols that behave *functionally* in the sense of computing a deterministic input-output relation for all branches of computation (interleavings) arising from non-determinism (due to concurrency or quantum measurement) in the system. These input-output relations can be abstracted by *superoperators* that enable us to exploit the linearity of quantum operators to extend equivalence checking to arbitrary inputs, not just stabilizer states.

Despite the fact that our current approach is limited to the use of Clifford operators and restricts the type of protocols that can be analysed, we demonstrate its applicability to many interesting and practical quantum communication, cryptographic and fault tolerant protocols. For example, we are able to analyse quantum teleportation, quantum secret sharing, quantum error correction and remote CNOT gates.

In Section 2, we give necessary background from quantum mechanics and quantum computing. In Section 3, Selinger's QPL (Quantum Programming Language) is used as the specification language for describing sequential quantum protocols. The syntax of QPL is explained, together with its type system and semantics in terms of superoperators. We also show how the equivalence of two protocols in QPL, containing only Clifford operators, can be checked efficiently. Section 4 introduces CCS^q, a concurrent language for the specification of quantum protocols. In Section 5, we show how the semantics of concurrent protocols can be understood by reduction to sequential interleavings and the superoperator semantics can be used by formally translating them to QPL. Section 6 describes verification algorithms for concurrent protocols and also presents QEC (Quantum Equivalence Checker), a tool for equivalence checking. In Section 7, we demonstrate various case studies of concurrent quantum protocols, and the result of their verification with QEC. Section 8 gives an overview of related work and Section 9 contains concluding remarks and directions for future work.

2 PRELIMINARIES

In this section, we give a concise introduction to quantum information processing (QIP). For more detail, we refer to [39]. The basic unit of quantum information is a *qubit* (quantum bit). A qubit can be in a *basis* state, represented by $|0\rangle$ or $|1\rangle$. These basis states correspond to the classical states 0 and 1. However, a qubit may be in a *superposition* of states, described by $\alpha|0\rangle + \beta|1\rangle$, with $|\alpha|^2 + |\beta|^2 = 1$ where α and β are complex numbers called *amplitudes*. More generally, we consider a state on n qubits, whose general form is $|\psi\rangle = \alpha_0|00\ldots0\rangle + \ldots + \alpha_{2^n-1}|11\ldots1\rangle$ with $\Sigma_i|\alpha_i|^2 = 1$.

The state of a single qubit is an element of a two-dimensional complex vector space, called *Hilbert space*, which is equipped with an inner product denoted by $\langle\phi|\psi\rangle$. Multi-qubit state-spaces are constructed by tensor product, and an n-qubit basis state such as $|00\ldots0\rangle$ is an abbreviation for $|0\rangle \otimes |0\rangle \otimes \cdots \otimes |0\rangle$.

There are two kinds of operations on quantum states. A unitary transformation is an invertible linear operation on the Hilbert space. In a two-dimensional Hilbert space, measurement is specified by a pair of orthogonal subspaces, and randomly projects the state onto one of them, with probability determined by the amplitudes and producing classical information as the result of the measurement. For example, if the state $\alpha|0\rangle + \beta|1\rangle$ is measured in the standard basis, then the result is 0 with probability $|\alpha|^2$ or 1 with probability $|\beta|^2$.

An important phenomenon in quantum information is *entanglement*. A multi-qubit state is entangled if it cannot be decomposed as a tensor product of simpler states. An example is the two-qubit state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, which is known as an EPR pair. It is one of a set of four important two-qubit entangled states, termed *Bell states*. In this state, if the first qubit is measured in the standard basis, then the overall state collapses to either $|00\rangle$ or $|11\rangle$, so the effect is to also fix the state of the second qubit. Therefore, there is a correlation between the two entangled qubits even when they are separated by a distance.

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},\ Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},\ Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

Fig. 1. Pauli operators

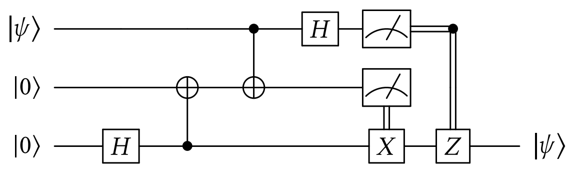

Fig. 2. Quantum Teleportation Circuit

Some basic single-qubit quantum operations and their matrix representations are shown in Figure 1. We write $\mathcal{P}$ for the set of *Pauli* operators $\{I, X, Y, Z\}$. The operator I is the *identity* operator; the operator X introduces a *bit-flip* and is the quantum version of the classical NOT gate; and the Z operator introduces a *phase-flip*. The Y operator combines bit- and phase-flip, with $Y = iXZ$.

A common model for describing a quantum system is the quantum circuit model, analogous to the classical circuit model. A quantum circuit consists of unitary gates and measurement. Unitary gates can be applied to one or more qubits. A two-qubit *controlled* gate has a control qubit and a target qubit. If the control qubit is set, then the quantum gate is applied to the target qubit. This can be generalised to multi-qubit gates. Controlled-X (or *CNot*) and *Toffoli* [39, p. 29] gates are examples of controlled gates.

Quantum circuits can be described in the following way. Single wires represent quantum information, while double wires represent classical information. Single gates and measurement are depicted with squares, whereas controlled gates are shown with a point representing the control qubit and a circle depicting the target qubit with a vertical wire connecting them. For example, Figure 2 illustrates a quantum circuit corresponding to the quantum teleportation protocol [11].

Teleportation is a quantum communication protocol designed to use only local quantum operations and classical communication (LOCC). It is an important primitive in QIP and many quantum computational schemes and models depend on it. Teleporting a given quantum state from Alice to Bob in this protocol can be achieved by only using entanglement, classical communications, local quantum operations and measurements.

In Figure 2, the three lines (from top to bottom) correspond to Alice's qubit, her share of an entangled pair and Bob's share of the entangled pair. Going from left to right, the first two steps of the circuit involve a Hadamard on the third qubit and a *CNot* on the third and second qubit. This represents the preparation of an entangled pair. The third and fourth step show Alice's operations, which involves a *CNot* on the first and second qubit followed by a Hadamard on the first qubit. Alice's measurements on the first two qubits are shown in the fifth step. Bob's operations, which depend on the outcome of Alice's measurements, are shown in the fifth and sixth step. At the end of the protocol, Bob's qubit is in the state that was the initial state of Alice's qubit.

The *density operator* is an alternative way of describing quantum states where we need to deal with uncertainty. For instance, an ensemble of quantum states $\{(|\phi_i\rangle, p_i)\}$, where the p_i are probabilities, can be represented by the following density operator:

$$\rho = \sum_i p_i |\phi_i\rangle\langle\phi_i|$$

where $|\phi_i\rangle\langle\phi_i|$ denotes outer product. Density operators are *positive semi-definite*, and *Hermitian*, meaning that they satisfy $\langle\varphi|\rho|\varphi\rangle \geq 0$ for all $|\varphi\rangle$, and $\rho^\dagger = \rho$ ($\dagger$ denotes transpose of the complex conjugate).

If quantum states are represented by density operators, then operations on states are represented by *superoperators*. These are linear operators on the space of Hermitian matrices, which also preserve the property of being positive semi-definite; therefore they map density operators to density operators. Superoperators encompass both unitary operators and measurements.

Definition 1. *A linear map between density operators ρ and ρ', $S : \rho \mapsto \rho'$, is a* superoperator *if it satisfies the following conditions.*

(1) *It preserves hermicity: ρ' is Hermitian $\Leftrightarrow$ ρ is Hermitian.*
(2) *It preserves trace: $tr(\rho') = tr(\rho)$.*
(3) *It is completely positive: ρ' is positive $\Leftrightarrow$ ρ is positive and for any new system R, $(I_R \otimes S)$ preserves positivity.*

For an n-qubit system, the space of Hermitian matrices has dimension 2^{2n}. A quantum program can be abstracted by a superoperator, where the input and output of the program are represented by density operators. In this paper, we take advantage of the linearity of superoperators: a superoperator is uniquely defined by its action on the elements of a *basis* of the space on which it acts, which in our case is a space of Hermitian matrices. We can therefore check equality of superoperators by checking that for every basis element as input, they produce the same output as each other.

Our approach to checking equivalence of quantum processes is based on the *stabilizer formalism*, which characterises a small but important part of quantum mechanics. The core idea of the stabilizer formalism is to represent certain quantum states, which are called *stabilizer states*, by their *stabilizer group*, instead of by an exponential number of complex amplitudes.

First we define the Pauli group of operators on n qubits, $\mathcal{P}_n$. It consists of n-fold tensor products of Pauli operators, combined with scalar factors from the set $\{+1, -1, +i, -i\}$.

$$\mathcal{P}_n = \{sP_1 \otimes \cdots \otimes P_n \mid P_i \in \mathcal{P}, s \in \{\pm 1, \pm i\}\}$$

For any n-qubit quantum state, its stabilizer group is the set of unitary operators that leave it unchanged; here we do not take account of the fact that states differing by scalar factors are physically indistinguishable, but simply consider the action of operators on vectors.

$$Stab(|\varphi\rangle) = \{S \mid S|\varphi\rangle = |\varphi\rangle\}$$

The stabilizer states are the states that are uniquely determined by the intersection of their stabilizer group with the Pauli group. A stabilizer state can be represented by a set of Pauli operators that generate its stabilizer group. The result is that an n-qubit state is represented by a *stabilizer tableau* consisting of n rows, each of which has n single-qubit Pauli operators (I, X, Y or Z) and a scalar factor of ± 1. This representation supports the simulation of *Clifford operators*, which consist of the Pauli operators together with the phase operator and the controlled not operator, in polynomial time.

Theorem 1. *(Gottesman-Knill, [39, p. 464]) Any quantum computation which consists of only the following components:*

(1) *state preparation, Hadamard gates, phase gates, controlled-not gates and Pauli gates;*
(2) *measurements in the standard basis;*

$$\begin{array}{llll}
\text{Statements} & S & ::= & \textbf{new qbit}\ q\ :=\ 0 \mid \textbf{discard}\ q \mid \widetilde{q}\ \mathrel{*}=\ U \\
 & & \mid & \textbf{skip} \mid S\ ;\ S \mid \textbf{measure}\ q\ \textbf{then}\ S\ \textbf{else}\ S \\
 & & & \\
\text{Programs} & P & ::= & \textbf{input}\ \tilde{q}\ ;\ S\ ;\ \textbf{output}\ \tilde{q}
\end{array}$$

Fig. 3. Syntax of QPL (excluding loops and procedures)

can be efficiently simulated on a classical computer.

As a consequence of Theorem 1, several algorithms for simulation of stabilizer formalism have been proposed [2, 4, 39, p. 463]. Our work is based on the algorithms in [2], which extend the stabilizer tableau with *destabilizer generators* in order to increase the efficiency of simulating measurements.

3 VERIFICATION OF SEQUENTIAL QUANTUM SYSTEMS

In this section we explain the key ideas of equivalence-checking by discussing sequential quantum protocols, which we express in Selinger's QPL language [41]. We expand on our previous work [6] by giving a full account of the semantic basis for our approach. After establishing this foundation, and introducing our concurrent language CCS^q in Section 4, it will be easy to explain equivalence-checking for concurrent quantum protocols in Sections 5 and 6.

3.1 The QPL Language and its Semantics

The syntax of QPL is defined in Figure 3. We exclude loops and procedures, and consider only quantum (not classical) variables. We assume an infinite supply of qubit identifiers: these are indicated by q in the grammar, but they can be any identifier. We use the notation $\widetilde{\cdot}$ for finite sequences. The top-level syntactic category is a program, which consists of a statement preceded by an input declaration and followed by an output declaration. There are several forms of statement. Qubits are created (and initialised) by **new qbit** $q\ :=\ 0$, and discarded by **discard** q. Application of a unitary operator U to a sequence $\widetilde{q}$ of qubits is denoted by $\widetilde{q}\ \mathrel{*}=\ U$. The construct **measure** q **then** S_1 **else** S_2 measures the qubit q and executes either S_1 or S_2 depending on whether the classical result of the measurement is 1 or 0. Sequential composition is denoted by semicolon, and **skip** does nothing.

The type system of QPL identifies well-formed programs P as those for which a judgement $\vdash \langle\Gamma\rangle\, P\, \langle\Delta\rangle$ is derivable by the rules in Figure 4. Here Γ and Δ are type environments $x_1 : \textbf{qbit}, \ldots, x_n : \textbf{qbit}$. The type system checks that unitary operators are applied correctly, and tracks the set of qubits that are in scope and available to be operated on. As an example, the teleportation protocol in QPL is illustrated in Figure 5.

Selinger defines the semantics of QPL by first representing a program as a flowchart. Flowcharts are constructed from the components in Figure 6. A flowchart can have multiple input paths and multiple output paths. This allows them to show the branching structure arising from different measurement outcomes. The annotations $\Gamma = A$ show the set of qubits in scope (Γ) and the effect of the component on a density matrix A. These components correspond to QPL statements, except for two cases. A *merge* node is used to combine the branches of a measurement so that they can be followed by a single flow of control. *Horizontal composition* allows multiple branches to be constructed, representing independent operations on separate parts of the quantum state. A QPL program

$$\vdash \langle\widetilde{q} : \textbf{qbit}\rangle\ \textbf{input}\ \widetilde{q}\ ;\ S\ ;\ \textbf{output}\ \widetilde{r}\ \langle\widetilde{r} : \textbf{qbit}\rangle$$

$$(\textit{newqbit}) \qquad \vdash \langle \Gamma \rangle \ \textbf{new qbit}\ q \ := \ 0 \ \langle q : \textbf{qbit}, \Gamma \rangle$$

$$(\textit{discard}) \qquad \vdash \langle q : \textbf{qbit}, \Gamma \rangle \ \textbf{discard}\ q \ \langle \Gamma \rangle$$

$$(\textit{unitary}) \qquad \frac{U \text{ is of arity } n}{\vdash \langle q_1 : \textbf{qbit}, \dots, q_n : \textbf{qbit}, \Gamma \rangle \ \widetilde{q} \ \mathrel{*}= \ U \ \langle q_1 : \textbf{qbit}, \dots, q_n : \textbf{qbit}, \Gamma \rangle}$$

$$(\textit{skip}) \qquad \vdash \langle \Gamma \rangle \ \textbf{skip} \ \langle \Gamma \rangle$$

$$(\textit{compose}) \qquad \frac{\vdash \langle \Gamma \rangle \ S \ \langle \Gamma' \rangle \qquad \vdash \langle \Gamma' \rangle \ T \ \langle \Gamma'' \rangle}{\vdash \langle \Gamma \rangle \ S \, ; T \ \langle \Gamma'' \rangle}$$

$$(\textit{measure}) \qquad \frac{\vdash \langle q : \textbf{qbit}, \Gamma \rangle \ S \ \langle \Gamma' \rangle \qquad \vdash \langle q : \textbf{qbit}, \Gamma \rangle \ T \ \langle \Gamma' \rangle}{\vdash \langle q : \textbf{qbit}, \Gamma \rangle \ \textbf{measure}\ q \ \textbf{then}\ S \ \textbf{else}\ T \ \langle \Gamma' \rangle}$$

$$(\textit{permute}) \qquad \frac{\vdash \langle \Gamma \rangle \ S \ \langle \Delta \rangle \qquad \Gamma', \Delta' \text{ permutations of } \Gamma, \Delta}{\vdash \langle \Gamma' \rangle \ S \ \langle \Delta' \rangle}$$

$$(\textit{program}) \qquad \frac{\vdash \langle q_1 : \textbf{qbit}, \dots, q_m : \textbf{qbit} \rangle \ S \ \langle r_1 : \textbf{qbit}, \dots, r_n : \textbf{qbit} \rangle}{\vdash \langle q_1 : \textbf{qbit}, \dots, q_m : \textbf{qbit} \rangle \ \textbf{input}\ \widetilde{q} \, ; S \, ; \textbf{output}\ \widetilde{r} \ \langle r_1 : \textbf{qbit}, \dots, r_n : \textbf{qbit} \rangle}$$

Fig. 4. Typing rules for QPL statements and programs

```
// Alice's input.
input q0 ;

// Preparing the entangled pair.
newqbit q1 ;
newqbit q2 ;
q1 *= H ;
q1,q2 *= CNot ;

// Entangling Alice's qubit.
q0,q1 *= CNot ;
q0 *= H ;

// Alice's measurement and Bob's corrections.
measure q0 then q2 *= Z else q2 *= I end ;
measure q1 then q2 *= X else q2 *= I end ;

// Bob's output.
output q2
```

Fig. 5. Teleportation in QPL

is represented by a flowchart $(\!|S|\!)$ with input qubits $\widetilde{q}$ and output qubits $\widetilde{r}$:

$$\begin{tikzcd}
{} \arrow[d, "\widetilde{q} : \mathbf{qbit} = A"] \\
\boxed{(\!|S|\!)} \arrow[d, "\widetilde{r} : \mathbf{qbit} = B"] \\
{}
\end{tikzcd}$$

The translation from QPL statements to flowcharts is trivial except for the translation of measurement:

$$(\!|\mathbf{measure}\ q\ \mathbf{then}\ S\ \mathbf{else}\ T|\!) = \mathbf{measure}\ q\ ;\ ((\!|S|\!) \mid (\!|T|\!))\ ;\ \mathbf{merge}$$

The semantics of a program $\vdash \langle\Gamma\rangle\ P\ \langle\Delta\rangle$ is a superoperator $[\![P]\!] : [\![\Gamma]\!] \to [\![\Delta]\!]$. The semantics of type environments is defined by $[\![x_1 : \mathbf{qbit}, \ldots, x_n : \mathbf{qbit}]\!] = [\![\mathbf{qbit}]\!] \otimes \cdots \otimes [\![\mathbf{qbit}]\!]$ where $[\![\mathbf{qbit}]\!] = \mathbb{C}^2$.

The typing of a flowchart F with m input paths and n output paths has the form $\vdash\ [\Gamma_1 ; \ldots ; \Gamma_m]\ F\ [\Delta_1 ; \ldots ; \Delta_n]$, where each input path has a type environment Γ_i and each output path has a type environment Δ_j. The typing rules for flowcharts are defined explicitly in Figure 7, which contains the same information as the Γ parts of the annotations in Figure 6. The semantics of F is a superoperator $[\![F]\!] : [\![\Gamma_1]\!] \oplus \cdots \oplus [\![\Gamma_m]\!] \to [\![\Delta_1]\!] \oplus \cdots \oplus [\![\Delta_m]\!]$, where $\oplus$ corresponds to tuples of density matrices. The semantics of flowcharts is defined in Figure 8, following Selinger [41].

3.2 Checking Equivalence of QPL Programs

Our approach to verification is to consider two QPL programs, P and Q, with the same typing: $\vdash \langle\Gamma\rangle\ P\ \langle\Delta\rangle$ and $\vdash \langle\Gamma\rangle\ Q\ \langle\Delta\rangle$ for some Γ and Δ, and compute whether or not $[\![P]\!] = [\![Q]\!]$. We do this by simulation in the stabilizer formalism. $[\![P]\!] = [\![Q]\!]$ if and only if $\forall b \in \mathcal{B}.[\![P]\!](b) = [\![Q]\!](b)$, where $\mathcal{B}$ is a basis for $[\![\Gamma]\!]$. We can choose $\mathcal{B}$ to consist of stabilizer states.

Working in the stabilizer formalism has the obvious consequence that we can only use unitary operators from the Clifford group. There is another consequence related to how we simulate measurements.

In Selinger's semantics, defined via the translation into flowcharts, $[\![\mathbf{measure}\ q\ \mathbf{then}\ S\ \mathbf{else}\ T]\!]$ requires addition of density matrices in order to merge the effects of S and T on the quantum state. We cannot compute this addition in the stabilizer formalism, because in general the sum of stabilizer states is not a stabilizer state. To resolve this problem, we do not simulate **merge** nodes. Instead, we keep the two branches of a measurement separate, and simulate each branch to the end of the program. For a given basis state as input, we require that all branches produce the same output. This means that the non-determinism of measurement is compensated by other parts of the protocol, so that the overall effect is a deterministic function from input to output.

We describe simulation of QPL programs in terms of the following operations on stabilizer tableaus τ, which are all standard [2, 9].

- $\mathrm{create}(q_1, \ldots, q_n, |\phi\rangle)$ creates a tableau with qubits $q_1, \ldots, q_n$ in state $|\phi\rangle$.
- $\mathrm{allocate}(q, \tau)$ produces a new tableau with an additional qubit q in state $|0\rangle$.
- $\mathrm{apply}(q_1, \ldots, q_n, U, \tau)$ produces a new tableau in which the unitary operator U has been applied to qubits $q_1, \ldots, q_n$.
- $\mathrm{possible}(q, v, \tau)$ returns a boolean value indicating whether or not $|v\rangle$ (either $|0\rangle$ or $|1\rangle$) is a possible result of measuring qubit q.
- $\mathrm{set}(q, v, \tau)$ produces a new tableau in which qubit q has been measured with result $|v\rangle$.
- $\mathrm{reduce}(q_1, \ldots, q_n, \tau)$ produces a new tableau in which all of the qubits except $q_1, \ldots, q_n$ have been traced out.

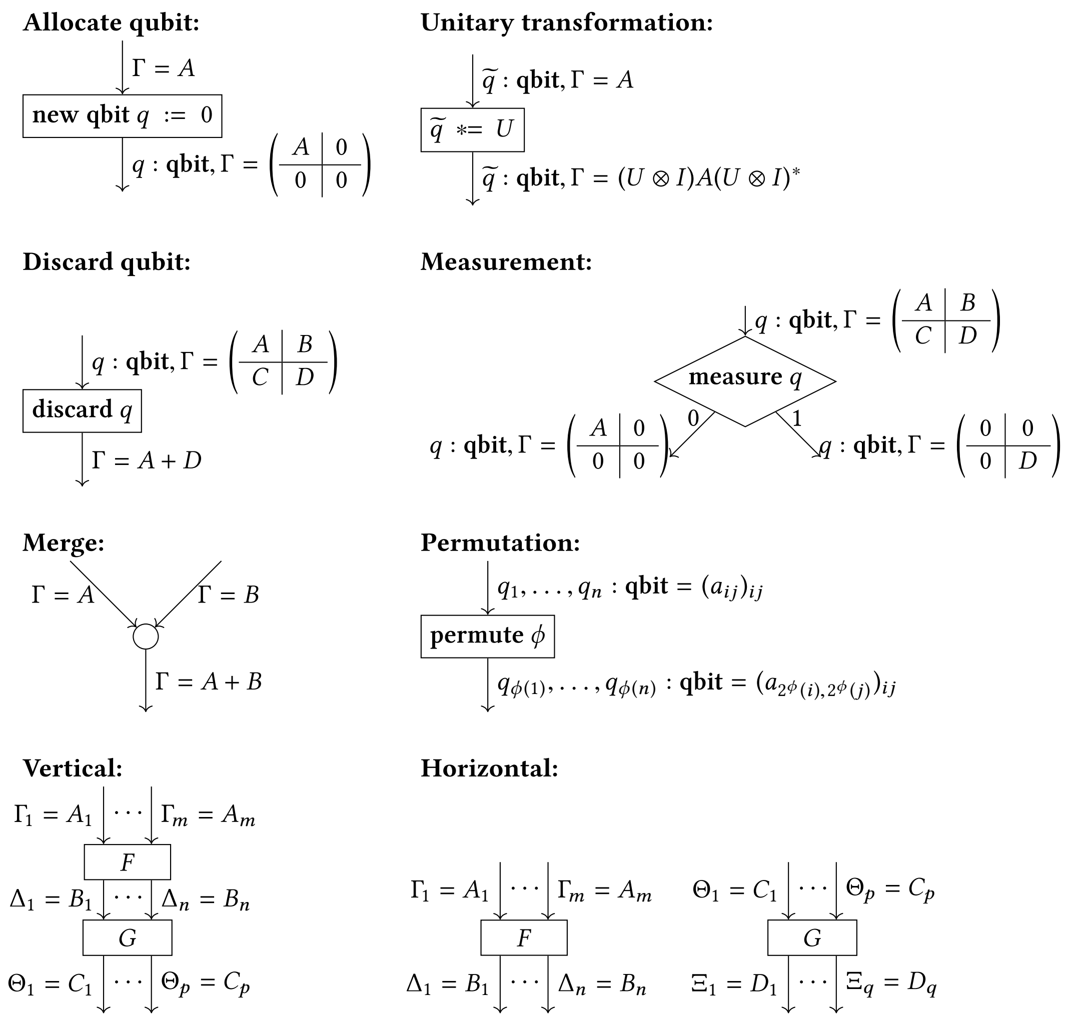

Fig. 6. QPL flowcharts

Given these operations, Figure 9 defines a transition relation on configurations $\langle \tau, S \rangle$ consisting of a stabilizer tableau τ and a QPL statement S. Simulation ends with a terminal configuration $\langle \tau \rangle$, reached when an **output** statement is executed. This transition relation enables us to define the key property of programs that our verification technique requires: that they are *functional.*

Definition 2. *Let* $\vdash \langle \widetilde{q} : \textbf{qbit} \rangle\ P\ \langle \widetilde{r} : \textbf{qbit} \rangle$ *be a program, where* $P = \textbf{input}\ \widetilde{q}\ ; S\ ; \textbf{output}\ \widetilde{r}$*. Define the transition relation* $\xrightarrow{P}$ *between stabilizer tableaus by* $\tau \xrightarrow{P} \tau'$ *if and only if:*

(1) τ *is a stabilizer tableau for qubits* $\widetilde{q}$*, and*
(2) τ' *is a stabilizer tableau for qubits* $\widetilde{r}$*, and*
(3) $\langle \tau, S\ ; \textbf{output}\ \widetilde{r} \rangle \rightarrow^* \langle \tau' \rangle$.

Definition 3. *A program P is* functional *if whenever* $\tau \xrightarrow{P} \tau_1$ *and* $\tau \xrightarrow{P} \tau_2$ *we have* $\tau_1 = \tau_2$*.*

$$(\mathit{newqbit}) \qquad \vdash [\Gamma]\ \mathbf{new\ qbit}\ q\ :=\ 0\ [q : \mathbf{qbit}, \Gamma]$$

$$(\mathit{discard}) \qquad \vdash [q : \mathbf{qbit}, \Gamma]\ \mathbf{discard}\ q\ [\Gamma]$$

$$(\mathit{unitary}) \qquad \frac{U \text{ is of arity } n}{\vdash [q_1 : \mathbf{qbit}, \dots, q_n : \mathbf{qbit}, \Gamma]\ \overline{q}\ \mathrel{*}= U\ [q_1 : \mathbf{qbit}, \dots, q_n : \mathbf{qbit}, \Gamma]}$$

$$(\mathit{vertical}) \qquad \frac{\vdash [\Gamma_1 ; \dots ; \Gamma_m]\ F\ [\Delta_1 ; \dots ; \Delta_n] \qquad \vdash [\Delta_1 ; \dots ; \Delta_n]\ G\ [\Theta_1 ; \dots ; \Theta_p]}{\vdash [\Gamma_1 ; \dots ; \Gamma_m]\ F ; G\ [\Theta_1 ; \dots ; \Theta_p]}$$

$$(\mathit{horizontal}) \qquad \frac{\vdash [\Gamma_1 ; \dots ; \Gamma_m]\ F\ [\Delta_1 ; \dots ; \Delta_n] \qquad \vdash [\Theta_1 ; \dots ; \Theta_p]\ G\ [\Xi_1 ; \dots ; \Xi_q]}{\vdash [\Gamma_1 ; \dots ; \Gamma_m ; \Theta_1 ; \dots ; \Theta_p]\ F \mid G\ [\Delta_1 ; \dots ; \Delta_n ; \Xi_1 ; \dots ; \Xi_q]}$$

$$(\mathit{measure}) \qquad \vdash [q : \mathbf{qbit}, \Gamma]\ \mathbf{measure}\ q\ [q : \mathbf{qbit}, \Gamma ; q : \mathbf{qbit}, \Gamma]$$

$$(\mathit{merge}) \qquad \vdash [\Gamma ; \Gamma]\ \mathbf{merge}\ [\Gamma]$$

$$(\mathit{permute}) \qquad \frac{\phi \text{ is a permutation}}{\vdash [q_1 : \mathbf{qbit}, \dots, q_n : \mathbf{qbit}]\ \mathbf{permute}\ \phi\ [q_\phi(1) : \mathbf{qbit}, \dots, q_\phi(n) : \mathbf{qbit}]}$$

Fig. 7. Typing rules for QPL flowcharts

We simulate a QPL program according to the transitions defined in Figure 9, exploring all branches due to non-deterministic measurements and discovering all possible final outputs for a given input. This enables functionality to be checked. Given two programs, we can also check equivalence. We now give a semantic justification for this approach to simulating branches, by separating the computational part of a flowchart from the part that merges branches resulting from measurements.

Definition 4. *A* merge network *is a flowchart constructed from* **merge**, *permutations and horizontal and vertical composition.*

Definition 5. *A flowchart is* normal *if horizontal composition is always immediately followed (in a vertical composition) by* **merge**. *That is, horizontal composition occurs only in structures of the form* $(F \mid G)$; **merge**.

Lemma 1. *(Weak commutativity) Let P be a flowchart that does not contain* **merge**, *and let M be a normal merge network. Then there exist flowcharts P' and M' such that P' does not contain* **merge**, *M' is a normal merge network, and $[\![M ; P]\!] = [\![P' ; M']\!]$.*

Proof. By induction on the construction of M.

- If M is **merge** then $[\![M ; P]\!](a, b) = [\![P]\!](a + b) = [\![P]\!](a) + [\![P]\!](b)$. Let $P' = P \mid P$ and $M' = \mathbf{merge}$. Then $[\![P' ; M']\!](a, b) = [\![(P \mid P) ; \mathbf{merge}]\!](a, b) = [\![P]\!](a) + [\![P]\!](b)$.
- If M is a permutation then we can combine it with P to give P', and take M' to be the identity permutation.
- If M is $M_1 ; M_2$ with M_1 and M_2 normal then by induction we have P_2' and M_2' such that P_2' does not contain **merge**, M_2' is a normal merge network, and $[\![M_1 ; M_2 ; P]\!] = [\![M_1 ; P_2' ; M_2']\!] =$

$[\![\textbf{new qbit } q := 0]\!]$	$[\![\Gamma]\!] \to [\![\textbf{qbit}]\!] \otimes [\![\Gamma]\!]$ $A \mapsto \begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix}$
$[\![\textbf{discard } q]\!]$	$[\![\textbf{qbit}]\!] \otimes [\![\Gamma]\!] \to [\![\Gamma]\!]$ $\begin{pmatrix} A & B \\ C & D \end{pmatrix} \mapsto A + D$
$[\![\widetilde{q} \mathrel{*}= U]\!]$	$[\![\textbf{qbit}]\!]^n \otimes [\![\Gamma]\!] \to [\![\textbf{qbit}]\!]^n \otimes [\![\Gamma]\!]$ $A \mapsto (U \otimes I)A(U \otimes I)^*$
$[\![F\,;G]\!]$	$[\![\Gamma_1]\!] \oplus \cdots \oplus [\![\Gamma_m]\!] \to [\![\Theta_1]\!] \oplus \cdots \oplus [\![\Theta_p]\!]$ $[\![G]\!] \circ [\![F]\!]$
$[\![F \mid G]\!]$	$[\![\Gamma_1]\!] \oplus \cdots \oplus [\![\Gamma_m]\!] \oplus [\![\Theta_1]\!] \oplus \cdots \oplus [\![\Theta_p]\!] \to$ $[\![\Delta_1]\!] \oplus \cdots \oplus [\![\Delta_n]\!] \oplus [\![\Xi_1]\!] \oplus \cdots \oplus [\![\Xi_q]\!]$ $[\![F]\!] \oplus [\![G]\!]$
$[\![\textbf{measure } q]\!]$	$[\![\textbf{qbit}]\!] \otimes [\![\Gamma]\!] \to ([\![\textbf{qbit}]\!] \otimes [\![\Gamma]\!]) \oplus ([\![\textbf{qbit}]\!] \otimes [\![\Gamma]\!])$ $\begin{pmatrix} A & B \\ C & D \end{pmatrix} \mapsto (\begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & D \end{pmatrix})$
$[\![\textbf{merge}]\!]$	$[\![\Gamma]\!] \oplus [\![\Gamma]\!] \to [\![\Gamma]\!]$ $(A, B) \mapsto A + B$
$[\![\textbf{permute } \phi]\!]$	$[\![\textbf{qbit}]\!]^n \to [\![\textbf{qbit}]\!]^n$ $(a_{ij})_{ij} \mapsto (a_{2^\phi(i), 2^\phi(j)})_{ij}$ where ϕ is a permutation of $\{1, \ldots, n\}$ and 2^ϕ is a permutation of the set of bit vectors of length n, defined by $2^\phi(x_1, \ldots, x_n) = (x_{\phi^{-1}(1)}, \ldots, x_{\phi^{-1}(n)})$.

Fig. 8. Semantics of QPL flowcharts [41]

$$\langle \tau, \textbf{new qbit } q := 0\,; S\rangle \to \langle \text{allocate}(q, \tau), S\rangle$$

$$\langle \tau, \textbf{skip}\,; S\rangle \to \langle \tau, S\rangle$$

$$\langle \tau, q_1, \ldots, q_n \mathrel{*}= U\,; S\rangle \to \langle \text{apply}(q_1, \ldots, q_n, U, \tau), S\rangle$$

$$\langle \tau, \textbf{measure } q \textbf{ then } S \textbf{ else } T\,; U\rangle \to \langle \text{set}(q, 1, \tau), S\,; U\rangle \quad \text{if possible}(q, 1, \tau)$$

$$\langle \tau, \textbf{measure } q \textbf{ then } S \textbf{ else } T\,; U\rangle \to \langle \text{set}(q, 0, \tau), T\,; U\rangle \quad \text{if possible}(q, 0, \tau)$$

$$\langle \tau, \textbf{output } q_1, \ldots, q_n\rangle \to \langle \text{reduce}(q_1, \ldots, q_n, \tau)\rangle$$

Fig. 9. Simulation of QPL programs by exploring all branches

$[\![M_1 ; P'_2]\!] ; [\![M'_2]\!]$. Again by induction we have P'_1 and M'_1 such that P'_1 does not contain **merge**, M'_1 is a normal merge network, and $[\![M_1 ; P'_2]\!] = [\![P'_1 ; M'_1]\!] = [\![P'_1]\!] ; [\![M'_1]\!]$. Taking $P' = P'_1$ and $M' = M'_1 ; M'_2$ gives the required result.

- If M is $(M_1 \mid M_2) ; \textbf{merge}$ then $[\![M ; P]\!](a, b) = [\![P]\!]([\![M_1]\!](a) + [\![M_2]\!](b)) = [\![P]\!]([\![M_1]\!](a)) + [\![P]\!]([\![M_2]\!](b)) = [\![M_1 ; P]\!](a) + [\![M_2 ; P]\!](b)$. By induction we have P'_1 and M'_1 such that P'_1 does not contain **merge**, M'_1 is a normal merge network, and $[\![M_1 ; P]\!] = [\![P'_1 ; M'_1]\!]$. Again by induction we have P'_2 and M'_2 such that P'_2 does not contain **merge**, M'_2 is a normal merge network, and $[\![M_2 ; P]\!] = [\![P'_2 ; M'_2]\!]$. So $[\![M ; P]\!](a, b) = [\![P'_1 ; M'_1]\!](a) + [\![P'_2 ; M'_2]\!](b)$. Therefore $[\![M ; P]\!] = [\![((P'_1 ; M'_1) \mid (P'_2 ; M'_2)) ; \textbf{merge}]\!] = [\![(P'_1 \mid P'_2) ; (M'_1 \mid M'_2) ; \textbf{merge}]\!]$. Taking $P' = P'_1 \mid P'_2$ and $M' = (M'_1 \mid M'_2) ; \textbf{merge}$ gives the required result. □

Lemma 2. *(Deferring merge) Let P be a normal flowchart. There exist flowcharts P' and M such that P' does not contain* **merge**, *M is a normal merge network, and $[\![P]\!] = [\![P' ; M]\!]$.*

Proof. By induction on the construction of P. The base cases are all trivial: either P is a basic node different from **merge**, and we take M to be the identity permutation; or P is a permutation, and we again take M to be the identity permutation; or P is **merge**, and we take P' to be the identity permutation and take $M = \textbf{merge}$.

- If P is $(P_1 \mid P_2) ; \textbf{merge}$ then by induction we have P'_1, M_1, P'_2, M_2 such that P'_1 and P'_2 do not contain **merge**, M_1 and M_2 are normal merge networks, $[\![P_1]\!] = [\![P'_1 ; M_1]\!]$ and $[\![P_2]\!] = [\![P'_2 ; M_2]\!]$. Now $[\![P]\!] = [\![(P_1 \mid P_2) ; \textbf{merge}]\!] = ([\![P_1]\!] \oplus [\![P_2]\!]); \textbf{merge} = ([\![P'_1 ; M_1]\!] \oplus [\![P'_2 ; M_2]\!]); \textbf{merge} = ([\![P'_1]\!] \oplus [\![P'_2]\!]) ; ([\![M_1]\!] \oplus [\![M_2]\!]) ; \textbf{merge} = [\![P'_1 \mid P'_2]\!] ; [\![M_1 \mid M_2]\!] ; \textbf{merge} = [\![P'_1 \mid P'_2]\!] ; [\![(M_1 \mid M_2) ; \textbf{merge}]\!]$, so we can take $P' = P'_1 \mid P'_2$ and $M = (M_1 \mid M_2) ; \textbf{merge}$.
- If P is $P_1 ; P_2$ then by induction we have P'_1, M_1, P'_2, M_2 such that P'_1 and P'_2 do not contain **merge**, M_1 and M_2 are normal merge networks, $[\![P_1]\!] = [\![P'_1 ; M_1]\!]$ and $[\![P_2]\!] = [\![P'_2 ; M_2]\!]$. Now $[\![P_1 ; P_2]\!] = [\![P'_1 ; M_1]\!] ; [\![P'_2 ; M_2]\!] = [\![P'_1 ; M_1 ; P'_2 ; M_2]\!]$. By Lemma 1 there exist P''_2 and M''_1 such that P''_2 does not contain **merge**, M''_1 is a normal merge network, and $[\![M_1 ; P'_2]\!] = [\![P''_2 ; M''_1]\!]$. We can therefore take $P' = P'_1 ; P''_2$ and $M = M''_1 ; M_2$. □

Lemma 3. *If $\vdash \langle\Gamma\rangle\ P\ \langle\Delta\rangle$ is a QPL program then $(\!|P|\!)$ is a normal flowchart.*

Proof. The only place in which the translation introduces horizontal composition is

$$(\!|\textbf{measure}\ q\ \textbf{then}\ S\ \textbf{else}\ T|\!) = \textbf{measure}\ q ; ((\!|S|\!) \mid (\!|T|\!)) ; \textbf{merge},$$

which constructs a normal flowchart. □

Proposition 1. *(Semantic consistency) Suppose $\vdash \langle\Gamma\rangle\ P\ \langle\Delta\rangle$ is a QPL program. If P is functional and $\tau \xrightarrow{P} \tau'$ then $[\![P]\!](\tau) = \tau'$.*

Proof. We also write P for the flowchart representation of the program, and we have $\vdash\ [\Gamma]\ P\ [\Delta]$. By Lemma 3, P is a normal flowchart. By Lemma 2 there exist P' and M such that P' does not contain **merge**, M is a normal merge network, and $[\![P]\!] = [\![P' ; M]\!] = [\![P']\!] ; [\![M]\!]$. We have $[\![P']\!] : [\![\Gamma]\!] \to [\![\Delta]\!] \oplus \cdots \oplus [\![\Delta]\!]$ and $[\![M]\!] : [\![\Delta]\!] \oplus \cdots \oplus [\![\Delta]\!] \to [\![\Delta]\!]$. Now P' consists of all the potential branches when P is simulated. For a given input state (tableau) τ, some branches might not occur because they correspond to measurement outcomes that are not possible for τ. We have $[\![P']\!](\tau) = (\tau_1, \ldots, \tau_n)$ where each τ_i is either a stabilizer tableau or 0, representing absence of a branch. For each non-zero τ_i we have $\tau \xrightarrow{P} \tau_i$ and because P is functional, all of these τ_i are equal. Let the common value be τ'. Then $[\![M]\!](\tau_1, \ldots, \tau_n) = \tau_1 + \cdots + \tau_n$ (including zero terms), which is equal to τ' up to normalisation. □

3.3 Equality of Stabilizer States

Because the representation of stabilizer states by tableaus is not unique, we need an algorithm to test tableaus for the equality of the states that they represent. Let $|\phi_1\rangle$ and $|\phi_2\rangle$ be stabilizer states with stabilizer groups G_1 and G_2, represented by tableaus τ_1 and τ_2. It is standard that $|\phi_1\rangle = |\phi_2\rangle$ if and only if $G_1 = G_2$. Each group G_i is generated by the rows $g_{i,1}, \ldots, g_{i,n}$ of its tableau, assuming that we are working with n qubits. G_1 and G_2 are equal if and only if each generator $g_{1,j}$ of G_1 can be produced by a combination of the generators of G_2, and vice versa. Multiplication of generators corresponds to row operations on the tableau, so we check that each row of τ_1 is linearly dependent on the rows of τ_2, and vice versa.

Linear dependence can be checked by calculating the rank of a set of generators (tableau rows). The RREF (row-reduced echelon form) algorithm of Audenaert and Plenio [9] uses row operations to put a tableau into a form in which the rank is revealed as the number of rows that do not consist only of identity operators. Using RREF, we can calculate $\mathsf{rank}(g_1, \ldots, g_m)$ for any set of rows.

Assuming that we are working with n-qubit states, it is possible that $\mathsf{rank}(\tau_i) < n$ if $|\phi_i\rangle$ is a mixed state. Let $r_i = \mathsf{rank}(\tau_i) = \mathsf{rank}(g_{i,1}, \ldots, g_{i,n})$. If $r_1 \neq r_2$ then $|\phi_1\rangle \neq |\phi_2\rangle$. Otherwise, $|\phi_1\rangle = |\phi_2\rangle$ if and only if $\mathsf{rank}(g_{1,1}, \ldots, g_{1,n}, g_{2,1}, \ldots, g_{2,n}) = r_1 = r_2$.

An alternative way of checking equality is to use the inner product algorithm for stabiliser tableaus [2] and the fact that $|\phi_1\rangle = |\phi_2\rangle$ if and only if $\langle\phi_1|\phi_2\rangle = 1$.

3.4 The Stabilizer Basis

When checking equivalence of QPL programs, we simulate them with inputs taken from a basis for the space of Hermitian matrices. For stabilizer simulation, we need a basis that consists of density matrices of stabilizer states, as previously defined by Gay [21]. We summarise the construction and proof here in order to explicitly describe the basis. It is straightforward to implement the construction of the stabilizer tableaus for these basis elements.

We denote the standard basis for n-qubit states by $\{|x\rangle \mid 0 \leqslant x < 2^n\}$, considering numbers to stand for their n-bit binary representations. For simplicity, we omit normalization factors. With this notation, for $n \geqslant 1$ let $\mathsf{GHZ}_n = |0\rangle + |2^n - 1\rangle$ and $\mathsf{iGHZ}_n = |0\rangle + i|2^n - 1\rangle$, as n-qubit states.

Lemma 4. *([21]) For all $n \geqslant 1$, GHZ_n and iGHZ_n are stabilizer states.*

Lemma 5. *([21]) If $n \geqslant 1$ and $0 \leqslant x, y < 2^n$ with $x \neq y$ then $|x\rangle + |y\rangle$ and $|x\rangle + i|y\rangle$ are stabilizer states.*

Theorem 2. *([21]) The space of $2^n \times 2^n$ Hermitian matrices, considered as a $(2^n)^2$-dimensional real vector space, has a basis consisting of density matrices of n-qubit stabilizer states.*

Proof. The obvious basis is the union of

$$\{|x\rangle\langle x| \mid 0 \leqslant x < 2^n\} \tag{1}$$
$$\{|x\rangle\langle y| + |y\rangle\langle x| \mid 0 \leqslant x < y < 2^n\} \tag{2}$$
$$\{-i|x\rangle\langle y| + i|y\rangle\langle x| \mid 0 \leqslant x < y < 2^n\}. \tag{3}$$

Now consider the union of

$$\{|x\rangle\langle x| \mid 0 \leqslant x < 2^n\} \tag{4}$$
$$\{(|x\rangle + |y\rangle)(\langle x| + \langle y|) \mid 0 \leqslant x < y < 2^n\} \tag{5}$$
$$\{(|x\rangle + i|y\rangle)(\langle x| - i\langle y|) \mid 0 \leqslant x < y < 2^n\}. \tag{6}$$

This is also a set of $(2^n)^2$ states, and it spans the space because we can obtain states of forms (2) and (3) by subtracting states of form (4) from those of forms (5) and (6). Therefore it is a basis, and by Lemma 5 it consists of density matrices of stabilizer states. □

3.5 Equivalence Checking

We can now combine the ideas from previous sections to describe the algorithm for equivalence checking. Given a QPL program P, there is a corresponding flowchart. By Lemma 2 there is an equivalent flowchart consisting of a merge-free flowchart combined with a merge network. Denote the merge-free part by P_f.

The structure of P_f is a tree in which the leaf nodes correspond to outputs. This is a tree of potential execution paths. According to the semantics of flowcharts (Figure 8), the denotational semantics of P for input τ is the sum of the density matrices at the leaf nodes. Some leaf nodes have a zero density matrix, because they arise from measurement outcomes that are not possible for the particular input τ; the branches leading to these leaf nodes are not simulated.

For a given input τ, let $\mathsf{paths}(P, \tau)$ be the set of simulated paths from the root node to leaf nodes. For a particular path p, let $\mathsf{result}(P, \tau, p)$ be the output when p is simulated. We denote equivalence of stabilizer tableaus τ_1 and τ_2 by $\tau_1 \cong \tau_2$. We write $\mathcal{B}$ for the stabilizer basis on the appropriate number of qubits.

Figure 10 shows the algorithm for checking equivalence of programs P_1 and P_2. It is presented as if $\mathsf{paths}(P_i, \tau)$ is known in advance, but actually the paths are calculated during simulation because this is when it becomes known that certain measurement outcomes are possible or impossible. In order to generate paths, we apply the following scheduling procedure. We start running the stabilizer simulation algorithm, and when we reach a random measurement outcome (determined by looking at the stabilizer tableau representation of the state which we intend to measure), we store both configurations after measurement. This will result in a program tree in which all execution paths can be generated by applying depth first search.

Remark 1. *The overall complexity of the above algorithm is $O(2^{2n+L}(m+n)^3)$, where n is the number of input qubits, m is the number of qubits inside the programs (i.e. those created by* ***new qbit****) and L is the length of program (this is the worst case where we assume the program consists of only measurement). The complexity of the stabilizer simulation algorithm is $O((m+n)^2)$ and the equality test has complexity $O((m+n)^3)$, where we run them on 2^L interleavings for 2^{2n} basis inputs.*

4 SPECIFICATION OF CONCURRENT QUANTUM SYSTEMS

We now present a concurrent language, CCS^q, for specifying quantum protocols, and illustrate it with a concurrent version of the teleportation protocol. CCS^q is based on the classical process calculus CCS [37] and is broadly similar to the quantum process calculi *qCCS* [43] and *CQP* [22]. The aim is to explicitly describe the structure of a quantum system as a collection of communicating components, in order to enable analysis of its communication behaviour as well as its quantum information processing behaviour.

A CCS^q program consists of a collection of processes which are described in a similar language to QPL. Additionally, processes can send classical and quantum data to each other along synchronous communication channels. Synchronous communication is semantically simpler, as there is no need to consider message queues or buffers; it also seems more realistic for current quantum technology, because the existence of queues for quantum messages would imply the existence of stable quantum memory.

The syntax of CCS^q is defined in Figure 11. We assume an infinite supply of identifiers, indicated by q or x in the grammar. We write v for a value, which can be either a boolean literal or a variable. We use the notation $\widetilde{\ }$ for finite sequences. The top-level syntactic category is a *process*. Termination is represented by **nil**, and parallel composition of P_1 and P_2 is $P_1 \mid P_2$. Processes can also be constructed by prefixing. The process $c!v \, . \, P$ sends the message v, which can be classical

```
for all $\tau \in \mathcal{B}$ do
    for all $i \in \{1, 2\}$ do
        $paths_i = \{p_{i,1}, \ldots, p_{i,n_i}\} = \mathsf{paths}(P_i, \tau)$
        $\tau_i = \mathsf{result}(P_i, \tau, p_{i,1})$
        for all $q \in \{p_{i,2} \ldots, p_{i,n_i}\}$ do
            if $\mathsf{result}(P_i, \tau, q) \ncong \tau_i$ then
                return $P_i$ non-functional
            end if
        end for
    end for
    if $\tau_1 \ncong \tau_2$ then
        return $P_1 \ncong P_2$
    end if
end for
return $P_1 \cong P_2$
```

Fig. 10. Algorithm for checking equivalence of QPL programs.

$$
\begin{array}{llll}
\text{Booleans} & b & ::= & 0 \mid 1 \\
\text{Prefixes} & e & ::= & x := \textbf{measure } q \mid U(\widetilde{q}) \mid \textbf{if } x \textbf{ then } U(\widetilde{q}) \mid \textbf{match } \widetilde{x} : \widetilde{b} \textbf{ then } U(\widetilde{q}) \\
 & & \mid & \textbf{new qbit } q \mid \textbf{input } \widetilde{q} \mid \textbf{output } \widetilde{q} \mid c!v \mid c?x \\
\text{Processes} & P & ::= & \textbf{nil} \mid (P \mid P) \mid e \,.\, P
\end{array}
$$

Fig. 11. Syntax of CCS^q

or quantum, on channel c and then continues with P. The process $c?x \,.\, P$ receives a classical or quantum message on channel c and binds it to the variable x, then continues with P. The process $E \,.\, P$ executes the expression E, which can have several forms, and then continues with P.

The forms of expressions E cover non-communication operations. They are similar to the constructs of QPL. The initial input and final output are defined by **input** $\widetilde{q}$ and **output** $\widetilde{q}$ respectively. Application of a unitary operator is expressed by the syntax $U(\widetilde{q})$. Conditional behaviour on the result of a measurement is separated into two constructs: x := **measure** q introduces a classical variable x to store the result, and **if** x **then** $U(\widetilde{q})$ conditionally applies a unitary operator depending on the value of a classical variable. Qubits are introduced by the **new qbit** construct, just as in QPL. Finally, the **match** construct generalises the conditional construct for convenience in expressing more complex conditions.

In order to state the correctness of the semantic definitions in Section 5, we define a type system for CCS^q by the rules in Figure 12. Types are **bit**, **qbit** and **chan**(T), which is the type of channels carrying values of type T. Some previous work on quantum process calculus [22] has used the type system to guarantee that parallel processes do not share access to qubits. We do not include this condition here, in order to simplify the semantic definitions.

The concurrent model of quantum teleportation can be described in CCS^q as follows. Three components interact with each other: *EPR*, *Alice* and *Bob*. These are modelled as processes that run in parallel. The process *EPR* prepares an entangled pair of qubits and sends one to *Alice* on

$$(bit) \qquad \Gamma \vdash 0, 1 : \mathbf{bit}$$

$$(newqbit) \qquad \frac{\Gamma, q : \mathbf{qbit} \vdash P}{\Gamma \vdash \mathbf{new\ qbit}\ q \ .\ P}$$

$$(output) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash \mathbf{output}\ \widetilde{q} \ .\ P}$$

$$(input) \qquad \frac{\Gamma, \widetilde{q} : \widetilde{\mathbf{qbit}} \vdash P}{\Gamma \vdash \mathbf{input}\ \widetilde{q} \ .\ P}$$

$$(send) \qquad \frac{\Gamma \vdash c : \mathbf{chan}(T), v : T \quad \Gamma \vdash P}{\Gamma \vdash c!v \ .\ P}$$

$$(receive) \qquad \frac{\Gamma \vdash c : \mathbf{chan}(T) \quad \Gamma, x : T \vdash P}{\Gamma \vdash c?x \ .\ P}$$

$$(measure) \qquad \frac{\Gamma \vdash q : \mathbf{qbit} \quad \Gamma, x : \mathbf{bit} \vdash P}{\Gamma \vdash x \ := \ \mathbf{measure}\ q \ .\ P}$$

$$(unitary) \qquad \frac{\Gamma \vdash q_1 : \mathbf{qbit}, \dots, q_n : \mathbf{qbit} \quad \Gamma \vdash P \quad U \text{ has arity } n}{\Gamma \vdash U(\widetilde{q}) \ .\ P}$$

$$(cond) \qquad \frac{\Gamma \vdash q_1 : \mathbf{qbit}, \dots, q_n : \mathbf{qbit}, x : \mathbf{bit} \quad \Gamma \vdash P \quad U \text{ has arity } n}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ U(\widetilde{q}) \ .\ P}$$

$$(match) \qquad \frac{\Gamma \vdash q_1 : \mathbf{qbit}, \dots, q_n : \mathbf{qbit}, x_1 : \mathbf{bit}, \dots, x_m : \mathbf{bit} \quad \Gamma \vdash P \quad U \text{ has arity } n}{\Gamma \vdash \mathbf{match}\ x_1 : b_1, \dots, x_m : b_m\ \mathbf{then}\ U(\widetilde{q}) \ .\ P}$$

$$(nil) \qquad \Gamma \vdash \mathbf{nil}$$

$$(parallel) \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}$$

Fig. 12. Typing rules for *CCS*^{q} programs

channel c and one to *Bob* on channel d. *Alice* inputs the qubit to be teleported, then receives one of the entangled qubits on channel c and performs the appropriate operations before making two measurements. The results of the measurements are sent to *Bob* on channel b. *Bob* applies a suitable transformation to his part of the entangled pair, in order to recover the state of the original qubit. This is shown in Figure 13.

In the concurrent model the physical separation of *Alice* and *Bob* is evident, a feature that cannot be captured easily within the quantum circuit model or a QPL program. Also, the communication

```
// Preparing EPR pair (y,z) and sending y to Alice and z to Bob:

newqubit y . newqubit z . H(y) . CNOT(y,z) . c!y . d!z . nil          |

// Alice's process:

(input x . c?y . CNOT(x,y) . H(x) . m := measure x . n := measure y .
 b!m . b!n . nil                                                      |

// Bob's process:

d?w . b?m . b?n . if n then X(w) . if m then Z(w) . output w . nil )
```

Fig. 13. Quantum teleportation

```
// The desired behaviour of teleportation: input and output the same qubit

input x . output x . nil
```

Fig. 14. The specification of quantum teleportation

between *EPR*, *Alice* and *Bob* is made explicit. The specification of quantum teleportation is a simple process, that just passes the input to the output. This is shown in Figure 14.

As an interesting example to show how the design of concurrent quantum protocols can be non-intuitive, suppose that in the *Alice* process of Figure 13, quantum measurements and sending outcomes are run concurrently.

```
m := measure x . b!m | n := measure y . b!n
```

Then specification and implementation become non-equivalent, because the order in which *Bob* receives the two measurement outcomes is no longer fixed.

Although CCS^q has a simple structure, its key advantage over quantum circuits is the ability to explicitly model aspects of protocols that are not visible in a circuit diagram. There is potential to extend CCS^q with additional programming constructs, to increase its flexibility.

5 SEMANTICS OF CCS^q

For each CCS^q program we define a set of QPL programs, which differ in the interleaving of concurrent operations. When we check equivalence of CCS^q programs, we first check that they implement deterministic functions, which means that their behaviour does not depend on the choice of interleaving.

Figure 15 defines transition rules that generate sequentialisations of CCS^q programs. We use them to define the translation from CCS^q to QPL. Because we have restricted QPL to quantum data, for simplicity, the translation represents the classical results of measurement by qubits in basis states. This could be optimised to avoid increasing the size of the stabilizer tableaus.

It is straightforward to prove the following result about transitions of typed CCS^q programs.

Lemma 6. *If* $\Gamma \vdash P$ *and* $P \xrightarrow{\alpha} Q$ *then there exists* Δ *such that* $\Delta \vdash Q$ *and* $\Gamma \subseteq \Delta$.

(*input*)	$\textbf{input}\ \widetilde{q}\ .\ P \xrightarrow{\mathsf{input}\ \widetilde{q}} P$
(*output*)	$\textbf{output}\ \widetilde{q}\ .\ P \xrightarrow{\mathsf{output}\ \widetilde{q}} P$
(*new*)	$\textbf{new qbit}\ q\ .\ P \xrightarrow{\mathsf{new\ qbit}\ q} P$
(*unitary*)	$U(\widetilde{q})\ .\ P \xrightarrow{U(\widetilde{q})} P$
(*measure*)	$x\ :=\ \textbf{measure}\ q\ .\ P \xrightarrow{\mathsf{measure}(x,q)} P$
(*conditional*)	$\textbf{if}\ x\ \textbf{then}\ U(\widetilde{q})\ .\ P \xrightarrow{\mathsf{cond}(x,U(\widetilde{q}))} P$
(*match*)	$\textbf{match}\ \widetilde{x} : \widetilde{b}\ \textbf{then}\ U(\widetilde{q})\ .\ P \xrightarrow{\mathsf{match}(\widetilde{x}:\widetilde{b},U(\widetilde{q}))} P$
(*communication*)	$c!v\ .\ P \mid c?x\ .\ Q \xrightarrow{\mathsf{comm}} P \mid Q[v/x]$
(*parallel-left*)	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$
(*parallel-right*)	$\dfrac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$

Fig. 15. Transition rules for sequentialising CCS^q

For a transition label α, we define a QPL statement $T(\alpha)$ as follows.

$$
\begin{array}{lcl}
T(\mathsf{input}\ \widetilde{q}) & = & \textbf{skip} \\
T(\mathsf{output}\ \widetilde{q}) & = & \textbf{skip} \\
T(\mathsf{new\ qbit}\ q) & = & \textbf{new qbit}\ q\ :=\ 0 \\
T(U(\widetilde{q})) & = & \widetilde{q}\ {*}{=}\ U \\
T(\mathsf{cond}(x,U(\widetilde{q}))) & = & \textbf{measure}\ x\ \textbf{then}\ \widetilde{q}\ {*}{=}\ U\ \textbf{else skip} \\
T(\mathsf{match}(\widetilde{x} : \widetilde{b}, U(\widetilde{q}))) & = & (M(\widetilde{x} : \widetilde{b}, U(\widetilde{q})) \\
T(\mathsf{measure}(x,q)) & = & \textbf{new qbit}\ x\ :=\ 0\ ;\ \textbf{measure}\ q\ \textbf{then}\ x\ {*}{=}\ X\ \textbf{else skip} \\
T(\mathsf{comm}) & = & \textbf{skip}
\end{array}
$$

The translation of match labels depends on the function M that translates a sequence $\widetilde{x} : \widetilde{b}$ of conditions into a nested structure of conditional statements.

$$
\begin{array}{lcl}
M(\epsilon, U(\widetilde{q})) & = & \widetilde{q}\ {*}{=}\ U \\
M((x:0,t), U(\widetilde{q})) & = & \textbf{measure}\ x\ \textbf{then skip else}\ M(t, U(\widetilde{q})) \\
M((x:1,t), U(\widetilde{q})) & = & \textbf{measure}\ x\ \textbf{then}\ M(t, U(\widetilde{q}))\ \textbf{else skip}
\end{array}
$$

Now for a typed CCS^q program $\Gamma \vdash P$ and a sequence σ of transition labels such that $P \xrightarrow{\sigma}{}^{*} P'$, we define a typed QPL program $T(\Gamma \vdash P, \sigma)$ by induction on σ, as follows. For a type environment Γ, we write $\overline{\Gamma}$ for the environment obtained by replacing **qbit** by **bit**.

$$\begin{array}{lcll} T(\Gamma \vdash P, \varepsilon) & = & \langle\overline{\Gamma}\rangle\ \mathbf{skip}\ \langle\overline{\Gamma}\rangle & \\ T(\Gamma \vdash P, \alpha\ .\ \sigma) & = & \langle\overline{\Gamma}\rangle\ T(\alpha)\ \langle\overline{\Delta}\rangle\ ; T(\Delta \vdash Q, \sigma) & \text{where } P \xrightarrow{\alpha} Q \text{ and } \Delta \vdash Q \end{array}$$

The appearance of **skip** in the translation is a technical convenience which could be avoided by combining the definitions of the translation of individual labels and the translation of sequences of labels. Extra **skip** statements are harmless because they correspond to identity functions.

It is straightforward to check that the correctness conditions for CCS^q programs, expressed by the type system, match those of QPL programs.

Proposition 2. *If $\Gamma \vdash P$ and $P \xrightarrow{\sigma}{}^{*} Q$ then $T(\Gamma \vdash P, \sigma)$, which is of the form $\langle\overline{\Gamma}\rangle\ R\ \langle\overline{\Delta}\rangle$, is derivable in the type system of QPL.*

In the implementation, we integrate construction of the sequentialisations with exploring all paths in each sequentialisation. During this process, we also check that the choice of sequentialisation does not affect the input/output behaviour of the program.

Definition 6. *Given a typed CCS^q program $\Gamma \vdash P$, its set of sequentialisations is defined by*

$$S(P) = \{T(\Gamma \vdash P, \sigma) \mid P \xrightarrow{\sigma}{}^{*} P'\}.$$

Definition 7. *A CCS^q program Q is* functional *if*

(1) *for all QPL programs $P \in S(Q)$, P is functional, and*
(2) *for all QPL programs $P, P' \in S(Q)$, $P \cong P'$.*

The semantics of a functional CCS^q program is the superoperator that is the semantics of all of its sequentialisations.

Example: Consider the CCS^q program

$$\mathbf{input}\ q\ .\ H(q)\ .\ c!q\ .\ \mathbf{nil} \mid c?r\ .\ H(r)\ .\ \mathbf{output}\ r\ .\ \mathbf{nil}$$

It has a unique sequentialisation, corresponding to the sequence of transitions shown in Figure 16. From this sequence of transitions we derive the following QPL program. The **skip** statements come from the translations of **input** q, **output** r and the communication on c.

$$\langle q : \mathbf{qbit}\rangle\ \mathbf{skip}\ ; H \ \mathrel{*}= \ q\ ; \mathbf{skip}\ ; H \ \mathrel{*}= \ q\ ; \mathbf{skip}\ \langle q : \mathbf{qbit}\rangle$$

6 AUTOMATED VERIFICATION OF CONCURRENT QUANTUM SYSTEMS

Our equivalence-checking tool for concurrent quantum protocols, QEC (Quantum Equivalence Checker) [5], was originally described in [7]. In this section we summarise how we combine the ideas of Sections 4 and 5 to check equivalence of CCS^q programs, and then discuss the implementation of QEC.

6.1 Equivalence-checking algorithm for concurrent quantum protocols

To check equivalence of concurrent protocols in CCS^q, we combine the sequentialisation and translation of CCS^q into QPL (Section 5) and the equivalence-checking algorithm for QPL (Section 3). CCS^q programs are simulated by a scheduler that generates sequentialisations and explores all execution paths of each sequentialisation. The rest of the analysis is exactly the same as for QPL. The description of the algorithm is the same as in Figure 10; the only change is that paths now includes sequentialisation.

The cost of running the equivalence-checking algorithm increases exponentially in the size of input qubits, because of iteration over all basis states, and the number of concurrent processes,

$$\begin{tikzcd}
\textbf{input}\ q\ .\ H(q)\ .\ c!q\ .\ \textbf{nil} \mid c?r\ .\ H(r)\ .\ \textbf{output}\ r\ .\ \textbf{nil} \arrow[d, "\text{input}\ q"] \\
H(q)\ .\ c!q\ .\ \textbf{nil} \mid c?r\ .\ H(r)\ .\ \textbf{output}\ r\ .\ \textbf{nil} \arrow[d, "H(q)"] \\
c!q\ .\ \textbf{nil} \mid c?r\ .\ H(r)\ .\ \textbf{output}\ r\ .\ \textbf{nil} \arrow[d, "\text{comm}"] \\
\textbf{nil} \mid H(q)\ .\ \textbf{output}\ q\ .\ \textbf{nil} \arrow[d, "H(q)"] \\
\textbf{nil} \mid \textbf{output}\ q\ .\ \textbf{nil} \arrow[d, "\text{output}\ q"] \\
\textbf{nil} \mid \textbf{nil}
\end{tikzcd}$$

Fig. 16. A sequence of transitions for a concurrent protocol

because of scheduling to determine interleavings. The following proposition gives the computational complexity of our equivalence-checking algorithm. Note that in classical computing, the equivalence-checking problem or *implementation verification* of concurrent systems (where only the containment problem is considered, not the simulation problem), is *PSPACE-complete* (see [28] for details).

Proposition 3. *Checking equivalence of concurrent quantum protocols has overall time complexity of $O(N\,2^{2n}(m+n)^3)$, where n is the number of input qubits (basis size), m is the number of qubits inside a program (*i.e *those created by* **newqbit**) *and N is the number of interleavings of processes (where $N = \frac{(\sum_i^M n_i)!}{\prod_i^M (n_i!)}$ for M processes each having n_i atomic instructions)*.

The analysis in Proposition 3 is based on the three phases of our algorithm, namely: scheduling (where N comes from), basis generation (factor 2^{2n}) and stabilizer simulation and equality test (factor $(m+n)^3$, see Remark 1).

6.2 The QEC tool

QEC is implemented in Java and consists of around 30k lines of code. The core is the scheduler, which implements the paths function used by the algorithm in Figure 10. The implementation of the scheduler and the stabilizer simulation operations are designed for clarity and extensibility, and have not been heavily optimised. The structure of QEC is illustrated in Figure 17.

QEC is a command-line tool that is given two files containing CCS^q models, and checks them for equivalence. There is no intrinsic difference between specifications and implementations; any two models can be compared. The output of QEC consists of a report of the number of basis states generated, the number of simulated runs for each model, the final result of equivalence-checking, and the time taken. If the models are not equivalent, then QEC shows a basis state that caused the models to produce different outputs. An outline example of QEC's output is shown in Figure 18.

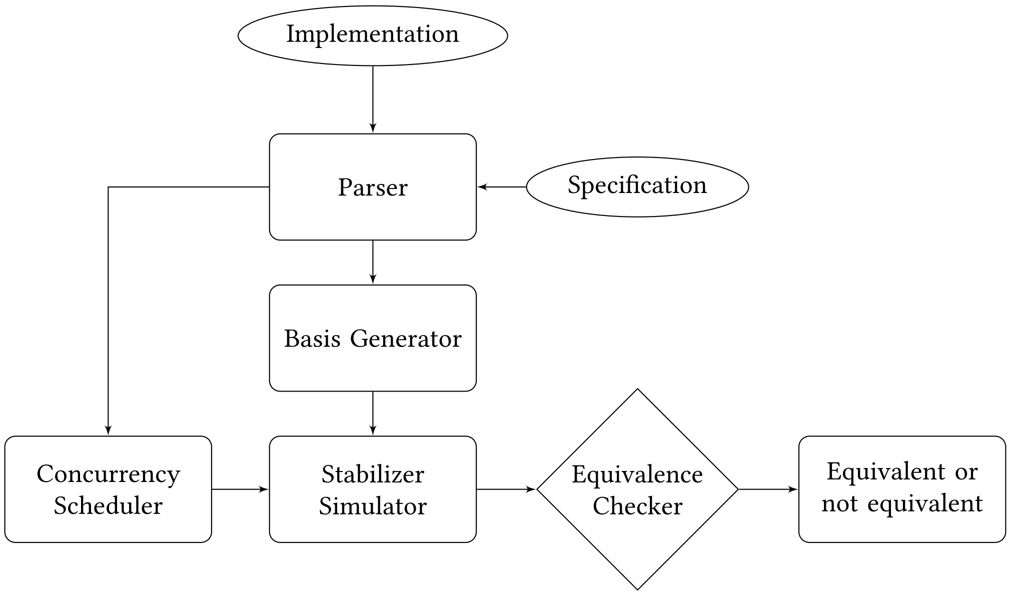

Fig. 17. Structure of QEC

```
> check Identity.ccs Teleportation.ccs
>
>
> Basis generated with 4 states.
> Final Evaluation:true
> Equivalence Checked on Spec: 4, Imple: 400 runs.
> Verification Time: 138 milliseconds
```

Fig. 18. QEC output for the specification (Figure 14) and implementation (Figure 13) of teleportation.

7 CASE STUDIES AND EXPERIMENTATION

7.1 Case Studies

We now describe several case studies that we have verified with QEC. These case studies are chosen from a range of areas within quantum information processing: quantum communication, quantum cryptography, quantum error-correction, and fault tolerant quantum computation. Together, they show that our approach and tool can be used to analyse a variety of important protocols.

For each protocol, we model the implementation and the specification in the concurrent language CCS^q. Sequential models of each protocol in QPL have previously been presented by Ardeshir-Larijani [8]. Our results, in Figure 30, compare the performance of verification using the CCS^q and QPL models.

7.1.1 Dense Coding. This is another quantum communication protocol that takes advantage of entanglement [12]. *Alice* communicates two classical bits to *Bob* by sending only one qubit. Because a pair of entangled qubits must also be distributed to *Alice* and *Bob*, there is no real reduction in

```
// Inputs x and y represent classical bits: only |0> and |1> are simulated

// Preparing EPR pair (a,b) and sending a to Alice and b to Bob:

newqubit a. newqubit b . H(a). CNOT(a,b) . c!a . d!b . nil          |

// Alice's process:

(input x,y . c?a . m := measure x . n := measure y .
if m then Z(a) . if n then X(a) . q!a .nil                          |

// Bob's process:

d?b . q?a . CNOT(a,b) . H(a) . output a,b . nil )
```

Fig. 19. Dense coding

```
// Input and output the same pair of qubits, representing classical bits

input x,y . output x,y . nil
```

Fig. 20. The specification of dense coding

the amount of communication; however, the protocol illustrates the effects that can be achieved through entanglement.

Our model of dense coding, shown in Figure 19, encodes the classical input bits as standard basis states of qubits. When we check equivalence, the protocol is simulated only for the standard basis states, not for the full stabilizer basis. Similarly to teleportation, there are three processes: *EPR*, *Alice* and *Bob*, where *Alice* receives the input and *Bob* produces the output of the protocol. The specification (Figure 20) is similar to the specification of teleportation but with two qubits.

An alternative way of analysing dense coding is to extend the equivalence checker to work with classical bits. In this case the inputs `x` and `y` are classical, and we replace *Alice's* measurements and conditional operations with

```
if x then Z(a) . if y then X(a)
```

This means that the semantics of the protocol is a classical binary function, rather than a superoperator, and the analysis does not require the theory from Section 3.

7.1.2 Quantum Secret-Sharing. The problem of secret sharing involves an agent *Alice* sending a message to two agents *Bob* and *Charlie*, one of whom is dishonest. *Alice* does not know which one of the agents is dishonest, so she encodes the message in a way that requires *Bob* and *Charlie* to collaborate in order to retrieve it. We describe a version of a protocol due to Hillery *et al.*[1] [29]. The message, in this case, is a qubit.

[1]There is another quantum secret sharing protocol, called *Graph State Secret Sharing* [35], for sharing a classical bit or a qubit, based on the same idea of using multi-party entangled states.

```
// Preparing GHZ state and sending to Alice, Bob and Charlie:
newqubit a . newqubit b . newqubit c . H(a) . CNOT(a,b) . CNOT(b,c) .
d!a . e!b . f!c . nil                                                |

// Alice, who commits her qubit as a secret:
(input x . d?a . CNOT(x,a) . H(x) . m := measure x .
n := measure a . t!m . w!n . nil                                     |

// Bob, who is chosen as a collaborator:
(e?b . H(b) . o := measure b . u!o .  nil                            |

//Charlie, who recovers the original qubit from Alice:
f?c . t?m . w?n . u?o . if o then Z(c) .
if m then X(c) . if n then Z(c) . output c . nil))
```

Fig. 21. The implementation of quantum secret sharing in *CCS*^{q}

```
// Input and output the same qubit

input x . output x . nil
```

Fig. 22. The specification of quantum secret sharing in *CCS*^{q}

The three agents need to share a maximally entangled three-qubit state, the *GHZ* state $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$, prior to the execution of the protocol. We assume that it is *Charlie* who retrieves the original qubit (a variation of the protocol allows *Bob* to retrieve it, or *Alice* can choose who retrieves it). The body of the protocol has two main phases: committing a secret by *Alice*, and collaboration between *Bob* and *Charlie* to retrieve the secret. First, *Alice* entangles the input qubit with her entangled qubit from a previously distributed GHZ state. Then *Alice* measures her qubits and sends the outcome to *Charlie* (committing the secret). *Bob* also measures his qubit and sends the outcome to *Charlie*. Finally, *Charlie* is able to retrieve the original qubit once he has access to the bits from *Alice* and *Bob* (collaboration and retrieval of the secret). The security of this protocol is a consequence of the no-cloning theorem and is discussed in [29].

The implementation of the secret-sharing protocol is shown in Figure 21. The specification of the protocol in *CCS*^{q} is the same as for teleportation: a process that inputs a qubit and immediately outputs it (Figure 22).

7.1.3 Error-Correcting Codes. Error-correction is an important requirement for building practical QIP systems. Several error-correction protocols are based on *stabilizer codes* [39, p. 453]. They can be analysed with QEC because they only require operations from the stabilizer formalism. The general structure of the examples that we consider is as follows. There is a process *Alice* which inputs a qubit and encodes it, an *Error* process which receives encoded data from *Alice* and introduces errors, and a process *Bob* which corrects the errors in the data received from the *Error* process.

We start with two simple codes that are able to correct bit-flip and phase-flip errors. We then discuss the five-qubit code [34], which combines bit-flip and phase-flip correction in such a way

```
// Alice encodes the input and sends three qubits to Error:
input x . newqubit a . newqubit b . CNOT(x,a) . CNOT(x,b) .
c!x . d!a . e!b . nil                                             |

// The Error process randomly flips at most one qubit:
(c?x . d?a . e?b . newqubit w . newqubit z . H(w) . H(z) .

k := measure w . l := measure z . match k:0 and l:1 then X(x) .
match k:1 and l:0 then X(a) . match k:1 and l:1 then X(b) .

f!x . g!a . h!b . nil                                             |

// Bob receives qubits and corrects errors:
f?x . g?a . h?b . newqubit s . newqubit t .

CNOT(x,s) . CNOT(a,s) . CNOT(x,t) . CNOT(b,t) .
m := measure s . n :=measure t .

match m:1 and n:0 then X(a) . match m:0 and n:0 then X(b) .
match m:1 and n:1 then X(x) .

CNOT(x,a) . CNOT(x,b) . output x . nil )
```

Fig. 23. The bit-flip error-correcting code

that it results in protecting a single qubit against a continuum of errors. The five-qubit code has also been implemented in the laboratory [31].

Bit Error Codes. There are two kinds of bit error codes, namely bit-flip and phase-flip codes. The bit-flip code corrects an error that consists of applying the X operator to a qubit. This is the analogue of inverting a classical bit. The phase-flip code corrects an error that consists of inverting the phase of a qubit, in other words applying a Z operator to a qubit. This has no classical analogue.

The encoding phase of a bit error code entangles the input of the protocol, which is the qubit that needs to be protected, with two ancillary qubits. Then errors are introduced, randomly, to the encoded qubit. Quantum measurement is used as a source of randomness. Finally, the original qubit is retrieved by applying the appropriate correcting operations. In the last step, new qubits are introduced and entangled with the message; these qubits are measured and act as *error syndromes*. They are necessary because simply measuring the message qubits would destroy their state.

The phase-flip code applies Hadamard operators in the encoding and decoding phases, which is because the syndrome measurements need to be in the diagonal basis.

The implementation of bit error codes is shown in Figures 23 and 24. The `match` construct is convenient for expressing the cases corresponding to different error possibilities. The specification for these protocols is the same as for teleportation (Figure 14).

In our models, errors (bit or phase-flip) are introduced on at most one qubit. However, in general errors can occur in more than one encoded qubit, resulting in a more complicated scenario. In this case, the original (input) qubit cannot be perfectly recovered, but errors can be corrected with high probability. We are not able to analyse this aspect of the protocols.

```
// Alice encodes the input and sends three qubits to Error:
input x . newqubit a . newqubit b . CNOT(x,a) . CNOT(x,b).
H(x) . H(a) . H(b) .
c!x . d!a . e!b . nil                                        |

// The Error process randomly flips the phase of at most one qubit:
(c?x . d?a . e?b . newqubit w . newqubit z . H(w) . H(z) .

k := measure w . l := measure z . match k:0 and l:1 then Z(x) .
match k:1 and l:0 then Z(a) . match k:1 and l:1 then Z(b) .

f!x . g!a . h!b . nil                                        |

// Bob receives qubits and corrects errors:
f?x . g?a . h?b .
H(x) . H(a) . H(b) .

newqubit s . newqubit t .

CNOT(x,s) . CNOT(a,s) . CNOT(x,t) . CNOT(b,t) .
m := measure s . n := measure t .

match m:1 and n:0 then Z(a) . match m:0 and n:0 then Z(b) .
match m:1 and n:1 then Z(x) .

CNOT(x,a) . CNOT(x,b) . output x . nil )
```

Fig. 24. The phase-flip error-correcting code

Five Qubit Code. The idea of combining phase-flip and bit-flip codes into a single code first appeared in Shor's nine qubit protocol [39, p. 430]. Then it was optimized to the Steane seven qubit protocol. Finally, Laflamme *et al.* [34] showed that there is a five qubit protocol which is equivalent to Shor's and Steane's codes. These codes can protect a single qubit not only from bit-flip and phase-flip errors, but from arbitrary errors.

The CCS^q model of the five qubit code is rather long, because of the large number of combinations of error syndrome measurements that need to be handled. The code can be found in [8]. Analysis of the five qubit code is included in our experimental results in Section 7.2.

Remark 2. *The five qubit code can also protect a single qubit against* adversary *measurements, in which the transmitted qubits are altered by unintended measurements rather than by unitary operations, with high probability. The current version of QEC, which checks exact equivalence, cannot analyse this situation.*

7.1.4 X and Z teleportation. In the standard teleportation protocol, quantum operations are only applied locally inside the *Alice*, *EPR* or *Bob* processes. However, if a CNot operation can be applied to *Alice's* input qubit and *Bob's* qubit, then variations of the protocol are possible in which only one classical bit is sent from *Alice* to *Bob*, and *Bob* only applies one correction to his qubit.

```
// Alice's process:
input x . d!x . f?x . b := measure x . g!b . nil                  |

// Intermediate process for applying joint operations:
(c?a . d?x . CNOT(a,x) . e!a . f!x . nil                          |

// Bob's process:
newqubit a . H(a). c!a . e?a . g?b . if b then X(a). output a . nil)
```

Fig. 25. X-teleportation

```
// Alice's process:
input x . d!x . f?x . H(x) . b := measure x . e!b . nil           |

// Intermediate process for applying joint operations:
( c?a . d?x . CNOT(x,a) . f!x . g!a . nil                         |

// Bob's process:
newqubit a . c!a . g?a . e?b . if b then Z(a) . output a . nil )
```

Fig. 26. Z-teleportation

Moreover, the initial entangled pair of qubits can be replaced by a single qubit allocated by *Bob*. These variations are called X-teleporation and Z-teleportation. The CNot operation is applied locally by first sending the necessary qubits to an intermediate process, then sending them back afterwards.

In the concurrent implementation of X- and Z-teleportation, specified in CCS^q, there are three processes: *Alice*, who sends her qubit; an intermediate process for joint operations; and *Bob*, who receives only one bit of classical information and retrieves *Alice's* qubit state. The models of the implementation of these protocols are shown in Figure 25 and Figure 26. The specification is the same as for teleportation (Figure 14).

7.1.5 Remote CNot *(1).* This is a fault-tolerant quantum protocol in which *Alice* and *Bob* want to perform a joint CNot gate between their qubits without exchanging any qubits with each other. However, they are only allowed to use prior entanglement and classical communication. Here we present a construction of such a protocol introduced in [45].

The idea of this protocol is that *Alice* runs X-teleportation and *Bob* runs Z-teleportation, in parallel. To achieve this, two pairs of entangled qubits must be used along with communicating four classical bits.

The concurrent implementation of the Remote CNot protocol has four processes: *Feeder*, *EPR*, *Alice* and *Bob*. Joint operations between two processes, namely *Alice* and *Bob*, are not allowed. The role of *Feeder* is to distribute input qubits to *Alice* and *Bob*. The *EPR* process distributes an entangled pair to *Alice* and *Bob*. Another pair of qubits will be entangled later by *Bob*.

The implementation and specification of Remote CNot in CCS^q are illustrated in Figures 27 and 28, respectively. In the implementation, *Alice* produces the final output of the protocol; alternatively it could be restructured so that *Bob* produces the final output.

```
// Feeder of inputs:
input x,y . e!x . p!y . nil                                            |

// EPR process sharing entanglement:
(newqubit a . newqubit b . H(a) . CNOT(a,b) . c!a . d!b . nil          |

// Alice's process (Block 1):
( e?x . c?a . CNOT(a,x) . u := measure x .

if u then X(a) . f!u . g?t . if t then Z(a) . h?b . output a,b . nil  |

// Bob's process (Block 2):
p?y . d?b . CNOT (y,b) . H(y). f?u . if u then X(b) .

t := measure y . if t then Z(b) . g!t . h!b . nil ))
```

Fig. 27. Implementation of Remote CNot (1).

```
// Directly apply CNOT to the input qubits
input x,y . CNOT(x,y) . output x,y . nil
```

Fig. 28. Specification of Remote CNot.

7.1.6 Remote CNot *(2).* This version of Remote CNot was introduced in [25]. The structure of this protocol is different in that it does not run two teleportation protocols, and therefore reduces classical communication to only three bits. Figure 29 shows the implementation; the specification is the same as before.

7.2 Experimental Results

Figure 30 presents the results of verification of the case studies that we have described. For each case study, we have used QEC to verify the concurrent model presented earlier in this section, as well as a sequential version of the model. We have also verified each case study with our previous tool SEC [6], which handled only sequential models. Experiments were run on a 2.6GHz Intel Core i7 machine with 16GB RAM.

As well as the running times, we show the number of interleavings for the concurrent models, and the number of branches (resulting from non-deterministic measurements) for the sequential models. The running times for concurrent models are longer than for sequential ones, because of the need to analyse more interleaving generated by concurrency. The running times for sequential models in the two tools are similar but in most cases QEC is slightly faster, which might be explained by its use of an abstract syntax tree instead of a full parse tree. The five-qubit code was not analysed with SEC. This is because the `match` construct necessary for effective definition of the model was not implemented in SEC.

The experimental results show how concurrency affects quantum systems. Not surprisingly, with more sharing of entanglement and increased classical and quantum communication, we have to deal with a larger amount of interleaving, particularly in the last three protocols of Figure 30. However,

```
//Feeder of inputs:
input x,y . e!x . p!y . nil                                             |

//EPR process sharing entanglement:
(newqubit a . newqubit b . H(b) . CNOT(b,a) . c!a . d!b . nil           |

//Alice process (Block 1):
( e?x . c?a . CNOT(x,a) . u := measure a . f!u .

g?t . if t then Z(x) . h?y . output x,y . nil                           |

//Bob process (Block2):
p?y . d?b . CNOT (b,y) . f?u . if u then X(b) .

if u then X(y) . H(b) . t:=measure b . g!t . h!y . nil ))
```

Fig. 29. Implementation of Remote CNot (2).

error correction protocols are inherently sequential and therefore verification of sequential and concurrent models in these cases produce similar results.

An advantage of our approach is that we can change the amount of concurrency in the models. Naturally, increasing the concurrency leads to slower verification due to the larger amount of interleaving. This is illustrated by the two Remote CNot protocols, in which the second version has fewer classical communication and, consequently, less interleaving.

There is a lack of other fully-automatic tools for equivalence-checking of concurrent quantum protocols; this is why our results are based on comparisons between concurrent and sequential models in QEC, and between QEC and our own previous tool SEC. We would like to compare our results with those produced by the model-checker QMC [23], but QMC has not been maintained and we have not been able to run all the examples. Moreover, QMC is based on a different approach to verification i. e., temporal logic model checking, rather than equivalence checking. The Quantomatic [15] tool is a well-developed system to support reasoning about quantum protocols, but it is not fully automatic. Therefore we have not included it in our comparison.

We conclude this section with two remarks. First, consider teleportation again (Figure 5). If we add another qubit x to the inputs and outputs in both the specification and the implementation, then equivalence-checking verifies that q0 is correctly teleported in cases in which it is entangled with x. This property follows from linearity, but it is not usually stated explicitly in standard presentations of teleportation. Similarly for the error-correcting codes: correct transmission of a qubit is guaranteed even when it is entangled with other quantum state. However, secret-sharing does not have this property. Our approach, by explicitly identifying the input and output qubits, enables us to investigate such questions.

Second, we can model different implementations of a protocol, for example by changing the amount of concurrency, as with the sequential (Figure 5) and concurrent (Figure 13) versions of teleportation. These differences are not apparent at the level of circuit diagrams or sequential programs.

Protocol	#Interleavings	QEC/conc	#Branches	QEC/seq	SEC
Teleportation	400	138	16	16	22
Dense Coding	100	75	4	12	17
Bit flip code	16	47	16	38	23
Phase flip code	16	37	16	31	21
Five qubit code	64	198	64	181	*
X-Teleportation	32	53	8	15	16
Z-Teleportation	72	62	8	16	16
Remote CNOT(1)	78400	7359	64	54	63
Remote CNOT(2)	23040	3855	64	52	69
Quantum Secret Sharing	88480	8015	32	34	31

Fig. 30. Experimental results for equivalence-checking of quantum protocols. The columns headed QEC/conc and QEC/seq show the results of verification of concurrent and sequential models of protocols with the QEC tool. Column SEC shows verification times for sequential models in our previous tool for sequential equivalence-checking [6]. The number of branches for the QEC/seq and SEC models are the same. Times are in milliseconds.

8 RELATED WORK

Quantum computation and quantum information processing were originally studied in the quantum circuit model, but the growing complexity of quantum technologies makes it necessary to develop higher-level techniques adapted from classical formal methods. A key characteristic of this approach is the use of structured modelling languages supported by automated analysis tools.

8.1 QMC

Papanikolaou *et al.* developed the QMC model-checking tool [23, 24, 40]. The modelling language is broadly similar to CCS^q, but the approach to specification is property-oriented rather than our process-oriented technique. Specifications are expressed in quantum computation tree logic (QCTL) [10], which combines two aspects: a logic for expressing properties of quantum states, and the standard connectives of temporal logic. Although QCTL has the potential to specify complex temporal properties of protocols, the case studies developed by Papanikoloau *et al.* only use properties of the initial and final quantum states; therefore the expressivity in practice is similar to that of our QEC.

8.2 Probabilistic Model-Checking

Gay and Nagarajan [38] investigated the use of the PRISM probabilistic model-checking system [33], which has been widely used for classical systems, to analyse quantum systems. A major challenge for this approach is the scalability of verifying larger protocols with more qubits, due to lack of an efficient representation of arbitrary quantum states. Also, without a dedicated language, specification of more complicated protocols and their properties becomes a difficult task.

Probabilistic model-checking represents systems in terms of Markov chains, consisting of a set of states and a probabilistic transition function between states. For quantum systems, a quantum Markov chain is defined on a set of quantum states and a set of superoperators that represent the transitions between states. Feng *et al.* [18] introduced a model-checking technique and algorithm for quantum Markov chains. The idea has been implemented in a model-checking tool, QPMC [19], which is an extension of the probabilistic model-checker IscasMC [27].

8.3 Quantum Process Calculus

Our modelling language, CCS^q, is based on previous work on quantum versions of classical process calculus: CQP (Communicating Quantum Processes) [22] and especially qCCS (Quantum Communicating Concurrent Processes) [43]. One aim of research on quantum process calculus was to support equivalence-based specification and verification of quantum systems by developing theories of bisimulation for quantum processes. There has been some theoretical work on bisimulation for CQP [14], and much more for qCCS [16, 17]. The key difference between bisimulation and the input/output equivalence that QEC checks, is that bisimulation allows for inputs and outputs at various points during the lifetime of a system. However, examples of quantum bisimulation in the literature do not yet take advantage of this aspect. Analogously to the situation for classical model-checking, it should be possible to implement tools to check quantum bisimulation. So far, the only work in this direction is by Kubota *et al.* [32], who have developed a semi-automated approach in which the user has to provide equations that can be used to verify bisimulation relationships by equational reasoning. Their technique has been used to verify a quantum key-distribution protocol.

8.4 Quantomatic

The category-theoretic approach to quantum mechanics [3] led to the development of the zx-calculus [13], a graphical formalism for quantum information processing. Quantomatic [15, 30] is a semi-automated tool that uses graph rewriting to verify equivalences between quantum systems described in zx-calculus. Its approach to specification, like ours, is process-oriented, and verification consists of progressively transforming a model of a system into a simpler model which is taken to be the specification. This is a form of symbolic analysis, in constrast to our explicit consideration of particular input states, so there is no restriction to Clifford operators; however, user interaction is required to guide the rewriting.

9 CONCLUSION AND FUTURE WORK

We have presented the theory and implementation of a technique for verifying quantum information processing systems by checking equivalence between a specification process and an implementation process, both defined in a modelling language based on process calculus. The implementation as QEC is the first fully automatic equivalence checking tool for concurrent quantum processes. Checking equivalence means checking that two processes have the same semantics, viewed as superoperators from an input space to an output space. By working in the stabilizer formalism, and restricting attention to processes that use Clifford group operators, we are able to check equality of superoperators by simulating their action on a basis consisting of stabilizer states. We have presented experimental results showing that it is feasible to verify a range of quantum systems with this approach.

There are two limitations of our approach. The first is that we can only analyse protocols and programs that use Clifford operators and can therefore be simulated in the stabilizer formalism. As universal quantum computation cannot be simulated efficiently on classical computers and our techniques are built on simulation, it is reasonable to accept this limitation. The examples in Section 7.1 show that despite this restriction, we can handle a representative range of protocols including teleportation, secret-sharing and error-correction. Although we can model quantum computing, our focus has been on modelling quantum protocols as many of them have been implemented and are used in applications, whereas practical quantum computers are still under development. Regarding the restriction to stabilizer simulation, it seems that realistic quantum algorithms and protocols that are outside the stabilizer formalism have only a few non-stabilizer gates. Stabilizer simulation can be extended to this situation [2] and such an extension could be included in QEC in the future; inevitably this would reduce efficiency. It is also important to note that, although our approach is limited to Clifford operators, we can check equivalence of protocols on arbitrary input states. Currently not many automatic tools exist which can model and analyse quantum protocols, so QEC is a valuable contribution. In the tool LIQ*Ui*|⟩, developed at Microsoft Research, there is a stabilizer simulator in addition to a universal simulator. The stabilizer simulator handles many more qubits than the universal simulator and is faster. Therefore, such tools can be useful in practice. However, we are doing more than just simulation.

The second limitation is that we can only analyse protocols and programs that produce a definite consistent output state for each input state: this is the requirement of *functionality*. It means that we exclude algorithms and protocols that succeed with a certain probability, rather than with certainty. However, the examples in Section 7.1 show that many natural quantum verification problems fall into this category. Again considering quantum protocols rather than programs, this is a natural restriction as it would not be desirable for a protocol to produce different outputs for a given input.

In conclusion, we have established the principle of verification of quantum systems via formal modelling and automatic equivalence-checking. We have shown that it works well with stabilizer simulation and is a practical approach to the verification of quantum protocols.

Future work will address at least the following areas; we are already working on the first four points.

- Classical model-checkers usually provide diagnostic information when a specification is not satisfied, in the form of a trace of execution steps leading to failure. We have not yet implemented such diagnostics in QEC, although it is not conceptually difficult.
- Some quantum protocols and algorithms are not intended to work exactly, but only up to a certain probability of error. It would be possible to extend QEC so that it calculates the probability of failure of a specification, instead of simply reporting failure in an absolute sense.
- There are techniques for extending the stabilizer formalism to handle quantum states in which some qubits are in non-stabilizer states, or in which some non-Clifford operators are applied [2]. This increases the complexity of simulation, but in a way that depends only on the non-stabilizer or non-Clifford parts of the system. It would be interesting to implement these techniques in QEC.
- The CCS^q language lacks programming constructs such as loops, and data structures such as arrays. It should be straightforward to add these features in order to support the analysis of more complex systems.
- We have only considered systems that take an initial input and produce a final output, so that their behaviour can be interpreted as a superoperator. Theories of quantum bisimulation [14, 16, 17] define equivalence between systems that receive inputs and produce outputs repeatedly at arbitrary points during their execution. An open question is whether our stabilizer-based technique can be generalised from input/output behaviour to bisimulation.
- The concept of verification as equivalence-checking, as implemented in QEC, doesn't cover all existing areas of reasoning about correctness of quantum systems. For example, correctness of quantum key-distribution protocols is expressed in terms of asymptotic limits on the amount of mutual information between an attacker and a generated key. At the moment it is not clear how to generalize our approach to include this kind of specification.
- It would be interesting to analyze other stabilizer-based systems such as Aaronson's stabilizer money [1]. This might require probabilistic analysis.

Acknowledgements

The first author is supported in part by a grant from the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Iran. The third author is partially supported by EU ICT COST Action IC1405 “Reversible Computation—Extending Horizons of Computing".

REFERENCES

[1] S. Aaronson. Quantum copy-protection and quantum money. In *Proceedings of the 24th Annual IEEE Conference on Computational Complexity (CCC)*, pages 229–242. IEEE Computer Society, 2009.

[2] S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *Phys. Rev. A*, 70:052328, 2004.

[3] S. Abramsky and B. Coecke. A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 415–425. IEEE Computer Society, 2004.

[4] S. Anders and H. J. Briegel. Fast simulation of stabilizer circuits using a graph-state representation. *Phys. Rev. A*, 73:022334, 2006.

[5] E. Ardeshir-Larijani. Quantum Equivalence Checker (QEC). http://www.dcs.gla.ac.uk/~simon/qec, 2013.

[6] E. Ardeshir-Larijani, S. J. Gay, and R. Nagarajan. Equivalence checking of quantum protocols. In N. Piterman and S. A. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *Lecture Notes in Computer Science*, pages 478–492. Springer, 2013.

[7] E. Ardeshir-Larijani, S. J. Gay, and R. Nagarajan. Verification of concurrent quantum protocols by equivalence checking. In E. Ábrahám and K. Havelund, editors, *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 500–514. Springer, 2014.

[8] E. Ardeshir-Larijani. *Automated Equivalence Checking of Quantum Information Systems.* PhD thesis, University of Warwick, 2014.

[9] K. M. R. Audenaert and M. B. Plenio. Entanglement on mixed stabilizer states: normal forms and reduction procedures. *New Journal of Physics*, 7(1):170, 2005.

[10] P. Baltazar, R. Chadha, and P. Mateus. Quantum computation tree logic: model checking and complete calculus. *International Journal of Quantum Information*, 6(2):219–236, 2008.

[11] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Lett.*, 70:1895–1899, 1993.

[12] C. H. Bennett and S. J. Wiesner. Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Phys. Rev. Lett.*, 69:2881–2884, 1992.

[13] B. Coecke and R. Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, 2011.

[14] T. A. S. Davidson. *Formal Verification Techniques Using Quantum Process Calculus.* PhD thesis, University of Warwick, 2011.

[15] L. Dixon and R. Duncan. Graphical reasoning in compact closed categories for quantum computation. *Annals of Mathematics and Artificial Intelligence*, 56(1):23–42, 2009.

[16] Y. Feng, Y. Deng, and M. Ying. Symbolic bisimulation for quantum processes. *ACM Transactions on Computational Logic*, 15(2):14:1–14:32, 2014.

[17] Y. Feng, R. Duan, and M. Ying. Bisimulation for quantum processes. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 523–534. ACM, 2011.

[18] Y. Feng, N. Yu, and M. Ying. Model checking quantum Markov chains. *Journal of Computer and System Sciences*, 79(7):1181–1198, 2013.

[19] Y. Feng, E. M. Hahn, A. Turrini, and L. Zhang. QPMC: A model checker for quantum programs and protocols. In N. Bjørner and F. de Boer, editors, *Proceedings of the 20th International Symposium on Formal Methods (FM)*, volume 9109 of *Lecture Notes in Computer Science*, pages 265–272. Springer, 2015.

[20] S. J. Gay. Quantum programming languages: survey and bibliography. *Mathematical Structures in Computer Science*, 16(4):581–600, 2006.

[21] S. J. Gay. Stabilizer states as a basis for density matrices. *arXiv:1112.2156*, 2011.

[22] S. J. Gay and R. Nagarajan. Communicating Quantum Processes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages (POPL)*, pages 145–157. ACM, 2005.

[23] S. J. Gay, R. Nagarajan, and N. Papanikolaou. QMC: A model checker for quantum systems. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 543–547. Springer, 2008.

[24] S. J. Gay, R. Nagarajan, and N. Papanikolaou. Specification and verification of quantum protocols. In S. J. Gay and I. C. Mackie, editors, *Semantic Techniques in Quantum Computation*. Cambridge University Press, 2010.

[25] D. Gottesman. The Heisenberg representation of quantum computers. Technical Report LA-UR-98-2848, Los Alamos National Laboratory, 1998.

[26] A. S. Green, P. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. An introduction to quantum programming in Quipper. In G. W. Dueck and D. M. Miller, editors, *Proceedings of the 5th International Conference on Reversible Computation (RC)*, volume 7948 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2013.

[27] E. M. Hahn, Y. Li, S. Schewe, A. Turrini, and L. Zhang. IscasMC: A web-based probabilistic model checker. In J. S. Cliff Jones, Pekka Pihlajasaari, editor, *Proceedings of the 19th International Symposium on Formal Methods (FM)*, volume 8442 of *Lecture Notes in Computer Science*, pages 312–317. Springer, 2014.

[28] D. Harel, O. Kupferman, and M. Y. Vardi. On the complexity of verifying concurrent transition systems. *Information and Computation*, 173(2):143–161, 2002.

[29] M. Hillery, V. Bužek, and A. Berthiaume. Quantum secret sharing. *Phys. Rev. A*, 59:1829–1834, 1999.

[30] A. Kissinger. *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing*. PhD thesis, University of Oxford, 2011. arXiv:1203.0202.

[31] E. Knill, R. Laflamme, R. Martinez, and C. Negrevergne. Benchmarking quantum computers: The five-qubit error correcting code. *Phys. Rev. Lett.*, 86:5811–5814, June 2001.

[32] T. Kubota, Y. Kakutani, G. Kato, Y. Kawano, and H. Sakurada. Semi-automated verification of security proofs of quantum cryptographic protocols. *Journal of Symbolic Computation*, 73:192–220, 2016.

[33] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. *International Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[34] R. Laflamme, C. Miquel, J. P. Paz, and W. H. Zurek. Perfect quantum error correcting code. *Phys. Rev. Lett.*, 77:198–201, July 1996.

[35] D. Markham and B. C. Sanders. Graph states for quantum secret sharing. *Phys. Rev. A*, 78:042309, 2008.

[36] Microsoft. Language-Integrated Quantum Operations. https://www.microsoft.com/en-us/research/project/language-integrated-quantum-operations-liqui, 2013.

[37] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[38] R. Nagarajan and S. J. Gay. Formal verification of quantum protocols. *arXiv:quant-ph/0203086*, 2002.

[39] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[40] N. Papanikolaou. *Model Checking Quantum Protocols*. PhD thesis, University of Warwick, 2009.

[41] P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[42] G. F. Viamontes, I. L. Markov, and J. P. Hayes. *Quantum Circuit Simulation*. Springer, 2009.

[43] M. Ying, Y. Feng, R. Duan, and Z. Ji. An algebra of quantum processes. *ACM Transactions on Computational Logic*, 10(3):19:1–19:36, 2009.

[44] M. Ying. *Foundations of Quantum Programming*. Morgan Kaufmann, 2016.

[45] X. Zhou, D. W. Leung, and I. L. Chuang. Methodology for quantum logic gate construction. *Phys. Rev. A*, 62:052316, 2000.

Received 2017