Formalization
of
Quantum Protocols using Coq

Jaap Boender Florian Kammüller
Rajagopal Nagarajan
Department of Computer Science, School of Science and Technology, Middlesex University, London, UK
J.Boender,F.Kammueller,R.Nagarajan@mdx.ac.uk

Quantum Information Processing, which is an exciting area of research at the intersection of physics and computer science, has great potential for influencing the future development of information processing systems. The building of practical, general purpose Quantum Computers may be some years into the future. However, Quantum Communication and Quantum Cryptography are well developed. Commercial Quantum Key Distribution systems are easily available and several QKD networks have been built in various parts of the world. The security of the protocols used in these implementations rely on information-theoretic proofs, which may or may not reflect actual system behaviour. Moreover, testing of implementations cannot guarantee the absence of bugs and errors. This paper presents a novel framework for modelling and verifying quantum protocols and their implementations using the proof assistant Coq. We provide a Coq library for quantum bits (qubits), quantum gates, and quantum measurement. As a step towards verifying practical quantum communication and security protocols such as Quantum Key Distribution, we support multiple qubits, communication and entanglement. We illustrate these concepts by modelling the Quantum Teleportation Protocol, which communicates the state of an unknown quantum bit using only a classical channel.

1 Introduction

Quantum cryptography aims to overcome the limitations of classical cryptography by providing perfect security. The Quantum Key Distribution protocol by Bennett and Brassard (BB84) [4] showed how quantum superposition and non-clonability of quantum states can be used to communicate a secret key, while *a posteriori* verifying on an open line whether the communication has been intercepted. Thus, by sacrificing a small portion of the transmitted key, it is possible to detect eavesdropping in almost 100% of cases, which in turn, guarantees (almost) perfect secrecy when using the transmitted one-time key in future communications. This protocol has been implemented in commercial products, e.g., by Id Quantique [6], MagiQ [25], NEC and Toshiba, amongst others, and has been used in practical applications, e.g., the Geneva election ballot count [29]. Various QKD networks have been built, including the DARPA Quantum Network in Boston, the SeCoQC network around Vienna and the Tokyo QKD Network.

Physical restrictions of quantum communication, like preserving photon states over long distances, are gradually being resolved, for example, by quantum repeaters [31] and using quantum teleportation, i.e., transmission of quantum bits through a classical medium. There is no doubt that quantum communication and quantum cryptographic protocols will become an integral part of our society's infrastructure.

On the theoretical side, quantum key distribution protocols such as BB84 have been proved to be unconditionally secure [26]. It is important to understand that this is an information-theoretic proof, which does not necessarily guarantee that *implemented systems* are unconditionally secure. That is why

Chris Heunen, Peter Selinger, and Jamie Vicary (Eds.):
12th International Workshop on Quantum Physics and Logic (QPL 2015).
EPTCS 195, 2015, pp. 71–83, doi:10.4204/EPTCS.195.6

alternative approaches, such as those based on formal methods, could be useful in analysing behaviour of implemented systems.

Quantum Formal Methods has been an active research area recently and quantum communication and protocols have been investigated and formalised. Formal investigations of quantum protocols [14, 22, 33] use languages for distributed communications, like CCS and the π-calculus [27] or related formalisms. Such formalisations enable reasoning about the properties of the quantum communication model using techniques like type checking [15], as well as bisimulation and equational reasoning [23, 11, 12] for these underlying languages. Other approaches which are more automated and tool-based include model checking and equivalence checking [16, 13, 17, 1, 2]. In our approach, which we believe is new, we directly formalise the mathematical model of quantum communication in constructive logic using the Coq proof assistant. In this way, we build a mechanised and partly automated logical theory of quantum communication. This theory allows the implementation of quantum cryptography protocols based on their mathematical models and the proof of their properties. Moreover, since Coq's logic is constructive, extraction of code is possible.

The expressivity of Coq and its code extraction properties make it an interesting tool for the creation of generic frameworks, i.e. abstract formalisations of practical specification and verification problems that may be instantiated to various applications. A famous application of Coq is the formal proof of the four colour theorem by Gonthier [18]. Another impressive achievement by Gonthier, together with others, is the Coq proof of the Odd Order Theorem [19]. Coq has also been used for the formal verification of large pieces of software, such as a C compiler in the CompCert project [24]. Other higher-order theorem provers, such as Isabelle, HOL, NuPRL and PVS have been used in a variety of applications. Mathematicians have also been using theorem proving techniques to formalise and prove difficult conjectures and theorems. Vladimir Voevodsky and others are using Coq to formalise homotopy theory. Thomas Hales has been working on a project called Flyspeck, formalising his proof of the Kepler conjecture in the theorem prover HOL Light.

The contribution of this paper is a framework for modelling and proving quantum protocols in Coq [21]. We assume that the reader is familiar with quantum theory and computing, for details we refer to the book by Nielsen and Chuang [28] or the paper by Rieffel and Polak [32] tailored to computer scientists. We provide a Coq library for quantum bits (qubits), quantum gates, and quantum measurement (Section 3). An important aspect of quantum information processing is the use of entanglement of qubits and their transmission. Our Coq library also provides this, through measurement of qubit vectors (Section 4). Finally, we illustrate the functioning and application of entanglement and measurement by modelling quantum teleportation (Section 5), which is a protocol that allows the use of so-called Einstein-Podolsky-Rosen (EPR) pairs of maximally entangled qubits to communicate the state of a qubit from Alice to Bob using a pair of classical bits. This technique is crucial for realistic implementations of quantum protocols over long distances by repeaters as mentioned earlier.

2 Interactive Proof in Coq

Interactive theorem proving is semi-automated proof development, usually in Higher Order Logic (HOL) or some variant, like the Calculus of Indictive Constructions that underlies the interactive theorem proving system Coq [5, 21]. Coq's logic is not classical: it is a constructive type theory interpreted as a logic according to the Curry-Howard isomorphism [20]. Although constructivity imposes restrictions that can be quite awkward at times, it offers one decisive advantage. Since all proofs are constructions of witnesses, they are executable as programs. Program code can be extracted automatically from the Coq tool into the programming language OCaml which is also the implementation language of the Coq system.

Another advantage of Coq is that its type system is more powerful than the simple type theory that is the basis for classical HOL. For example, the type system of Coq allows *dependent types*, which can be used to represent universal and existential quantification. However, the type system is sufficiently restricted that it remains decidable. Note, however, that there are some constructions in Coq that violate decidability of type checking. Generally, type checking in Coq is decidable, but, as an example, pattern matching constructions over dependent types can become undecidable. Coq also offers dependent records and additionally a separate module system [8].

Coq's original foundation, the Calculus of Inductive Constructions, has been extended to a calculus of *inductive* constructions [9] that includes inductive definitions. Similar to a datatype definition in a programming language like ML, an inductive definition in Coq consists of a set of rules describing the signature of the constructors of the type. However, Coq's logic, according to the Curry-Howard isomorphism, is defined by its types. Therefore, an inductive definition may also be used to define logical formulas. Using inductive definitions for the formalisation of computer science related subjects is very natural, because types defined by an inductive definition automatically contain an induction principle and so-called exhaustion properties that enable to reason by *inversion*. This means that, if we need to show a property for all elements of a type, it suffices to make a case analysis over all different manifestations of elements of the type given by the constructors of the inductive definition.

An excellent introduction to Coq is [7]. More concrete features of the Coq system will be explained when we use them in the following sections.

Our quantum formalisation is work in progress and the current version is available online [30]. It comprises around 2000 lines of Coq code. About 700 of these lines make up the specification of some elementary matrix theory (there are existing matrix libraries in Coq, but these do not work together well with the C-CoRN library). The development is discussed in the rest of the paper.

3 Qubits and Quantum Gates in Coq

For the formalisation of qubits and their basic infrastructure in Coq, we need theory libraries supporting vector spaces and matrix operations. The constructive Coq repository C-CoRN at Nijmegen [10] offers a mathematical library that we use as a starting point for the theory of real and complex numbers. We give here just a short overview of the matrix operations defined in our framework. Further information can be found in the attached online resources. Based on the C-CoRN module `CRings` for constructive rings, we have built a module `MatrixOps` specifying the signature of $m \times n$-matrices and their operations over a ring. As operations, we define matrix equality using the notation {=}, matrix addition {+}, matrix multiplication {*}, matrix scalar multiplication {**}. We also define tensor multiplication {o} needed for multiple qubits (see Section 4). We specify the corresponding properties of these operations in module `MatrixSpec`. As a first consistency check of our specification, we can already prove some algebraic properties of matrices. Examples of such properties include facts such as matrices with addition form an abelian (commutative) group. These proofs are contained in the module `MatrixSetoid`, which is used in the main formalisation in the file `Quantum.v`, and will be further explained below.

Given this foundation, we can define qubits as complex column vectors.

```
Definition qubit (n: nat) :=
 {q: matrix (2^n) 1 | vector_length q [=] [1]}
```

This notation defines `qubit` as a $2^n \times 1$-matrix over the complex numbers `CC`, such that the vector length is 1 (i.e. the vector is a unit vector). In this section, we only consider single qubits, i.e., `n` is 1.

We provide basic infrastructures for `qubits`, for example, an equivalence relation and related proofs and auxiliary lemmas. In order to create an element of the `qubit n` type, we will need to provide both a matrix and a proof that the vector length of this matrix is 1.

We use our own matrix library for matrix operations. We have a type constructor that, given two natural numbers, constructs a new Coq type representing a matrix:

```
Parameter matrix: nat -> nat -> Type.
```

The basis vectors can be defined for any number of qubits as follows. The following function `basis` is auxiliary.

```
Definition basis n k: (matrix n 1) :=
 create n 1 (fun i j =>
  if eq_nat_dec (‘i) (S k)
  then [1]
  else [0]).
```

Then we add a lemma stating that any matrix created by calling `basis` represents a unit vector:

```
Lemma basis_length: forall n k,
  k < 2^n -> vector_length (basis (2^n) k) [=] [1].
```

And finally we combine the function and the lemma into the definition of a qubit. We use the `exist` function to combine the matrix created by calling `basis` and the proof from `basis_length` into a member of the `qubit n` type.

```
Definition basis_q {n} (i: nat | i < 2^n): qubit n :=
 exist _ (basis (2^n) (‘i))
  (basis_length (n) (‘i) (proj2_sig i)).
```

Note that, `basis_q 2 0 = [1; 0; 0; 0]` and `basis_q 2 1 = [0; 1; 0; 0]`, etc. This is not a bug, but a feature, since the superposition states of qubits necessitate higher dimensional vector spaces, as we will see in Section 4.

In this development, we are using the standard basis for qubits. Of course, other bases can be used just as easily, because measurement with respect to another basis is equivalent to applying a suitable transformation and then measuring with respect to the standard basis.

Quantum gates over n qubits are now defined as the type of $2^n \times 2^n$ matrices.

```
Definition gate (n: nat) :=
 {g:matrix (2^n) (2^n)|unitary g}.
```

Our matrix library is capable of using any ring as a coefficient algebra. The type `gate` is a definition in the module `Quantum` representing the specific “subtype” of quadratic matrices over complex numbers. This is achieved by the trailer in `Quantum` instantiating our module `MatrixSetoid` with C-CoRN’s complex numbers `CC` as the base ring `C` and opening it with `Import` in the current context of the module `Quantum`.

```
Declare Module Matrix: MatrixSetoid with
  Definition C := (cf_crr CC).
Import Matrix.
```

We also use a dependent type to specify that gates are unitary matrices.

With this infrastructure at hand, we can define quantum gates in Coq. For example, the X gate is defined as follows.

```
Definition x_function
 (i: nat | 0 < i <= 2) (j: nat | 0 < j <= 2): C :=
 match `i, `j with
 | 1, 2 | 2, 1 => [1]
 | _, _        => [0]
 end.

Program Definition x_gate: gate 1 :=
 exist (fun x => unitary x) (create 2 2 x_function) _.
```

We first define a function that matches the specific matrix coordinates with the value at that coordinate, and use this function to create a matrix. The `create` function works in such a way that if M is the result of calling `create` with function `f`, then $M_{x,y} = \texttt{f}\ x\ y$ (for all x and y that are valid coordinates for the matrix M).

We use Coq's `Program` extension to make working with dependent types easier. It will generate the proof obligations that are needed and try to solve them automatically. Any obligations that cannot be solved are left to the user to prove. In this case, there is only one proof obligation: we need to prove that the matrix is unitary. This is easy to do, since the matrix is only 2×2 in size.

A gate represents a qubit transformation function. In order to apply a gate to a qubit we define the following operator.

```
Program Definition apply {n} (q: qubit n) (g: gate n):
 qubit n :=
  exist (fun x => vector_length x [=] [1])
   ((`g) {*} (`q)) _.
```

This function multiplies the qubit and gate matrices. It needs some more syntax because of the dependent types: first we discard the proofs from the g and q parameters (by using the backquote operator) so that we keep only the actual matrices. These we multiply, and then we use `exist` to create a new inhabitant of a dependent type. This is necessary, since the result is a qubit, so we will need to prove that it is a unit vector. The `Program` extension again helps us to achieve this easily.

We formalise the Hadamard gate just like any of the other quantum gates as `hadamard`. In fact, it is the same as the `x_gate` shown above with the matrix entries `Zero, One, One, Zero` replaced by `onestwo, onestwo, onestwo, --onestwo`.

Here `onestwo` represents $\frac{1}{\sqrt{2}}$ and the definition in Coq is as follows.

```
Definition onestwo: CC :=
  Build_CC_set (One [/] NRootIR.sqrt Two
  (less_leEq _ _ _ (pos_two _)) [//] stwo_pos) ZeroR.
```

The definition of division and square root in C-CoRN may seem complex but it integrates mathematical accuracy. The division operator has an extra argument (`stwo_pos`) in order to ensure that the divisor is not zero. Similarly, the square root function takes an extra argument that ensures that we do not take the square root of a negative number.

4 Formalising Multiple Qubits, Measurement, and Entanglement

We now explain how tensor products, measurement and entanglement are formalised in our Coq framework.

The Coq definition for the tensor product is a straightforward type of matrices.

```
Parameter tensor_mult: forall m n p q,
 matrix m n -> matrix p q -> matrix (m*p) (n*q).
```

The specification of tensor multiplication is the following.

$$M_{i,j}N_{k,l} = R_{p(i-1)+k,q(j-1)+l} \tag{1}$$

When we formalise this definition in Coq, we need to provide the proofs that the indices are within bounds (for example, that $0 < p(i-1)+k \leq m*p$). To state the fact that two indices that are within the source matrices are also within bounds in the result matrix, we use the lemma `tensor_bounds` (the backquote is used to extract the number from the dependent type):

```
Lemma tensor_bounds: forall m p
 (i: nat | 0 < i <= m) (k: nat | 0 < k <= p),
  0 < p*(`i-1)+(`k) <= m*p.
```

This lemma can then be used in the specification of tensor product multiplication. To improve the readability of formulas we define the infix notation `{o}` for `tensor_mult`. The definition then maps the tensor multiplication to the complex number multiplication `[*]` following the indexing of definition (1) while supplying proofs that indices are in range.

```
Parameter tensor_mult_spec1:
 forall m n p q (mx1: matrix m n) (mx2: matrix p q)
 i j k l,
  (mx1 {o} mx2) {exist (fun x => 0 < x <= m*p)
                  (p*(`i-1)+`k)
                 (tensor_bounds m p i k),
                 exist (fun x => 0 < x <= n*q)
                  (q*(`j-1)+`l)
                 (tensor_bounds n q j l)}
 [=]
 mx1 {i, j} [*] mx2 {k, l}.
```

In order to represent the use of the tensor product to combine gates and qubits, we prove the following theorem (in `MatrixTheory`).

```
Theorem mult_dist_tensor: forall {m1 n1 p1 m2 n2 p2}
 (mx1: matrix m1 n1) (mx2: matrix n1 p1)
 (mx3: matrix m2 n2) (mx4: matrix n2 p2),
 (mx1 {*} mx2) {o} (mx3 {*} mx4) {=}
 (mx1 {o} mx3) {*} (mx2 {o} mx4).
```

4.1 Measurement

For modelling purposes, we can treat measurement of multiple qubits as sequential measurement of single qubits.

In our model, measurement of a quantum state consists of taking the state (represented by the `qubit n` Coq type) and a bit number, and then computing the probabilities of measuring a 0 or a 1 for that bit.

As a quantum state is represented by a matrix, and each element of this matrix represents one possible measurement result, measuring one specific bit amounts to taking the sum of the relevant elements.

For example, in a three-qubit state, there are eight elements, of which the first represents the probability of measuring 000, the second the probability of measuring 001, and so on. If we want to measure the second bit, the probability of measuring 0 for this bit is equal to the sum of the probabilities of measuring 000, 001, 100 and 101.

The `sum_pair` function takes a bit number *i* and a quantum state *q*. It returns a pair of probabilities for bit *i*: that of measuring 0 and that of measuring 1.

```
Definition sum_pair n i (q: qubit n): IR * IR :=
 foldn (2^n) (2^n) (le_refl (2^n)) (fun k Hk acc =>
  match acc with
  | (acc1, acc2) =>
    if digit_is_zero i k
    then (acc1 [+]
     AbsCC ((`q) {exist _ _ (plusone _ _ Hk),
      exist _ _ (plusone _ _ (lt_0_Sn 0))}) [^] 2,
     acc2)
    else (acc1, acc2 [+]
     AbsCC ((`q) {exist _ _ (plusone _ _ Hk),
      exist _ _ (plusone _ _ (lt_0_Sn 0))}) [^] 2)
  end
 ) ([0], [0]).
```

In order to compute the new quantum state after a measurement, half of the values in the matrix will become 0 (if we have just measured that bit 5 is 0, then all the states that represent bit 5 being 1 have become impossible!). However, a quantum state is a unit vector, so we will have to normalise the remaining values to keep this property. We do this by dividing each value by the sum of all remaining values. In this way, the probability distribution for the bits that have not been measured remains the same.

We need to be sure that this is not a division by zero. Therefore we introduce the following axiom, saying that it is decidable whether the probability of measuring 0 or 1 for a given bit is 0 or not. This is not very farfetched, since an event that has probability 0 will never happen. These axioms internalise that knowledge.

```
Axiom sum_pair1: forall {n} (i: nat | i < n)
 (q: qubit n),
  fst (sum_pair (`i) q) [=] [0] or
  [0] [<] fst (sum_pair (`i) q).

Axiom sum_pair2: forall {n} (i: nat | i < n)
 (q: qubit n),
  snd (sum_pair (`i) q) [=] [0] or
  [0] [<] snd (sum_pair (`i) q).
```

Now we can define the function `nqv` (for ‘new qubit vector’) that computes the new quantum state after a measurement. The parameter *f* should be either the identity function or the boolean NOT function. This allows us to compute both the new quantum state after a measurement of 0 and after a measurement of 1 with the same function. We also add the sum, or the probability of measuring the result, and a proof that it is greater than 0, needed for the division.

```
Definition nqv {n} (k: nat) (f: bool -> bool)
 (sum: IR) (sum_proof: [0] [<] sum) (q: qubit n) :=
 create (2^n) 1 (fun i j =>
  if f (digit_is_zero k (‘i))
  then (‘q) i, j [/]
   cc_IR (NRootIR.sqrt sum (less_leEq _ _ _ sum_proof))
   [//] (re_ap_zero _ (ap_symmetric _ _ _
    (less_imp_ap _ _ _ (NRoot_pos _ _ _ _ sum_proof))))
  else [0]
 ).
```

The division needs three arguments: two for the division operation and one for a proof that the divisor is not 0.

Note that these functions are all auxiliary and not intended to be called on their own. The main function that takes care of measurement is shown below.

```
Program Definition measure {n} (i: nat | i < n)
 (q: qubit n): list (IR * qubit n) :=
 match sum_pair1 i q with
 | inl _ => (* zero *) [([1], q)]%list
 | inr sum0_gt =>
   match sum_pair2 i q with
   | inl _ => (* zero *) [([1], q)]%list
   | inr sum1_gt =>
     [(fst (sum_pair i q),
      existT _ (nqv (‘i) negb (fst (sum_pair i q))
       sum0_gt q) _);
      (snd (sum_pair i q),
      existT _ (nqv (‘i) (fun x => x)
       (snd (sum_pair i q)) sum1_gt q) _)]%list
   end
 end.
```

This function uses the sum_pair axioms and the nqv function. It takes the two probabilities (of measuring 0 and 1 for bit i) and computes the new quantum states that occur after this measurement. If either probability is zero, we cannot compute the new state, because this would involve a divition by zero. However, the fact that the probability is zero means that this situation will never occur. The sum_pair axioms are a way of externalising that knowledge.

4.2 Entanglement and EPR pairs

Measurement also allows us to think about entanglement. As mentioned earlier, definition of an entangled state is a state that cannot be broken down as the tensor product of smaller states. However, in constructive logics such as the one used by Coq, it is difficult to prove the absence of something—it would require proving that any breakdown in some way leads to a contradiction. Fortunately, we can use an alternative definition of entanglement which is specified in terms of measurement. Two qubits in a quantum state are entangled if measuring one qubit changes the state of the other; a quantum state space is entangled if it contains entangled qubits.

The advantage of this definition is that it is much easier to work with: we just need to show that two qubits are entangled. In Coq, this definition becomes:

```
Definition entangled_p n (q: qubit n)
 (p1: nat | p1 < n) (p2: nat | p2 < n) :=
  exists pr, exists res,
    List.In (pr, res) (measure p1 q) /\
         ~(distribution_equal (measure p2 res) (measure p2 q)).
```

This then allows us to define entanglement: a state is entangled if it contains entangled qubits:

```
Definition entangled n (q: qubit n) :=
  exists p1 p2, (‘p1) <> (‘p2) /\ entangled_p q p1 p2.
```

The distribution_equal function determines whether two probability distributions are equal.

The entangled_p definition, therefore, says that the probability of $p2$ being v in q must not be the same as the probability of $p2$ being v in res, where res is the result of measuring $p1$ in q.

As a sanity check, we can now prove that the 'maximally entangled' EPR pair $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is indeed entangled:

```
Definition epr_function (i: nat | 0 < i <= 4)
 (j: nat | 0 < j <= 1): C :=
  match ‘i with
  | 1 | 4 => onestwo
  | _     => [0]
  end.

Program Definition epr_1: qubit 2 :=
  (exist (fun x => vector_length x [=] [1])
  (create _ _ epr_function) _).

Lemma entangled: entangled epr_1.
```

Note how we specify a qubit: we simply give a vector with 4 elements and then explicitly note that this is an element of qubit 2. Coq automatically checks that the types are equivalent.

We can also prove that the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is not entangled:

```
Lemma entangled2: ~entangled ([cc_IR Half;
  cc_IR Half; cc_IR Half; cc_IR Half]: qubit 2).
```

5 Quantum Teleportation in Coq

The quantum teleportation procedure [3] allows the communication of the state of an unknown qubit, from one party to another, via a classical (*i.e.*, non-quantum) medium. According to the procedure, Alice has a qubit $\phi = a|0\rangle + b|1\rangle$ whose state she does not know and which she wishes to send to Bob. Alice and Bob each control one qubit of the maximally entangled EPR pair $\psi = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, i.e., we assume that the entangled EPR pair ψ is shared between Alice and Bob. The translation of this protocol into our Coq framework is relatively straightforward. We can use the definition of the maximally entangled EPR pair ψ from the previous section (epr_1). Alice's function is a sequence of transformations and measurements applied to the qubit-vector composed of ϕ and ψ [32] leaving Bob's third qubit unchanged (I-gate).

$$\text{Alice}_f \quad = \quad (H \otimes I \otimes I)(C_{not} \otimes I)(\phi \otimes \psi)\,.$$

Alice then measures the first two qubits, thereby affecting Bob's qubit via entanglement, and sends two classical bits x, y encoding the outcome of this measurement to Bob. Bob can now restore ϕ in his second qubit of the EPR-pair by simply applying the I, X, Z or Y gate to it depending on x and y. The details of the mathematical description [32] can be used one to one in the Coq definitions of teleportation below.

To compose the application of quantum gates, Alice employs the `{o}` operator which represents the tensor product. She needs the `c_not_gate` and the Hadamard transformations composed with identity transformations.

```
Definition firstgate: gate 3 :=  c_not_gate {o} identity.
Definition sndgate: gate 3 :=  hadamard {o} identity {o} identity.
```

The function `Alice` applies first the two gates `firstgate` and `secondgate` to the qubit triple `phi {o} epr_1` as defined in Alice$_f$ above. The first two qubits `p1` and `p2` of this application are then measured using the constructor `measure` (see Section 4). The result of function Alice lists the different outcomes of the measurement with their probabilities attached.

```
Definition Alice (phi: qubit 1): list (IR * qubit 3):=
measure p1 (measure p2 (apply (apply (phi {o} epr_1) firstgate) sndgate)).
```

An application of function `Alice` to a single qubit `phi` results in the following four possibilities; `alpha phi` and `beta phi` are the components a, b, respectively, of qubit $\phi = a|0\rangle + b|1\rangle$.

```
Definition Alice_pos (phi: qubit 1)(q: qubit 3):=
‘q = [alpha phi;beta phi;[0];[0];[0];[0];[0];[0]]
or
‘q = [[0];[0];beta phi;alpha phi;[0];[0];[0];[0]]
or
‘q = [[0];[0];[0];[0];alpha phi;[--](beta phi);[0];[0]]
or
‘q = [[0];[0];[0];[0];[0];[0];[--](beta phi);alpha phi]
```

The fact that the application of Alice's function yields precisely those four possibilities is encoded in the following theorem.

```
Theorem Alice_case: forall phi: qubit 1, forall q: qubit 3, forall r: IR,
                    List.In (r,q)(‘(Alice phi)) -> Alice_pos phi q.
```

For the transmission of the measurement of Alice's first two qubits we need classical bits encoded as an inductive type to enable subsequent pattern matching.

```
Inductive Bit : Set := | z | o.
```

Depending on the outcome of the measurement according to `Alice_pos`, we define the following function that encodes this measurement in a pair of bits. The constructors `inl` and `inr` allow to pattern match a disjunction in Coq.

```
Definition Alice_out(phi: qubit 1)(q: qubit 3)(qp: Alice_pos phi q): (Bit * Bit) :=
  match qp with
  | inl _ => (z,z)
  | inr(inl _ ) => (z,o)
  | inr(inr(inl  _)) => (o,z)
  | inr(inr(inr _)) => (o,o)
  end.
```

Bob operates on the transformed qubit triple `psix`, i.e. $\phi \otimes \psi$ with the first two bits collapsed and the third one—the one he controls—in an unknown state. But, using Coq's pattern matching, Bob can restore the original state of ϕ in his third qubit from the two classical bits he receives.

```
Definition Bob (pxy: qubit 3 * (Bit* Bit)): qubit 3 :=
  match pxy with
  | (psix,(z,z)) => apply psix (exist _ (‘identity {o} ‘identity {o} ‘identity) _)
  | (psix,(z,o)) => apply psix (exist _ (‘identity {o} ‘identity {o} ‘x_gate) _)
  | (psix,(o,z)) => apply psix (exist _ (‘identity {o} ‘identity {o} ‘z_gate) _)
  | (psix,(o,o)) => apply psix (exist _ (‘identity {o} ‘identity {o} ‘y_gate) _)
  end.
```

Now, to formalise that the protocol does actually achieve the teleportation of Alice's qubit ϕ, we can simply state that the combination of Alice's and Bob's functions results in a triple of qubits whose last element is the same as ϕ. Since Alice's function used ϕ as its first qubit, this statement then encodes that ϕ has been "teleported" from the first position to the third position, i.e., from Alice to Bob. The protocol composes Alice's function, given by the theorem `Alice_case`, followed by sending the classical bits using `Alice_out`, and finally applying Bob's decoding function. This combination can be applied to any of the possible qubits `q` in the result list `qp` that is returned by `Alice phi`. The first two elements measured and encoded in classical bits determine the qubit `q`. We represent these first two measured bits by an existentially quantified qubit pair `z`. The proof of teleportation consists of showing that, for each of the four possible outcomes for `q`, a suitable `z` exists.

```
Theorem teleportation:
forall phi: qubit 1, forall q: qubit 3,
forall pr: IR, forall qp: List.In (pr, q)‘(Alice phi),
 exists z: qubit 2,
   ‘Bob(q, Alice_out phi q (Alice_case phi q pr qp)) {=} ‘(z {o} phi).
```

6 Conclusions and Future Work

We have presented a framework for modelling and analysing quantum protocols using the proof assistant Coq. The framework makes it possible to represent and reason about all the essential features of quantum systems such as single and multiple qubits, quantum gates, measurement and entanglement. We have modelled and analysed the canonical example of Quantum Teleportation. As mentioned earlier, model-checking and equivalence checking techniques have been used for verification of quantum protocols. The work in this paper also opens up the possibility of combining theorem proving with these (more automatic) approaches to greater gain.

The framework uses existing Coq libraries as much as possible, though a new matrix library had to be developed. This library can also be used independently. Both the library and the framework make use of dependent types. This can make the proofs more complicated, but it allows for more concise formulations of key lemmas. We intend to push the use of dependent types still further during future development of the framework.

Future work will aim to look at simplifying some definitions, improving the specifications and enhancing the general readability of the Coq code. We will also investigate using linear typing as a way to incorporate the notion of non-cloneability of quantum states in our framework. Our Coq development is already quite substantial and contains most of the necessary features to analyse large scale examples. In the near future, we hope to apply it to various case studies ranging from Quantum Bit Commitment and Blind Quantum Computing to Quantum Error Correction Protocols. Of course, an ambitious and challenging project which we have in mind is to formalise and prove the security of Quantum Key Distribution using Coq.

Acknowledgements

We would like to thank Andrei Popescu for reading the paper carefully and providing valuable comments.

References

[1] Ebrahim Ardeshir-Larijani, Simon J. Gay & Rajagopal Nagarajan (2013): *Equivalence Checking of Quantum Protocols*. In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*, *Springer LNCS* 7795, pp. 478–492. Available at `http://dx.doi.org/10.1007/978-3-642-36742-7_33`.

[2] Ebrahim Ardeshir-Larijani, Simon J. Gay & Rajagopal Nagarajan (2014): *Verification of Concurrent Quantum Protocols by Equivalence Checking*. In: *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '14)*, *Springer LNCS* 8413, pp. 500–514. Available at `http://dx.doi.org/10.1007/978-3-642-54862-8_42`.

[3] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres & W K Wootters (1993): *Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels*. *Physical Review Letters* 70(13), doi:10.1103/PhysRevLett.70.1895.

[4] Charles H. Bennett & Gilles Brassard (1984): *An Update on Quantum Cryptography*. In G. R. Blakley & David Chaum, editors: *CRYPTO*, *Lecture Notes in Computer Science* 196, Springer, pp. 475–480. Available at `http://dx.doi.org/10.1007/3-540-39568-7_39`.

[5] Y. Bertot & P. Castéran (2004): *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science, Springer, doi:10.1007/978-3-662-07964-5.

[6] Cerberis: *Layer 2 Link Encryprion with Quantum Key Distribution*. Available at `http://www.idquantique.com/wordpress/wp-content/uploads/Cerberis-Datasheet.pdf`. Retrieved 27.9.2015.

[7] Adam Chlipala (2011): *Certified Programming with Dependent Types*. MIT Press. Available at `http://adam.chlipala.net/cpdt/`. `http://adam.chlipala.net/cpdt/`.

[8] Jacek Chrzaszcz (2003): *Implementing Modules in the Coq System*. In David A. Basin & Burkhart Wolff, editors: *TPHOLs*, *Lecture Notes in Computer Science* 2758, Springer, pp. 270–286. Available at `http://dx.doi.org/10.1007/10930755_18`.

[9] Th. Coquand & C. Paulin-Mohring (1990): *Inductively defined types*. In P. Martin-Löf & G. Mints, editors: *Proceedings of Colog'88*, *Lecture Notes in Computer Science* 417, Springer-Verlag. Available at `http://dx.doi.org/10.1007/3-540-52335-9_47`.

[10] Luís Cruz-Filipe, Herman Geuvers & Freek Wiedijk (2004): *C-CoRN, the Constructive Coq Repository at Nijmegen*. In Andrea Asperti, Grzegorz Bancerek & Andrzej Trybulec, editors: *MKM*, *Lecture Notes in Computer Science* 3119, Springer, pp. 88–103. Available at `http://dx.doi.org/10.1007/978-3-540-27818-4_7`.

[11] Timothy A. S. Davidson (2011): *Formal Verification Techniques using Quantum Process Calculus*. Ph.D. thesis, University of Warwick.

[12] Yuan Feng, Runyao Duan & Mingsheng Ying (2011): *Bisimulation for quantum processes*. In: *38th ACM Symposium on Principles of Programming Languages (POPL 2011)*, ACM, doi:10.1145/1926385.1926446.

[13] Yuan Feng, Nengkun Yu & Mingsheng Ying (2012): *Model checking quantum Markov chains*. arXiv:1205.2187 [quant-ph].

[14] S. J. Gay & R. Nagarajan (2005): *Communicating Quantum Processes*. In: *POPL'05*, ACM, doi:10.1145/1040305.1040318.

[15] Simon J. Gay & Rajagopal Nagarajan (2006): *Types and typechecking for Communicating Quantum Processes*. *Mathematical Structures in Computer Science* 16(3), pp. 375–406, doi:10.1017/S0960129506005263.

[16] Simon J. Gay, Nikolaos Papanikolaou & Rajagopal Nagarajan (2008): *QMC: a model-checker for quantum systems*. In: *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008), Springer LNCS* 5123, pp. 543–547. Available at `http://dx.doi.org/10.1007/978-3-540-70545-1_51`.

[17] Simon J. Gay, Nikolaos Papanikolaou & Rajagopal Nagarajan (2010): *Specification and verification of quantum protocols*. In: *Semantic Techniques in Quantum Computation*, Cambridge University Press.

[18] Georges Gonthier (2008): *Formal proof - the four color theorem*. *Notices of the American Mathematical Society* 55(11), pp. 1382–1393. Available at `http://dx.doi.org/10.1007/978-3-540-87827-8_28`.

[19] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi & Laurent Théry (2013): *A Machine-Checked Proof of the Odd Order Theorem*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *ITP*, *Lecture Notes in Computer Science* 7998, Springer, pp. 163–179. Available at `http://dx.doi.org/10.1007/978-3-642-39634-2`.

[20] William Howard (1980): *The formulae-as-types notion of construction*. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp. 479–490.

[21] INRIA (2013): *The Coq Proof Assistant*. Available at `http://coq.inria.fr/`. Accessed on 8.5.2013.

[22] Philippe Jorrand & Marie Lalire (2004): *Toward a Quantum Process Algebra*. In: *1st ACM Conference on Computing Frontiers*, doi:10.1145/977091.977108.

[23] Marie Lalire (2006): *Relations among quantum processes: bisimilarity and congruence*. *Mathematical Structures in Computer Science* 16(3), pp. 407–428, doi:10.1017/S096012950600524X.

[24] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Communications of the ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814. Available at `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`.

[25] MagiQ: *Quantum Key Distribution System (Q-Box)*. Available at `http://www.magiqtech.com/Products.html`. Retrieved 27.9.2015.

[26] D. Mayers (2001): *Unconditional Security in Quantum Cryptography*. *Journal of the ACM* 48(3), pp. 351–406, doi:10.1145/382780.382781.

[27] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I*. *Inf. Comput.* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90009-5.

[28] Michael A. Nielsen & Isaac L. Chuang (2000): *Quantum Computation and Quantum Information*. Cambridge University Press.

[29] M.E. Peck (2007): *Geneva Vote Will Use Quantum Cryptography*. *IEEE Spectrum*. Available at `http://spectrum.ieee.org/computing/networks/geneva-vote-will-use-quantum-cryptography`.

[30] Quantum-Coq (2014): *Coq sources for quantum protocol proofs*. Available at `https://www.dropbox.com/sh/fhldyg1mfnxdmcl/AACOT7tWxDrOnt4SZGshOKeua`. Shared Dropbox folder, read only – work in progress.

[31] H. de Riedmatten, I. Marcikic, W. Tittel, H. Zbinden, D. Collins & N. Gisin (2004): *Long distance quantum teleportation in a quantum relay configuration*. *Physical Review Letters* 92(4), p. 047904, doi:10.1103/PhysRevLett.92.047904.

[32] Eleanor G. Rieffel & Wolfgang Polak (2000): *An introduction to quantum computing for non-physicists*. *ACM Comput. Surv.* 32(3), pp. 300–335. Available at `http://doi.acm.org/10.1145/367701.367709`.

[33] Mingsheng Ying, Yuan Feng, Runyao Duan & Zhengfeng Ji (2009): *An algebra of quantum processes*. *ACM Transactions on Computational Logic* 10(3), p. 19, doi:10.1145/1507244.1507249.