Entangλe: A Translation Framework from Quipper Programs to Quantum Markov Chains [*]

Linda Anticoli[1], Carla Piazza[1], Leonardo Taglialegne[1], and Paolo Zuliani[2]

[1] Dept. of Mathematics, Computer Science and Physics, University of Udine, Italy
anticoli.linda@spes.uniud.it,carla.piazza@uniud.it
[2] School of Computing, Newcastle University, United Kingdom
paolo.zuliani@ncl.ac.uk

Abstract. Entangλe is a framework for translating the quantum programming language Quipper to the QPMC model checker. It has been developed in order to formally verify Quipper-like programs. Quipper is a functional circuit description language, allowing an high-level approach for manipulating quantum circuits. Quipper uses the vector state formalism and provides high-level operations. QPMC is a model checker designed for quantum protocols specified as Quantum Markov Chains, and it is based on the density matrix formalism; QPMC supports the temporal logic QCTL. We have developed Entangλe to deal with the notion of tail recursive quantum programs in Quipper, and so we are able to verify QCTL properties over such programs. The tool implementation has been tested on several quantum protocols, including the BB84 protocol for quantum key distribution.

Keywords: Quantum Languages, Quantum Circuits, Model Checking

1 Introduction

Entangλe is a framework allowing to define – by using a sublanguage of the quantum programming language Quipper, called *Quip-E* – and automatically verify – using the quantum model checker QPMC – formal properties of quantum algorithms and protocols by abstracting away from low-level features. A preliminary version of Entangλe has been presented in [1], and the new, extended version is freely available[3].

Quipper [16] is a functional quantum programming language based on Haskell that allows to build and simulate quantum circuits and programs by describing them in a simple programming style. QPMC [6] is a PRISM-inspired model checker that uses the quantum temporal logic QCTL to verify properties of quantum protocols.

Currently, Quipper lacks a built-in formal verification tool, while QPMC supports formal verification but it is based on a low-level specification language.

[*] This work has been partially supported by INdAM GNCS, and by the PRID ENCASE UniUD project.

[3] https://github.com/miniBill/entangle

`Entang`λ`e` translates Quipper-like programs (written in *Quip-E*) into QPMC structures (i.e., Quantum Markov Chains). We used `Entang`λ`e` to translate several protocols and quantum algorithms, including Quantum Key Distribution protocols (herein QKD) in both their recursive and non-recursive version, and entanglement-based protocols. `Entang`λ`e` can be used to verify classical properties, i.e., measurement outcomes or probability distributions over them, but also for verifying whether quantum effects such as correlations and entanglement are preserved throughout a computation. The extensions of `Entang`λ`e`, with respect to the previous version, are:

- a new initialization operator (due to the need to devise a translation suitable for QPMC);
- support for tail recursion in quantum programs;
- an easy-to-use graphical user interface.

The paper is organized as follows: Section 2 introduces the basic concepts and notation regarding the quantum mechanics formalisms used in this paper, plus a short description of the Quipper and QPMC languages. Section 3 shows the tool and describes the main implementation choices made throughout its development. Finally, Section 4 presents two examples of quantum algorithms translated and tested: Grover's algorithm for quantum search, and our tail-recursive version of the QKD protocol BB84. (For further references about the non-recursive version see [14, 5].) Section 5 concludes the paper.

2 Setting the Context

2.1 Quantum Formalisms

Quantum systems are represented in a complex Hilbert space $\mathcal{H}$ i.e., a complete vector space equipped with an inner product. The elements of $\mathcal{H}$ (vectors) are denoted by $|\psi\rangle$ (i.e., ket notation). The notation $\langle\psi|$ (i.e., bra notation) denotes the transposed conjugate of $|\psi\rangle$. The scalar product of two vectors φ and ψ in $\mathcal{H}$ is denoted by $\langle\varphi|\psi\rangle$, whereas $|\varphi\rangle\langle\psi|$ denotes the linear operator defined by $|\varphi\rangle$ and $\langle\psi|$. We use I to denote the identity matrix and $tr(\cdot)$ for the matrix trace.

There are two possible formalisms based on Hilbert spaces for quantum systems: the *state vector* formalism and the *density matrix* one.

State Vectors The *state* of a quantum system is described by a *normalized vector* $|\psi\rangle \in \mathcal{H}$, i.e., $\||\psi\rangle\| = \sqrt{\langle\psi|\psi\rangle} = 1$. The normalization condition is related to the probabilistic interpretation of quantum mechanics.

The temporal *evolution* of a quantum system is described by a *unitary operator* (see, e.g., [14]). A linear operator U is unitary if and only if $U^\dagger = (U^T)^* = U^{-1}$. Unitary operators preserve inner products and, as a consequence, norms of vectors. In absence of any measurement process, the state $|\psi_0\rangle$ at time t_0 evolves at time t_1 through the unitary operator U to the state

$$|\psi_1\rangle = U\ |\psi_0\rangle.$$

If a measurement occurs, the state collapses to one of the eigenstates of the observable measured.

An *observable* is a property that can be measured, i.e., a physical quantity such as position, spin, etc. Observables are *Hermitian operators* (see, e.g., [15]) i.e., $A = A^\dagger$. If no degeneracy occurs, an Hermitian operator A can be decomposed as

$$A = \sum_{i=1}^{n} a_i |\varphi_i\rangle\langle\varphi_i|$$

where the a_i's ($|\varphi_i\rangle$'s) are the eigenvalues (eigenvectors, respectively) of A. The eigenvalues of a Hermitian operator are real numbers.

The outcome of measuring observable A, given a system in a state $|\psi\rangle$, is one of its eigenvalues a_i. The state vector of the system after the measurement is:

$$\frac{(|\varphi_i\rangle\langle\varphi_i|)|\psi\rangle}{||(|\varphi_i\rangle\langle\varphi_i|)|\psi\rangle||}$$

with probability

$$p(a_i) = ||(|\varphi_i\rangle\langle\varphi_i|)|\psi\rangle||^2 = \langle\psi|(|\varphi_i\rangle\langle\varphi_i|)|\psi\rangle.$$

Density Matrices The *state* of a quantum system is here described by an Hermitian, positive matrix ρ with $tr(\rho) = 1$. A matrix ρ is positive if for each vector $|\phi\rangle$ it holds that $\langle\phi|\rho|\phi\rangle \geq 0$. Such matrices are called *density* matrices.

Given a normalized vector $|\psi\rangle$ representing the state of a system through the state vector formalism, the corresponding density matrix is $|\psi\rangle\langle\psi|$.

Evolution and *measurement* of quantum systems are now described by *superoperators* [14]. A superoperator is a (linear) function $\mathcal{E} : \rho_0 \to \rho_1$ which maps density matrices to density matrices and satisfies the following properties: $\mathcal{E}$ preserves hermiticity; $\mathcal{E}$ is trace preserving; $\mathcal{E}$ is completely positive. Requiring complete positivity allows a linear map to be positivity preserving even if the system under consideration has previously been correlated with another, unknown, system. In this case, indeed, positivity alone does not guarantee a positive evolution of the density matrix.

Given a unitary operator U the corresponding superoperator $\mathcal{E}_U$ can be defined as follows:

$$\mathcal{E}_U(\rho) = U\rho U^\dagger.$$

A quantum *measurement* is described by a collection $\{M_i\}$ of linear operators, called measurement operators, satisfying the following condition:

$$\sum_i M_i^\dagger M_i = I$$

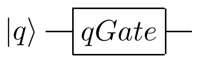
where the index i refers to the measurement outcomes that may occur.

2.2 Quipper

Quipper is an Haskell-like quantum circuit description language based on Knill's QRAM model [12] of quantum computation. The language is endowed with quantum and classical data types, combinators, and libraries of functions within Haskell, together with an idiom, i.e., a preferred style of writing quantum programs [10]. Quipper is based on the state vector formalism and its semantics is given in terms of quantum circuits [14], which involve qubits and unitary gates.

Quipper programs are functions in which qubits are held in variables and gates are applied to them one at a time. The Haskell monad `Circ`, which from an abstract point of view returns a quantum circuit, encapsulates this behavior. An example of a simple Quipper program with one qubit, where `qgate_at` refers to a generic quantum gate, and its corresponding circuit representation can be seen in the following:

```
qExample :: Qubit -> Circ Qubit
qExample q = do
  qgate_at q
  return q
```

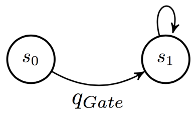


2.3 QPMC

QPMC is a model checker for quantum programs and protocols available in both web-based and off-line version[4], and it based on the density matrix formalism of quantum mechanics [14][15]. The semantics of a QPMC program is given in terms of a *quantum Markov chain* (herein QMC), a Markov chain in which the state space is taken classical, while all quantum effects are encoded in the superoperators labelling the transitions (see, e.g., [6, 7]).

QPMC takes in input programs written in an extension of PRISM probabilistic model checker's language [13] allowing, in addition, the specification of types `vector`, `matrix`, and `superoperator`. A translation of the QMC relative to the Quipper example introduced above, and its graphical representation, can be seen in the following:

```
qmc
const matrix A1 = QGATE;
module resetCirc
  s: [0..1] init 0;
  [] (s = 0) -> <<A1>> : (s' = 1);
  [] (s = 1) -> true;
endmodule
```



Properties of quantum protocols are expressed as formulae of the quantum computation tree logic (QCTL [7]), a temporal logic for reasoning about quantum systems, and defined as an extension of PCTL [11] in which classical probabilities are replaced by quantum probabilities represented by superoperators.

[4] Available at `http://iscasmc.ios.ac.cn/too/qmc`

A QCTL formula is a formula over the following grammar:

$$\Phi ::= a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathbb{Q}_{\sim\epsilon}[\Phi]$$

$$\phi ::= X\Phi \mid \Phi U^{\leq k}\Phi \mid \Phi U \Phi$$

where a is an atomic proposition, $\sim \in \{\lesssim, \gtrsim, \eqsim\}$, $\mathcal{E}$ is a superoperator, $k \in \mathbb{N}$. Φ is a *state* formula, while ϕ is a *path* formula. For instance, the formula $\mathbb{Q}_{\sim\epsilon}[\phi]$ means that the probability that the paths from a certain state satisfy the formula ϕ is constrained by ϵ; $Q = ?[\phi]$ computes the superoperator satisfying ϕ and $qeval((Q = ?)[\varphi], \rho)$ returns the density matrix obtained at the selected step. Finally, the formula $qprob((Q = ?)[\phi], \rho) = tr(qeval((Q = ?)[\phi], \rho)))$ computes the probability of satisfying ϕ, starting from the quantum state ρ [6].

3 Entangλe

The framework `Entang`λ`e`[5] has been implemented using Haskell, which allowed to import and re-use libraries already developed for Quipper. The first version of `Entang`λ`e`, which was limited to the translation of quantum circuits, has been presented in [1]. In order to translate a larger class of programs, we restricted Quipper to an *ad hoc* sublanguage for `Entang`λ`e`, called *Quip-E*, allowing the use of reset operators, unitary and measurement gates, and tail-recursion. In particular, in the tail-recursive programs we considered, the results of measurements are used as guard conditions for recursive calls (after being converted to Boolean values by using a dynamic lifting instruction). The `Body` of a tail-recursive quantum program written in *Quip-E*, should be written using the following instructions:

1. **reset:** A sequence of unitary operators is used to initialize a qubit as $|0\rangle$;
2. **unitary:** A unitary operator is applied to a list of qubits;
3. **measure:** A list of qubits is measured in the standard basis through the `measure` Quipper operator resulting in a list of bits;
4. **dynamic_lift:** A bit is lifted to a boolean throught the `dynamic_lift` Quipper operator;
5. **if-then-else:** Depending on the evaluation of a Boolean expression either a body `Body_C_1` or a body `Body_C_2` are used;
6. **exit_On:** It has been introduced in *Quip-E* to guarantee the translation of tail-recursive programs only, without other syntactical checks; this instruction can only be used as last instruction and its effect is the evaluation of a Boolean expression: if it is true, the program terminates, otherwise a loop to the first instruction occurs.

In the following there is a small example showing the main differences between the Quipper-like *Quip-E* code (on the left, with user-defined functions) and

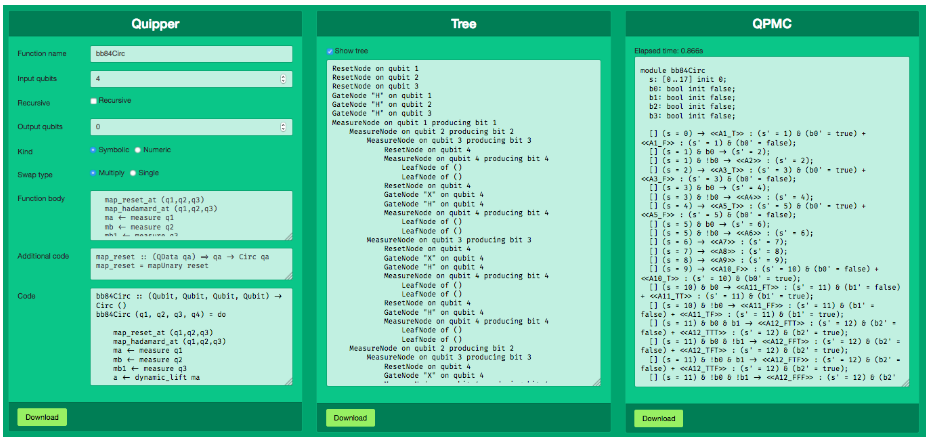
[5] Available at `https://github.com/miniBill/entangle`

an equivalent program using the standard syntax of Quipper (on the right).

```
exampleCirc :: (Qubit, Qubit) ->
    Circ RecAction
      exampleCirc (q1, q2) = do
          reset_at q1
          reset_at q2
          gate_X_at q2
          hadamard_at q1
          m <- measure q2
          bool <- dynamic_lift m
          if bool
              then gate_X_at q1
              else gate_Z_at q1
          m1 <- measure q1
          exitOn bool
```

```
exampleCirc :: (Qubit, Qubit) ->
    Circ ()
      exampleCirc (q1, q2) = do
          [q1, q2] <- qinit [True,
              False]
          hadamard_at q1
          m <- measure q2
          bool <- dynamic_lift m
          if bool
              then gate_X_at q1
              else gate_Z_at q1
                   exampleCirc(q1,
                       q2)
          m1 <- measure q1
```

Moreover, Entangλe now features a graphical interface to provide a more intuitive layout for writing quantum programs. A snapshot of Entangλe's interface is shown in Figure 1. It is divided into three main blocks: *Quipper*, *Tree* and *QPMC*, which will be analyzed in the following.



Function name: bb84Circ

Input qubits: 4

Recursive: [ ] Recursive

Output qubits: 0

Kind: [● Symbolic] ○ Numeric

Swap type: [● Multiply] ○ Single

Function body:

```
  map_reset_at (q1,q2,q3)
  map_hadamard_at (q1,q2,q3)
  ma ← measure q1
  mb ← measure q2
  mb1 ← measure q3
```

Additional code:

```
map_reset :: (QData qa) ⇒ qa → Circ qa
map_reset = mapUnary reset
```

Code:

```
bb84Circ :: (Qubit, Qubit, Qubit, Qubit) →
Circ ()
bb84Circ (q1, q2, q3, q4) = do

    map_reset_at (q1,q2,q3)
    map_hadamard_at (q1,q2,q3)
    ma ← measure q1
    mb ← measure q2
    mb1 ← measure q3
    a ← dynamic_lift ma
```

[✓] Show tree

```
ResetNode on qubit 1
ResetNode on qubit 2
ResetNode on qubit 3
GateNode "H" on qubit 1
GateNode "H" on qubit 2
GateNode "H" on qubit 3
MeasureNode on qubit 1 producing bit 1
    MeasureNode on qubit 2 producing bit 2
        MeasureNode on qubit 3 producing bit 3
            ResetNode on qubit 4
            MeasureNode on qubit 4 producing bit 4
                LeafNode of ()
                LeafNode of ()
            ResetNode on qubit 4
            GateNode "X" on qubit 4
            GateNode "H" on qubit 4
            MeasureNode on qubit 4 producing bit 4
                LeafNode of ()
                LeafNode of ()
        MeasureNode on qubit 3 producing bit 3
            ResetNode on qubit 4
            GateNode "X" on qubit 4
            GateNode "H" on qubit 4
            MeasureNode on qubit 4 producing bit 4
                LeafNode of ()
                LeafNode of ()
            ResetNode on qubit 4
            GateNode "H" on qubit 4
            MeasureNode on qubit 4 producing bit 4
                LeafNode of ()
                LeafNode of ()
    MeasureNode on qubit 2 producing bit 2
        MeasureNode on qubit 3 producing bit 3
            ResetNode on qubit 4
            GateNode "X" on qubit 4
```

Elapsed time: 0.866s

```
module bb84Circ
  s: [0..17] init 0;
  b0: bool init false;
  b1: bool init false;
  b2: bool init false;
  b3: bool init false;

  [] (s = 0) → <<A1_T>> : (s' = 1) & (b0' = true) +
<<A1_F>> : (s' = 1) & (b0' = false);
  [] (s = 1) & b0 → (s' = 2);
  [] (s = 1) & !b0 → <<A2>> : (s' = 2);
  [] (s = 2) → <<A3_T>> : (s' = 3) & (b0' = true) +
<<A3_F>> : (s' = 3) & (b0' = false);
  [] (s = 3) & b0 → (s' = 4);
  [] (s = 3) & !b0 → <<A4>> : (s' = 4);
  [] (s = 4) → <<A5_T>> : (s' = 5) & (b0' = true) +
<<A5_F>> : (s' = 5) & (b0' = false);
  [] (s = 5) & b0 → (s' = 6);
  [] (s = 5) & !b0 → <<A6>> : (s' = 6);
  [] (s = 6) → <<A7>> : (s' = 7);
  [] (s = 7) → <<A8>> : (s' = 8);
  [] (s = 8) → <<A9>> : (s' = 9);
  [] (s = 9) → <<A10_F>> : (s' = 10) & (b0' = false) +
<<A10_T>> : (s' = 10) & (b0' = true);
  [] (s = 10) & b0 → <<A11_FT>> : (s' = 11) & (b1' = false)
+ <<A11_TT>> : (s' = 11) & (b1' = true);
  [] (s = 10) & !b0 → <<A11_FF>> : (s' = 11) & (b1' =
false) + <<A11_TF>> : (s' = 11) & (b1' = true);
  [] (s = 11) & b0 & b1 → <<A12_FTT>> : (s' = 12) & (b2' =
false) + <<A12_TTT>> : (s' = 12) & (b2' = true);
  [] (s = 11) & b0 & !b1 → <<A12_FFT>> : (s' = 12) & (b2' =
false) + <<A12_TFT>> : (s' = 12) & (b2' = true);
  [] (s = 11) & !b0 & b1 → <<A12_FTF>> : (s' = 12) & (b2' =
false) + <<A12_TTF>> : (s' = 12) & (b2' = true);
  [] (s = 11) & !b0 & !b1 → <<A12_FFF>> : (s' = 12) & (b2'
```

Fig. 1: Entangλe GUI.

3.1 Quipper

This block is devoted to the writing of programs that should be translated; it takes in input only *Quip-E* and Quipper code, and consists of eight sections:

Function name: the name of the Quipper program that we want to write;

Input qubits: the number of qubits in input to the Quipper program;

Recursive: if checked it automatically changes the type signature of the program;

Output qubits: the number of output qubits, provided that the program is non-recursive;

Type: switches the matrix representation from symbolic, which uses embedded QPMC instructions, to numeric, which provides a numeric, MATLAB-style, representation of matrices, and vice versa;

Swap: allows the selection of a different algorithm to build the swap matrices, `multiply` generates the swaps using an algorithm based on permutation by transposition, while `single` uses an algorithm which outputs the matrix representation of the swaps;

Function body: the body (i.e., the set of instructions in the right order) of the quantum program to be translated. In this box it is not required to provide the method signature since the complete *Quip-E* program will be generated automatically by `Entangλe` in the Code section;

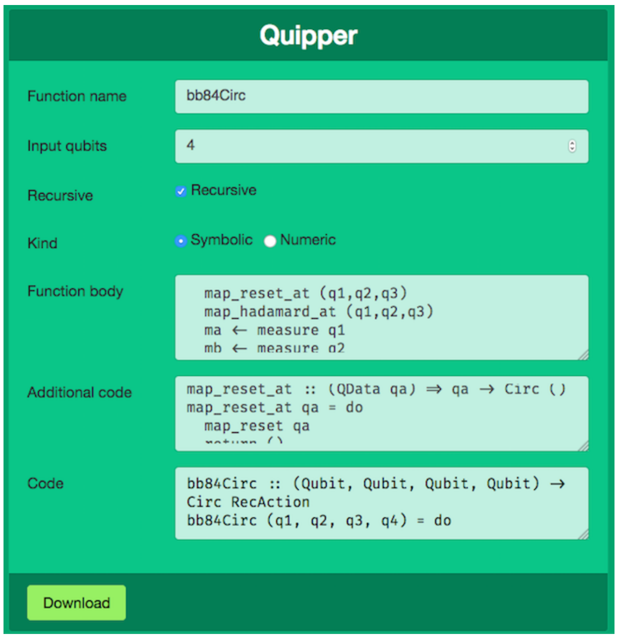
Additional code: auxiliary Quipper/Haskell code that cannot be translated from the function body;

Code: the final Quipper program, generated by `Entangλe` by using all the information provided above. A .hs file with the code can be downloaded by clicking the Download button.

In the **Function body** we constrained the programmer to a preferred coding style, i.e., we restrict the *Quip-E* syntax to certain choices: the name of the variables of type `Qubit`, if more than one, are denoted by $\mathtt{q}_i$, with $i = 1, \ldots, n$, with n number of input qubits, otherwise, if there is only one input qubit it should be labeled `q`. Moreover, measured qubits cannot be re-initialized in the last part of the body. The instruction order to be preserved is: unitary and reset instructions, measurements, dynamic lifting, and control flow. `Entangλe` translates tail recursive programs, thus the recursive call must be (when present) the last instruction. A snapshot of the Quipper block can be seen in Figure 2.

3.2 Tree Block

The Tree block displays an abstract representation of the instructions in the program. It represents the intermediate translation steps. In this block, each quantum gate is associated to the qubits on which it is acting. Measurements produce a binary branch terminating with two leaves which can be of different type, according to the type of program that we want to translate. This block can be hidden.



Function name: bb84Circ

Input qubits: 4

Recursive: [✓ Recursive]

Kind: [✓ Symbolic] Numeric

Function body:

```
  map_reset_at (q1,q2,q3)
  map_hadamard_at (q1,q2,q3)
  ma ← measure q1
  mb ← measure q2
```

Additional code:

```
map_reset_at :: (QData qa) ⇒ qa → Circ ()
map_reset_at qa = do
  map_reset qa
```

Code:

```
bb84Circ :: (Qubit, Qubit, Qubit, Qubit) →
Circ RecAction
bb84Circ (q1, q2, q3, q4) = do
```

Fig. 2: Quipper Block

3.3 QPMC Block

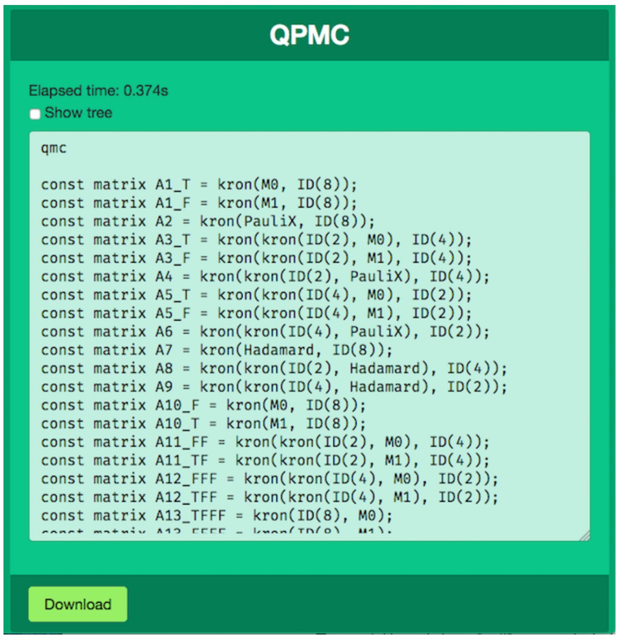
The QPMC block displays the translation of the Quipper program into the corresponding QPMC model. It provides both matrices and module to be checked using QCTL. This final code can be downloaded as a .prism file and used as an input for the QPMC model checker. A snapshot of the QPMC block can be seen in Figure 3. For further references about the intermediate translation steps see [1], [2].

4 Case Studies

We present here some results on the tests we performed on two quantum algorithms. We tested **Entangλe** on both an instance of Grover's quantum search algorithm and on a tail-recursive version of the BB84 protocol.

4.1 Grover's Quantum Search

The aim of this algorithm is searching for the index x of an element in a unstructured array of length N. We take $N = 2^n$, so that the indexes are represented by n-bit strings. The algorithm solves the problem by considering a function $f : \{0,1\}^n \to \{0,1\}$ such that $f(x) = 1$ if and only if the string x points to the



```
qmc

const matrix A1_T = kron(M0, ID(8));
const matrix A1_F = kron(M1, ID(8));
const matrix A2 = kron(PauliX, ID(8));
const matrix A3_T = kron(kron(ID(2), M0), ID(4));
const matrix A3_F = kron(kron(ID(2), M1), ID(4));
const matrix A4 = kron(kron(ID(2), PauliX), ID(4));
const matrix A5_T = kron(kron(ID(4), M0), ID(2));
const matrix A5_F = kron(kron(ID(4), M1), ID(2));
const matrix A6 = kron(kron(ID(4), PauliX), ID(2));
const matrix A7 = kron(Hadamard, ID(8));
const matrix A8 = kron(kron(ID(2), Hadamard), ID(4));
const matrix A9 = kron(kron(ID(4), Hadamard), ID(2));
const matrix A10_F = kron(M0, ID(8));
const matrix A10_T = kron(M1, ID(8));
const matrix A11_FF = kron(kron(ID(2), M0), ID(4));
const matrix A11_TF = kron(kron(ID(2), M1), ID(4));
const matrix A12_FFF = kron(kron(ID(4), M0), ID(2));
const matrix A12_TFF = kron(kron(ID(4), M1), ID(2));
const matrix A13_TFFF = kron(ID(8), M0);
const matrix A13_FFFF = kron(ID(8), M1);
```

Fig. 3: QPMC Block

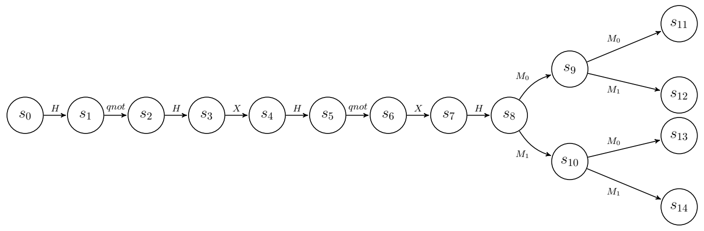
searched element. Classically, this problem can be solved in $O(N)$ steps, while using a quantum oracle it can be probabilistically solved in $O(\sqrt{N})$ steps. The Grover algorithm gives in input to the quantum oracle all the possible strings at the same time. Then it marks the strings corresponding to possible solutions and it performs some steps to maximize the probability of getting the desired result after the measurement. The result is the index of the searched element. In general, the algorithm is probabilistic, but for $N = 4$ after one iteration it behaves in a deterministic way, giving the right result with probability equal to 1. In our experiments we used $N = 4$.

We wrote an oracle which returns the string $x = 3$. The *Quip-E*, which in this case is the same as the Quipper one, implementation can be seen below:

```
grover :: (Qubit, Qubit, Qubit) -> Circ (Bit, Bit)
grover (q1,q2,q3) = do
    hadamard_at q1
    hadamard_at q2
    hadamard_at q3
    qnot_at q3 `controlled` [q1, q2]
    hadamard_at q1
    hadamard_at q2
    gate_X_at q1
    gate_X_at q2
    hadamard_at q2
    qnot_at q2 `controlled` q1
    hadamard_at q2
    gate_X_at q1
    gate_X_at q2
```

```
hadamard_at q1
hadamard_at q2
hadamard_at q3
measure (q1,q2)
```

Verification of Grover's Quantum Search For space reasons we omit the automatically generated QPMC code. A representation of the QMC can be seen in the following:



According to the calculations we should reach the terminal state s_{14} with probability equal to 1, while the other terminal states must have an associated probability equal to 0. We tested QCTL formulae to evaluate the density matrix associated to each terminal state with input state $|1\rangle\langle 1|$ and the results are the following:

```
qeval(Q=? [F (s = 11)], |1>_8 <1|_8);

    0  0  0  0  0  0  0  0
    0  -0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
```

```
qeval(Q=? [F (s = 12)], |1>_8 <1|_8);

    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
```

```
qeval(Q=? [F (s = 13)], |1>_8 <1|_8);

    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
```

```
qeval(Q=? [F (s = 14)], |1>_8 <1|_8);

    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  -0
    0  0  0  0  0  0  0   1
```

The trace of the first three matrices is equal to 0, thus the probability of reaching those states is null. The last matrix has trace equal to 1, thus the computation will reach that state (s_{14}). We also tested formulas to calculate the accumulated superoperators for each state, but since the resulting matrices have size $2^6 \times 2^6$ for space reasons we do not report them here. The results can be found in the Examples folder at `https://github.com/miniBill/entangle`.

4.2 Recursive BB84

In this example we focus on protocols that use quantum effects, such as entanglement, to guarantee secure communication between two parties (Alice and Bob) communicating on classical channels. Errors of different type can occur during the communication, in particular due to noise and decoherence effects. Moreover, in QKD protocols such as BB84, we want not only to guarantee that the channel is free of noise effects that could change the output, but also that no eavesdropper could obtain the key the two parties are exchanging. In particular, the BB84 protocol enables two parties sharing a random and secure key, which could be used for classical encryption schemes such as the one-time pad.

The BB84 protocol between parties Alice and Bob works as follows:

1. Alice generates two classical, *random*, n-bit strings $\mathbf{a} = [a_1, \dots, a_n]$ and $\mathbf{b} = [b_1, \dots, b_n]$;
2. Alice generates a quantum state $|\Psi\rangle = \bigotimes_{i=1}^{n} |\psi_{a_i b_i}\rangle$ using the two strings; the symbol $\bigotimes$ represents the tensor product; string `b` is used to determine in which basis each element of string `a` will be encoded:
$$\begin{array}{ll} |\psi_{00}\rangle = |0\ \rangle & |\psi_{10}\rangle = |1\rangle \\ |\psi_{01}\rangle = |+\rangle & |\psi_{11}\rangle = |-\rangle \end{array}$$
3. The state $|\Psi\rangle$ is transmitted along a quantum channel to Bob;
4. Bob generates a classical, *random*, n-bit string $\mathbf{b}' = [b'_1, \dots, b'_n]$ used to determine in which basis measure $|\Psi\rangle$, obtaining the classical string $\mathbf{a}' = [a'_1, \dots, a'_n]$;
5. Alice and Bob compare, sending classical information on a public channel, the strings $\mathbf{b}$ and $\mathbf{b}'$: if there is any difference between them, the strings $\mathbf{a}$ and $\mathbf{a}'$ are considered not reliable, and then discarded, if they match the check is passed. In the recursive version, instead of manually checking whether the strings are reliable or not, this is done automatically by the algorithm that restarts the protocol if the final check fails.

In this experiment, we implemented a BB84 protocol where only one-bit strings are transmitted. This is the simplest setting possible and can be generalized to strings of arbitrary length. Since quantum measurement outcomes are random, we decided to generate the classical strings using this kind of technique, i.e., we measure two qubits previously put in a uniform superposition by the application of an Hadamard gate and then we get the classical outcome. The Quipper code can be seen in the following:

BB84 rec (1)

```
bb84RecCirc :: (Qubit, Qubit, Qubit
     , Qubit) -> Circ RecAction
bb84RecCirc (q1, q2, q3, q4) = do

      map_reset_at (q1,q2,q3)
      map_hadamard_at (q1, q2,q3)
      ma <- measure q1
      mb <- measure q2
      mb1 <- measure q3
      a <-dynamic_lift ma
      b <- dynamic_lift mb
      b1 <- dynamic_lift mb1
      if (False) && (not b)
                then do
                  reset_at q4

                else if True && (
                    not b)
                  then do
```

BB84 rec (2)

```
                    reset_at q4
                    gate_X_at q4

                  else if (False)
                      && b
                    then do
                      reset_at q4
                      hadamard_at
                          q4

                    else do
                        reset_at q4
                        gate_X_at
                            q4
                        hadamard_at
                            q4
      ma1<- measure q4
      a1 <- dynamic_lift ma1
      exitOn $ b==b1 && a==a1
```

For space reasons we omit in this paper the QPMC code automatically generated by `Entangλe` (it can be found in the Examples folder on the `Entangλe` repository). The abstract model of the quantum Markov chain can be seen in Figure 4.

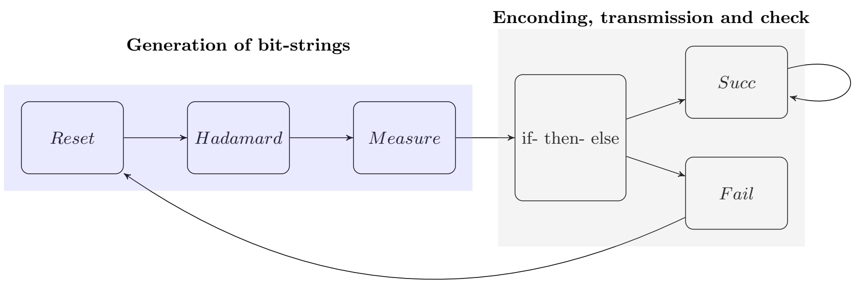
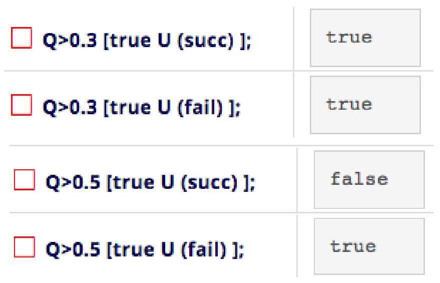


Fig. 4: High level representation of recursive BB84 steps.

4.3 Verification of the BB84 model

We show here a test performed by using QCTL formulae [6]. In Figure 5 we show an example of *until* formulae. The first two formulae show that both *success* and *failure* states can be reached with probability > 0.3. The second set of formulae bounds the probability that a state *success* is reached with a value < 0.5. By



☐ Q>0.3 [true U (succ)];	true
☐ Q>0.3 [true U (fail)];	true
☐ Q>0.5 [true U (succ)];	false
☐ Q>0.5 [true U (fail)];	true

Fig. 5: QCTL example test for recursive BB84 protocol.

generating the density matrices, and by computing the trace, these results are consistent, since `trace(SUCC)= 0.37500` and `trace(FAIL) = 0.62500`[6]

5 Conclusions

In this work we presented `Entangλe`, a translation framework from Quipper programs to QPMC models. Other model checking techniques for quantum protocols have been considered, e.g., in [8][9][3] and [4]. These techniques are interesting, but they are either restricted to a particular class of quantum circuits, while Quipper and QPMC provide a more flexible paradigm, or their implementation was unavailable. `Entangλe` allows to both write and verify quantum protocols, using an high-level programming language. It is able to translate also Quipper programs in which measurement results may control termination of quantum protocols and algorithms. We used `Entangλe` to translate and verify different protocols, with the final results validating our expectations. Some steps have been moved towards the optimization of our framework to verify more complex quantum programs. More optimizations should be done from the model checking point of view, involving the automatic verification of more complex properties, i.e., entanglement and other quantum effects.

References

1. L. Anticoli, C. Piazza, L. Taglialegne, and P. Zuliani. Towards quantum programs verification: From quipper circuits to QPMC. In *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*, pages 213–219, 2016.
2. L. Anticoli, C. Piazza, L. Taglialegne, and P. Zuliani. Verifying quantum programs: From quipper to QPMC. *CoRR*, abs/1708.06312, 2017.

[6] Computed with MATLAB, using the density matrices generated by QPMC and the formula `qeval`. Further examples are available at `https://github.com/miniBill/entangle/tree/master/res/Entangle_Tests`

3. P. Baltazar, R. Chadha, and P. Mateus. Quantum computation tree logic – model checking and complete calculus. *International Journal of Quantum Information*, 2008.
4. P. Baltazar, R. Chadha, P. Mateus, and A. Sernadas. Towards model-checking quantum security protocols. In *Proceedings of the first workshop on Quantum Security: QSEC'07*. ieee Press, 2007.
5. C.H. Bennett and G. Brassard. Quantum public key distribution reinvented. *SIGACT News*, 18(4), 1987.
6. Y. Feng, E. M. Hahn, A. Turrini, and L. Zhang. QPMC: A Model Checker for Quantum Programs and Protocols. In Nikolaj Bjørner and Frank D. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, June 24-26, 2015, Proceedings*, Lecture Notes in Computer Science. Springer, 2015.
7. Y. Feng, N. Yu, and M. Ying. Model checking quantum Markov chains. *Journal of Computer and System Sciences*, 2013.
8. S. Gay, R. Nagarajan, and N. Papanikolaou. Probabilistic model-checking of quantum protocols. In *Proceedings of the 2nd International Workshop on Developments in Computational Models*, 2006.
9. S. J. Gay, N. Papanikolaou, and R. Nagarajan. QMC: A model checker for quantum systems. In *In Proceeding of the 20th International Conference on Computer Aided Verification*, 2008.
10. A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. Quipper: A Scalable Quantum Programming Language. *SIGPLAN Not.*, 48(6), 2013.
11. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5), 1994.
12. E. Knill. Conventions for Quantum Pseudocode. Technical report, Los Alamos National Laboratory, 1996.
13. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *LNCS*, volume 6806, 2011.
14. M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2011.
15. J. Preskill. *Lecture Notes for Physics 229: Quantum Information and Computation.* CreateSpace Independent Publishing Platform, 1998.
16. J.M. Smith, N.J. Ross, P. Selinger, and B. Valiron. Quipper: concrete resource estimation in quantum algorithms. Extended abstract for a talk given at the 12th International Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2014, Grenoble. Available from arxiv1412.0625, 2014.