



Exploring a Quantum Programming Language with Concurrency

Manisha Jain ✉ 

CIDMA – Research Center in Mathematics and Applications, Aveiro, Portugal
Mathematics Department of University of Aveiro, Portugal
International Iberian Nanotechnology Laboratory, Braga, Portugal

Vitor Fernandes ✉ 

HasLab INESC TEC, University of Minho, Braga, Portugal

Alexandre Madeira ✉ 

CIDMA – Research Center in Mathematics and Applications, Aveiro, Portugal
Mathematics Department of University of Aveiro, Portugal

Luís S. Barbosa ✉ 

HasLab INESC TEC, University of Minho, Braga, Portugal
UNU-EGOV, United Nations University, Tokyo, Japan

Abstract

In quantum programming, as in the classical case, concurrent control is a form of program coordination that proves well suited to express complex composition patterns. This paper introduces a quantum programming language with explicit parallel and synchronization primitives and its semantics. The language is explored through a MAUDE implementation, and illustrated with two non trivial examples.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases Quantum programming, semantics prototyping, Maude

Digital Object Identifier 10.4230/OASICS.Programming.2025.16

Supplementary Material *Software (Maude implementation)*: <https://github.com/jmanishajain/CQDL> [11], archived at `swb:1:dir:21be0d264bc1d4ab0a9c3a40e5237a4bb2f1b9ce`

Funding This work is financed by National Funds through FCT – Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference 10.54499/PTDC/CCI-COM/4280/2021, and by CIDMA with UIDP/04106/2025 and UIDB/04106/2025.

1 Introduction

This paper adds concurrent control to a standard quantum programming language. A form of parallel composition and a synchronization mechanism are introduced to support a more expressive and detailed form of coordination of quantum computations. The language LQC and its semantics is prototyped in MAUDE [4] a well-known implementation of rewriting logic. This provides a flexible tool to execute the proposed semantics and observe the language in action through a number of examples.

The use of parallelism has proved fruitful to speed-up the execution of some algorithms (*e.g.*: leader election problem [15], dining philosophers [1], quantum Fourier transform [5]). Furthermore, the development of proof-of-concept implementations of quantum computers, entails corresponding developments in the design of suitable operating systems [6].



© Manisha Jain, Vitor Fernandes, Alexandre Madeira, and Luís S. Barbosa;
licensed under Creative Commons License CC-BY 4.0

Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025).

Editors: Jonathan Edwards, Roly Perera, and Tomas Petricek; Article No. 16; pp. 16:1–16:9



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The use of MAUDE to animate quantum circuits and their composition has been discussed elsewhere [14, 9]. Our own contribution is the introduction of concurrency as a programming construct of its own. A logic counterpart of this work was developed by the first authors in [12], where a corresponding dynamic logic to reason about this sort of programs was introduced.

The paper is organized as follows: Section 2 introduces the syntax and semantics of the proposed language. Then, in section 3, its MAUDE implementation [11] (where file *main.maude* was retrieved from [8]) is illustrated through two examples encoding quantum teleportation and the Grover's unstructured search algorithm. Finally, section 4 concludes and points out some future work.

2 A Concurrent Quantum Programming Language and Its Semantics

The programming language presented below, LQC, is inspired by the work of Mingsheng Ying [16, Section 5]. The syntax of some commands was modified and we introduced a new **await** construct instead of the atomic region (which is indeed an unconditional **await**). LQC has an imperative flavour.

Its programs act over a shared-variable memory consisting of a finite number N of qubits interpreted, as usual, over a Hilbert space $\mathcal{H} = \mathbb{C}^{2^N}$, where \mathbb{C} denotes the field of complex numbers. A set $\mathcal{U}(\mathcal{H})$ of unitary gates over \mathcal{H} is assumed. The syntax is given by the following grammar

$$\begin{aligned} C &::= \text{skip} \mid U(\vec{n}) \mid M(n, C_1, C_2) \mid C_1 ; C_2 \mid C_1 + C_2 \mid C_1 \parallel C_2 \mid \text{await}(n, D) \\ D &::= \text{skip} \mid U(\vec{n}) \mid M(n, D_1, D_2) \mid D_1 ; D_2 \end{aligned}$$

where $U(\vec{n})$ stands for the application of unitary U to the list of qubits \vec{n} ; $M(n, C_1, C_2)$ is a conditional on the measurement of qubit n in the computational basis, evolving to C_1 if the result is 0, to C_2 otherwise; and $C_1 ; C_2$ and $C_1 + C_2$ denote sequential composition and non deterministic choice, respectively.

The new, distinguished operators are parallel composition $C_1 \parallel C_2$ and **await**(n, D). The latter triggers the execution of a (restricted) sequence of commands D depending on the value of a qubit n , referred to as the *await qubit*. In a classical setting [3], the **await** command can be executed infinitely often, but in quantum computing, measurement is destructive. To avoid state collapse, each **await** command has an auxiliary qubit to serve as a control, which is never allowed into a superposition state. Finally, note that the measurement command can be written as $\Phi_0(n)? ; C_1 + \Phi_1(n)? ; C_2$. Hence, from $\langle M(q, C_1, C_2), v \rangle$ one can transit to $\langle C_1, v_0 \rangle$, where v_0 is the result of applying projector $\Phi_0(n)?$ to v , or to $\langle C_2, v_1 \rangle$, where v_1 comes from a similar application to v of projector $\Phi_1(n)?$.

The language's small-step operational semantics is given in Fig 1. Based on configurations, *i.e.* command/continuation state pairs, the rules are self-explicative. Note, however, the introduction of auxiliary qubits to support the **await** command, resulting in an extended state $v = u \otimes u_{\text{await}} = |1 \dots n\rangle \otimes |n+1 \dots m\rangle$, where $u = |1 \dots n\rangle$ and $u_{\text{await}} = |n+1 \dots m\rangle$. The two rules for this construct specify when the sequence of commands guarded within **await** is executed, basically depending on the value of the corresponding **await** qubit.

Figure 2 presents the corresponding big-step semantics specifying for a configuration $\langle C, v \rangle$ a possible final state (rather than *the* final state, as the language is non-deterministic).

$$\begin{array}{c}
\frac{}{\langle \text{skip}, v \rangle \longrightarrow v} \text{ (sk)} \quad \frac{}{\langle U(n), v \rangle \longrightarrow U_n v} \text{ (un)} \\
\\
\frac{}{\langle M(n, C_1, C_2), v \rangle \longrightarrow \langle C_1, v_0 \rangle} \text{ (meas}_0\text{)} \quad \frac{}{\langle M(n, C_1, C_2), v \rangle \longrightarrow \langle C_2, v_1 \rangle} \text{ (meas}_1\text{)} \\
\\
\frac{\langle C_1, v \rangle \longrightarrow v'}{\langle C_1; C_2, v \rangle \longrightarrow \langle C_2, v' \rangle} \text{ (seq1)} \quad \frac{\langle C_1, v \rangle \longrightarrow \langle C'_1, v' \rangle}{\langle C_1; C_2, v \rangle \longrightarrow \langle C'_1; C_2, v' \rangle} \text{ (seq2)} \\
\\
\frac{\langle C_1, v \rangle \longrightarrow v'}{\langle C_1 \parallel C_2, v \rangle \longrightarrow \langle C_2, v' \rangle} \text{ (parL1)} \quad \frac{\langle C_1, v \rangle \longrightarrow \langle C'_1, v' \rangle}{\langle C_1 \parallel C_2, v \rangle \longrightarrow \langle C'_1 \parallel C_2, v' \rangle} \text{ (parL2)} \\
\\
\frac{\langle C_2, v \rangle \longrightarrow v'}{\langle C_1 \parallel C_2, v \rangle \longrightarrow \langle C_1, v' \rangle} \text{ (parR1)} \quad \frac{\langle C_2, v \rangle \longrightarrow \langle C'_2, v' \rangle}{\langle C_1 \parallel C_2, v \rangle \longrightarrow \langle C_1 \parallel C'_2, v' \rangle} \text{ (parR2)} \\
\\
\frac{\langle C_1, v \rangle \longrightarrow v'}{\langle C_1 + C_2, v \rangle \longrightarrow v'} \text{ (ndL1)} \quad \frac{\langle C_1, v \rangle \longrightarrow \langle C'_1, v' \rangle}{\langle C_1 + C_2, v \rangle \longrightarrow \langle C'_1, v' \rangle} \text{ (ndL2)} \\
\\
\frac{\langle C_2, v \rangle \longrightarrow v'}{\langle C_1 + C_2, v \rangle \longrightarrow v'} \text{ (ndR1)} \quad \frac{\langle C_2, v \rangle \longrightarrow \langle C'_2, v' \rangle}{\langle C_1 + C_2, v \rangle \longrightarrow \langle C'_2, v' \rangle} \text{ (ndR2)} \\
\\
\frac{}{\langle \text{await}(n, D), u \otimes |n_{m+1} \dots 0 \dots n_{m'}\rangle \longrightarrow \langle \text{await}(n, D), u \otimes |n_{m+1} \dots 0 \dots n_{m'}\rangle} \text{ (aw1)} \\
\\
\frac{\langle D, u \otimes |n_{m+1} \dots 1 \dots n_{m'}\rangle \rightarrow u' \otimes |n'_{m+1} \dots n'_n \dots n'_{m'}\rangle}{\langle \text{await}(n, D), u \otimes |n_{m+1} \dots 1 \dots n_{m'}\rangle \longrightarrow u' \otimes |n'_{m+1} \dots 0 \dots n'_{m'}\rangle} \text{ (aw2)}
\end{array}$$

■ **Figure 1** LQC: small-step operational semantics.

$$\frac{\langle C, v \rangle \longrightarrow v'}{\langle C, v \rangle \rightarrow v'} \text{ (trans1)} \quad \frac{\langle C_1, v \rangle \longrightarrow \langle C_2, v' \rangle \quad \langle C_2, v' \rangle \rightarrow v''}{\langle C_1, v \rangle \rightarrow v''} \text{ (trans2)}$$

■ **Figure 2** LQC: big-step operational semantics.

3 Exploring LQC Through a Maude Implementation

The implementation of LQC in MAUDE follows a declarative discipline. Sorts, subsorts and operators are declared as usual. Functions `smallStep` and `bigStep` encode LQC small-step and big-step semantics, respectively, acting over a sequence of configurations, an implementation detail which guarantees that non-deterministic behaviors are effectively captured and intermediate steps of computations are accessible. This approach simplifies the implementation, avoids introducing additional sorts, and allows for the systematic evaluation of all potential outcomes in quantum computations. As an example, consider the small step semantics of the atomic program corresponding to a *CNOT* gate.

`eq smallStep(< CX(N1,N2), Q >) = (Q).CX(N1, N2) .`

where `(Q).CX(N1, N2)` applies the gate over qubits N1 and N2 of state Q.

► **Example 1.** To illustrate the use of concurrency and the `await` command, consider the execution of the following program: $\langle (X(2); \text{await}(2, M(1, \text{skip}, \text{skip}))) \parallel H(1), |00\rangle \rangle$ where

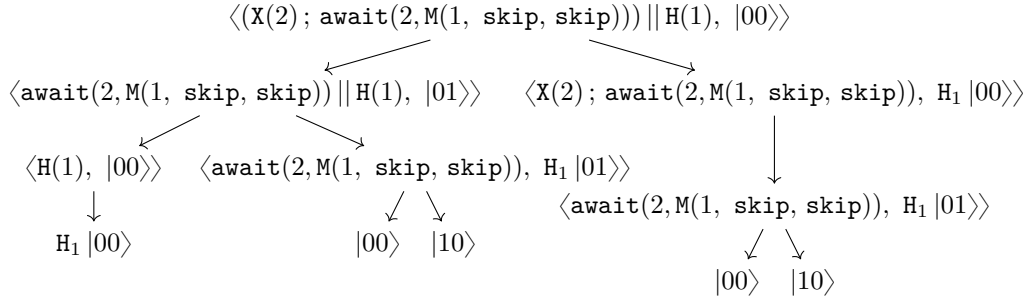
```
red in AWAIT : rmvRep(bigStep(< P1 || P2, inQSt >)).
result ListQConf:
(q[1 2]: (1 ./ Sqrt(2)) . |0>(x)|0> +
         (1 ./ Sqrt(2)) . |1>(x)|0>),
(q[1 2]: (1 ./ Sqrt(2)) . |0>(x)|0>),
(q[1 2]: (1 ./ Sqrt(2)) . |1>(x)|0>)
```

where P1 and P2 are the obvious abbreviations and $\text{inQSt} = q[1\ 2] : |0\rangle \otimes |0\rangle$.

As expected, the result corresponds to the execution tree depicted in Figure 3.

Notice the use of function `rmvRep`. This is because evaluating a program with the big-step semantics implemented in MAUDE returns a list with all the possible final states. In order to reduce the final states to the strictly necessary ones, repeated states are removed.

As expected, the results obtained coincide with those in Figure 3.

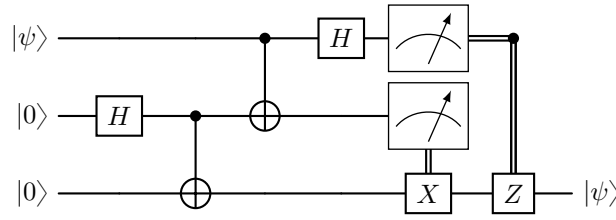


■ **Figure 3** Execution tree of $\langle (X(2); \text{await}(2, M(1, \text{skip}, \text{skip}))) \parallel H(1), |00\rangle \rangle$.

To further explore the potential of the language, two non trivial, but well-known examples are considered in the sequel: the teleportation protocol, which makes further use of the `await` command, and the Grover's algorithm for unstructured search.

► **Example 2 (Quantum Teleportation).** The quantum teleportation protocol [2] makes possible the transmission of a quantum state without directly resorting to quantum communication channels: only classical binary information is transmitted. The whole procedure, which relies on a very specific quantum resource, entanglement, is easy to explain. The scenario, depicted in Figure 4, unfolds as follows: Alice holds a quantum state $|\psi\rangle = a|0\rangle + b|1\rangle$ to be transferred to Bob, with whom she shares an entangled state. Alice begins by entangling the state she wants to transfer with her own component of the shared entangled state. Then, she measures both qubits and sends the results of such measurements to Bob. Depending on them, Bob performs one of the following operations.

- If the measurement result is 00, Bob does nothing to his qubit.
- If the result is 01, he applies a Z gate to his qubit.
- If the result is 10, he applies an X gate to his qubit.
- If the result is 11, he applies the X gate and then the Z gate to his qubit.



■ **Figure 4** Quantum Teleportation.

This protocol is implemented in LQC as described below. However, a third participant, Charlie, encodes the entanglement procedure of Alice's & Bob's shared qubits.

```

Charlie = H(2); CNOT[2,3]
Alice = CNOT[1,2]; H(1)
Bob = M(2, skip, X(3)); M(1, skip, Z(3))
qTele = Charlie; Alice; Bob
inQSt = (q[1] : a. |0> + b. |1>)(q[2] : |0>)(q[3] : |0>)

```

where $a, b \in \mathbb{C}$ s.t. $|a|^2 + |b|^2 = 1$, and $(q[1] : a. |0\rangle + b. |1\rangle)$ represents the state to teleport. Note that, although the description of the protocol states that Bob receives the bits measured by Alice, in the MAUDE encoding, the measurement is indeed done by Bob. This does not lead to any loss of generality as communication occurs by accessing shared variables in the state. As before, the protocol is checked by evaluation, confirming the expected results.

```

red in TELEPORT : rmvRep(bigStep(< qTele, inQSt >)) .
result ListQConf:
(q[1 2 3]: (a .* 1/2) . |0>(x)|1>(x)|0> + (b .* 1/2) . |0>(x)|1>(x)|1>),
(q[1 2 3]: (a .* 1/2) . |1>(x)|1>(x)|0> + (b .* 1/2) . |1>(x)|1>(x)|1>),
(q[1 2 3]: (a .* 1/2) . |0>(x)|0>(x)|0> + (b .* 1/2) . |0>(x)|0>(x)|1>),
(q[1 2 3]: (a .* 1/2) . |1>(x)|0>(x)|0> + (b .* 1/2) . |1>(x)|0>(x)|1>)

```

Let us now explore concurrency in the context of this protocol. The basic observation, which explains why concurrent control is not trivial here, concerns the fact that Bob's actions depend on Alice's measurement outcomes. Actually, the critical case occurs when Alice's measurement yields 11. Thus, Bob must apply the X gate, followed by the Z gate. This order must be preserved, which entails the need for some mechanism to inform that the Z gate was performed. This is achieved through an auxiliary qubit, the qubit 4, which is switched to $|1\rangle$ after the application of gate Z. Also note how the `await` command is used in order to allow Bob to effectively synchronize his actions. The updated specification for Bob's operations is as follows:

```

BobX = await(6, M(4, M(2, skip, X(3)), M(2, skip, X(3); Z(4))))
BobZ = await(5, M(1, skip, Z(3); X(4)))
BobP = BobZ || BobX

```

What is the intuition behind `BobX` and `BobZ` (to simplify, assume that `await` qubits 5 and 6 are already set to $|1\rangle$)? As `BobX` and `BobZ` are in parallel, one of them is executed first. In the former case, the instructions inside the `await` command are accessed to check whether the Z gate was already executed, or not, by measuring qubit 4. If the response is negative, then the state of qubit 4 is $|0\rangle$, which means that the quantum teleportation protocol is

16:6 Exploring a Quantum Programming Language with Concurrency

behaving as expected. If not, the state of qubit 4 is $|1\rangle$. This entails the need to cancel out the phase introduced by first executing the Z gate, which can be done by applying a Z gate to qubit 4, which is $|1\rangle$, as we already know. This representation ensures proper synchronization of Bob's operations, regardless of the sequence in which **BobX** and **BobZ** are executed. The complete quantum teleportation protocol with concurrency, together with the update initial state, is defined as

```
qTeleP = Charlie; Alice; BobP
inQStP = (q[1] : a. |0> + b. |1>)(q[2 3 4 5 6] : |0> ⊗ |0> ⊗ |0> ⊗ |1> ⊗ |1>)
```

The protocol outputs are checked as before, confirming the parallel implementation also works as expected:

```
red in TELEPORT : rmvRep(bigStep(< qTeleP, inQStP >)) .
result ListQConf:
(q[1 2 3 4 5 6] : (a . * 1/2) . |0>(x) |1>(x) |0>(x) |0>(x) |0>(x) |0> +
                  (b . * 1/2) . |0>(x) |1>(x) |1>(x) |0>(x) |0>(x) |0>),
(q[1 2 3 4 5 6] : (a . * 1/2) . |1>(x) |1>(x) |0>(x) |1>(x) |0>(x) |0> +
                  (b . * 1/2) . |1>(x) |1>(x) |1>(x) |1>(x) |0>(x) |0>),
(q[1 2 3 4 5 6] : (a . * 1/2) . |0>(x) |0>(x) |0>(x) |0>(x) |0>(x) |0> +
                  (b . * 1/2) . |0>(x) |0>(x) |1>(x) |0>(x) |0>(x) |0>),
(q[1 2 3 4 5 6] : (a . * 1/2) . |1>(x) |0>(x) |0>(x) |1>(x) |0>(x) |0> +
                  (b . * 1/2) . |1>(x) |0>(x) |1>(x) |1>(x) |0>(x) |0>)
```

► **Example 3 (Grover's Algorithm).** Grover's algorithm [10] is one of the most well known quantum algorithms, with quadratic advantage, whose purpose is to perform unstructured search on an unsorted data space. It is divided into three parts, as depicted in Figure 5. the first one is the initialization component, in which the qubits holding a state of N elements are put in superposition, through an Hadamard gate applied to each of them, and the ancilla qubits prepared. The oracle component flips the phase of the state one is looking for. Finally, the diffusion component aims at increasing the amplitude of the targeted state. For a search space with N elements, the oracle and the diffusion operator are applied \sqrt{N} times.

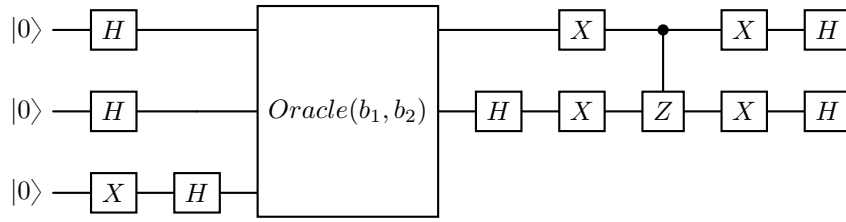
For illustrative purposes, consider a search space with four elements (00, 01, 10, and 11), which can be encoded into two qubits. An extra qubit, an ancilla qubit is introduced. Thus, a sequential implmentation of the algorithm emerges as follows:

```
init = H(1); H(2)
anc = X(3); H(3)
oracleij = X(1-i)(1); X(1-j)(2); CCX[1, 2, 3]; X(1-i)(1); X(1-j)(2)
diff = H(1); X(1); H(2); X(2); CZ[1, 2]; X(1); H(1); X(2); H(2)
groverij = init; anc; oracleij; diff
inQSt = q[1 2 3] : |0> ⊗ |0> ⊗ |0>
```

where $i, j \in \{0, 1\}$.

Note that for each possible solution a different oracle is defined. For instance, to find element 11 the oracle is simply $\text{CCX}[1, 2, 3]$, where CCX is the controlled CNOT gate. Having this into consideration, let us check the execution of the algorithm for each element in the search space

```
red rmvRep(bigStep(< grover00, inQSt >)) .
result EnQubit: q[1 2 3] : (-1 . * 1 ./ Sqrt(2)) . |0>(x) |0>(x) |0> +
                  (1 . / Sqrt(2)) . |0>(x) |0>(x) |1>
```



■ **Figure 5** Grover's algorithm.

```
red rmvRep(bigStep(< grover01, inQSt >)) .
result EnQubit: q[1 2 3]: (-1 .* 1 ./ Sqrt(2)) . |0>(x)|1>(x)|0> +
(1 ./ Sqrt(2)) . |0>(x)|1>(x)|1>

red rmvRep(bigStep(< grover10, inQSt >)) .
result EnQubit: q[1 2 3]: (-1 .* 1 ./ Sqrt(2)) . |1>(x)|0>(x)|0> +
(1 ./ Sqrt(2)) . |1>(x)|0>(x)|1>

red rmvRep(bigStep(< grover11, inQSt >)) .
result EnQubit: q[1 2 3]: (-1 .* 1 ./ Sqrt(2)) . |1>(x)|1>(x)|0> +
(1 ./ Sqrt(2)) . |1>(x)|1>(x)|1>
```

Again, the results are the expected ones.

To explore concurrency in this example, one needs to analyze which actions can be put in parallel. In the initialization component, the application of Hadamard gates is made on disjoint qubits. Hence, it can be put in parallel. Furthermore, this can also be put in parallel with the preparation of the ancilla qubit. In what concerns the oracle, the only case where disjoint actions are found is in searching for the element 00. Finally, a similar analysis leads to some parallelization of the diffusion component. Therefore, we end up with the following implementation:

```
initP = H(1) || H(2)
oracle00P = (X(1) || X(2)); CCX[1, 2, 3]; (X(1) || X(2))
diffP = ((H(1); X(1)) || (H(2); X(2))) ; CZ[1, 2] ; ((X(1); H(1)) || (X(2); H(2)))
groverijP = (initP || anc); oracleij; diffP
```

with $i, j \in \{0, 1\}$. As expected, both sequential and parallel implementations of the Grover's algorithm in LQC lead to the same results:

```
red rmvRep(bigStep(< grover00P, inQSt >)) .
result EnQubit: q[1 2 3]: (-1 .* 1 ./ Sqrt(2)) . |0>(x)|0>(x)|0> +
(1 ./ Sqrt(2)) . |0>(x)|0>(x)|1>

red rmvRep(bigStep(< grover01P, inQSt >)) .
result EnQubit: q[1 2 3]: (-1 .* 1 ./ Sqrt(2)) . |0>(x)|1>(x)|0> +
(1 ./ Sqrt(2)) . |0>(x)|1>(x)|1>

red rmvRep(bigStep(< grover10P, inQSt >)) .
result EnQubit: q[1 2 3]: (-1 .* 1 ./ Sqrt(2)) . |1>(x)|0>(x)|0> +
(1 ./ Sqrt(2)) . |1>(x)|0>(x)|1>

red rmvRep(bigStep(< grover11P, inQSt >)) .
result EnQubit: q[1 2 3]: (-1 .* 1 ./ Sqrt(2)) . |1>(x)|1>(x)|0> +
(1 ./ Sqrt(2)) . |1>(x)|1>(x)|1>
```

4 Conclusions and Future Work

The paper introduced LQC, a quantum programming language with concurrency, and the implementation of its operational semantics in MAUDE. Reference [13] pursues a similar objective but for a different language without concurrent control.

The language can be revised in a number of ways, namely to introduce some form of recursion, or achieving finer control of the so-called await qubits. Actually, it turns out that flipping the await qubit n to $|0\rangle$ is not mandatory, since there is no risk of another program accessing the associated await command.

Reference [7] introduces a dedicated processor for LQC. Current work within the doctoral project of the second author targets the study of different sorts of denotational semantics for the language. Specification and verification of properties of LQC, in the context of a dedicated dynamic logic, are also being studied, with previous results presented in reference [12].

References

- 1 Dorit Aharonov, Maor Ganz, and Loick Magnin. Dining philosophers, leader election and ring size problems, in the quantum setting, 2017. [arXiv:1707.01187](#).
- 2 Charles H Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Physical review letters*, 70(13):1895, 1993.
- 3 Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996. doi:10.1006/INCO.1996.0056.
- 4 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. doi:10.1016/S0304-3975(01)00359-0.
- 5 Richard Cleve and John Watrous. Fast parallel circuits for the quantum fourier transform, 2000. doi:10.1109/SFCS.2000.892140.
- 6 Henry Corrigan-Gibbs, David J. Wu, and Dan Boneh. Quantum operating systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 76–81, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3102980.3102993.
- 7 Maria Inês Machado Correia Brioso Dias. An interpreter for a concurrent quantum language. Master's thesis, University of Minho, 2024.
- 8 Canh Minh Do. Automated Quantum Protocol Verification Based on Dynamic Quantum Logic. URL: <https://github.com/canhminhdo/dql>.
- 9 Canh Minh Do and Kazuhiro Ogata. Symbolic model checking quantum circuits in maude. *PeerJ Computer Science*, 10:e2098, 2024. doi:10.7717/PEERJ-CS.2098.
- 10 Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996. doi:10.1145/237814.237866.
- 11 Manisha Jain and Vitor Fernandes. Maude Implementation for Concurrent Quantum Programming Language. Software, swhId: `swh:1:dir:21be0d264bc1d4ab0a9c3a40e5237a4bb2f1b9ce` (visited on 2025-07-16). URL: <https://github.com/jmanishajain/CQDL>, doi:10.4230/artifacts.23607.
- 12 Manisha Jain, Vitor Fernandes, and Alexandre Madeira. Adding concurrency to a quantum dynamic logic. In Songmao Zhang and Luis Soares Barbosa, editors, *Artificial Intelligence Logic and Applications*, pages 17–31, Singapore, 2025. Springer Nature Singapore.
- 13 Canh Minh Do and Kazuhiro Ogata. An executable operational semantics of quantum programs and its application. In *International Symposium on Software Fault Prevention, Verification, and Validation*, pages 15–31. Springer, 2024. doi:10.1007/978-981-96-1621-3_2.

- 14 Tsubasa Takagi, Canh Minh Do, and Kazuhiro Ogata. Automated quantum program verification in a dynamic quantum logic. In Nina Gierasimczuk and Fernando R. Velázquez-Quesada, editors, *Proc. DaLi'2023: Dynamic Logic. New Trends and Applications*, pages 68–84. Springer Lecture Notes in Computer Science, 14401, 2024.
- 15 Seiichiro Tani, Hirotada Kobayashi, and Keiji Matsumoto. Exact quantum algorithms for the leader election problem. *ACM Trans. Comput. Theory*, 4(1), March 2012. doi:10.1145/2141938.2141939.
- 16 Mingsheng Ying, Li Zhou, and Yangjia Li. Reasoning about parallel quantum programs. *arXiv preprint arXiv:1810.11334*, 2018. doi:10.1145/1122445.1122456.