Imperial College London

Department of Computing

Automatic Verification of Quantum Protocols with MCMAS

by

Pavel Gonzalez

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science / Artificial Intelligence of Imperial College London

September 2010

Abstract

We propose a technique for automatic verification of quantum information protocols via symbolic model checker for multi-agent systems, `MCMAS`. The method is based on logical framework [16] for investigating epistemic and temporal notions built on top of the model for Distributed Measurement-based Quantum Computation [13], an extension of the Measurement Calculus [15] to distributed quantum systems. We present the translation map from this formalism to the input language of the model checker and we implement `dmc2ispl`, a source-to-source compiler which does the translation automatically. We detail the `C++` implementation of the toolkit along with its modelling language. We apply the technique to verification of Quantum Teleportation, Quantum Key Distribution and Superdense Coding protocols. We discuss expressive power and efficiency of the technique.

Source Code: `http://www.doc.ic.ac.uk/~pg809/dmc2ispl.tar.gz`

Acknowledgements

First and foremost, I wish to thank my supervisor and personal tutor, Alessio Lomuscio, for all his help and guidance throughout the completion of this project.

I wish express my great thanks to Francesco Belardinelli. Without his suggestions and critical review this report would not have been possible.

I would also like to thank my good friends, Pavel for his artistic leanings and for borrowing me a pen, and Dasha for proofreading this report and for all the free beer she has provided me with over the years.

Lastly, I wish to thank my parents. There are not enough words to express how grateful I am for their unconditional love, support and patience.

Contents

Chapter 1

Introduction

> *Quantum computation is... a distinctively new way of harnessing nature... It will be the first technology that allows useful tasks to be performed in collaboration between parallel universes.*
>
> David Deutsch

Quantum computing has gained prominence in the last decade due to a huge potential in applications such as information processing, security and distributed systems. With this increase of activity, the need for validation of correctness of the algorithms has arisen. Model checking [18], and in particular the epistemic [20] approach, has been proved to be a successful technique for verification of classical distributed systems and security protocols. Recently, logics which can be used for reasoning about knowledge in the context of distributed quantum computation have been suggested, however formalisation for automatic verification of epistemic properties has not been developed yet.

The goal of this project is to bridge the gap and provide fully automatic verification of distributed quantum systems. The fundamental question from the epistemic point of view is how to model non-classical flow of quantum information. Is there such thing as "quantum knowledge", and if there is how can we express it? One of the proposed approaches, presented in [16], argues that "quantum knowledge" does not exist, and so it seems to be a well suited theoretical framework to use in combination with existing symbolic model checker for multi-agent systems, `MCMAS` [28].

The logic is based on the Distributed Measurement-based Quantum Computation (DMC) [13], which is a natural extension of the Measurement Calculus [15], a formal model for quantum computations. Measurement-based models provide an intriguing framework for thinking about these computations and physicists believe they lead to easier implementations [32]. The greatest advantage of the approach based on DMC is, though, that its underlying operational semantics is similar to the semantics of interpreted systems, around which `MCMAS` is built.

In this report, we describe a translation map from DMC to Interpreted Systems Programming Language (ISPL), which is a modular input language of the model checker. The idea is that, given a protocol specification in DMC, we can translate it into ISPL such that possible worlds, temporal accessibility relations and epistemic accessibility relations are preserved.

Initially, we have rewritten several quantum protocols by hand and successfully verified the epistemic properties of the protocols in `MCMAS`. But to make the whole verification process automatic, we have developed a source-to-source compiler, `dmc2ispl`, which reads a protocol specified in machine-readable adaptation of DMC together with a set of formulae to be verified and translates it into the corresponding ISPL code. The toolkit is implemented in `C++` and

exploits the fast lexical analyser generator `flex`, the general-purpose parser generator `GNU Bison` and the high-level interactive language for numerical computations `GNU Octave`.

This report has the following structure:

- Chapter 2 lays the foundation for this project by reviewing the three theoretical frameworks for investigating epistemic properties in the context of distributed quantum computation. We briefly introduce Qubit Message Passing Environments [39] and Dynamic Epistemic Quantum Logic [1, 2, 3, 4] before explaining Distributed Measurement Calculus [12, 13, 15, 16, 17] in greater detail.

- Chapter 3 presents model checking as a method for formal verification of the correctness of a system. First, we mention the fundamental techniques and then we concentrate on the theoretical background of `MCMAS` and describe its input language ISPL.

- Chapter 4 details our design of the toolkit. In the first part, we define the translation map between DMC and ISPL and disscus solutions to problems associated with the translation. In the second part, we introduce our adaptation of DMC which serves as the input language for the compiler.

- Chapter 5 outlines the implementation of `dmc2ispl`. We discuss the implementation language and justify our choices of the development tools. We then decompose the system into the three main modules respectively responsible for parsing the input file, analysing the reachable quantum state space, and generating the valid ISPL program.

- Chapter 6 methodically describes the evaluation of the final version of the compiler. We consider three quantum protocols: Quantum Teleportation [6], Quantum Key Distribution [19] and Superdense Coding [7]. For each protocol, we interpret the operational semantics of the underlying DMC network, investigate its epistemic properties, and finally verify these properties automatically using `dmc2ispl` and `MCMAS` toolkits. At the end of the chapter, we present and discuss results of several performance tests.

- Chapter 7 concludes this report. We summarise our overall contributions, present a qualitative evaluation of `dmc2ispl` compiler, and propose the directions for further development.

- Appendices provide some additional material. Appendix A introduces the fundamental concepts of quantum computing, appendix B revises interpreted systems and temporal epistemic logics, appendix C details the complete grammar for parsing our input language, and appendix D contains listings of translated ISPL files for the three protocols.

Chapter 2

Quantum Epistemic Logics

The aim of this chapter is to investigate three theoretical logical frameworks that can be used to reason about the knowledge in multi-agent systems in which quantum computation is performed.

We have found the approach based on the Distributed Measurement-Based Computation (DMC) semantics to be the most well-developed and the best suitable for implementation of this project. This framework is described in detail in section 2.3. We introduce the other two logics, Qubit Message Passing Environments and Dynamic Epistemic Quantum Logic, in section 2.1, respective 2.2. However, we include them for completeness of the discussion only as they are not essential in the development of the project itself. At the end of this chapter, we evaluate the approaches and justify why we have chosen DMC as the theoretical base for the project.

2.1 Qubit Message Passing Environments

A first attempt to define knowledge for quantum distributed systems can be found in [39]. A formal model of quantum message passing systems is developed and then the definition for semantics of a modal operator for knowledge in this model is considered.

Two different notions of knowledge are defined. First, an agent i can *classically know* a formula φ to hold, denoted $K_i^c\varphi$, in which case the possibility relation is based on equality of local classical states. Second, an agent can *quantumly know* a formula to hold, denoted $K_i^q\varphi$. For the latter, the possibility relation is based on equality of reduced density matrices, which embody what an agent, in principle, could determine from its local quantum state.

2.1.1 States

Decription of formal model of computational setting, in which the agents synchronously communicate by sending classical messages, by transmitting qubits and by operating on qubits, is considered in this section. An *environment*, denoted by $\mathcal{E}$, specifies the number n of agents in the system, possible classical and quantum states of the system, the actions that agents can perform, and the effect of these actions on the system.

The set of *global states* S of $\mathcal{E}$ is a cartesian product of the set of *quantum states* S^q and the set of *classical states* S^c. A global state s is denoted $\langle s^q, s^c\rangle$, where $s^q \in S^q$ is a quantum state in a N-qubit space $(\mathbb{C}^2)^{\otimes_N}$, where N is the number of qubits that agents can operate on, and $s^c \in S^c$ is a classical state $\langle$`var`,`loc`,`chan`,`res`$\rangle$, where classical bit assignment `var`, qubit location assignment `loc`, channel value assignment `chan`, and measurement result assigment `res` are defined as follows:

- Each agent i has a set of variables $\texttt{Var}_i$. A *classical bit assignment* is a function $\texttt{var}$ mapping each agent i to a function $\texttt{var}(i) : \texttt{Var}_i \mapsto \{0, 1\}$ assigning a truth value to each variable of agent i.

- Each qubit in the system is in the possession of some agent. A *qubit location assignment* in a N-qubit environment with n agents is a function $\texttt{loc} : [0, N] \mapsto [0, n]$ mapping each qubit to an agent possessing it. $\texttt{loc}^{-1}(i) = \{i_1, \ldots, i_k\}$ is then a set of indices of k qubits located at agent i.

- Between each pair of agents i, j, there is a single channel for communication from i to j. At each step of a computation, a single message may be transmitted along this channel. A *channel value assignment* is a function in a system with n agents $\texttt{chan} : [1 \ldots n]^2 \mapsto \texttt{Msg}$, where $\texttt{Msg}$ is a set of messages containing the special value $\perp$, which represents the fact that no message was sent.

- An agent is able to perform a measurement on its qubits. A measurement M^i on the qubits located at agent i produces some outcome m_i. A *measurement result assigment* is a function $\texttt{res}$ from agents to outcomes, such that $\texttt{res}(i) = (M^i, m_i)$

2.1.2 Actions

Agents are able to perform actions, which affect the global states of a qubit message passing environment.

- For each classical variable $v \in \texttt{Var}_i$, and boolean value $b \in \{0, 1\}$, the action "$v := b$" assigns $\texttt{Var}(i)(v)$ the value b. Additionally, a random assignment is represented by an action $\texttt{flip}(v)$.

- An agent i is able to transmit a qubit to agent j via the action $\texttt{transmit}(b, j)$, where b is an index of the qubit in $\texttt{loc}^{-1}(i)$. This action changes the value of $\texttt{loc}(b)$ to equal j.

- An agent i is able to transmit classical messages to agent j via the action $\texttt{send}_{i,j}(m)$, where $m \in \texttt{Msg}$. This action sets $\texttt{chan}(i, j) = m$ in the next state. Note that for all $j' \neq j : \texttt{chan}(i, j') = \perp$ in the next state and $\texttt{chan}(i, j) = \perp$ in the next state for all j if agent i does not send any message.

- An agent is able to perform measurements on the qubits in its possession. A measurement action has two effects. It transforms the quantum state s^q of the system because the measured qubit colapse to an element of the computational basis of $(\mathbb{C}^2)^{\otimes_N}$ in which it was measured, and sets $\texttt{res}(i) = (M^i, m_i)$.

A *joint action* in an environment $\mathcal{E}$ is a tuple $\langle a_1, \ldots, a_n \rangle$, where each a_i is an action of agent i. There is a *transition* from a global state s to a global state t if one of the possible combined effects of the actions is to transform the state s into the state t with some non-zero probability.

2.1.3 Protocols

A *run* is defined to be a function $r : \mathbb{N} \mapsto S$ mapping the natural numbers to the set of global states of a qubit passing environment $\mathcal{E}$. A *point* is a pair (r, m) consisting of a run r and a time m, with $r(m)$ reprsenting the global state of the system at that point. The information that agent i acquires about the environment is a function O_i mapping global states to some set of *observations* $\mathcal{O}_i$. The *perfect recall local state* of an agent i at a point (r, m),

denoted $r_i(m)$, is the sequence of observations made by the agent up to time m in r, i.e. $r_i(m) = O_i(r(0)), \ldots, O_i(r(m))$.

Agents engage in a particular patterns of behaviour, called *protocols*, which generate specific sets of runs. More formally, a protocol for an agent i is a function $P : \mathcal{O}_i^+ \mapsto Act_i$, where $\mathcal{O}_i^+$ is sequence of observations representing the information that the agent i has acquired along the computation steps, and Act_i is the next action of the agent. A *joint protocol* is a tuple $\mathbf{P} = \langle P_1, \ldots, P_n \rangle$, where each P_i is a protocol for agent i.

Run r is "a run of the joint protocol $\mathbf{P} = \langle P_1, \ldots, P_n \rangle$ in the environment $\mathcal{E}$" if for each time m, there exists a transition of $\mathcal{E}$ on the joint action $\langle P_1(r_1(m)), \ldots, P_n(r_n(m)) \rangle$ from global state $r(m)$ to global state $r(m+1)$. The set of all runs of the joint protocol $\mathbf{P}$ in $\mathcal{E}$ is denoted $\mathcal{R}(\mathcal{E}, \mathbf{P})$.

2.1.4 Temporal Epistemic Logic in Quantum Systems

The environment $\mathcal{E}$ is *interpreted* if it specifies an interpetation function $\pi : S \times \texttt{Prop} \mapsto \{0, 1\}$ for some set of propositions $\texttt{Prop}$ defined over the set of global states S. The grammar of the language is

$$\varphi ::= \; p \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i^c\varphi \mid K_i^q\varphi \mid \Box\varphi \mid \texttt{init}(\varphi),$$

where $p \in \texttt{Prop}$ is a basic proposition. The formula $K_i^c\varphi$ is read "agent i classically knows φ", the formula $K_i^q\varphi$ is read "agent i quantumly knows φ", the formula $\Box\varphi$ is read "at all times in the future, φ", and the formula $\texttt{init}(\varphi)$ is read "initially φ".

Formulas are evaluated with respect to triples $\mathcal{E}, \mathbf{P}, (r, m)$ where $\mathcal{E}$ is an interpreted qubit passing environment, $\mathbf{P}$ is a joint protocol and (r, m) is a point on a run $r \in \mathcal{R}(\mathcal{E}, \mathbf{P})$. Two equivalence relations on points for each agent i, denoted $\sim_i^c$ and $\sim_i^q$, capture different notions of two points being *indistinguishable* to the agent. The semantics of the above formulas is then given in the familiar pattern for the temporal epistemic logic:

$$\begin{aligned}
&\mathcal{E}, \mathbf{P}, (r, m) \vDash p \text{ if } \pi(r(m), p) = 1, \text{ when } p \in \texttt{Prop},\\
&\mathcal{E}, \mathbf{P}, (r, m) \vDash K_i^c\varphi \text{ if } \mathcal{E}, \mathbf{P}, (r', m') \vDash \varphi \; \forall (r', m') \in \mathcal{R}(\mathcal{E}, \mathbf{P}) : (r, m) \sim_i^c (r', m'),\\
&\mathcal{E}, \mathbf{P}, (r, m) \vDash K_i^q\varphi \text{ if } \mathcal{E}, \mathbf{P}, (r', m') \vDash \varphi \; \forall (r', m') \in \mathcal{R}(\mathcal{E}, \mathbf{P}) : (r, m) \sim_i^q (r', m'),\\
&\mathcal{E}, \mathbf{P}, (r, m) \vDash \Box\varphi \text{ if } \mathcal{E}, \mathbf{P}, (r, m') \vDash \varphi \; \forall m' \geq m,\\
&\mathcal{E}, \mathbf{P}, (r, m) \vDash \texttt{init}(\varphi) \text{ if } \mathcal{E}, \mathbf{P}, (r, 0) \vDash \varphi.
\end{aligned}$$

The definitions of the equivalence relations $\sim_i^c$ and $\sim_i^q$ arise by using different definitions of the observation function O_i in the notion of an agent's perfect recall local state $r_i(m)$.

The first notion of observation corresponds to what an agent is able to learn from the part of the classical state of the system that it is able to control. Given a global state s with the classical state s^c the observation of agent i is defined by $O_i^c(s) = \langle \texttt{var}(i), \texttt{loc}^{-1}(i), \texttt{chan}(i), \texttt{res}(i) \rangle$. This means that the agent is aware of the values of its classical bits, of the qubits it possesses, of what messages it has received, and of the outcomes of the measurements that it has performed. In this case the notion of $\sim_i^c$-indistinguishability is derived.

A different observation function O_i^q, which takes into account the possible measurements on the qubits possessed by an agent, is used for $\sim_i^q$-indistinguishability. To capture what information an agent is able to obtain from all measurements it is able to perform, a function d_{i,s^q} is defined as a map from the possible measurements on qubits $\texttt{loc}^{-1}(i)$ to their outcome distributions (i.e. the reduced density matrix of the quantum state of agent's qubits). Both the classical observation and the results of all possible quantum measurements are observable to the agent in this model, so $O_i^q(s)$ is defined as the pair $\langle O_i^c(s), d_{i,s^q} \rangle$. In other words, the two

points are $\sim_i^q$-indistinguishable if both the sequence of classically observed information obtained by agent i in the past, and the sequence of all possible information it could gather by quantum measurement are the same in the two points.

2.1.5 Quantum Key Distribution Protocol Example

To illustrate the definitions, a version of Quantum Key Distribution Protocol, called B92 [5], is formalised using the qubit message passing environment framework. The objective of the protocol is to establish a shared secret key between two parties, who can only communicate via a channel controlled by an eavesdropper. Only a fragment of the protocol, which shows how a single shared classical bit is established, is presented here for simplicity.

Initially agent **A** possesses a single qubit $|\psi\rangle$ and a classical bit a, agent **B** has two classical bits a' and b. The protocol proceeds as follows. First, **A** flips its classical bit a and then prepares the qubit in the state $|0\rangle$ if the outcome is 0 and in the state $|+\rangle$ if the outcome is 1. **A** then transmits the qubit to **B**. Upon receipt of the qubit, **B** flips its classical bit a' and if the outcome is 0, it measures the received qubit in the basis $|0\rangle$, $|1\rangle$, otherwise it measures in the basis $|+\rangle$, $|-\rangle$. If the measurement transforms the qubit to first element of the basis measured, **B** sets bit b to 0, otherwise it sets it to 1, and then sends a classical message to **A** stating the value of b. **A** and **B** reach agreement on the value of a only if $b = 1$ in the end of the protocol.

In order to model agent **E**'s ability to eavesdrop on this protocol, all messages and qubit transmissions are "doubled up". That is, **A** prepares two qubits in the same state, sending one to **B** and one to **E**. Similarly, **B** sends his classical message both to **A** and to **E**. Then the following can be shown

$$\mathcal{E}, \mathbf{P}, (r, 0) \vDash \Box(b = 1 \Rightarrow K_A^c(a) \wedge K_B^c(a)), \tag{2.1}$$

$$\mathcal{E}, \mathbf{P}, (r, 0) \vDash \Box(b = 1 \Rightarrow \neg K_E^c(a)), \tag{2.2}$$

$$\mathcal{E}, \mathbf{P}, (r, 0) \vDash \Box(b = 1 \Rightarrow K_E^q(a)), \tag{2.3}$$

where $K_i^x(a)$ is a shorthand for $K_i^x(a = 0) \vee K_i^x(a = 1)$, i.e. agent i knows the value of a. Then statement 2.1 expresses that in successful runs, agents **A** and **B** will classically know the value of bit a. The statement 2.2 means that agent **E** will never know the value of this bit, based on its classical observations alone. However, statement 2.3 says that agent **E** will quantumly know (i.e. if it was allowed repeatable measurements on the qubit) the value of bit a.

2.2 Dynamic Epistemic Quantum Logic

Dynamic Epistemic Quantum Logic was gradually developed in [1, 2, 3, 4] to model and interpret behaviour of quantum systems. The whole framework is very complex and contains many elements of non-classical quantum logic. This section gives a brief review needed for examination of epistemic properties of quantum system using this formalism.

2.2.1 Quantum Transition Systems

Axiomatization of the logic of quantum actions in terms of quantum transition systems [2], also called *quantum dynamic frames*, is a semantical abstraction of quantum systems, in the form of a relational structure, of the type known as "labeled transition systems". It takes as fundamental the notion of (quantum) state and represents quantum actions as relations between states.

2.2.1.1 Dynamic Frames

For a given binary relation $R \subseteq \Sigma \times \Sigma$ on a set Σ, and subsets $S, P \subseteq \Sigma$, the *image* of S is defined via R as $R(S) = \{t \in \Sigma : \exists s \in S(s,t) \in R\}$, and the *weakest precondition* of R with respect to *postcondition* P as $[R]P = \{s \in \Sigma : \forall t \in \Sigma((s,t) \in R \Rightarrow t \in P)\}$. If R is the input–output relation of a program then $[R]P$ captures the weakest condition that must be satisfied by the input-state, so that any output-state will satisfy condition P.

A *labeled transition system* is a relational structure $(\Sigma, \{\xrightarrow{a}\}_{a\in\mathcal{A}})$, consisting of a set Σ of *states* and a family of *transition relations* $\xrightarrow{a} \subseteq \Sigma \times \Sigma$ between states, relations labeled by a set $\mathcal{A}$ called *basic actions*.

Definition 2.1. *A* dynamic frame *is a labeled transition system* $\mathcal{F} = (\Sigma, \{\xrightarrow{P?}\}_{P\in\mathcal{L}}, \{\xrightarrow{U}\}_{U\in\mathcal{U}})$*, consisting of:*

1. *A set* Σ *of objects, called* states*.*
2. *A family of binary transition relations* $\xrightarrow{P?} \subseteq \Sigma \times \Sigma$*, labeled by test actions* $P?$*; the action labels come from a given family* $\mathcal{L} \subseteq \mathcal{P}(\Sigma)$ *of subsets* $P \subseteq \Sigma$*, called* testable properties*.*
3. *A family of binary transition relations* $\xrightarrow{U} \subseteq \Sigma \times \Sigma$*, labeled by basic actions* $U \in \mathcal{U}$*, called* basic unitary evolutions*.*

Any dynamic frame $\mathcal{F}$ can be equiped with a *measurement relation* $s \to t \Leftrightarrow s \xrightarrow{P?} t$ for some $P \in \mathcal{L}$, which means that state t can be obtained from state s by performing a measurement. The negation of this gives an *orthogonality relation* on states $s \perp t \Leftrightarrow s \not\to t$. *Orthocomplement* of the set S is defined by $\sim S := \{t \in \Sigma : t \perp s\ \forall s \in S\}$.

For example, Kripke frames for propositional dynamic logic are a special case of dynamic frames, in which $\mathcal{L} := \mathcal{P}(\Sigma)$, the transition relation for a test is $s \xrightarrow{P?} t \Leftrightarrow s = t \in P$, and the transitions $\xrightarrow{U}$ are arbitrarily chosen binary relations on Σ.

2.2.1.2 Quantum Frames

Definition 2.2. *A* quantum transition system *is a dynamic frame* $\mathcal{F}$*, satisfying the following list of conditions*

1. *Closure under* arbitrary conjunctions*: If* $\mathcal{L}' \subseteq \mathcal{L}$ *then* $\bigcap \mathcal{L}' \in \mathcal{L}$*.*
2. Atomicity*. States are testable, i.e.* $\{s\} \in \mathcal{L}$*.*
3. Adequacy*. Testing a true property does not change the state: if* $s \in P$ *then* $s \xrightarrow{P?} s$*.*
4. Repeatability*. Any property holds after it has been successfully tested: if* $s \xrightarrow{P?} t$ *then* $t \in P$*.*
5. Covering Law*: if* $s \xrightarrow{P?} w \neq t \in P$ *then* $\exists v \in P : t \to v \not\to s$*.*
6. Self-Adjointness Axiom*: if* $s \xrightarrow{P?} w \to t$ *then* $\exists v \in \Sigma : t \xrightarrow{P?} v \to s$*.*
7. Proper Superposition Axiom*. Every two states of a quantum system can be properly superposed into a new state:* $\forall s, t \in \Sigma,\ \exists w \in \Sigma : s \to w \to t$*.*
8. Reversibility and Totality Axioms*. Basic unitary evolutions are total bijective functions:* $\forall t \in \Sigma,\ \exists s \in \Sigma : s \xrightarrow{U} t$ *and* $\forall s \in \Sigma,\ \exists t \in \Sigma : s \xrightarrow{U} t$*.*

9. Orthogonality Preservation. *Basic unitary evolutions preserve (non) orthogonality: Let* $s, t, s', t' \in \Sigma$ *be such that* $s \xrightarrow{U} s'$ *and* $t \xrightarrow{U} t'$. *Then:* $s \to t$ *iff* $s' \to t'$.

10. Mayet's Condition*: Orthogonal Fixed Points. There exists some unitary evolution* $U \in \mathcal{U}$ *and some property* $P \in \mathcal{L}$*, such that* U *maps* P *into a proper subset of itself; and moreover the set of fixed-point states of* U *has dimension* ≥ 2.

Variables P, Q *range over testable properties in* $\mathcal{L}$*, variables* s, t, s', t', v, w *range over states in* Σ *and* U *ranges over evolutions.*

A Hilbert space $\mathcal{H}$ can be structured as a quantum dynamic frame $\mathcal{F}(\mathcal{H})$ by taking the family of all one-dimensional closed linear subspaces of $\mathcal{H}$, called "rays", as set of states Σ, the family of closed linear subspaces of $\mathcal{H}$ as class of testable properties $\mathcal{L}$, the maps induced on Σ by the projectors on the closed subspace as test actions $P?$, and the family of all unitary operators on $\mathcal{H}$ as the set of basic unitary actions $\mathcal{U}$. Any subframe of $\mathcal{F}(\mathcal{H})$ satisfying this conditions is called a *concrete quantum dynamic frame.*

2.2.1.3 Quantum Actions

Given a quantum dynamic frame $\mathcal{F}$, the class of quantum actions, also called *quantum programs*, over $\mathcal{F}$ is defined as the smallest family of binary relations $\mathcal{Q} \subseteq \mathcal{P}(\Sigma \times \Sigma)$ which contains all tests actions $\{\xrightarrow{P?}\}_{P \in \mathcal{L}}$, and all basic unitary evolutions $\{\xrightarrow{U}\}_{U \in \mathcal{U}}$ as well as their inverses $\xrightarrow{U^{-1}} := \xleftarrow{U}$, and is closed under the operations of relational composition $R \cdot R'$, defined by $(s,t) \in R \cdot R' \Leftrightarrow \exists w : (s,w) \in R \wedge (w,t) \in R'$, and arbitrary union of relations $\bigcup_{i \in I} R_i$, defined by $(s,t) \in \bigcup_{i \in I} R_i \Leftrightarrow \exists i \in I : (s,t) \in R_i$. The family of all quantum actions over $\mathcal{F}$ is denoted by $\mathcal{Q}(\mathcal{F})$.

Intuitively, relational composition represents sequential composition of actions (do first action π then action π'), while arbitrary union gives us nondeterministic choice (do either one of the actions $\{\pi_i\}_{i \in I}$). An action π, that can be expressed without the use of choice $\cup$ is called *deterministic*. As relations, such actions are partial functions, i.e. for a given input state s, they have at most one output state t, such that $s \xrightarrow{\pi} t$.

2.2.2 Logic of Quantum Programs

Ideas presented in the previous section were extended in [1] and [3] to a full-fledged dynamic Logic of Quantum Programs (LQP).

2.2.2.1 Syntax of LQP

Most importantly, spatial features were added to distinguish local properties of given subsystems of a quantum system, with I-local unitary actions of various types performed only on a subsystem. The syntax of LQP is an extension of PDL, with a set of propositional *formulas* and a set of *programs*, defined by following grammar:

$$\varphi ::= \top_I \mid p \mid c \mid \neg\varphi \mid \varphi \wedge \varphi \mid [\pi]\varphi$$
$$\pi ::= \top_I \mid \varphi? \mid U \mid \pi^\dagger \mid \pi \cup \pi \mid \pi;\pi \mid \pi^*$$

The sentence $\top_I$ expresses *I-separation*, which is true iff the qubits in I form a separated subsystem (i.e. are not entangled). p is a element of a set $\mathcal{Q}$ of propositional variables, c is a element of a set $\mathcal{C}$ of propositional constants. The constructs $\neg\varphi'$ and $\varphi \wedge \varphi$ denote classical

negation and conjunction, while the construct given by dynamic modalities $[\pi]\varphi$ denotes the *weakest precondition* that ensures that property φ will hold after running program π.

On the program side, $\top_I$ denotes the *trivial I-local action*, which acts on any given I-separated state. The meaning of quantum test $\varphi?$, adjoint $\pi^\dagger$, union $\pi \cup \pi$, composition $\pi;\pi$, and iteration π^* is given by the corresponding operations on quantum actions. U is a constant from a set $\mathcal{U}$ of basic programs, to be interpreted as quantum gates.

The extention of the basic language of LQP allowing for the measurement modalities and connectives of quantum logic, such as orthocomplement and quantum join, is not covered in this paper.

2.2.2.2 Semantics of LQP

An LQP-model is a multi-partite quantum frame $\Sigma = \Sigma(\mathcal{H})$ based on an n-dimensional Hilbert space $\mathcal{H}$, together with a valuation function, mapping each propositional variable p into a set of states $\|p\| \subseteq \Sigma$. The valuation map is used to give an interpretation $\|\varphi\| \subseteq \Sigma$ to all formulas in terms of quantum properties of the multi-partite frame, i.e. sets of states in Σ. In the same time, an interpretation $\|\pi\| \subseteq \Sigma \times \Sigma$ is given to all programs in terms of quantum actions. The interpretations of programs are defined by mutual recursion as follows:

$$\|\top_I\| := \top_I^{\Sigma\times\Sigma} \qquad \|\varphi?\| := \|\varphi\|? \qquad \|\pi_1 \cup \pi_2\| := \|\pi_1\| \cup \|\pi_2\|$$
$$\|U\| := U \qquad \|\pi^\dagger\| := \|\pi\|^\dagger \qquad \|\pi_1;\pi_2\| := \|\pi_2\|;\|\pi_1\|$$

The interpretation $\|\pi\|$ allows for extending the notation $\xrightarrow{\pi}$ to all programs, by putting: $s \xrightarrow{\pi} t$ iff $(s,t) \in \|\pi\|$. The valuation $\|p\|$ can be extended from propositional variables to all formulas by putting for the others:

$$\|1\| = |1\rangle^{\otimes_n} \qquad \|\top_I\| = \top_I^\Sigma \qquad \|\varphi \wedge \psi\| = \|\varphi\| \cap \|\psi\|$$
$$\| + \| = |+\rangle^{\otimes_n} \qquad \|\neg\varphi\| = \Sigma\backslash\|\varphi\| \qquad \|[\pi]\varphi\| = [\|\pi\|]\|\varphi\|$$

2.2.2.3 Proof Theory for LQP

A sound proof system for this logic is presented in [3], where the autors show how to characterize by logical means various forms of entanglement (e.g. the Bell states) and various quantum gates. The full system is very extensive and only the propositions, without proofs, needed for derivation of the teleportation protocol in the next section are included here. The notation $\overline{\pi}_{ij}$ means that states i and j are "entangled according to π".

Proposition 2.1. Teleportation Property. *If i, j, k are distinct indices then*

$$\vdash [\overline{\sigma_{jk}}?;\overline{\pi_{ij}}?](p_i) =_k [\pi_{ij};\sigma_{jk}](p_i)$$

Corollary. *If i, j, k are distinct then* $\vdash [\overline{\pi_{ij}}?](p_i \wedge \overline{\sigma_{jk}}) =_k [\pi_{ij};\sigma_{jk}](p_i)$.

Proposition 2.2. *The Bell states $\beta_{xy}^{i,j}$ are characterized by the logic Bell formulas $\beta_{xy}^{ij} := \overline{[Z_1^x;X_1^y]}_{ij}$, with $x,y \in \{0,1\}$ and distinct indices $i,j \in N$. In other words, a state satisfies one of these formulas iff it coincides with the corresponding Bell state.*

Proposition 2.3. *For all $x,y \in \{0,1\}$:* $\vdash [H_i;{}^CNOT_{i,j}](x_i \wedge y_j) = \beta_{xy}^{ij}$.

Corollary. *If i, j, k are all distinct then* $\vdash [{}^CNOT_{ij};H_i;(x_i \wedge y_j)?](p) =_k \beta_{xy}^{i,j}?(p)$.

2.2.3 Teleportation Protocol Example

In the syntax of LQP, the program for quantum teleportation protocol is:

$$\pi = \bigcup_{x,y\in\{0,1\}} {}^{C}NOT_{12}; H_1; (x_1 \wedge y_2)?; X_3^y; Z_3^x$$

and the validity expressing the correctness of the program is:

$$\vdash [\pi](q_1 \wedge \beta_{00}^{2,3}) =_3 id_{13}(q_1).$$

To show this is the case, the corollary of proposition 2.3, with $i = 1$, $j = 2$ and $k = 3$, is applied to program π:

$$\vdash [\bigcup_{x,y\in\{0,1\}} {}^{C}NOT_{12}; H_1; (x_1 \wedge y_2)?; X_3^y; Z_3^x](q_1 \wedge \beta_{00}^{2,3}) =_3 [\bigcup_{x,y\in\{0,1\}} \beta_{xy}^{1,2}?; X_3^y; Z_3^x](q_1 \wedge \beta_{00}^{2,3})$$

The following equivalent validity is obtain by replacing the logical Bell formulas with their definitions from proposition 2.2:

$$\vdash [\bigcup_{x,y\in\{0,1\}} \beta_{xy}^{1,2}?; X_3^y; Z_3^x](q_1 \wedge \beta_{00}^{2,3}) =_3 [\bigcup_{x,y\in\{0,1\}} \overline{[Z_1^x; X_1^y]_{12}}?; X_3^y; Z_3^x](q_1 \wedge \overline{[Z_1^0; X_1^0]}_{23})$$

This next validity follows from applying the corollary of Teleportation Property 2.1:

$$\vdash [\bigcup_{x,y\in\{0,1\}} \overline{[Z_1^x; X_1^y]_{12}}?; X_3^y; Z_3^x](q_1 \wedge \overline{[Z_1^0; X_1^0]}_{23}) =_3 [\bigcup_{x,y\in\{0,1\}} [Z_1^x; X_1^y]_{12}; [Z_1^0; X_1^0]_{23}; X_3^y; Z_3^x](q_1)$$

The last step follows from the fact that X^0, Z^0, $[X^y; X^y]$ and $[Y^x; Y^x]$ are all identities:

$$\vdash [\bigcup_{x,y\in\{0,1\}} [Z_1^x; X_1^y]_{12}; [Z_1^0; X_1^0]_{23}; X_3^y; Z_3^x](q_1) =_3 id_{13}(q_1).$$

2.2.4 Dynamic Epistemic Quantum Logic

The epistemic part of the logical framework for reasoning about quantum mechanical behaviour appeared in [4] and is based on the formal quantum-logical investigations from the previous sections. It provides an analysis of the dynamic-informational aspects of compound systems.

A compound system S consists of n subsystems $S_1, \ldots, S_n$. The quantum transition system includes the global actions U as well as i-local unitary actions U_i performed only on the respective subsystems. A constant symbol c is a special separated state $c = c_i \otimes c_j = (c_i, c_j) \in H_i \times H_j \subseteq H_i \times H_j$.

An equivalence relation $\simeq_i$ is defined on global states as follows: $s \simeq_i s'$ iff there exists a j-local, $j \neq i$, unitary action U_j such that $s' = U_j(s)$. Intuitively, this says that two possible global states s and s' of the compound system S are *indistinguishable* from the point of view of subsystem S_i iff s' can be obtained from s by performing a unitary action only on the environment S_j. The local state s_i of subsystem S_i in global state s can now be defined as the $\simeq_i$-equivalence class $s_i := \{s' : s \simeq_i s'\}$ of s. It is natural to introduce an "epistemic" operator K_i as the modality associated with the indistinguishability relation $\simeq_i$. The following property is defined for every property $\varphi \subseteq S$:

$$K_i\varphi := \{s : t \in \varphi \text{ for all } t \simeq_i s\} = \{s : s_i \subseteq \varphi\}.$$

K_i has the formal properties of a "knowledge" operator, satisfying the axioms of the modal system $S5$, however it does not represent a knowledge of some agent but rather a potential local information attainable in subsystem S_i. Thus, $K_i\varphi$ is read "the information φ is potentially available at location i".

In particular, for a separated state c, the *local state* of subsystem S_i does in fact correspond to a pure local state of S_i. A property φ is i-*local* (i.e. it is a property of the separated subsystem S_i) if it entails i-separation and it can only hold when φ is logically equivalent to $K_i\varphi$. Subsystems S_i and S_j are separated in a global state s iff there exists s' such that $s \simeq_j s' \simeq_i c$, otherwise the subsystems are entangled. An "epistemic" characterization of entanglement can be then obtained using the knowledge modal operators: S_i and S_j are entangled iff $K_2K_1\neg c$. In other words, two subsystems are entangled if they have some specific "knowledge" about each other prior to any communication.

2.3 Distributed Measurement Calculus

A technique of modelling knowledge in distributed quantum systems has been presented in [16] which is built on top of the semantics for Measurement Calculus developed in [15] and later extended to multi-agent systems in [13]. This approach has many advantages with regard to model checking and this section explains the ideas in more detail.

2.3.1 Measurement Calculus

The Measurement Calculus [15] provides a formalisation for one-way quantum computing, developed by Raussendorf and Briegel [36], which centres on 1-qubit measurements and a multi-party entanglement as the main ingredients of quantum computation. More precisely, a computation consists of a phase in which a collection of qubits is set up in a standard entangled state, then measurements are applied to individual qubits and finally local unitary operators, called corrections, are applied to some qubits. The phrase "one-way" is used to emphasize that the computation is driven by irreversible measurements.

In this section the computation pattern is defined, followed by its operational semantics, denotational semantics and universality. In the end, the rewrite theory is developed together with the standard form. For brevity some advanced issues, like the No Dependency Theorems and other approaches to measurement-based computation, are not included in this review.

2.3.1.1 Computation Patterns

The basic commands that can be used in a pattern are:

- 1-qubit auxiliary preparation N_i
- 2-qubit entanglement operators E_{ij}
- 1-qubit measurements M_i^α
- and 1-qubit Pauli operators corrections X_i and Z_i

The indices i, j represent the qubits on which each of these operations apply, and α is a parameter in $[0, 2\pi]$. Preparation N_i prepares qubit i in state $|+\rangle_i := \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, the entanglement commands are defined as controlled-Z gates ($E_{ij} := {}^CZ_{ij}$), the correction commands are the

Pauli gates X_i and Z_i and measurement M_i^α is defined by orthogonal projections on

$$|+_\alpha\rangle_i := \frac{1}{\sqrt{2}}(|0\rangle + e^{i\alpha}|1\rangle),$$
$$|-_\alpha\rangle_i := \frac{1}{\sqrt{2}}(|0\rangle - e^{i\alpha}|1\rangle)$$

followed by a trace-out operator. The parameter $\alpha \in [0, 2\pi]$ is called the *angle* of the measurement. The *outcome* of a measurement done at qubit i will be denoted by $s_i \in \mathbb{Z}_2$ and $s_i = 0$ if under the corresponding measurement the state collapses to $|+_\alpha\rangle_i$, and $s_i = 1$ if to $|-_\alpha\rangle_i$. Outcomes can be summed together resulting in expressions of the form $s = \sum_{i\in I} s_i$ which is called *signals*, and where the summation is being done in $\mathbb{Z}_2$.

Corrections and measurements may depend on signals. Dependent corrections will be written X_i^s and Z_i^s and dependent measurements will be written ${}^t[M_i^\alpha]^s$, where $s, t \in \mathbb{Z}_2$ and $\alpha \in [0, 2\pi]$. The meaning of dependencies for corrections is straightforward: $X_i^0 = Z_i^0 = I$, no correction is applied, while $X_i^1 = X_i$ and $Z_i^1 = Z_i$. In the case of dependent measurements, the measurement angle will depend on s, t and α as follows:

$${}^t[M_i^\alpha]^s := M_i^{(-1)^s\alpha + t\pi} \tag{2.4}$$

so that, depending on the parities of s and t, one may have to modify the α to one of $-\alpha$, $\alpha + \pi$ and $-\alpha + \pi$. These modifications correspond to conjugations of measurements under X and Z:

$$\begin{aligned} X_i M_i^\alpha X_i &= M_i^{-\alpha}, \\ Z_i M_i^\alpha Z_i &= M_i^{\alpha+\pi} \end{aligned} \tag{2.5}$$

This completes the catalog of basic commands, including dependent ones, and now measurement patterns can be defined [15].

Definition 2.3. *A pattern $\mathcal{P} = (V, I, O, \mathcal{A})$ consists of three finite sets V, I, O, together with two injective maps $i : I \to V$ and $o : O \to V$ and a finite sequence $\mathcal{A}$ of commands $A_n \dots A_1$, read from right to left, applying to qubits in V in that order, i.e. A_1 first and A_n last, such that:*

(D0) *no command depends on an outcome not yet measured;*

(D1) *no command acts on a qubit already measured;*

(D2) *no command acts on a qubit not yet prepared, unless it is an input qubit;*

(D3) *a qubit i is measured if and only if i is not an output.*

The set V is called the pattern *computation space*, and $\mathfrak{H}_V$ is the associated quantum state space $(\mathbb{C}^2)^{\otimes_{i\in V}}$. The sets I, O are called respectively the pattern *inputs* and *outputs*, and $\mathfrak{H}_I$ and $\mathfrak{H}_O$ are the associated quantum state spaces. The sequence $A_n \dots A_1$ is called the pattern *command sequence*, while the triple (V, I, O) is called the pattern *type*.

For example, a pattern implementing the Hadamard gate $\mathcal{H}$:

$$\mathcal{H} := (\{1, 2\}, \{1\}, \{2\}, X_2^{s_1} M_1^0 E_{12} N_2)$$

with computation space $\{1, 2\}$, inputs $\{1\}$, and outputs $\{2\}$. To run $\mathcal{H}$, the first qubit is prepared in some input state $|\psi\rangle$, and the second qubit in state $|+\rangle$, then these are entangled, then the first qubit is measured in the $|+\rangle$, $|-\rangle$ basis and finally an X correction is applied on the output qubit, if the measurement outcome was $s_1 = 1$.

Composition and tensoring are two ways to combine patterns in order to obtain larger ones. Two patterns $\mathcal{P}_1$ and $\mathcal{P}_2$ may be composed if $V_1 \cap V_2 = O_1 = I_2$. Provided that $\mathcal{P}_1$ has as many outputs as $\mathcal{P}_2$ has inputs, by renaming the pattern qubits, the patterns are always composable.

Definition 2.4. *The composite pattern $\mathcal{P}_2 \circ \mathcal{P}_1$ is defined as: $V := V_1 \cup V_2, I = I_1,$ and $O = O_2$, commands are concatenated.*

Two patterns $\mathcal{P}_1$ and $\mathcal{P}_2$ may be tensored if $V_1 \cap V_2 = \emptyset$. Again this condition can always be met by renaming qubits in a way that these sets are made disjoint.

Definition 2.5. *The tensor pattern $\mathcal{P}_1 \otimes \mathcal{P}_2$ is defined as: $V = V_1 \cup V_2, I = I_1 \cup I_2,$ and $O = O_1 \cup O_2$, commands are concatenated.*

To rename qubits of a pattern, the following concrete notation is used: if $\mathcal{P}$ is a pattern over $\{1, \ldots, n\}$, and f is an injection, $\mathcal{P}(f(1), \ldots, f(n))$ stands for the same pattern with qubits renamed according to f.

2.3.1.2 Operational Semantics

A formal operational semantics for the pattern language is given as a probabilistic labeled transition system [15]. To record the outcomes of the successive measurements performed in a pattern a classical state is required. It is natural to define the computation state space as:

$$S := \sum_{V,W} \mathfrak{H}_V \times \mathbb{Z}_2^W,$$

where V is the finite set of qubits that are still active (i.e. not yet measured) and W is the set of qubits that have been measured (i.e. they are now just classical bits recording the measurement outcomes). The computation states form a V, W-indexed pairs of q and Γ, where q is a quantum state from $\mathfrak{H}_V$ and Γ is an *outcome map* from some W to the outcome space $\mathbb{Z}_2$.

For any signal s and classical state $\Gamma \in \mathbb{Z}_2^W$, such that the domain of s is included in W, we take s_Γ to be the value of s given by the outcome map Γ. That is to say, if $s = \sum_I s_i$, then $s_\Gamma := \sum_I \Gamma(i)$ where the sum is taken in $\mathbb{Z}_2$. Also if $\Gamma \in \mathbb{Z}_2^W$, and $x \in \mathbb{Z}_2$, a map in $\mathbb{Z}_2^{W \cup \{i\}}$ is defined:

$$\Gamma[x/i](i) = x,\ \Gamma[x/i](j) = \Gamma(j) \text{ for } j \neq i.$$

Each command can be now viewed as acting on the state space S, V and W are suppressed in the first four commands [15]:

$$\begin{array}{lll}
q, \Gamma & \xrightarrow{N_i} & q \otimes |+\rangle_i, \Gamma, \\
q, \Gamma & \xrightarrow{E_{ij}} & {}^C Z_{ij} q, \Gamma, \\
q, \Gamma & \xrightarrow{X_i^s} & X_i^{s_\Gamma} q, \Gamma, \\
q, \Gamma & \xrightarrow{Z_i^s} & Z_i^{s_\Gamma} q, \Gamma, \\
V \cup \{i\}, W, q, \Gamma & \xrightarrow{{}^t[M_i^\alpha]^s} & V, W \cup \{i\}, \langle +_{\alpha_\Gamma}|_i \, q, \Gamma[0/i], \\
V \cup \{i\}, W, q, \Gamma & \xrightarrow{{}^t[M_i^\alpha]^s} & V, W \cup \{i\}, \langle -_{\alpha_\Gamma}|_i \, q, \Gamma[1/i],
\end{array}$$

where $\alpha_\Gamma = (-1)^{s_\Gamma}\alpha + t_\Gamma \pi$ follows equation 2.4. All commands except measurements are deterministic and only modify the quantum part of the state. The measurement actions on S are not deterministic, so that these are actually binary relations on S, and modify both the quantum and classical parts of the state.

An additional (classical) command called *signal shifting* performs shifting of the measurement outcome at i by the amount s_Γ:

$$q, \Gamma \xrightarrow{S_i^s} q, \Gamma[\Gamma(i) + s_\Gamma/i]$$

2.3.1.3 Denotational Semantics

The execution of a pattern $\mathcal{P}$ starts with some input state q in $\mathfrak{H}_I$, together with the empty outcome map $\emptyset$. The input state q is then tensored with as many $|+\rangle$ as there are noninputs in V, so as to obtain a state in the full space $\mathfrak{H}_V$. Then E, M and C commands in $\mathcal{P}$ are applied in sequence from right to left. The situation can be summarised as follows:

$$\begin{array}{ccccc} \mathfrak{H}_I & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots & & & \mathfrak{H}_O \\ \downarrow & & & & \uparrow \\ \mathfrak{H}_I \times \mathbb{Z}_2^{\emptyset} & \xrightarrow{prep} & \mathfrak{H}_V \times \mathbb{Z}_2^{\emptyset} & \xrightarrow{A_1 \ldots A_n} & \mathfrak{H}_O \times \mathbb{Z}_2^{V \setminus O} \end{array}$$

If m is the number of measurements then the run may follow 2^m different branches described by a unique binary string $\mathbf{s}$ of length m, representing the classical outcomes of the measurements, and a unique *branch map* $A_{\mathbf{s}}$ representing the linear transformation from $\mathfrak{H}_I$ to $\mathfrak{H}_O$. This map is proven [15] to be a completely positive trace preserving map and it is obtained from the operational semantics via the sequence (q_i, Γ_i) with $1 \leq i \leq n+1$, such that:

$$\begin{array}{l} q_1, \Gamma_1 = q \otimes |+\ldots+\rangle, \emptyset \\ q_{n+1} = q' \neq 0 \\ \text{and for all } i \leq n : q_i, \Gamma_i \xrightarrow{A_i} q_{i+1}, \Gamma_{i+1}. \end{array}$$

Definition 2.6. *A pattern is said to be* deterministic *if it realizes a trace-preserving completely-positive map that sends pure states to pure states. A pattern is said to be* strongly deterministic *when branch maps are equal.*

This is equivalent to saying that for all $q \in \mathfrak{H}_I$ and all $\mathbf{s_1}$, $\mathbf{s_2} \in \mathbb{Z}_2^n$, $A_{\mathbf{s}_1}(q)$ and $A_{\mathbf{s}_2}(q)$ differ only up to a scalar, for a strongly deterministic pattern $A_{\mathbf{s}_1}(q) = A_{\mathbf{s}_2}(q)$. For example $\mathcal{P} := (\{1,2\}, \{1\}, \{1\}, M_2^{\alpha})$, which implements the identity, is a deterministic pattern:

$$q \otimes |+\rangle, \emptyset \xrightarrow{M_2^{\alpha}} \begin{cases} \frac{1}{2}(1 + e^{-i\alpha})q, \emptyset[0/2], \\ \frac{1}{2}(1 - e^{-i\alpha})q, \emptyset[1/2] \end{cases}$$

but it is not strongly deterministic since the branches have respective probabilities $\frac{1}{2}(1+\cos\alpha)$ and $\frac{1}{2}(1-\cos\alpha)$, which are not generally equal. A pattern $\mathcal{P} := (\{1,2\}, \{1\}, \{2\}, M_1^0 E_{12})$ starting with input $q = a|0\rangle + b|1\rangle$ is nondeterministic:

$$\begin{array}{rl} (a|0\rangle + b|1\rangle) \otimes |+\rangle, \emptyset & \xrightarrow{E_{12}} \frac{1}{\sqrt{2}}(a|00\rangle + a|01\rangle + b|10\rangle - b|11\rangle), \emptyset \\ & \xrightarrow{M_1^0} \begin{cases} \frac{1}{2}((a+b)|0\rangle + (a-b)|1\rangle, \emptyset[0/1], \\ \frac{1}{2}((a-b)|0\rangle + (a+b)|1\rangle, \emptyset[1/1]. \end{cases} \end{array}$$

Both transitions happen with equal probabilities $\frac{1}{2}$ but, in general, the outputs in each branch are in different state. However, if correction X_2 is applied on one of the branches, both outputs would be the same with the same probabilities and the pattern would be strongly deterministic. In fact, correction applied on the second branch is the pattern $\mathcal{H}$, which realizes Hadamard gate.

2.3.1.4 Universality

It has been shown [14] that a universal set for $(\mathbb{C}^2)^{\otimes n}$ is form by unitaries $J(\alpha)$ and ${}^{C}Z$, defined:

$$J(\alpha) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & e^{i\alpha} \\ 1 & -e^{i\alpha} \end{bmatrix} \text{ and } {}^{C}Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

These universal generators can be realised as patterns $\mathcal{J}(\alpha) = (\{1,2\}, \{1\}, \{2\}, X_2^{s_1} M_1^{-\alpha} E_{12})$ and ${}^{C}\mathcal{Z} = (\{1,2\}, \{1,2\}, \{1,2\}, E_{12})$, and therefore combining the corresponding patterns will generate patterns realizing any unitary in $(\mathbb{C}^2)^{\otimes n}$.

2.3.1.5 Standardization

Definition 2.7. *Two patterns $\mathcal{P}$ and $\mathcal{P}'$ are equivalent if and only if for any branch $s \in \{0,1\}^m$, $A_s^{\mathcal{P}} = A_s^{\mathcal{P}'}$, where $A_s^{\mathcal{P}}$ and $A_s^{\mathcal{P}'}$ are the branch map A_s defined in Section 2.3.1.3.*

The first set of equations gives the means to propagate local Pauli corrections through the entangling operator E_{ij}:

$$\begin{aligned} E_{ij} X_i^s &= X_i^s Z_j^s E_{ij}, & E_{ij} Z_i^s &= Z_i^s E_{ij}, \\ E_{ij} X_j^s &= X_j^s Z_i^s E_{ij}, & E_{ij} Z_j^s &= Z_j^s E_{ij}. \end{aligned}$$

The second set of equations allows for pushing corrections through measurements acting on the same qubit. There are two cases:

$$\begin{aligned} {}^{t}[M_i^\alpha]^s X_i^r &= {}^{t}[M_i^\alpha]^{s+r}, \\ {}^{t}[M_i^\alpha]^s Z_i^r &= {}^{t+r}[M_i^\alpha]^s. \end{aligned}$$

A set of rewrite rules is obtained by orienting the equations above:

$$\begin{array}{lcll} E_{ij} X_i^s & \Rightarrow & X_i^s Z_j^s E_{ij} & EX, \\ E_{ij} X_j^s & \Rightarrow & X_j^s Z_i^s E_{ij} & EX, \\ E_{ij} Z_i^s & \Rightarrow & Z_i^s E_{ij} & EZ, \\ E_{ij} Z_j^s & \Rightarrow & Z_j^s E_{ij} & EZ, \\ {}^{t}[M_i^\alpha]^s X_i^r & \Rightarrow & {}^{t}[M_i^\alpha]^{s+r} & MX, \\ {}^{t}[M_i^\alpha]^s Z_i^r & \Rightarrow & {}^{t+r}[M_i^\alpha]^s & MZ. \end{array}$$

Free commutation rules, obtained when commands operate on disjoint sets of qubits, are defined:

$$\begin{array}{lcll} E_{ij} A_{\vec{k}} & \Rightarrow & A_{\vec{k}} E_{ij} & \text{where } A \text{ is not an entanglement,} \\ A_{\vec{k}} X_i^s & \Rightarrow & X_i^s A_{\vec{k}} & \text{where } A \text{ is not a correction,} \\ A_{\vec{k}} Z_i^s & \Rightarrow & Z_i^s A_{\vec{k}} & \text{where } A \text{ is not a correction,} \end{array}$$

where $\vec{k}$ represents the qubits acted upon by command A, which are supposed to be distinct from i and j.

Write $\mathcal{P} \Rightarrow \mathcal{P}'$, respectively $\mathcal{P} \Rightarrow^* \mathcal{P}'$, if both patterns have the same type, and the command sequence of $\mathcal{P}'$ can be obtained from the command sequence of $\mathcal{P}$ by applying one, respectively any number, of the rewrite rules. Pattern $\mathcal{P}$ is *standard* if for no $\mathcal{P}'$, $\mathcal{P} \Rightarrow \mathcal{P}'$ and the procedure of writing a pattern to standard form is called *standardization*.

Definition 2.8. *A pattern has a* NEMC *form if its commands occur in the order of Ns first, then Es , then Ms, and finally Cs.*

Usually just "EMC" form is used since it is assumed that all the auxiliary qubits are prepared in the $|+\rangle$ state and N commands are omitted in a command sequence. It has been proved in [15] that for all $\mathcal{P}$, there exists a unique standard $\mathcal{P}'$, such that $\mathcal{P} \Rightarrow^* \mathcal{P}'$, and $\mathcal{P}'$ is in EMC form.

The calculus can be extended to include the signal shifting command S_i^t. This allows for disposal of dependencies induced by the Z-action (see equation 2.5), which may sometimes lead to standard patterns with smaller computational depth complexity.

$$\begin{array}{lcl}
{}^t[M_i^\alpha]^s & \Rightarrow & S_i^t[M_i^\alpha]^s \\
X_j^s S_i^t & \Rightarrow & S_i^t X_j^{s[t+s_i/s_i]} \\
Z_j^s S_i^t & \Rightarrow & S_i^t Z_j^{s[t+s_i/s_i]} \\
{}^t[M_i^\alpha]^s S_i^r & \Rightarrow & S_i^r\ {}^{t[r+s_i/s_i]}[M_j^\alpha]^{s[r+s_i/s_i]} \\
S_i^s S_j^t & \Rightarrow & S_j^t S_i^{s[t+s_j/s_j]}
\end{array}$$

where $s[t/s_i]$ denotes the substitution of s_i with t in s, s, t being signals.

We now illustrate standardization on quantum teleportation. Consider the composite pattern $\mathcal{J}(\beta)(2,3) \circ \mathcal{J}(\alpha)(1,2) := (\{1,2,3\},\{1\},\{3\}, X_3^{s_2} M_2^{-\beta} E_{23} X_2^{s_1} M_1^{-\alpha} E_{12})$. The standardization procedure to obtain an equivalent standard pattern is as follows:

$$\begin{array}{lll}
\mathcal{J}(\beta)(2,3) \circ \mathcal{J}(\alpha)(1,2) & = & X_3^{s_2} M_2^{-\beta} \boxed{E_{23} X_2^{s_1}} M_1^{-\alpha} E_{12} \\
& \Rightarrow_{EX} & X_3^{s_2} \boxed{M_2^{-\beta} X_2^{s_1}} Z_3^{s_1} M_1^{-\alpha} E_{23} E_{12} \\
& \Rightarrow_{MX} & X_3^{s_2} Z_3^{s_1} [M_2^{-\beta}]^{s_1} M_1^{-\alpha} E_{23} E_{12}
\end{array}$$

If $\alpha = \beta = 0$, then $\mathcal{J}(0)(2,3) \circ \mathcal{J}(0)(1,2) = X_3^{s_2} Z_3^{s_1} M_2^x M_1^x E_{23} E_{12}$ and since $\mathcal{J}(0)$ implements H and $H^2 = I$, this pattern implements the identity, or in other words it teleports qubit 1 to qubit 3.

2.3.2 Quantum Networks

A formal model for distributed measurement-based quantum computations adopting an agent-based view was developed in [13] as a natural extention of the measurement calculus. Because of the inherently distributed aspect, measurement-based model for quantum computation is well-suited as a starting point for a formal model for distributed quantum computations. The system is described as a set of agents communicating synchronously and operating on a globally entangled quantum state. The operational semantics for systems of agents is given by a probabilistic transition system.

2.3.2.1 Networks of Agents

Agents are localized processes which, executing concurrently, make up a distributed system.

Definition 2.9. *An agent* $\mathbf{A(i, o)} : Q.\mathcal{E}$*, with classical input* $\mathbf{i}$ *and output* $\mathbf{o}$*, and sort given by a set of qubit references* Q*, is defined by a finite event sequence* $\mathcal{E}$ *composed of*

1. *patterns command sequences* $\mathcal{A}$*, with input qubits in* Q*;*
2. *classical message reception* $\mathrm{c}?x$ *and sending* $\mathrm{c}!y$*, where* c *is a classical channel, and* x *and* y *are names;*

3. *qubit reception* qc?x *and sending* qc!q*, where* qc *is a quantum channel and* q *a qubit reference.*

An agent's state *is given by a classical environment* Γ*, which is a partial mapping from names, i.e. classical variables and qubit references, to values.*

An expression of the form $\mathbf{A}(\mathbf{i},\mathbf{o}) : Q.\mathcal{E}$ is read "the agent with *name* $\mathbf{A}$ runs the *program* $\mathcal{E}$ with qubits, inputs and outputs as specified". Any pattern $\mathcal{P}(V,I,O,\mathcal{A})$ trivially corresponds to agent $\mathbf{A} : I.\mathcal{A}$ and, in general, the sort Q equals $I \uplus I^s$, where I is the local quantum input and I^s are qubits of a shared entangled state. Agents with the same agent names can be composed together. The composition of agents $\mathbf{A}(\mathbf{i}_1,\mathbf{o}_1) : Q_1.\mathcal{E}_1$ and $\mathbf{A}(\mathbf{i}_2,\mathbf{o}_2) : Q_2.\mathcal{E}_2$ is denoted $\mathbf{A}[(\mathbf{i}_2,\mathbf{o}_2) : Q_2.\mathcal{E}_2] \circ [(\mathbf{i}_1,\mathbf{o}_1) : Q_1.\mathcal{E}_1]$ and given by $\mathbf{A}(\mathbf{i},\mathbf{o}) : Q.\mathcal{E}_2\mathcal{E}_1$ with $\mathbf{i} = \mathbf{i}_1 \cup (\mathbf{i}_2 \backslash \mathbf{o}_1)$, $\mathbf{o} = \mathbf{o}_1 \cup \mathbf{o}_2$ and $Q = Q_1 \cup (Q_2 \backslash Q_1')$, where Q_1' is the output sort of the first agent.

A network of agents consists of several agents executing their event sequence concurrently, together with a global shared entangled state [13].

Definition 2.10. *A network of agents* $\mathcal{N}$ *is defined by a set of concurrently acting agents together with a shared quantum state, that is*

$$\begin{aligned}\mathcal{N} &= \mathbf{A}_1(\mathbf{i}_1,\mathbf{o}_1) : Q_1.\mathcal{E}_1 \mid \ldots \mid \mathbf{A}_m(\mathbf{i}_m,\mathbf{o}_m) : Q_m.\mathcal{E}_m \parallel \sigma \\ &= |_i\ \mathbf{A}_i(\mathbf{i}_i,\mathbf{o}_i) : Q_i.\mathcal{E}_i \parallel \sigma,\end{aligned}$$

where $\sigma \in \mathcal{D}(\mathcal{H}_{\uplus_i I_i^s})$*, with* $Q_i = I_i \cup I_i^s$ *for all* i*.*

Individual agents $\mathbf{A}(\mathbf{i},\mathbf{o}) : Q.\mathcal{E}$ trivially correspond to a network $\mathbf{A}(\mathbf{i},\mathbf{o}) : Q.\mathcal{E} \parallel \mathbf{0}$. The network state σ is the initial entanglement resource. Local quantum inputs specified in I_i are added to the network state σ during initialization.

There are two different ways of composing networks of agents. The sequential composition of networks $\mathcal{N}_1 = |_{i=1}^m\ \mathbf{A}_i(\mathbf{i}_{1,i},\mathbf{o}_{1,i}) : Q_{1,i}.\mathcal{E}_{1,i} \parallel \sigma_1$ and $\mathcal{N}_2 = |_{i=1}^m\ \mathbf{A}_i(\mathbf{i}_{2,i},\mathbf{o}_{2,i}) : Q_{2,i}.\mathcal{E}_{2,i} \parallel \sigma_2$ is defined as

$$\mathcal{N}_2 \circ \mathcal{N}_1 = |_{i=1}^m\ \mathbf{A}_i[(\mathbf{i}_{2,i},\mathbf{o}_{2,i}) : Q_{2,i}.\mathcal{E}_{2,i}] \circ [(\mathbf{i}_{1,i},\mathbf{o}_{1,i}) : Q_{1,i}.\mathcal{E}_{1,i}] \parallel \sigma_1 \otimes \sigma_2.$$

Sequential composition is only defined for networks containing the same agents, where agents carry out event sequences of both networks one after the other. On the other hand, parallel composition, where agents operate in parallel and independently of each other, is only defined for networks containing different agents. The parallel composition of networks $\mathcal{N}_1 = |_{i=1}^m\ \mathbf{A}_i(\mathbf{i}_{1,i},\mathbf{o}_{1,i}) : Q_{1,i}.\mathcal{E}_{1,i} \parallel \sigma_1$ and $\mathcal{N}_2 = |_{i=1}^n\ \mathbf{B}_i(\mathbf{i}_{2,i},\mathbf{o}_{2,i}) : Q_{2,i}.\mathcal{E}_{2,i} \parallel \sigma_2$ is defined as

$$\mathcal{N}_1 \otimes \mathcal{N}_2 = |_{i=1}^m\ \mathbf{A}_i(\mathbf{i}_{1,i},\mathbf{o}_{1,i}) : Q_{1,i}.\mathcal{E}_{1,i}\ |_{i=1}^n\ \mathbf{B}_i(\mathbf{i}_{2,i},\mathbf{o}_{2,i}) : Q_{2,i}.\mathcal{E}_{2,i} \parallel \sigma_1 \otimes \sigma_2.$$

The above definitions can be summarised by an abstract grammar. Alternatives are separated by [] instead of |, as the latter denotes parallel composition.

$$\begin{aligned}\mathcal{A} &::= \text{nil} \ [] \ E \ [] \ M \ [] \ C \ [] \ \mathcal{A} \otimes \mathcal{A} \ [] \ \mathcal{A}.\mathcal{A} \\ \mathcal{E} &::= \text{c?}x \ [] \ \text{c!}x \ [] \ \text{qc?}q \ [] \ \text{qc!}q \ [] \ \mathcal{A} \ [] \ \mathcal{E}.\mathcal{E} \\ \mathbf{a} &::= \mathbf{A}(\mathbf{i},\mathbf{o}) : Q.\mathcal{E} \ [] \ \mathbf{A}[(\mathbf{i}_2,\mathbf{o}_2) : Q_2.\mathcal{E}_2] \circ [(\mathbf{i}_1,\mathbf{o}_1) : Q_1.\mathcal{E}_1] \\ \mathcal{N} &::= |_i\ \mathbf{a}_i \parallel \sigma \ [] \ \mathcal{N} \circ \mathcal{N} \ [] \ \mathcal{N} \otimes \mathcal{N}\end{aligned}$$

2.3.2.2 Operational Semantics

The following example shows how execution proceeds in the local view. An agent $\mathbf{A}$ owns the first two qubits of the system's state $\sigma(1,2,3,4,5)$ and executes the Hadamard pattern $\mathcal{H}$ on its first qubit. Qubits not in agent's sort are assumed to be initialized to $|+\rangle$ before the pattern execution. The output of $\mathcal{H}$ will be qubit 6, while qubit 1 will be destroyed by measurement, however the corresponding measurement outcome s_1 will be recorded in the state Γ, via the added binding $\Gamma[s_1 \mapsto v]$, where v is the measurement outcome. The network state will become $\sigma'(6,2,3,4,5)$. Execution of pattern occurs in a single transition step by relying on its big-step semantics. The evaluation step is as follows:

$$\sigma, \mathbf{A} : \{1,2\}.X_6^{s_1} M_1^0 E_{16} \Longrightarrow \sigma', \mathbf{A} : \{6,2\}$$

The small-step transitions for distributed computations describe how the network evolves over time. A shorthand notation for agents, leaving out classical inputs and output, which do not change with small-step reductions is given:

$$\begin{aligned} \mathbf{a}_i = \mathbf{A}_i : Q_i.\mathcal{E}_i, \qquad & \mathbf{a}^{-q} = \mathbf{A} : Q \backslash q.\mathcal{E}, \\ \mathbf{a}_i.E = \mathbf{A}_i : Q_i.[\mathcal{E}_i.E], \qquad & \mathbf{a}^{+q} = \mathbf{A} : Q \uplus q.\mathcal{E}[q/x]. \end{aligned}$$

A *configuration* is given by the system state σ together with a set of agent programs, and their states, specifically

$$C = \sigma, |_i\ \Gamma_i, \mathbf{a}_i = \sigma, \Gamma_1, \mathbf{a}_1 \mid \Gamma_2, \mathbf{a}_2 \mid \ \ldots\ \mid \Gamma_m, \mathbf{a}_m. \tag{2.6}$$

The small-step rules [13] for configuration transitions, denoted $\Longrightarrow$, are specified below with some explanations afterwards. If the system state is not changed in an evaluation step, $\sigma \vdash$ precedes a rule.

$$\frac{\sigma, \mathcal{P}(V, I, O, \mathcal{A}) \longrightarrow_\lambda \sigma', \Gamma'}{\sigma, \Gamma, \mathbf{A} : I \uplus R.[\mathcal{E}.\mathcal{P}] \Longrightarrow_\lambda \sigma', \Gamma \cup \Gamma', \mathbf{A} : O \uplus R.\mathcal{E}} \tag{2.7}$$

$$\frac{\Gamma_2(y) = v}{\sigma \vdash (\Gamma_1, \mathbf{a}_1.\mathrm{c}?x \mid \Gamma_2, \mathbf{a}_2.\mathrm{c}!y \Longrightarrow \Gamma_1[x \mapsto v], \mathbf{a}_1 \mid \Gamma_2, \mathbf{a}_2)} \tag{2.8}$$

$$\frac{}{\sigma \vdash (\Gamma_1, \mathbf{a}_1.\mathrm{qc}?x \mid \Gamma_2, \mathbf{a}_2.\mathrm{qc}!q \Longrightarrow \Gamma_1, \mathbf{a}_1^{+q} \mid \Gamma_2, \mathbf{a}_2^{-q})} \tag{2.9}$$

$$\frac{L \Longrightarrow_\lambda R}{L \mid L' \Longrightarrow_\lambda R \mid L'} \tag{2.10}$$

The first rule is for local operations. Because a pattern's big-step semantics is given by a probabilistic transition system described by $\longrightarrow$, a probability λ is introduced here. Also, an agent changes its sort depending on pattern's output O. The next rule is for classical rendez-vous and is straightforward. For quantum rendez-vous a substitution q for x in the event sequence of the receiving agent is performed, and agents need to update qubit sorts. The last rule is a metarule, which is required to express that any of the other rules may fire in the context of a larger system. Terms in the parallel composition of agents may need to be rearranged in order to make the context rule applicable.

The big-step semantics of a system of agents can be defined from the small-step rules. In the *initial configuration*, all local states are given by the map containing the classical inputs $\Gamma_{\mathbf{i}_i}$, while the network state is determined by the entanglement resource together with local quantum inputs. A *final configuration* is one in which all agents have an empty command sequence, and in which all local states have been restricted to classical outputs $\Gamma_{i,\mathbf{o}_i}$. A computation *path*, which run from initial to final configurations via small-step transitions, is defined as follows.

Definition 2.11. *Given a network of agents* $\mathcal{N} = |_i\ \mathbf{A}_i(\mathbf{i}_i, \mathbf{o}_i) : Q_i.\mathcal{E}_i \parallel \sigma$ *and quantum inputs* ρ_i*, a* path γ *is a* maximal *sequence of configurations* $C_j = \sigma_j, |_i\ \Gamma_i^j, \mathbf{a}_i^j, j = 1, \ldots, k-1$*, i.e.*

$$\begin{aligned} &C_1 = \sigma \otimes_{i=1}^m \rho_i, |_i\ \Gamma_{\mathbf{i}_i}, \mathbf{A}_i : Q_i.\mathcal{E}_i \\ &C_j \Longrightarrow_{\lambda_j} C_{j+1} \\ &C_k = \sigma_k, |_i\ \Gamma_{i,\mathbf{o}_i}^k, \mathbf{A}_i : Q_i^k, \end{aligned}$$

written as $C_1 \overset{\gamma}{\Longrightarrow}_{\lambda_\gamma} C_k$ *where* $\lambda_\gamma = \prod_{j=1}^k \lambda_j$*, and* C_k *is a final configuration of* $\mathcal{N}$*.*

The operational semantics of a system $\mathcal{N}$ is defined as the probabilistic transition system defined by paths leading to the same observable behaviour of the network. However, because of the nondeterminism in the order in which concurrent agents execute their event sequence, the operational semantics of a network has to be defined with respect to a *schedule*, which is a particular order in which agents execute events.

Definition 2.12. *The* operational semantics *of a network* $\mathcal{N} = |_i\ \mathbf{A}_i(\mathbf{i}_i, \mathbf{o}_i) : Q_i.\mathcal{E}_i \parallel \sigma$*, with respect to a particular schedule, is a probabilistic transition system relating initial with final configurations,*

$$[\![\mathcal{N}]\!]_{op} : \uplus_i Q_i \to \uplus_i Q_i'.\ \otimes_i \rho_i, |_i\ \Gamma_{\mathbf{i}_i} \Longrightarrow_\lambda \sigma', |_i\ \Gamma_{\mathbf{o}_i}$$

with $\lambda = \sum_\gamma \lambda_\gamma$ *and the sum runs over all paths* γ *such that*

$$\sigma \otimes_i \rho_i, |_i(\Gamma_{\mathbf{i}_i}, \mathbf{A}_i(\mathbf{i}_i, \mathbf{o}_i) : Q_i.\mathcal{E}_i) \overset{\gamma}{\Longrightarrow}_{\lambda_\gamma} \sigma', |_i(\Gamma_{\mathbf{o}_i}, \mathbf{A}_i(\mathbf{i}_i, \mathbf{o}_i) : Q_i').$$

$\uplus_i Q_i \to \uplus_i Q_i'$ *is called the* type *of the network.*

Two networks $\mathcal{N}_1$ and $\mathcal{N}_2$ are *operationally equivalent*, denoted $[\![\mathcal{N}_1]\!]_{op} \equiv [\![\mathcal{N}_2]\!]_{op}$, if their operational semantics, given by the probabilistic transition system, is identical. Operational equivalence is therefore identified with *bisimilarity* since definition of computation paths imposes bisimilarity relation on final configurations

2.3.2.3 Denotational Semantics

The denotational semantics is a mapping from classical inputs to classical outputs and a quantum operation, which determines quantum states evolution in the network. There are two types of classical outputs. The *external* outputs $\mathbf{o}_e = \uplus_i \mathbf{o}_{i,e}$ depend only on the classical input $\mathbf{i}$, and the *signal* outputs $\mathbf{o}_s = \uplus_i \mathbf{o}_{i,s}$ that depend on the quantum operation and vice versa. Since local events operate on disjoint sets of qubits, it does not matter in which order these operations are applied. Therefore, any schedule of the computation leads to the same quantum operation $\mathcal{L}$. So to determine the operation elements of $\mathcal{L}$, a particular schedule is chosen, and then patterns are composed in the order in which they are executed. Each operation element L_j then corresponds to a sequential composition of actualizations for each of these patterns. When there are signal outputs particular measurement outcomes are preserved, therefore excluding actualizations of $\mathcal{L}$ that do not correspond to that outcome. Actualizations compatible with output $\mathbf{o}_s$ are denoted by $L_i^{\mathbf{o}_s}$, and the quantum operation with these operation elements by $\mathcal{L}^{\mathbf{o}_s}$, they are called *restricted*.

Definition 2.13. *The* denotational semantics *of a network* $\mathcal{N} = |_i\ \mathbf{A}_i(\mathbf{i}_i, \mathbf{o}_i) : Q_i.\mathcal{E}_i \parallel \sigma$ *is given by*

$$[\![\mathcal{N}]\!]_{de} : \uplus_i Q_i \to \uplus_i Q_i'.\mathbf{i} \to \{(\mathbf{o}, \mathcal{L}^{\mathbf{o}_s}), \forall \mathbf{o}_s\}$$

with

$$\mathcal{L} : \mathcal{D}(\mathcal{H}_I) \to \mathcal{D}(\mathcal{H}_O) : \otimes_i \rho_i \to \sum_j L_j(\sigma \otimes_i \rho_i)L_j^\dagger,$$

where $\mathbf{o} = \mathbf{o}_e \uplus \mathbf{o}_s$*,* ρ_i *is the quantum input,* Q_i' *the final sort of agent* $\mathbf{A}_i$*, and* I *and* O *are quantum input and output spaces respectively. In case there are no outputs, we have* $[\![\mathcal{N}]\!]_{de} : \uplus_i Q_i \to \uplus_i Q_i'.\mathbf{i} \to \mathcal{L}$*, or just* $.\mathcal{L}$ *if there are no inputs either.*

For example, the one-agent network $\mathbf{A}(-, \{s_2\}) : \{1\}.X_1^{s_2} M_2^{-\alpha}$ has denotational semantics $[\![\mathcal{N}]\!]_{de} : \{(0, p\rho), (1, (1-p)X\rho X)\}$, for all ρ, where probability p is a function of α.

It has been proven in [13] that there is a precise correspondence between the operational and the denotational semantics of networks of agents, that means $\forall \mathcal{N}_1, \mathcal{N}_2 : \mathcal{N}_1 \equiv_{op} \mathcal{N}_2 \iff \mathcal{N}_1 \equiv_{de} \mathcal{N}_1$. Another important result proven in the paper is that semantics of networks is conserved with respect to network composition, i.e. the semantics of networks is compositional: $[\![\mathcal{N}_2.\mathcal{N}_1]\!] = [\![\mathcal{N}_2]\!].[\![\mathcal{N}_1]\!]$ and $[\![\mathcal{N}_1 \otimes \mathcal{N}_2]\!] = [\![\mathcal{N}_1]\!] \otimes [\![\mathcal{N}_2]\!]$. This means the composing any two networks is sound and the computations on the composed network are correct.

2.3.3 Knowledge in Quantum Networks

A formal framework for investigating epistemic and temporal notions in the context of distributed quantum computation has been constructed in [12, 16, 17] on top of structures developed in Distributed Measurement Calculus. A proper notion of quantum knowledge, which captures the information an agent can obtain about its quantum state, needs to be defined. Authors' approach to the problem is following: an agent knows a state that it has prepared and a state that it has measured. Also, it may obtain knowledge of one of the above by classical communication. While knowledge of preparation states is contained in the description of the protocol, the notion of equivalence from definition 2.14 captures the latter two cases. In this sense the quantum knowledge is about classically knowing facts about quantum systems.

This notion of quantum knowledge is model-independent, however in this section it is used in the context of *quantum networks*. A network determines a set of configurations $\mathcal{C}_\mathcal{N}$ that can potentially occur during execution of $\mathcal{N}$. Configurations are defined by equation 2.6. Formally, $\mathcal{C}_\mathcal{N}$ is obtained by following the rules for the small-step operational semantics of networks from the section 2.3.2.2.

Primitive propositions are defined abstractly rather then in a full-fledged language. An interpretation of $\mathcal{N}$ is a truth-value assignment for configurations in $\mathcal{C}_\mathcal{N}$ for some basic set of primitive propositions p. Writing $I(C, \varphi)$ for the interpretation of fact φ in configuration C gives

$$C, \mathcal{N} \vDash \varphi \iff I(C, \varphi) = \text{true}.$$

Composite formulae can be built from primitive propositions and the logical connectives $\wedge$, $\vee$ and $\neg$ in the usual way.

2.3.3.1 Knowledge

In order to define knowledge, an equivalence relation needs to be defined on configurations for each of the agents, embodying what an agent knows about the global configuration from its own information only.

All classical information an agent has is stored in its local state Γ. As for quantum information, an agent knows which qubits it owns, what local operations it applies on these qubits, and what non-local entanglement it shares initially. It can also have information on its local quantum inputs. All this information is captured by an agent's local state and its event sequence in a particular configuration. The following definition follows naturally [16].

Definition 2.14. *Given a network $\mathcal{N}$ and configurations $C = \sigma, |_i\ \Gamma_i, \mathbf{A}_i : Q_i.\mathcal{E}_i$ and $C' = \sigma', |_i\ \Gamma'_i, \mathbf{A}_i : Q'_i.\mathcal{E}'_i$ in $\mathcal{C}_\mathcal{N}$, an agent $\mathbf{A}_i$ considers C and C' to be* equivalent*, denoted $C \sim_i C'$, if $\Gamma_i = \Gamma'_i$ and $\mathcal{E}_i = \mathcal{E}'_i$. For each agent $\mathbf{A}_i$ the relation $\sim_i$ is an equivalence relation on $\mathcal{C}_\mathcal{N}$, called the* possibility relation *of $\mathbf{A}_i$.*

Via possibility relations modal operators for knowledge (i.e. what it means for an agent $\mathbf{A}_i$ to know a fact φ in a configuration C) can be defined in the usual way:

$$C, \mathcal{N} \vDash K_i\varphi \Longleftrightarrow \forall C' \sim_i C : C' \vDash \varphi.$$

There is one caveat to this. For each set of possible classical input values there is a group of corresponding configurations in $\mathcal{C}_\mathcal{N}$. However, this cannot be done for quantum inputs, since these occupy a continuous space, and so configurations have to be parameterised by these inputs, writing $C(|\psi\rangle)$ whenever there are some. Either a quantum input is known, in which case there is only one possible initial configuration, or it is unknown, and hence all configurations in the set $\{C(|\psi\rangle), |\psi\rangle \in I_A\}$ are considered equivalent by $\mathbf{A}$. Basically, whenever a configuration $C(|\psi\rangle)$ is written, it should be interpreted as a *set* of states, all considered equivalent by all agents of the network.

2.3.3.2 Time

To formalise time related logical statements, the approach of *computational tree logic* (CTL) is used in the model. The reason for this is the fact that quantum networks typically have a branching structure, and so there is a need for statements concerning all paths as well as those pertaining to some paths. This can be expressed by placing restrictions on the paths considered in a particular statement. And this is precisely captured in the definition of modal path operators. In this way it is possible to abstract away from actual path definitions, which are determined by the formal semantics for networks described in section 2.3.2.2, and denoted abstractly as $\Longrightarrow$ below.

Traditional temporal state operators $\Box$ and $\Diamond$, combined with the path operators A ("*for all paths*") and E ("*there exists a path*"), are introduced into the model as follows:

$$C, \mathcal{N} \vDash A\Box\varphi \Longleftrightarrow \forall\gamma, \forall C' \text{ with } C \overset{\gamma}{\Longrightarrow} C' : C' \vDash \varphi,$$
$$C, \mathcal{N} \vDash E\Box\varphi \Longleftrightarrow \exists\gamma, \forall C' \text{ with } C \overset{\gamma}{\Longrightarrow} C' : C' \vDash \varphi,$$
$$C, \mathcal{N} \vDash A\Diamond\varphi \Longleftrightarrow \forall\gamma, \exists C' \text{ with } C \overset{\gamma}{\Longrightarrow} C' : C' \vDash \varphi,$$
$$C, \mathcal{N} \vDash E\Diamond\varphi \Longleftrightarrow \exists\gamma, \exists C' \text{ with } C \overset{\gamma}{\Longrightarrow} C' : C' \vDash \varphi,$$

where $\overset{\gamma}{\Longrightarrow}$ is the closure of the small-step transition relation $\Longrightarrow$. In other words, $C \overset{\gamma}{\Longrightarrow} C'$ if C' can be reached form C by a series of consecutive small-step transitions, specified by path γ.

2.4 Evaluation

In this section, we summarise the three frameworks and analyse their usability in the context of model checking multi-agent systems.

Qubit Message Passing Environments

There are two main problems with this approach. First, the modal operator for quantum knowledge K_i^q is an information-theoretic idealisation of knowledge. It embodies what an agent

would determine from its local quantum state if it measured the qubit repeatedly. But observing a state alters it irreversibly and the only way is to measure many copies of the qubit, which is not always possible. So, quantum knowledge does not consist of possession of a quantum state, i.e. if an agent has a qubit it does not imply that the agent knows anything about it.

The second problem with the approach is that the quantum operations on qubits, apart from measurements, are not included in the set of actions which an agent may perform. This makes it somewhat limited.

Moreover, an untrue property of the teleportation protocol has been verified using this formalism in [31], which means that Qubit Message Passing Environments is not a sound logic. And thus, together with the problematic interpretation of quantum knowledge, it is not suitable for use in epistemic model checking.

Dynamic Epistemic Quantum Logic

This formalism has very abstract approach to modelling knowledge in distributed quantum systems. The knowledge operator embodies the information that a subsystem "has" about the global system rather then facts that an external agent in possession of the subsystem could possibly know about it. Although this is more sensible than the previous notion of quantum knowledge, it still does not reflect the reality of multi-agent systems.

Quantum logic is included in the framework, even though it is not the essential part of it, and so a model checker should possibly be able to handle this non-classical logic. Also, the method is very theoretical, and in fact, the proof theory is not yet complete, and so direct application to automatic model checking would be difficult.

Although epistemic properties of the teleportation protocol have been shown valid, the above problems render this approach unsuitable for automatic verification.

Distributed Measurement Calculus

The logic based on DMC stands out as the most suitable starting point for automatic, epistemic verification of quantum protocols. One of the main strengths of this approach is the fact that it has been built on top of formalisation for one way quantum computing, which is not only able to perform any quantum computation (see section 2.3.1.4), it also seems to be the easiest way to realise an actual quantum computer at the moment [32].

Another advantage is that DMC does not consider tricky notion of the quantum epistemic modality. And the idea that "quantum knowledge is about classically knowing facts about quantum systems" makes sense independently of any concrete agent-based model of quantum networks. It has also been shown [31] that this approach epistemically verifies the correctness of the quantum teleportation protocol without any problems.

But what makes this approach really outstanding is its similarity to the formalism of interpreted systems . Small-step rules for configurations construct a transition system necessary for automatic generation of runs from a given quantum protocol. The model has CTL approach to time, which is commonly used in model checkers. Equivalence relations are based on local classical states and local quantum operations only, and as such they are purely classical. These features make it possible to translate a network specification in DMC such that the epistemic model checker `MCMAS` is able to verify its properties automatically as we will show in this report.

Chapter 3

Model Checking with MCMAS

In this chapter, we introduce `MCMAS` [28], a symbolic model checker for multi-agent systems, and its dedicated input language ISPL.

Model checking [18] is a technique for automatic verification of finite concurrent systems such as communication protocols and hardware circuits. It has many advantages over other approaches, it is fast, produces counterexamples and handles partial specifications.

The approach consists of three main tasks. *Modeling* means converting an algorithm into a formalism accepted by a model checker, which is usually a simple compilation task. *Specification* states the properties that the algorithm must satisfy, this is given in some logical formalism. *Verification* phase should be as automatic as possible but it usually involves human interaction, such as error tracing and result analysis. There are several model checking techniques.

- *Temporal model checking* is based on Kripke structures $M = (S, R, L)$ that represents a concurrent system. Given a temporal formula φ expressing some desired specification, the goal is to find all states S that satisfy φ: $\{s \in S : M, s \vDash \varphi\}$. The system specifies the requirements if all initial states are it this set. Linear temporal logic (LTL) and computation tree logic (CTL) are logical formalisms useful for specifying the systems in this setting.

- *Symbolic model checking* [9] has been born out of the need for handling large system. Symbolic representation for the state transition graphs allows for this. It is based on manipulation of Boolean formulas. Ordered binary decision diagrams (OBDDs) are a canonical form representation of these Boolean formulas. They are directed acyclic graphs obtained by optimising binary decision trees which represent these binary formulas. OBDDs are useful for obtaining representations of relations over finite domains and can be used to represent and analyse Kripke structures.

- *Bounded model checking* [8] technique reduces the problem of model checking to a problem of satisfiability for a Boolean formula (SAT). A state transition system is encoded as a propositional formula, which is satisfiable if and only if there exists a counterexample of bounded length k. The formula is given to a satisfiability solver and if the formula cannot be satisfied at length k the search is continued for larger k. SAT solvers do not require exponential space and large designs can be checked very fast, however the method generally lacks completeness and properties that can be verified are limited.

- *Abstraction* [10] is a method for reducing the state space explosion problem. The model checker builds a simplified model of a system if the actual model is too complex. It then can verify a property of the smaller model if the property is preserved in the original one.

3.1 Model Checker for Multi-Agent Systems

`MCMAS` (Model Checker for Multi-Agent Systems) [27, 29, 35] is a symbolic model checker which takes input specified in Interpreted Systems Programming Language (ISPL) and evaluates formulae using algorithms based on OBDDs. It supports a number of modalities including CTL, ATL[1] and epistemic operators. The tool is implemented in `C++` and exploits the CUDD library [38] for BDD operations.

In this section we explain the theoretical background of `MCMAS` and give pseudo-algorithms for the verification of temporal and epistemic modalities.

3.1.1 Interpreted Systems Semantics for Multi-Agent Systems

The interpreted systems formalism has been put forward in [20] as a basic model of multi-agent systems (MAS) and for reasoning about knowledge in such systems. It was originally proposed only for LTL and MAS formalisms are typically built on extensions of CTL. However, interpreted systems can be suitably extended [30] to support these modalities.

In interpreted systems, each agent $i \in A$ is modelled by a set of possible local states L_i, a set of actions Act_i that they may perform according to their protocol function P_i, and an evolution function t_i. Apart from standard agents, a special agent E, representing the environment in which the other agents operate, is analogously described by a set of local states L_E, a set of actions Act_E, a protocol P_E, and an evolution function t_E. The protocol P_i is defined as a function $P_i : L_i \rightarrow 2^{Act_i}$, assigning a set of actions to a given local state. The evolution function is a transition functions returning the target local state given the current local state and the set of actions performed by all agents, formally defined[2] as $t_i : L_i \times L_E \times Act_1 \times \cdots \times Act_n \times Act_E \rightarrow L_i$. Agents evolve simultaneously in every state of the system.

A joint action α is a tuple of agents' actions (one per agent) and a set of all joint actions Act is defined as the Cartesian product of all agents' actions, formally $Act = Act_1 \times \cdots \times Act_n \times Act_E$. The Cartesian product of the agents' local states S, defined $S = L_i \times \cdots \times L_n \times L_E$ is a set of all global states. The local state of agent i in the global state $g \in S$ is denoted $l_i(g)$. The set of reachable global states $G \subseteq S$ is obtained by all the possible runs, generated according to the protocols and the evolution functions, from the set of initial global states $I \subseteq S$. The description of an interpreted system is finalised by inclusion of the set of atomic propositions AP and the evaluation relation $V \subseteq AP \times S$. Formally, An interpreted system is defined as the following tuple

$$IS = \langle (L_i, Act_i, P_i, t_i)_{i \in A}), (L_E, Act_E, P_E, t_E), I, V \rangle \,.$$

The extended interpreted systems provide a semantics for modalities by means of the language $\mathcal{L}$ built from a set of propositional atoms $p \in AP$, and a set of agents $i \in A$

$$\begin{aligned} \varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E[\varphi U \psi] \mid K_i\varphi \mid E_\Gamma\varphi \mid C_\Gamma\varphi \mid D_\Gamma\varphi \mid O_i\varphi \mid \\ \langle\langle\Gamma\rangle\rangle X\varphi \mid \langle\langle\Gamma\rangle\rangle G\varphi \mid \langle\langle\Gamma\rangle\rangle[\varphi U \psi]\varphi \end{aligned}$$

where $\Gamma \subseteq A$ denotes a group of agents. EX, EG, and EU are the standard CTL operators from which all the remaining operators EF, AX, AG, AU and AF can be derived. K_i, E_Γ C_Γ and D_Γ are epistemic operators. The deontic operator O_i means "whenever agent i is working correctly φ holds" and the ATL operators $\langle\langle\Gamma\rangle\rangle X$ ("group Γ can enforce φ at the next step"), $\langle\langle\Gamma\rangle\rangle G$ ("group Γ can enforce a sequence of states in which φ holds globally") and $\langle\langle\Gamma\rangle\rangle U$ ("group Γ can enforce a sequence of states in which ψ eventually holds, and φ holds until then") complete the list of modal operators.

[1] ATL stands for Alternating-time Temporal Logic.

[2] This definition differs from the single evolution function $t : S \times Act \rightarrow S$ proposed in [20].

3.1.2 Verification Algorithms

MCMAS implements verification algorithms for all operators of language $\mathcal{L}$, however an interpreted system has to be first encoded using Boolean variables and Boolean formulae.

The number of Boolean variables required to encode the set of local states L_i of agent i is $nv(i) = \lceil \log_2 |L_i| \rceil$. This means that a global state g can be represented with a vector of Boolean variables of length $N = \sum_{i \in A} nv(i)$ as follows

$$g \rightarrow (\underbrace{x_1, \ldots, x_{nv(1)}}_{L_1}, \ldots, \underbrace{x_j, \ldots, x_{nv(k)}}_{L_k}, \ldots, x_N).$$

Similarly, the number of Boolean variables required to encode the set of actions Act_i of agent i is $na(i) = \lceil \log_2 |Act_i| \rceil$. This means that a joint action α can be associated with a Boolean vector of length $M = \sum_{i \in A} na(i)$ as follows.

$$\alpha \rightarrow (a_1, \ldots, a_M).$$

Each vector then can be identified with a Boolean formula in form of a conjunction of literals. A set of vectors (i.e. a set of global states and actions) is described as a disjunction of the formulae encoding the vectors. The definition of Boolean formulae encoding agents' protocols f_{P_i} and evolution functions f_{t_i} capitalise on the Boolean representations of local states and actions. A local protocol has the following definition

$$f_{P_i} = \bigvee_{l_i \in L_i} \left[bf(\{l_i\}) \wedge \left(\bigoplus_{a \in P_i(l_i)} bf(\{a\}) \right) \right],$$

where $bf(Q)$ is a function that transforms set Q of states or actions into its corresponding encoding set. The rightmost formula expresses the requirement that a single action has to be selected in a given local state. Boolean formula f_{t_i} representing evolution functions is the disjunction of all possible Boolean functions $bf(\{l_p\}) \wedge bf(\{a_q\} \wedge bf'(\{l_r\}))$ which encode a generic pair $t_i(l_p, a_q) = l_r$, where $a_q \in Act$ and $l_p, l_r \in L_i$. Formulae for a joint protocol f_P and global evolution function f_t are defined as conjunction of respective local ones as follows

$$f_P = \bigwedge_{i \in A} f_{P_i}, \qquad f_t = \bigwedge_{i \in A} f_{t_i}.$$

Finally, the Boolean function f_I associated with the set I of initial global states is defined as the disjunction of the Boolean formulae representing all global states in I, and the Boolean function f_V corresponding to evaluation function V is such that $f_V(p)$ is a Boolean function encoding the set of all global state in which $p \in AP$ holds.

Model checking algorithms then operate on these structures by computing the set of global states $[\![\varphi]\!]$ in which formula φ holds. This class of algorithms is called *labelling* because they "label" the state with φ. Pseudo-algorithm 3.1 describes a labelling procedure for verification of a temporal epistemic fragment of the CTLK formulae in interpreted systems. All algorithms in this section are adapted from [35].

The verification of temporal formulae requires definition of a temporal transition relation $R_t \subseteq S \times S$ between two two global states $g, g' \in S$. The formal definition of the relation is as follows

$$R_t(g, g') \text{ iff } \exists a \in Act \text{ s.t. } t(g, a) = g'.$$

Now, a CTL model $M = (S, R_t, V)$ can be constructed from the set of global states S, the temporal relation R_t, and the evaluation function V. The labelling algorithms 3.2, 3.3 and 3.4

Algorithm 3.1 $\text{MC}_{\mathbf{CTLK}}(\varphi, IS)$

1: **if** $\varphi \in AP$ **then**
2: **return** $V(\varphi)$
3: **else if** $\varphi = \varphi_1$ **then**
4: **return** $S \setminus \text{MC}_{\mathbf{CTLK}}(\varphi_1, IS)$
5: **else if** $\varphi = \varphi_1 \vee \varphi_2$ **then**
6: **return** $\text{MC}_{\mathbf{CTLK}}(\varphi_1, IS) \cup \text{MC}_{\mathbf{CTLK}}(\varphi_2, IS)$
7: **else if** $\varphi = EX\varphi_1$ **then**
8: **return** $\text{MC}_{\mathbf{CTLK},EX}(\varphi_1, IS)$
9: **else if** $\varphi = EG\varphi_1$ **then**
10: **return** $\text{MC}_{\mathbf{CTLK},EG}(\varphi_1, IS)$
11: **else if** $\varphi = E[\varphi_1 U \varphi_2]$ **then**
12: **return** $\text{MC}_{\mathbf{CTLK},EU}(\varphi_1, \varphi_2, IS)$
13: **else if** $\varphi = K_i\varphi_1$ **then**
14: **return** $\text{MC}_{\mathbf{CTLK},K}(\varphi_1, i, IS)$
15: **else if** $\varphi = E_\Gamma\varphi_1$ **then**
16: **return** $\text{MC}_{\mathbf{CTLK},E}(\varphi_1, \Gamma, IS)$
17: **else if** $\varphi = D_\Gamma\varphi_1$ **then**
18: **return** $\text{MC}_{\mathbf{CTLK},D}(\varphi_1, \Gamma, IS)$
19: **else if** $\varphi = C_\Gamma\varphi_1$ **then**
20: **return** $\text{MC}_{\mathbf{CTLK},C}(\varphi_1, \Gamma, IS)$
21: **end if**

for model checking CTL has been introduced in [25]. The following function $\text{pre}_\exists(X)$ generates the set of global states $Y \subseteq S$ which can transition into a state in Y

$$Y = \text{pre}_\exists(X) = \{g \in S \mid \exists g' \text{ s.t. } g' \in X \text{ and } R_t(g, g')\}.$$

Additionally, the last two algorithms exploit monotonicity of operators $\tau_{EG,\varphi}(Q) = [\![\varphi]\!] \cap \text{pre}_\exists(Q)$ and $\tau_{EG,\varphi,\psi}(Q) = [\![\varphi]\!] \cup ([\![\varphi]\!] \cap \text{pre}_\exists(Q))$ to build the sets $[\![\varphi]\!]$ which are the greatest, respective the least fixed-points of the operators.

The epistemic operators[3] are verified using accessibility relations rather than temporal transition relation. Specifically, accessibility relations $\sim_i \subseteq S \times S$ are defined as equivalence relations on local states of agent i and group accessibility relations have following definition

$$R_\Gamma^E = \bigcup_{i \in \Gamma} \sim_i, \qquad R_\Gamma^D = \bigcap_{i \in \Gamma} \sim_i, \qquad R_\Gamma^C = (R_\Gamma^E)^*,$$

where $\Gamma \subseteq A$ is a group of agents. Note, that R_Γ^C is the transitive closure of R_Γ^E.

The evaluation of epistemic operators has to be restricted to the set of reachable global states G. This set must be determined before evaluating epistemic operators and it is computed by iterating $\tau(\emptyset)$ until the least fixed-point of the operator $\tau : S \to S$ is found. The operator has the following definition

$$\tau(Q) = I \cup Q \cup \{g \in S \mid \exists g' \text{ s.t. } g' \in Q \text{ and } R_t(g, g')\},$$

where I is the set of initial states.

The algorithms 3.5, 3.6, 3.7 and 3.8 outline the procedures for computing sets of global states of the respective epistemic operators. The last algorithm, for common knowledge, is based on the greatest fixed-point of the operator $\tau_{C,\varphi}(Q) = [\![E_\Gamma(\varphi \wedge (Q))]\!]$.

[3]The introduction to epistemic logic can be found in appendix B.

Algorithm 3.2 $\text{MC}_{\mathbf{CTLK}_E X}(\varphi, IS)$

1: $X \leftarrow \text{MC}_{\mathbf{CTLK}}(\varphi, IS)$
2: $Y \leftarrow \text{pre}_\exists(X)$
3: **return** Y

Algorithm 3.3 $\text{MC}_{\mathbf{CTLK},EG}(\varphi, IS)$

1: $X \leftarrow \text{MC}_{\mathbf{CTLK}}(\varphi, IS)$, $Y \leftarrow S$, $Z \leftarrow \emptyset$
2: **while** $Z \neq Y$ **do**
3: $\quad Z \leftarrow Y$
4: $\quad Y \leftarrow X \cap \text{pre}_\exists(Y)$
5: **end while**
6: **return** Y

Algorithm 3.4 $\text{MC}_{\mathbf{CTLK},EU}(\varphi_1, \varphi_2, IS)$

1: $X \leftarrow \text{MC}_{\mathbf{CTLK}}(\varphi_1, IS)$, $Y \leftarrow \text{MC}_{\mathbf{CTLK}}(\varphi_2, IS)$, $Z \leftarrow \emptyset$, $W \leftarrow S$
2: **while** $Z \neq W$ **do**
3: $\quad W \leftarrow Z$
4: $\quad Z \leftarrow Y \cup (X \cap \text{pre}_\exists(Z))$
5: **end while**
6: **return** Z

Algorithm 3.5 $\text{MC}_{\mathbf{CTLK},K}(\varphi, i, IS)$

1: $X \leftarrow \text{MC}_{\mathbf{CTLK}}(\neg\varphi, IS)$
2: $Y \leftarrow \{g \in G \mid \exists g' \in X \text{ s.t. } g \sim_i g'\}$
3: **return** $\neg Y \cap G$

Algorithm 3.6 $\text{MC}_{\mathbf{CTLK},E}(\varphi, \Gamma, IS)$

1: $X \leftarrow \text{MC}_{\mathbf{CTLK}}(\neg\varphi, IS)$
2: $Y \leftarrow \{g \in G \mid \exists g' \in X \text{ s.t. } R^E_\Gamma(g, g')\}$
3: **return** $\neg Y \cap G$

Algorithm 3.7 $\text{MC}_{\mathbf{CTLK},D}(\varphi, \Gamma, IS)$

1: $X \leftarrow \text{MC}_{\mathbf{CTLK}}(\neg\varphi, IS)$
2: $Y \leftarrow \{g \in G \mid \exists g' \in X \text{ s.t. } R^D_\Gamma(g, g')\}$
3: **return** $\neg Y \cap G$

Algorithm 3.8 $\text{MC}_{\mathbf{CTLK},C}(\varphi, \Gamma, IS)$

1: $X \leftarrow \text{MC}_{\mathbf{CTLK}}(\varphi, IS)$, $Y \leftarrow G$
2: **while** $X \neq Y$ **do**
3: $\quad X \leftarrow Y$
4: $\quad Y \leftarrow \{g \in G \mid \exists g' \in G \text{ s.t. } g' \in \text{MC}_{\mathbf{CTLK}}(\varphi, IS) \text{ and } g' \in X \text{ and } R^E_\Gamma(g, g')\}$
5: **end while**
6: **return** Y

3.2 ISPL Language

ISPL (Interpreted Systems Programming Language) [29] is a modular input language of `MCMAS` derived from the formalism of interpreted systems. An ISPL program fully describes a multi-agent system and it is composed of six sections:

- *Declarations of agents.* An agent is characterised by a set of variables representing its local state, by a set of actions the agent can perform, and by protocol and evolution functions. There are two types of agents, standard agents and the *Environment* agent. The latter describes background conditions for the former but both types are modelled similarly. A declaration of an agent has the following syntax

  ```
  Agent <ID>
      <body>
  end Agent
  ```

  where `<ID>` is either the agent's identifier or a reserved keyword `Environment` and `<body>` is a description of the agent explained below in more detail.

- *Evaluation function* consists of a set of statements defining atomic propositions and has the following declaration

  ```
  Evaluation
      <ID> if <condition>;
  end Evaluation
  ```

  where `<ID>` is an identifier of the atomic proposition and `<condition>` is a Boolean expression over the local states of the agents. The proposition holds in global states that satisfy the condition.

- *Initial states* is a Boolean formula describing the set of initial global states. It is given by Boolean conditions on the local variables of the agents. It is defined by the following syntax

  ```
  InitStates
      ...
  end InitStates
  ```

- *Groups declaration.* Groups of agents are used in the verification of formulae containing group modalities. This section may be removed from an ISPL program if no group modality is involved in any formula. The following statement gives declaration of the section

  ```
  Groups
      <ID> = {...};
  end Groups
  ```

  where `<ID>` is a group identifier and set `{...}` contains at least one agent, including the *Environment* agent.

- *Fairness formulae* section contains a list of Boolean formulae defined over atomic propositions using logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$. All the logic expressions listed here constrains the reachable state space since they must hold infinitely often along all paths. The section is optional and has the following syntax

```
Fairness
    ...
end Fairness
```

- *Formulae* section consists of a list of formulae to be verified. A formula is defined over atomic propositions using the language $\mathcal{L}$. The section has the following identification.

  ```
  Formulae
      ...
  end Formulae
  ```

  The propositional atoms and groups involved in these formulae must be defined in *Evaluation*, respective *Groups* section.

The declaration of an agent's `<body>` is further further modularised into the following subsections which closely follows the definition of an agent in interpreted systems

- *Local states* of an agent are described in terms of local variables which only the agent can observe. An exception to this are *local observable variables* of *Environment* agent accessible to other agents, in which case they are part of their extended local states. Variables and local observable variables accessible to all agents are listed in `Vars`, respective `Obsvars` section. All variables have types. The supported types are Boolean (`true`, `false`), enumeration (a list of values) and bounded integer (set of integers within lower and upper bound). An optional section `RedStates` defines extended local states used to evaluate the correctness modalities O. Red states are represented by a Boolean formula over the agent's local and local observable variables. All other states are implicitly green.

- *Actions.* All agents, including *Environment*, may perform actions. Contrary to local variables, actions are public and visible to all other agents. This section is given by the statement `Actions = { <list> };` where `<list>` is the list of all agent's actions.

- *Protocols.* The protocol is a rule establishing which actions can be performed in a given local state. Formally, it is a function assigning a set of actions to a local state. If the set associated with the local state contains more then one action the agent selects non-deterministically which action to perform. The `Protocol` section is a list statements. A statement in the protocol function consists of a Boolean condition over local states and a set of actions. The actions from the set are enabled in all local states that satisfy the condition. The conditions of different statements are not required to be mutually exclusive and the list of statements do not have to be exhaustive. The condition `Other` in statement `Other : { <list> };` represents all local states that are not explicitly specified in the protocol function.

- *Evolution function.* The local evolution function determines how local states of an agent evolve, based on the agent's current local state and on the actions performed by all agents. Formally, the evolution function is modelled by a function from the set of local variable assignments and actions to the set of local variable assignments. The global evolution function is the union of the local evolution functions of all agents. ISPL provides two ways to specify evolution functions `MultiAssignment` and `SingleAssignment`. In the former, an evolution item in an evolution function consists of a set of assignments of local variables and an enabling condition and the items are mutually excluded. In the latter there is only one assignment and items are mutually excluded only if they update the same variable. The enabling condition in both semantics is a Boolean expression over local variables and agents' actions.

Chapter 4

Design of the Compiler

In this chapter, we present a method for automatic verification of quantum MAS. The idea is that, given a protocol specification in DMC, we can translate it into ISPL such that possible worlds, temporal accessibility relations and epistemic accessibility relations are preserved. Once a translation is complete, the `MCMAS` model checker takes over and verifies the protocol. First, we devise the translation map from DMC to ISPL, and then we introduce an adaptation of DMC that can be read by a machine.

4.1 Translation Map

If we compare semantics of DMC and `MCMAS` we can see that both have roots in interpreted systems [20]. The former is defined in terms of configurations and transitions between them. A configuration is given by $C = \sigma, |_i\ \Gamma_i, \mathbf{A}_i : Q_i.\mathcal{E}_i$, where σ denotes quantum state of the system and $\mathbf{A}_i$ is an agent with local state Γ_i, qubit references Q_i and set of operations $\mathcal{E}_i$. Similarly in the latter, the system consists of a special agent *Environment* and other agents defined by their local states and a set of actions. Also, transition functions change local states of agents according to their actions to a new states.

It seems very natural that a mapping from DMC to ISPL exists and we demonstrate our solution in this section. We split the problem into classical states, quantum states and to what can be verified and deal with each part separately. Once a prospective protocol is translated into a valid ISPL code, we process it in `MCMAS`.

4.1.1 Local States of Agents

A local state Γ_i of agent $\mathbf{A}_i$ consists of her classical variables and qubit references. The other part of classical information of the agent is captured by her event sequence $\mathcal{E}_i$. To represent all this in ISPL we map the variables and qubit references as they are in Γ_i to `Variable` section of ISPL and we characterise the event sequence by one variable in the same section.

Classical bits are represented by variable of enumeration type. Possible values of bit x of agent $\mathbf{A}$ depends on whether the bit is an input, a signal or a received bit from another agent. The first one has only two enumeration values

```
x : {zero, one};
```

A value of a measurement outcome is assigned to signal s. Before $\mathbf{A}$ measures her qubit, the bit is not initialised and its value is meaningless. Similarly, if the agent assigns a value, which she receives from agent $\mathbf{B}$, to variable y. Therefore, we define the two

```
s : {undefined, zero, one};
y : {undefined, zero, one};
```

and both variables are initialised to the enumeration value `undefined` in `InitStates` section of ISPL.

In the DMC specification, agents have associated a set of qubit references to qubits in their sorts. We extend this a little, so that every agent has a qubit reference `qi` to all qubits i in the computation state space. Moreover, we add some extra meaning to this bounded integer variable later in section 4.1.3.1. For now we define the qubit reference as

```
qi : 0..2;
```

To represent the event sequence $\mathcal{E}_{\mathbf{A}}$ of agent $\mathbf{A}$, we introduce a new variable called *program counter*. This is important feature because without it the agent would not be able to determine what to do next. Therefore, we associate every event $\mathcal{E}_{\mathbf{A},n} \in \mathcal{E}_{\mathbf{A}}$ with the natural number n of the agent's *program counter*. We also define relative position of events in the sequence such that $\mathcal{E}_{\mathbf{A},n} = \mathcal{E}_{\mathbf{A},m}$, $\mathcal{E}_{\mathbf{A},n} < \mathcal{E}_{\mathbf{A},m}$ and $\mathcal{E}_{\mathbf{A},n} > \mathcal{E}_{\mathbf{A},m}$ if and only if $n = m$, $n < m$ and $n > m$ respectively.

Each agent has her own *program counter* independent of counters of other agents. The counter is incremented only when the associated even is successfully executed. It is represented in ISPL by a variable of bounded integer type `pc` defined

```
pc : 1..l;
```

where `l` is the length of the agent's event sequence $\mathcal{E}_{\mathbf{A}} + 1$. This means that `pc` is always initialised to 1, which corresponds to the first event in the sequence, and after the agent executes all her commands, the counter is in its upper bound value.

Closely related to `pc` is *global counter* variable of the *Environment*. This counter increases with every action in the network, keeps global time and enumerates configurations according to their occurrence in the path. Again, it is defined as bounded integer variable

```
pc : 1..k;
```

where `k` corresponds to the final configuration C_k of the network.

4.1.1.1 Actions, Protocols and Transitions

For each local event $\mathcal{E}_{i,n} \in \mathcal{E}_i$ in the specification of agent i in DMC we create one or more actions in the set of agent's actions in the `Agent` section of ISPL. We may need more actions for one event because agents in ISPL do not have access to other agents local variables, and so if the event depends on a variable we have to express this with different action for each value of the variable. This is important for classical communication between agents and for executing dependent quantum commands since the quantum state of the system is represented by a special kind of agent, which we describe in the next section.

The event sequence itself then translates to the protocol function of the agent. In DMC, an agent's events depend only on values in her local state. On ISPL side, protocols are functions from local variable assignments to actions. Thus this translation is consistent.

The DMC configuration transitions are given in terms of small-step rules 2.7 to 2.10. In ISPL, an agent's evolution function describes how the local state of the agent changes as a result of actions performed by all agents. The global evolution function is then the conjunction of all local evolution functions.

We now describe the ISPL actions, the protocol function and the classical part of evolution function for all types of events defined in DMC, starting with communications. The evolution of quantum state of the network is covered in section 4.1.2.1. Each classical and quantum message reception event has a corresponding classical, respectively quantum, message sending event, and so we examine them both in pairs.

Classical rendez-vous events are c! for sending the message and c? for its reception. Suppose that agent **A** sends the value of variable x to agent **B** and she then assigns this value to variable y. The specification of agents in this scenario in DMC is $\Gamma_{\mathbf{A}}, \mathbf{A} : Q_{\mathbf{A}}.\mathcal{E}_{n,\mathbf{A}}$ and $\Gamma_{\mathbf{B}}, \mathbf{B} : Q_{\mathbf{B}}.\mathcal{E}_{m,\mathbf{B}}$ with $\mathcal{E}_{n,\mathbf{A}} = \text{c!}x$ and $\mathcal{E}_{m,\mathbf{B}} = \text{c?}y$. We introduce two actions of agent **A**, `snd_`B`_`x`0` and `snd_`B`_`x`1`, corresponding to the sending event. We have one action for each value of x. Agent **B** need just `rcv_`A`_`y action for the reception. Agent **A** performs the event as follows

```
pc = n and x = zero :  {snd_B_x0};
pc = n and x = one :  {snd_B_x1};
```

similarly, protocol function of agent **B** is defined as

```
pc = m :  {rcv_A_y};
```

The configuration transition for classical communication events is described by rule 2.8. Because the exchange of a message has to be synchronised, the transition occurs only if both agents have appropriate events at the beginning of their event sequences. Otherwise nothing happens and one of the agent is busy waiting for the other. Note, that successful execution of the events discards them from the event sequences. The evolution function of agent **A** subsequently reflects these facts

```
pc = pc + 1 if (Action = snd_B_x0 or Action = snd_B_x1) and
               B.Action = rcv_A_y;
```

as does the evolution function of agent **B**. Moreover, her function contains two statements, one for each received value. They are defined as

```
y = zero and pc = pc + 1 if Action = rcv_A_y and A.Action = snd_B_x0
y = one and pc = pc + 1 if Action = rcv_A_y and A.Action = snd_B_x1
```

The quantum communication, denoted qc! and qc? respectively, does not change the quantum state of the system and it can be abstracted as qubit reference exchange between the agents. Assume that agent **A** sends qubit i to agent **B**. The DMC description of agents in this scenario is almost the same as before $\Gamma_{\mathbf{A}}, \mathbf{A} : Q_{\mathbf{A}}.\mathcal{E}_{n,\mathbf{A}}$ and $\Gamma_{\mathbf{B}}, \mathbf{B} : Q_{\mathbf{B}}.\mathcal{E}_{m,\mathbf{B}}$, just with different events $\mathcal{E}_{n,\mathbf{A}} = \text{qc!}i$ and $\mathcal{E}_{m,\mathbf{B}} = \text{qc?}i$. We have only one action for each agent, `qsnd_`B`_`i and `qrcv_`A`_`i respectively. Also, each agent has one statement in her protocol function.

```
pc = n :  {qsnd_B_i};
```

for agent **A** and the following for agent **B**

```
pc = m :  {qrcv_A_i};
```

Rule 2.8 defines the configuration transition for quantum rendez-vous in terms of sets of qubit references $Q_{\mathbf{A}}$ and $Q_{\mathbf{B}}$. When agent **A** sends the qubit i, it is removed from her set as follows

```
qi = 0 and pc = pc + 1 if Action = qsnd_B_i and B.Action = qrcv_A_i;
```

and conversely, when agent **B** receives the qubit, it is added to her set

```
qi = 1 and pc = pc + 1 if Action = qrcv_A_i and A.Action = qsnd_B_i;
```

This concludes the communication between agents and a description of the basic quantum commands follows. However, we address only actions and protocols of the commands in this section and come back to their evolution functions later. The first command is entanglement. On entanglement, agent **A** applies controlled-Z operator on qubits i and j. The definition of the agent in this case is $\Gamma_{\mathbf{A}}, \mathbf{A} : i, j \uplus R_{\mathbf{A}}.\mathcal{E}_{n,\mathbf{A}}$ with $\mathcal{E}_{n,\mathbf{A}} = E_{ij}$. Since this event is independent of signals, we have only one corresponding action `eixj_n`. Note, that we add suffix `n` because we want to distinguish between two E_{ij} events executed in different time steps. This eliminates ambiguity in evolution function of the *Environment* agent. We then insert the following statement to the protocol of agent **A**

```
pc = n :  {eixj_n};
```

The correction commands X_i^s and Z_i^s together with pattern macros $\mathcal{P}$, realising some single qubit unitary operators, can be dealt with as a group since they differ only in their matrix representations. Agent **A** executes Pauli operator X or Pauli operator Z or pattern $\mathcal{P}$ on qubits i if signal $s \mapsto 1$. These scenarios have the following description in DMC, $\Gamma_{\mathbf{A}}, \mathbf{A} : i \uplus R_{\mathbf{A}}.\mathcal{E}_{n,\mathbf{A}}$ with $\mathcal{E}_{n,\mathbf{A}} = X_i^s$ or $\mathcal{E}_{n,\mathbf{A}} = Z_i^s$ or $\mathcal{E}_{n,\mathbf{A}} = \mathcal{P}_i^s$. The corresponding actions are `xi_n`, `zi_n` and `pi_n` respectively. Since the agent applies the events conditionally, we add action `skip` to her set of actions. For the same reason, each event translates into a protocol function with two rules

```
pc = n and s = zero :  {skip};
pc = n and s = one :  {Ui_n};
```

where $U \in \{\mathtt{x}, \mathtt{z}, \mathtt{p}\}$.

The last quantum command, measurement, is a complex event. It modifies the quantum state of the network as well as the local state of an agent. A measurement is stochastic. We utilise the feature of ISPL that protocols may be non-deterministic also. We associate a pair of actions to the given variable assignment. Each action represents possible measurement outcome. Additionally, measurements may depend on signals s and/or t. Suppose agent **A** measures her qubit i in the $\{|+_\alpha\rangle, |-_\alpha\rangle\}$ basis. In DMC, this operation is specified as $\Gamma_{\mathbf{A}}, \mathbf{A} : i \uplus R_{\mathbf{A}}.\mathcal{E}_{n,\mathbf{A}}$ with $\mathcal{E}_{n,\mathbf{A}} = {}^t[M_i^\alpha]^s$. Due to the possible signal dependency, there are four different sets of actions/protocol rules, shown in table 4.1. The parameter α is not important for identification of actions because `n` suffix uniquely determines the measurement and during generation of the reachable quantum state space these actions are matched with the correct operations.

This covers all types of events declared in DMC. However, we define a null action `none`, which indicates that the agent has finished her event sequence and is waiting for other agents who are still running their programs. The following statement is the last in the protocol function of all agents. It means that if the agent has no defined action for her local state she does nothing.

```
Other :  {none};
```

4.1.2 Quantum State of the System

The quantum state of a network is embodied in the local satate of a special type of agent called *Environment*. We divide the quantum state σ of the whole system into the smallest possible subsystems, which are in a well-defined state. The state of the system can be then expressed as the tensor product of states of these subsystems.

$\emptyset$ $\emptyset$	Actions	mip_n, mim_n
	Rules	pc = n : {mip_n, mim_n};
s $\emptyset$	Actions	mi_s0p_n, mi_s0m_n, mi_s1p_n, mi_s1m_n
	Rules	pc = n and s = zero : {mi_s0p_n, mi_s0m_n}; pc = n and s = one : {mi_s1p_n, mi_s1m_n};
$\emptyset$ t	Actions	mi_t0p_n, mi_t0m_n, mi_t1p_n, mi_t1m_n
	Rules	pc = n and t = zero : {mi_t0p_n, mi_t0m_n}; pc = n and t = one : {mi_t1p_n, mi_t1m_n};
s t	Actions	mi_s0_t0p_n, mi_s0_t0m_n, mi_s0_t1p_n, mi_s0_t1m_n, mi_s1_t0p_n, mi_s1_t0m_n, mi_s1_t1p_n, mi_s1_t1m_n
	Rules	pc = n and s = zero and t = zero : {mi_s0_t0p_n, mi_s0_t0m_n}; pc = n and s = zero and t = one : {mi_s0_t1p_n, mi_s0_t1m_n}; pc = n and s = one and t = zero : {mi_s1_t0p_n, mi_s1_t0m_n}; pc = n and s = one and t = one : {mi_s1_t1p_n, mi_s1_t1m_n};

Table 4.1: Actions and protocol rules for various degree of dependency of measurements.

We represent qubits and quantum registers in ISPL as local (non-observable) variables of enumeration type of the *Environment* agent. The enumeration values of a variable corresponds to a unique state of a qubit or a register associated with the variable. This is the raison d'être of `dmc2ispl` compiler. Quantum states are described by complex vectors and `MCMAS` cannot deal with these. Therefore the compiler generates reachable quantum state space of the system and enumerates encountered states with names in form `state$`, where `$` is a unique natural number.

The smallest possible unit is a single qubit. The subsystem of several entangled qubits is called a quantum register, though we may sometimes refer to a single qubit as a register and we use terms register and system interchangeably.

Because qubits are basic construction blocks of the system, every qubit i has associated following variable in the *Environment*

```
qi : {undefined, state$,..., state$};
```

where `undefined` is a special case indicating that the qubit is not in a well-defined local state but it is entangled with other qubits. `state$,..., state$` is a subset of all enumeration values. Therefore different qubits may have the same enumeration values in their enumeration types.

Quantum registers have also associated variables in the *Environment* but only if they are actually encountered in a run of the protocol. The name of the variable is in form `e_i_j_...`, which denotes that the register consists of qubits i, j, ... The following line defines the register

```
e_i_j_... : {undefined, state$,..., state$};
```

where `undefined` has a slightly different meaning than before. The state of the register may be defined as the tensor product of its constituent substates, however we want to stress that the global state is divided into the *smallest* possible subsystems, which are in a well-defined state. This implies that each qubit is part of precisely one register at any given time. More precisely, if $\mathcal{R}$ is a set of all registers and qubits in a well-defined state and S the computation space then

$$\bigcup_{r_k \in \mathcal{R}} r_k = S, \qquad \forall r_k, r_l \in \mathcal{R} : r_k \neq r_l \Rightarrow r_k \cap r_l = \emptyset.$$

Additionally, we employ an auxiliary set of enumeration variables. Each qubit has associated a variable which records the initial state of the qubit. This enables comparing states of qubits in later stages of a protocol with the initial one. For qubit i we define

```
qi_ :  {state$};
qi_ :  {undefined};
```

depending on whether the qubit is isolated or entangled at the beginning.

This brings us to the initial quantum state of the system before we move to its dynamics. Every qubit and every defined quantum register has its initial state defined in `InitStates` section of ISPL (e.g `qi = undefined$`, `e_i_j_...= state$`). Variables `qi_` do not need explicitly declared initial state because they attain only one value.

4.1.2.1 Dynamics of Quantum States

In this section we describe the translation of transition system of the global quantum state. Dynamics of the system is defined by evolution functions of *Environment* agent, and depends on computation patterns of normal agents.

An important issue here is the fact that DMC has *probabilistic* transition system but `MCMAS` does not deal with probabilities. However, this is not as problematic as it may seem because we are not interested in the probability of being in some state but rather in the truth of logic formulae in this state. So, we only need to prevent reaching impossible states. It would not be very useful if verification failed because of situation that can never occur. As it turns out solution to the obstacle exists.

Before we explain the respective transitions we make two more remarks about the rules. First, rule 2.7 is defined on patterns. Execution of a pattern occurs in a single transition step and depends on the big-step semantics of the pattern. But we handle transitions at the level of individual commands rather then full patterns, so that execution of a pattern is decomposed into its command sequence and relays on the small-step semantics of MC. We modify the rule because we want configurations to be more dense so we can examine properties of the MAS in the middle of execution of a pattern. However we provide the modeller with a possibility to declare whole patterns as macros, in which case the behaviour is identical to the rule.

Second remark concerns rule 2.10. The rule states that the order, in which agents execute their local operations, does not matter. However, we assume that quantum commands that happen in the same time step are ordered. This is because we would have to consider all possible combinations of actions on entangled registers and this would potentially lead to massive branching of the reachable quantum state space. Therefore, we order the sequence of commands in the order of declarations of the agents (e.g. if agent **A** is declared before agent **B** then **A** always acts ahead of **B**).

As a result of the smoother transitions rule, we have only five possible transitions. All quantum commands except measurements are deterministic and only modify the quantum part of the state. The measurement actions are not deterministic and modify both the quantum and classical parts of the state. We start with entanglement.

The small-step rule for entanglement in MC is given as $|\psi\rangle, \Gamma \xrightarrow{E_{ij}} {}^{C}Z_{ij}\,|\psi\rangle, \Gamma$. Since we divide the global state $|\psi\rangle$ into its smallest well-defined substates, we have two possible cases. We first consider the scenario whit $i \in$ *R1* and $j \in$ *R2* where *R1* and *R2* are either isolated qubits i and/or j only or entangled registers. The resulting subsystem *R3* is then union of the two registers. We define the evolution function of *Environment* agent as

```
gc = gc + 1 and R1 = undefined and R2 = undefined and R3 = state$
            if R1 = state$ and R2 = state$ and A.Action = eixj_n;
```

where we omit register *R3* from the enabling condition because it can be derived by tensoring current states of registers *R1* and *R2*, and so the register is never defined by our definition of

the global quantum state. If qubits i and j are already part of the same entangled register *R1* we simply have

```
gc = gc + 1 and R1 = state$ if R1 = state$ and A.Action = eixj_n;
```

Even though entanglement does not change the local state $\Gamma_{\mathbf{A}}$ nor qubit reference set $O_{\mathbf{A}}$, it changes knowledge the agent has about qubits i and j, we explain what this means in section 4.1.3.1 We also remove the command from agent's event sequence by incrementing her *program counter*. All put together, we add the following line to the agent's evolution function

```
qi = 1 and qj = 1 and pc = pc + 1 if Action = eixj_n;
```

The correction commands and pattern macros can be again covered together. Their small-step semantics is defined as $|\psi\rangle, \Gamma \xrightarrow{U_i^s} U_i^{s_\Gamma} |\psi\rangle, \Gamma$ where U stands for X, Z or $\mathcal{P}$ depending on the event. This time we have only one situation $i \in$ *R1*. *Environment* agent's evolution is augmented with the following statement

```
gc = gc + 1 and R1 = state$ if R1 = state$ and A.Action = Ui_n;
```

where *U* $\in \{\mathtt{x}, \mathtt{z}, \mathtt{p}\}$. The local state of the agent changes only in the sense of `pc` increment. Because the action depends on some signal from $\Gamma_{\mathbf{A}}$ we must account for action `skip`. The evolution function of agent **A** then contains

```
pc = pc + 1 if Action = Ui_n or Action = skip;
```

Measurement rules, $|\psi\rangle, \Gamma \xrightarrow{t[M_i^\alpha]^s}_\lambda \langle +_{\alpha_\Gamma}|_i |\psi\rangle, \Gamma[0/i]$ and $|\psi\rangle, \Gamma \xrightarrow{t[M_i^\alpha]^s}_\lambda \langle -_{\alpha_\Gamma}|_i |\psi\rangle, \Gamma[1/i]$ define two transitions for the command in MC. The resulting states are left unnormalised, so that the probability of reaching a state can be read off its norm. This is the source of non-determinism in quantum MAS. We are not interested in the respective probabilities as long as they are non-zero. If a resulting probability equals zero then the respective transition cannot happen and we must avoid reaching the state. But first, we consider cases with both transitions possible.

Because measurements may depend on signals, there are four types of the operation. However, they differ only in the name as far as the translations rules are concern. The difference is in the computation of quantum states and since we use generic enumerations of these (`state$`), we describe the rules only for independent measurements. The rest is translated analogously. In the first scenario, agent **A** measures isolated qubit i. The two possible outcomes are represented by two actions, `mip_n` and `mim_n`, which are selected non-deterministically. We express the evolution as follows

```
gc = gc + 1 and qi = state+ if qi = state$ and A.Action = mip_n;
gc = gc + 1 and qi = state- if qi = state$ and A.Action = mim_n;
```

where `state+` and `state-` are enumerations of $\{|+_\alpha\rangle, |-_\alpha\rangle\}$ measurement basis. If the qubit i is part of entangled register *R3* then the register disintegrates on measurement of the qubit. The qubit collapses to a base state and the rest of the qubits will form a new register *R2*. The state of the *R3* becomes therefore undefined. The following two lines are added to the evolution function of the *Environment* agent

```
gc = gc + 1 and qi = state+ and R2 = state$ and R3 = undefined
            if R3 = state$ and A.Action = mip_n;
gc = gc + 1 and qi = state- and R2 = state$ and R3 = undefined
            if R3 = state$ and A.Action = mim_n;
```

The measurement outcome is assigned to the signal variable of agent **A**. The evolution of the agent's local state is identical for both above cases because the outcome must correspond to the collapsed state and it does not matter if the qubit is entangled in the current state or not. Also, **A** gains some knowledge about the qubit by measuring it. All this is captured in the following evolution function of the agent

```
s = zero and qi = 2 and pc = pc + 1 if Action = mip_n;
s = one and qi = 2 and pc = pc + 1 if Action = mim_n;
```

If a transition has zero probability then the quantum state of the system is such that only one measurement outcome is possible. We can deal easily with evolution of the quantum state of the system by changing the enabling condition as follows (we show the case of $|+_\alpha\rangle$ state, $|-_\alpha\rangle$ state is done similarly)

```
gc = gc + 1 and qi = state+ if qi = state$
            and (A.Action = mip_n or A.Action = mim_n);
```

if qubit i is isolated and

```
gc = gc + 1 and qi = state+ and R2 = state$ and R3 = undefined
            if R3 = state$ and (A.Action = mip_n or A.Action = mim_n);
```

otherwise.

The problem is with matching the outcome assignment to the agent's signal variable because normal agents cannot access local variables of the *Environment* agent, unless they are declared as local observable variables. But in this case these variables are part of the local states of the agents which is not desirable since agents cannot observe quantum states. To overcome this obstacle we let the *Environment* "signal" that a measurement of qubit $i \in$ `R1` executed in configuration C_n has only one possible outcome by introducing action `envn` and adding the line

```
gc = n and R1 = state$ :  {envn };
```

to the protocol function of the *Environment*. Assuming that $|+_\alpha\rangle$ is the only possible state then evolution of the local state of the agent is defined

```
s = zero and qi = 2 and pc = pc + 1 if Action = mip_n
         or (Action = mim_n and Environment.Action = envn);
s = one and qi = 2 and pc = pc + 1 if Action = mim_n
        and !Environment.Action = envn;
```

Many commands may be executed concurrently by the agents because quantum MAS are parallel. Some of the operations may have only local effects but those on entangled resources change the state of the same register at one time step. To avoid ambiguity, we use `MultiAssignment` semantics of evolution functions in ISPL. So in general, the set of assignments of states to registers is union of LHS of the rules applicable in a given time step and the enabling condition is conjunction of RHS of these rules. For instance, assume **A** applies E_{ij} on qubits $i, j \in$ `R1` and **B** preforms X_k^s on qubit $k \in$ `R2`. Then the evolution function is

```
gc = gc + 1 and R1 = state$ and R2 = state$ if R1 = state$
            and R2 = state$ and A.Action = eixj_n and B.Action = xi_m;
```

Furthermore, `gc` needs to be incremented even if no action in a given time step changes the quantum state of the system. In such case the enabling condition of the evolution function consists of a formula defined over one of the following action: classical receive, quantum receive or skip. We group all conditions that occur in the run of the protocol into one disjunction.

```
gc = gc + 1 if A.Action = rcv_B_y or A.Action = qrcv_B_i or
               A.Action = skip;
```

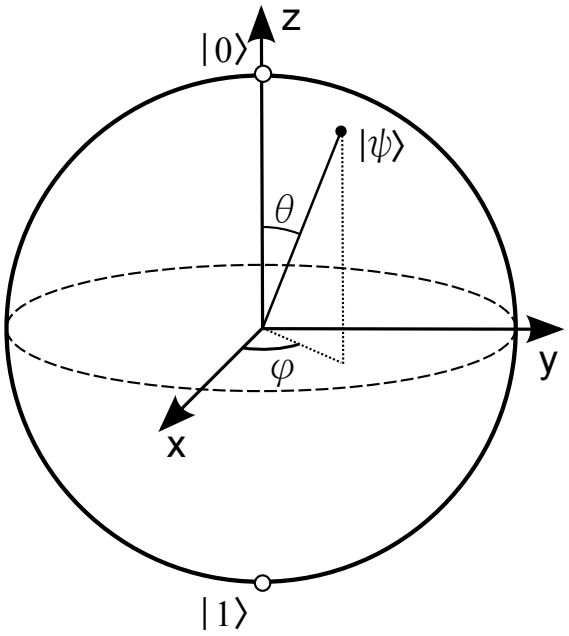

Figure 4.1: Bloch sphere.

4.1.2.2 Global Phase Factor

The only measurable quantities of a state are the probabilities $|\alpha|^2$ and $|\beta|^2$. So multiplying the state by an arbitrary factor $e^{i\gamma}$ has no observable consequences because

$$|e^{i\gamma}\alpha|^2 = (e^{i\gamma}\alpha)^*(e^{i\gamma}\alpha) = (e^{-i\gamma}\alpha)(e^{i\gamma}\alpha) = \alpha^*\alpha = |\alpha|^2,$$
$$|e^{i\gamma}\beta|^2 = (e^{i\gamma}\beta)^*(e^{i\gamma}\beta) = (e^{-i\gamma}\beta)(e^{i\gamma}\beta) = \beta^*\beta = |\beta|^2.$$

The factor $e^{i\gamma}$ is called a global phase. We illustrate the concept on the Bloch sphere (see figure 4.1). It is a geometric representation of a qubit state as a point on the surface of the unit sphere in $\mathbb{R}^3$. Although the qubit is represented by a vector in $\mathbb{C}^2$, its insensitivity to a global phase change means that certain rotations have no observable effects and we can visualise the qubit in $\mathbb{R}^3$.

A generic qubit of the standard form $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ can be rewritten in the polar form $|\psi\rangle = r_\alpha e^{i\phi_\alpha}|0\rangle + r_\beta e^{i\phi_\beta}|1\rangle$, where $r_\alpha, r_\beta, \phi_\alpha, \phi_\beta \in \mathbb{R}$ are parameters. Using the normalization constraint $|r_\alpha e^{i\phi_\alpha}|^2 + |r_\beta e^{i\phi_\beta}|^2 = 1$, we get $r_\alpha^2 + r_\beta^2 = 1$, and so we can replace r_α, r_β with a single parameter $\theta \in \{0, \pi\}$ such that $r_\alpha = \cos(\frac{\theta}{2})$ and $r_\beta = \sin(\frac{\theta}{2})$. If we set $\gamma = \phi_\alpha$ and $\phi = \phi_\beta - \phi_\alpha$ then the qubit equation simplifies to $|\psi\rangle = e^{i\gamma}(\cos(\frac{\theta}{2})|0\rangle + e^{i\phi}\sin(\frac{\theta}{2})|1\rangle)$. Finally, since the qubit is invariant to the global phase factor $e^{i\gamma}$ in the front, we can safely ignore it to get the canonical representation

$$|\psi\rangle = \cos(\frac{\theta}{2})|0\rangle + e^{i\phi}\sin(\frac{\theta}{2})|1\rangle,$$

with only two real parameters $\theta \in \{0, \pi\}$ and $\phi \in \{0, 2\pi\}$.

We can imagine the global phase as a rotation of the qubit around its own axis, e.g. for two special states at the north and south poles[1] we have $|\psi\rangle = e^{i\phi}|0\rangle$ and $|\psi\rangle = e^{i\phi}|1\rangle$ respectively, and so ϕ is a global phase with no significance. In other words, rotating a bit around z-axis is meaningless and intangible.

[1]Note that opposite points correspond to orthogonal qubit states.

Since a state of the system is invariant to a global phase factor, we do not differentiate between states that differ only in the global phase. A unique name is therefore assigned to the whole equivalence class of quantum states rather than to the particular state. For example states $|0\rangle$, $-|0\rangle$, $i|0\rangle$ and$(\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}i)|0\rangle$ would all have identical name.

It is not explicitly said in the reference paper [16] but from the sequence of configuration of quantum teleportation protocol on page 9, we can assume that the authors adopt the same approach because in case of

$$C_3^{11}(|\psi\rangle) = X^1 Z^1 |\psi\rangle ; [s_1 s_2 \mapsto 11], \mathbf{A} \mid [x_1 x_2 \mapsto 11], \mathbf{B} : \{3\}.X_3^{x_2} Z_3^{x_1},$$

the quantum state of the system is $XZ|\psi\rangle$ and agent **B** performs local operations $X_3 Z_3$, which gives the final state $XZXZ|\psi\rangle = -|\psi\rangle$. Therefore, if the states $|\psi\rangle$ and $-|\psi\rangle$ were treated as two different states then the network, as defined in the paper, would not realise the quantum teleportation.

4.1.2.3 General and Mixed States

General states in form $\alpha|0\rangle + \beta|1\rangle$ and their evolution are not possible to verify at the moment on two accounts. First, there is a prohibitive overhead in dealing with equality of general states such as $\alpha_0|0\rangle + \beta_0|1\rangle$ and $\alpha_1|0\rangle + \beta_1|1\rangle$. The atomic propositions would have to be extended to incorporate analytical solutions to these equations, and the enumeration of states would have to be on the level of coordinates rather than registers. This would mean enormous increase in the number of variables (e.g. we need only one variable for a ten-qubit entangled system but 1024 variables for its coordinates).

The second reason is that our chosen tool for quantum computations, `GNU Octave`, does not support symbolic mathematical expressions and variable-precision arithmetic. Although `MATLAB` includes a toolbox for symbolic computing, we felt that its disadvantages outweigh this feature, especially when we were not going to implement generic states. We debate the selection of tools in section 5.1.

As a workaround, we allow the modeller to use arbitrary quantum states instead. These are random normalised complex vectors that cannot be a result of an evolution of the system unless the state is part of the initial global state of the system. After we randomly generate the vector we test whether there is a way of manipulating the other input states using event sequences of the agents so that they are equal to the generated state. If this is possible we try a new state until a unique state is obtained. This way we know that the verification process does not happen by chance.

In the real world, the quantum registers are subject to phenomenon of decoherence. The state of a quantum system loses its purity as the result of entanglement with the environment. A pure state is a well-defined state of a quantum register, it may or may not be entangled. When we take out the register from isolation, the particles that represent qubits tend to couple with other particles in the environment. After a while, the global state of the register is no longer pure but rather becomes a probabilistic mix of pure states, known as a mixed state.

Consider the family of qubits $\frac{1}{\sqrt{2}}(|0\rangle + e^{i\theta}|1\rangle)$ characterised by a specific relative phase θ. If we treat these states as pure we can distinguish between them by measuring the states in $\{|+\rangle, |-\rangle\}$ basis. However, if we take the states as a statistical combination of states $|0\rangle$ and $|1\rangle$ there is no way to discriminate between them because we get $|\psi\rangle$ 50% of the times and $|\phi\rangle$ 50% of the times when measuring the states in any orthogonal $\{|\psi\rangle, |\phi\rangle\}$ basis. The relative phase θ and its associated information is lost.

Our `dmc2ispl` compiler does not currently support mixed quantum states. All states are assumed to be pure. The reason is that measurement projections depend on density operator

$|\psi\rangle\langle\psi|$ associated with a pure state $|\psi\rangle$ whereas a mixed state is characterised by a linear combination of density operators of pure states, called generic density operator. Implementation of measurements of mixed states would increase complexity of reachable state space generator. However, we can add this functionality in the future.

4.1.3 Logic Formulae

In this section we describe the interpretation of logical statements. First, we discuss our approach to quantum knowledge, then we define a set of propositional atoms, and finally we look at modal operators which act on these primitives.

4.1.3.1 Quantum Knowledge

The authors argue that there is no such thing as quantum knowledge and that knowledge is rather about classically knowing facts about quantum systems. This premise allows us to verify multi-agents quantum systems using `MCMAS` model checker which supports standard epistemic logic.

There is however one problem. Even if a state of a qubit is the same in all configurations C' that agent $\mathbf{A}$ considers possible in configuration C, she may still not know the state. This is not compatible with $K_{\mathbf{A}}$ operator

$$C, \mathcal{N} \vDash K_{\mathbf{A}}\varphi \iff \forall C' \sim_{\mathbf{A}} C : C' \vDash \varphi,$$

where atomic proposition φ is defined as

$$C, \mathcal{N} \vDash \varphi \iff q = |\psi\rangle .$$

This is because, from the point of view of the agent, the qubit is in infinitely many possible states and she cannot distinguish between them. However, we cannot deal with infinity in practice. To work around this, we introduce a local variable that indicates the relation of the agent towards her qubits. Concretely in ISPL, all agents have bounded integer variable `qi` for every qubit i in the protocol

```
qi :  0..2;
```

which has the following meaning: Value 0 indicates that the agent is not in possession of the qubit i and so knows nothing about it. Value 1 means that she owns the qubit but still has no knowledge of its state. Finally, value 2 denotes that the agent owns and knows the qubit.

So, whenever we want to make a statement involving knowledge of agent $\mathbf{A}$ about quantum state of qubit i in a configuration C, we have to refine the atomic proposition φ as follows

$$C, \mathcal{N} \vDash \varphi \iff q = |\psi\rangle \wedge \Gamma_{\mathbf{A}}(q_i) = 2.$$

There are only two ways of getting knowledge about a state of a qubit. The agent either prepares the qubit in that state or she measures it. On the other hand, the agent can lose the knowledge. When she sends the qubit through a quantum channel to another agent she cannot know what happens to it. And when the agent entangles the qubit with another qubit, the state of the qubit is no longer in a well-defined state.

No other operation on the qubit changes the epistemic state of the agent towards the qubit because we assume that she knows what the action does to the state of the qubit and she remembers it. In fact, the agent may trace the state of a qubit even after she entangles it and when the qubit becomes isolated again she may know the state. But this would involve tracing

evolution of all entangled qubits as well as their epistemic states just before entanglement and then determine if the agent still knows the state. We simplify the situation by saying that any entanglement cancels the agent's knowledge of the qubit.

Initial value of `A.qi` depends on circumstances of agent **A** and qubit i. If the qubit is not in possession of the agent at the beginning of a protocol then `A.qi = 0`. If **A** prepares the qubit in $|+\rangle$ state then `A.qi = 2`. We assume that all necessary preparations of qubits happen before execution of the protocol. The user can also explicitly specify that the agent knows the state of the qubit in her input sort for whatever reason, in which case `A.qi = 2` again. If the qubit is in the input sort but unknown then `A.qi = 1`.

4.1.3.2 Propositional Atoms

We now consider all atomic propositions p available to the modeller. The DMC does not have a full-fledged language for primitive propositions and atoms are defined in an ad-hoc manner. Propositional atoms in ISPL are defined over global states in `Evaluation` section as Boolean variables. Each variable is associated with Boolean expression over the local states of the agents and the Environment. An atom evaluates to true in all global states that satisfy the formula.

The primitives are currently limited to the following expressions but more atoms can be added if necessary

$$C,\mathcal{N} \vDash (\mathbf{A}_i(x) = v) \Longleftrightarrow \Gamma_i(x) = v \tag{4.1}$$
$$C,\mathcal{N} \vDash (\mathbf{A}_i(x) = \mathbf{A}_j(y)) \Longleftrightarrow \Gamma_i(x) = \Gamma_j(y) \tag{4.2}$$
$$C,\mathcal{N} \vDash (\mathbf{A}_i \text{ has } q) \Longleftrightarrow q \in Q_i \tag{4.3}$$
$$C,\mathcal{N} \vDash (\mathbf{A}_i\ [=<>]n) \Longleftrightarrow \mathcal{E}_{i,n}[=<>]\mathcal{E}_{i,C} \tag{4.4}$$
$$C,\mathcal{N} \vDash (q_i = q_j) \Longleftrightarrow \exists\, |\psi\rangle\, .q_i = q_j = |\psi\rangle \tag{4.5}$$
$$C,\mathcal{N} \vDash (q_i = \ \text{init}(q_j)) \Longleftrightarrow \exists\, |\psi\rangle\, .q_i = \ \text{init}(q_j) = |\psi\rangle \tag{4.6}$$
$$C,\mathcal{N} \vDash (q = |\psi\rangle) \Longleftrightarrow q = |\psi\rangle \tag{4.7}$$
$$C,\mathcal{N} \vDash (q = |\psi\rangle) \Longleftrightarrow q = |\psi\rangle \wedge \Gamma_{\mathbf{i}}(q) = 2 \tag{4.8}$$
$$C,\mathcal{N} \vDash (q) \Longleftrightarrow \exists\, |\psi\rangle\, .q = |\psi\rangle \wedge \Gamma_{\mathbf{i}}(q) = 2 \tag{4.9}$$

The first two expressions concern classical variable value and variable equality. The next atom is about qubit ownership and the fourth proposition refers to local operations. The rest of the primitives are genuine quantum statements.

We now describe the specification of these atomic propositions in ISPL. First, we assign to each atom a Boolean variable with unique name in the form `f$`, where `$` is a non-negative integer, and then we define the Boolean expression for the proposition.

Equation 4.1 means that classical variable x of agent **A** is equal to value v, which can be either 0 or 1. The translation into ISPL is as follows

```
f$ if A.x = v;
```

Similarly, the translation of equation 4.2, which expresses that classical variable x of agent **A** is equal to classical variable y of agent **B**, is

```
f$ if A.x = B.y;
```

However, some variables may be uninitialised until a value is assigned to them during the protocol run. Since equality of two undefined values has no meaning, we use the following expression in case that `undefined` is an enumeration value of variable y of agent **B**.

```
f$ if A.x = B.y and !B.y = undefined;
```

The equation 4.3 has an obvious connotation. Although we change it slightly from its original form in DMC, where the proposition is true if and only if the qubit is in the output sort of the agent. We formulate the atom as follows

```
f$ if A.qi = 1 or A.qi = 2;
```

which is true in all global states where agent **A** has or knows qubit i.

The next equation 4.4 concerns the *program counter* of agent A. We may be interested to know what properties hold in certain stage of the protocol and this primitive facilitate this. Three logical operators =, < and > denote global states during, before or after execution of event n by agent **A**

```
f$ if A.pc = n;
f$ if A.pc < n;
f$ if A.pc > n;
```

The first quantum statement is represented by equation 4.5. It states that two qubits i and j have identical states, though the actual state they are in may not be known to the agents. In the following translation

```
f$ if Environment.qi = Environment.qj and !Environment.qi = undefined;
```

we again cannot compare qubits that are not in well-defined local states, i.e., they are part of an entangled quantum system. The last condition prevents this from happening.

The equation 4.6 is almost identical to the previous one. Only this time we compare the quantum state of qubit i in configuration C with the quantum state of qubit j in the initial configuration C_0

```
f$ if Environment.qi = Environment.qj_ and !Environment.qi = undefined;
```

The next two equations 4.7 and 4.8 are about the same thing, the exact state of the qubit, however they differ in their usage. As we argue in section 4.1.3.1, the concept of quantum knowledge enforces a different approach to knowledge modalities. Therefore this atomic proposition has an alternative if it is part of an epistemic formula. In all other cases we define it as follows

```
f$ if Environment.qi = state$;
```

where `state$` is enumeration value of the quantum state. In epistemic formulae we distinguish three cases depending on the epistemic operator

```
f$ if Environment.qi = state$ and A.qi = 2;
f$ if Environment.qi = state$ and (A0.qi = 2 or ... or An.qi = 2);
f$ if Environment.qi = state$ and (A0.qi = 2 and ... and An.qi = 2);
```

which correspond respectively to $K_{\mathbf{A}}$, D_G and E_G/C_G. In the last two statements regarding group modalities, agents $\mathbf{A}_0 \ldots \mathbf{A}_n$ are members of group **G**. The reason why we discriminate between the cases is that every agent has her own epistemic relation towards the qubit, which reflects in notions of distributed and common knowledge.

The last equation 4.9 has meaning only in epistemic context, and so it must be used immediately after epistemic operator. It states that agent **A** (or members $\mathbf{A}_0 \ldots \mathbf{A}_n$ of **G**) knows the state of the qubit but the actual state is not important. Again, there are three expressions for the same reason as before.

```
f$ if A.qi = 2;
f$ if A0.qi = 2 or ... or An.qi = 2;
f$ if A0.qi = 2 and ... and An.qi = 2);
```

4.1.3.3 Temporal Epistemic Logic

The translation of formulae defined over atomic propositions is straightforward. In fact, we rather unify notation then do any conversion. The formulae in DMC are constructed using logical connectives, epistemic operators and CTL temporal operators, which is extremely convenient since `MCMAS` has been particularly built to support these. The formulae are defined in `Formulae` section of ISPL.

DMC	ISPL	Meaning	DMC	ISPL	Meaning
$K_{\mathbf{A}}\varphi$	`K(A, f$)`	"agent **A** knows φ"	$A\Box\varphi$	`AG f$`	"all paths globally"
$E_{\mathbf{G}}\varphi$	`GK(G, f$)`	"everyone in **G** knows φ"	$E\Box\varphi$	`EG f$`	"exists path globally"
$C_{\mathbf{G}}\varphi$	`GCK(G, f$)`	"common knowledge"	$A\Diamond\varphi$	`AF f$`	"all paths finally"
$D_{\mathbf{G}}\varphi$	`DK(G, f$)`	"distributed knowledge"	$E\Diamond\varphi$	`EF f$`	"exists path finally"

Table 4.2: Basic epistemic and temporal operators.

The elementary modalities are reviewed in table 4.2. ISPL supports some more operators, and so does `dmc2ispl` compiler, which translates all CTL, ATL and epistemic operators available in `MCMAS`.

4.2 DMC Input Format

Our adaptation of DMC provides a standardised input format for describing quantum multi-agent systems. We aim to emulate the syntax of the original DMC as closely as possible but some adjustments are necessary to create a modular language, so that our `dmc2ispl` can read it. Also, some features of ISPL are inevitably reflected in the language.

In this section, we clarify our choice for the syntax and the semantics of the compiler's input language. Fundamentally, a DMC file consists of five modules:

- `Definitions` allow the user to define a pattern (a set of local quantum operations) which can be later used in an agent's event sequence as just one command. This enables agents to do more complex quantum operations in one time step, and also substitutes in some degree if-else statement. This module is optional.

- `Agents` is a set of participating agents. Declaration of each agent consist of her input qubits, prior quantum knowledge of these qubits, classical inputs and event sequence. At least one agent must be defined in a valid DMC program.

- `Qubits` is a set of qubits, of which initial states are explicitly declared. A qubit can be declared either in an arbitrary state or in some concretely defined state. Qubits, that are used in the protocol but do not appear here, are assumed to be initialised in $|+\rangle$ state. This module is optional.

- `Groups` is used if there are formulae involving group modalities and it has exactly the same format as in ISPL. This module is optional.

- `Formulae` contains a set of formulae to be verified. All modal operators available in `MCMAS` are supported and the format is very similar to the corresponding section of ISPL. The difference is in handling of atomic propositions. Unlike ISPL, these are defined in formulae themselves. At least one formula must be defined in a valid DMC program.

There are some common features across the modules such as identifiers. We employ two types of identifiers. One for agents, classical variables, patterns and groups, the other for qubits. The latter is just an enumeration of qubits by natural numbers, the former is alphanumeric.

A valid alphanumeric identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid and identifiers always have to begin with a letter. In addition, identifiers cannot match any keyword of the ISPL language nor the compiler's specific ones. The reserved keywords are:

`A`, `AF`, `AG`, `Action`, `Actions`, `Agent`, `and`, `AX`, `boolean`, `cos`, `cX`, `cZ`, `DK`, `E`, `EF`, `EG`, `En`, `end`, `Environment`, `Evaluation`, `Evolution`, `EX`, `exp`, `F`, `Fairness`, `false`, `fin`, `Formulae`, `has`, `G`, `GCK`, `GK`, `GreenStates`, `Groups`, `if`, `init`, `InitStates`, `K`, `ln`, `Lobsvars`, `MA`, `Me`, `MultiAssignment`, `O`, `Obsvars`, `or`, `Other`, `Protocol`, `RedStates`, `SA`, `Semantics`, `sin`, `SingleAssignment`, `sqrt`, `true`, `U`, `Vars`, `X`.

Also, our DMC language is case sensitive. That means that an identifier written in capital letters is not equivalent to another one with identical name but written in small letters.

Measurement angles and complex coordinates of vectors representing quantum states require the user to type in real numbers. In our language, a real number is a mathematical expression, which may consist of decimal numbers, constants (`pi`, `e`), arithmetic operations (`+`, `-`, `*`, `/`, `^`) and functions (`sin`, `cos`, `exp`, `ln`, `sqrt`). For instance, 23, -0.4, $cos(pi/8)$ and $exp(sqrt(e+1))$ are all valid real numbers. Because exponential form of complex numbers is not implemented in our language, the user has to rewrite the coordinates in terms of trigonometric functions when such a form is needed.

Additionally, we support the capability for source code documentation, as present in most programming languages, to allow the user to include inline comments to document her protocols. The comments start with the double hyphen (--) as a comment delimiter and continue until the end of the line.

For demonstration purposes, we refer to listing 4.1, which presents a description of a simple protocol. In this scenario, *Alice* applies unitary operator $R_x(\frac{\pi}{4})$ on her qubit, which is originally in $|0\rangle$ state. This action rotates the qubit about the $x-$axis by angle $\alpha = \frac{\pi}{4}$. The example uses all five modules of the language and we look at all of them in detail.

4.2.1 Agents Module

Even though this is not the first module in the sequence, it is the most important one. Here are defined all the agents as well as their classical and quantum inputs and their event sequences.

```
1  -- DEFINITIONS
2  #Rx: {1},
3       {3},
4       {cX(3,s2),cZ(3,s1),Me(2,7/4*pi,s1,-,s2),Me(1,0,-,-,s1),En(2,3),En(1,2)};
5
6  -- AGENTS
7  Alice: {1},
8         {1},
9         {x: 1},
10        {Rx(1,x)};
11
12 -- QUBITS
13 1: {(1,0), (0,0)};
14
15 -- GROUPS
16 Imperial = {Alice};
17
18 -- FORMULAE
19 AG ({Alice >1} -> GK(Imperial,
20     {1 = {
21          (cos(pi/8), 0          ),
22          (0        , -sin(pi/8))
23     }}));
```

Listing 4.1: Code listing showing the DMC file for the x-rotation.

We start with the abstract grammar of the module.

$$
\begin{aligned}
\text{agents} &::= \text{agents agent} \mid \text{agent} \\
\text{agent} &::= \textit{ID} : \{ \text{ qubits } \} , \{ \text{ qubits } \} , \{ \text{ bits } \} , \{ \text{ events } \} ; \\
\text{qubits} &::= \text{qubits} , \textit{qubit} \mid \text{nil} \\
\text{bits} &::= \text{bits} , \text{bit} \mid \text{nil} \\
\text{events} &::= \text{events} , \text{event} \mid \text{nil} \\
\text{bit} &::= \textit{ID} : \textit{value} \\
\text{event} &::= \text{sendBit} \mid \text{receiveBit} \mid \text{sendQubit} \mid \text{receiveQubit} \mid \text{pattern} \mid \\
&\qquad \text{entanglement} \mid \text{correctionX} \mid \text{correctionZ} \mid \text{measurement}
\end{aligned}
$$

where *ID* is an identifier of the agent and a classical input bit respectively, *value* is the value 0, 1 or ? of the bit and *qubit* is an identifier of a qubit. The first list of qubits refers to the agent's input sort, the second list are qubits in a state a priori known to the agent, the third set consists of input classical bits of the agent and the last set defines the agent's event sequence composed of communication events and basic quantum commands.

- The first operation is a communication via a classical channel during which the agent sends value of a bit to another agent. The formal definition is `sendBit` ::= *c!* (*agentID* , *bitID*), where *agentID* is a valid identifier of the receiver and the bit must be initialised to value 0 or 1 at the time of sending. This means the bit is either an input bit of the sender, or an outcome of some prior measurement, or the agent receives the value from someone else.
- The opposite action is the reception of a classical message and both actions have to be paired together. The event is specified as `receiveBit` ::= *c?* (*agentID* , *bitID*), where *agentID* indicates the sender and *bitID* designates the variable to which is the received value assigned.

- The other pair of communication events consists of sending a qubit via quantum channel, formally `sendQubit` ::= *qc!* (*agentID* , *qubit*), where again *agentID* is the receiver and *qubitID* it the qubit to be send with the condition that the sender must be in possession of it at this point.

- The compulsory other half of the quantum communication is the reception of the qubit, defined `receiveQubit` ::= *qc?* (*agentID* , *qubit*). The *qubit* identifier has to refer to the same qubit as the corresponding `sendQubit` operation.

- The first quantum action is a "macro" operation, which we cover in detail in section 4.2.3. It is defined `pattern` ::= *ID* (*qubit* , *bitID*), where *ID* is an identifier of the pattern declared in `Definitions` module, *qubit* refers to the qubit on which the pattern acts and *bitID* identifies the classical variable on which the pattern depends. The agent must have the qubit in her sort and the bit has to be initialised to value 0 or 1. If the value of the bit is 0, the event is skipped.

- The quantum command witch applies the controlled-Z operator on two qubits is specified as `entanglement` ::= *En* (*qubit* , *qubit*), where both *qubit* denote the participating qubits, which must be distinct and in the agent's possession. The order of the qubits does not matter.

- Next are two correction commands which conditionally apply Pauli operators X and Z, defined `correctionX` ::= *cX* (*qubit* , *bitID*) and `correctionZ` ::= *cZ* (*qubit* , *bitID*) respectively. The definitions and conditions are identical to the `pattern` event.

- The last of the events is a quantum command which performs complement orthogonal projections of a qubit on the specified basis. Its formal abstract grammar is defined as `measurement` ::= *Me* (*qubit* , *angle* , *signalS* , *signalT* , *bitID*), where *qubit* is an identifier of the qubit on which the measurement acts, *angle* is the measurement angle in radians, *signalS* and *signalT* are classical variables on which the operator depends, if any, and *bitID* refers to the bit to which is the measurement outcome assigned. There are several requirements. The agent must be in possession of the qubit, the measurement angle must be a real number in $[0, 2\pi]$, and if the operator depends on a signal the corresponding bit has to be initialised to either 0 or 1. Otherwise it has to be indicated that the signal is irrelevant by hyphen (-).

The `Agents` module is compulsory, so at least one agent has to be defined. It would make little sense to have a MAS without agents but protocols without qubits are perfectly valid. In the definition of an agent, any or all of the agent's four sets can be left empty. However, at least one agent must have non-empty event sequence.

There are some other constraints and remarks we need to consider.

- All qubits in the agent's input sort must have their initial state explicitly declared in the `Qubits` module which we describe in section 4.2.2. Set of the known qubits must be subset of qubits in the agent's input sort. In addition, all qubits implicitly prepared in state $|+\rangle$ by the agent are known to her because it is her conscious action.

- For implementation purposes and clarity, we require that all qubits that appear anywhere in the protocol, apart from the `Definitions` module, are enumerated continually, i.e. their identifiers are consecutive natural numbers beginning with one.

- If a classical input bit has value '?' it does not mean that it is not initialised but rather that both 0 and 1 are possible initial values, and so we consider two initial local states of the agent. One with the value 0 assigned to the variable and the other with value 1.

- Because the communication events must be paired, an agent, who wants to exchange information with another agent, waits until the other executes the opposite operation and the communication channel can be opened. The user needs to be careful with specification of the protocol because this obviously may lead to deadlocks when two or more agents are "busy waiting" for each other. Moreover, an agent cannot send nor receive a message from herself.

Also, note that we use commands `cX` and `En` instead of X and E as in DMC developed in paper [13] because these denote modal operators in ISPL. We change notation to `cZ` and `Me` from Z and M for consistency.

We now investigate the `Agent` module of our example. The appropriate snippet is given in listings 4.2. In this simple protocol, we define just one agent and we call her *Alice*. She has one qubit in her input sort and she knows the exact state of it. *Alice* also has a classical variable x initialised to value 1. We describe her event sequence on line 4 and the only action in it is the Rx pattern. She applies the pattern on qubit *1* unconditionally because the value of x is fixed.

```
1 Alice: {1},
2        {1},
3        {x: 1},
4        {Rx(1,x)};
```

Listing 4.2: The `Agents` module of the DMC file for the x-rotation.

4.2.2 Qubits Module

The `Qubits` module contains declarations of quantum states of qubits in the input sorts of agents. All these qubits have to have their initial states explicitly defined here. All other qubits, that occur in the protocol, are implicitly initialised in $|+\rangle$ state. A qubit may not be in a well-defined local state but it can be a part of an entangled quantum register instead. The module is omitted if there are no input qubits in the protocol. We establish the following grammar,

$$\begin{aligned} \text{registers} &::= \text{registers register} \mid \text{register} \\ \text{register} &::= \text{qubits} : \{ \text{coordinates} \} ; \mid \textit{qubit} : ? ; \\ \text{qubits} &::= \text{qubits} , \textit{qubit} \mid \textit{qubit} \\ \text{coordinates} &::= \text{coordinates} , \textit{coordinate} \mid \textit{coordinate} \end{aligned}$$

where *qubit* is an identifier of a qubit and *coordinate* is a complex number. Complex number is an ordered pair of two real numbers. The first represents the real part and the second represents the imaginary part.

The coordinates specify a vector which realise the quantum state with respect to standard basis $\{|0\rangle, |1\rangle\}$. The length of the vector has to be proportional to the number of qubits in the register. Specifically, if the register consists of n qubits the representing vector has dimension 2^n and must have as many coordinates. Moreover, the sum of the squares of the absolute values of the coordinates must equal one because these represent probabilities of corresponding measurement results.

The quantum state of two or more qubits should be entangled and not separable, otherwise the register should be broken into subsystems in well-defined local states. This is desirable because not only the evolution of smaller registers requires exponentially less computational resources but also we can investigate the properties of the subsystems individually and not just as a unit.

The special case is when we are not interested in an exact state of a qubit. We can express that a single qubit is in an arbitrary state by typing '?' in place of the complex vector. This, however, is not possible for quantum systems of two or more entangled qubits, which have to be declared exactly, because an arbitrary state of such system is not necessarily entangled.

In our example shown in listings 4.3, we have only one quantum register ant it comprises of just one qubit, *1*, which we initialise to $|0\rangle$ state.

```
1 1: {(1,0), (0,0)};
```

Listing 4.3: The `Qubits` module of the DMC file for the x-rotation.

4.2.3 Definitions Module

This module allows the modeller to declare computation patterns in the standard EMC form and to use them later in the protocol. The effect of applying a pattern on a qubit is almost the same as applying the whole set of basic commands, which define the pattern, on the qubit. At the moment only patterns which implement some single qubit unitary operator are supported.

The differences between using a pattern and using its defining commands directly are that qubits used in the definition of the pattern do not appear in the translated protocol, they are thought of as auxiliary qubits. Also, the measurement outcomes are not registered in agents' local variables, and it takes just one time step to execute the pattern. But importantly, the resulting quantum states of a qubit are identical, regardless the method applied on it. This imposes several restriction on definitions of the patterns.

- A pattern must realise some unitary operator. This means that the resulting matrix U, which represents the operator, satisfies the condition $U^\dagger U = UU^\dagger = I$ where $U^\dagger$ is the conjugate transpose matrix of U and I is the identity matrix. In other words, the matrix U has an inverse equal to its conjugate transpose matrix $U^\dagger$.

- A pattern must be deterministic. This is to say that the final matrix representing the operator is independent of measurement outcomes, which implies that in all possible executions of the pattern, the output state of a qubit is always the same for a given input state of the qubit. Therefore appropriate corrections must be in place to offset measurement outcomes.

- A pattern must be defined in the EMC form, i.e. its commands must occur in the order of entanglements, measurements and corrections.

- Any signal, on which a correction depends, must be a result of some prior measurement.

- All qubits that appear in a pattern must be identified by consecutive positive integers, although these are completely separate from qubits in the rest of the protocol. The output qubit cannot be measured and all others must be measured at some point. This conditions

are devised to make consistency check and implementation smoother whilst not being prohibitive to functionality[2].

The idea is that we extract the matrix representation of the operator from the matrix representations of the set of the defining commands. There are more than one of these matrices due to measurements. We then use this unitary matrix in computations the same way we use unitary matrices of corrections. There are three main advantages to this approach:

- Conditional execution of more complex operators. Normally, only the basic commands `cX` and `cZ` are dependent on values of classical bits but we may require that a set of commands is executed conditionally. Since agents apply predefined patterns in one time step, we can easily allow for this feature by using the patterns the same way as the two correction commands.

- Reduction of state space. Each correction has a branching factor of two and a measurement of up to eight. Patterns usually consist of several corrections and measurements but their branching factor is just two. Moreover, they are represented by matrices which can be computed before the generation of the state space. Therefore using predefined patterns leads to significant pruning of the tree.

- User convenience. A pattern can be used and reused in a protocol and it is obviously easier to handle use than the whole set of commands that define the pattern. With patterns we achieve more clarity and brevity in the code.

We think that these points justify slight deviation from the original DMC, although the authors themselves sometimes tend to overload and simplify the notation. The module can be summarised by the following formal grammar,

$$\begin{aligned}
\text{definitions} &::= \text{definitions definition} \mid \text{definition} \\
\text{definition} &::= \#\ \mathit{ID} : \{\ \mathit{input}\ \} , \{\ \mathit{output}\ \} , \{\ \text{commands}\ \} ; \\
\text{commands} &::= \text{commands , command} \mid \text{command} \\
\text{command} &::= \text{entanglement} \mid \text{correctionX} \mid \text{correctionZ} \mid \text{measurement}
\end{aligned}$$

where *ID* is an identifier of the pattern, *input* is an identifier of the input qubit, *output* is an identifier of the output qubit and individual commands are as defined in section 4.2.1.

We now explain the above constructs on our exemplary protocol. Listing 4.4 shows the part of the DMC file with the `Definitions` module. The following pattern from the reference paper [15] implements rotation about the $x-$axis by angle α up to a global phase. We remind the reader that the command sequence is read from right to left.

$$R_x(\alpha) := (\{1,2,3\}, \{1\}, \{3\}, X_3^{s_2} Z_3^{s_1} [M_2^{-\alpha}]^{s_1} M_1^x E_{23} E_{12})$$

Our notation is kept almost identical with the MC, although we omit the computation space $\{1,2,3\}$, and instead, we require that the participating qubits are identified by consecutive natural numbers. In the example, we set the angle of rotation, α, to $\frac{\pi}{4}$.

Every pattern definition starts with the hash symbol (#) because we want to visually distinguish the module from the rest of the code. Then comes a unique alphanumeric identifier for the pattern, in this case we call it *Rx*, and we refer to the pattern by this name throughout

[2]We discuss universality in section 2.3.1.4 and it has been shown that any unitary can be implemented with patterns $\mathcal{J}(\alpha) = (\{1,2\}, \{1\}, \{2\}, X_2^{s_1} M_1^{-\alpha} E_{12})$ and ${}^{C}\mathcal{Z} = (\{1,2\}, \{1,2\}, \{1,2\}, E_{12})$, which satisfy these conditions.

```
1 #Rx: {1},
2       {3},
3       {cX(3,s2),cZ(3,s1),Me(2,7/4*pi,s1,-,s2),Me(1,0,-,-,s1),En(2,3),En(1,2)};
```

Listing 4.4: The `Definitions` module of the DMC file for the x-rotation.

the program. The parameter in the first curly brackets is a natural number representing the input qubit of the pattern and the parameter in the curly brackets on line 2 is a natural number representing the output qubit.

The command sequence is specified in the curly brackets on line 3. We cover the detailed syntax of local operations, including the quantum commands, in section 4.2.1 and here we briefly describe this particular sequence. The first command is the entanglement `En` of qubits *1* and *2*, followed by the entanglement `En` of qubits *2* and *3*. Then the qubit *1* is measured `Me` in the basis $\{|+\rangle, |-\rangle\}$ since the angle of the measurement is 0. This measurement does not depend on any signals and its outcome is assign to variable *s1*.

The next measurement `Me` depends on the signal and *s1* has parameter $\alpha = -\frac{\pi}{4}$. Note that, since parameter α has to be in $[0, 2\pi]$, we write "7/4∗pi". Thus, the qubit *2* is measured in the basis $\{|+_{-\frac{\pi}{4}}\rangle, |-_{-\frac{\pi}{4}}\rangle\}$ and the outcome is mapped to variable *s2*. Then the two corrections are accordingly applied on qubit *3*. First correction `cZ` depends on signal *s1* and second correction `cX` depends on signal *s2*.

We move to the semantics of patterns. In our example, the rotation about the $x-$axis by angle $\alpha = \frac{\pi}{4}$ is a quantum gate implemented via the following unitary matrix [42].

$$R_x(\frac{\pi}{4}) := \begin{bmatrix} \cos\frac{\pi}{8} & -i\sin\frac{\pi}{8} \\ -i\sin\frac{\pi}{8} & \cos\frac{\pi}{8} \end{bmatrix}$$

The quantum operator acts on a single qubit and the operation can be characterised by multiplying the matrix representing the gate with the vector which represents the quantum state of the qubit. However, there are four different matrices for the pattern defining sequence of commands, one for each measurement outcome.

$$(I \otimes I \otimes I)(I \otimes I \otimes I)(I \otimes |+_{-\frac{\pi}{4}}\rangle\langle+_{-\frac{\pi}{4}}| \otimes I)(|+\rangle\langle+| \otimes I \otimes I)(I \otimes {}^{C}Z)({}^{C}Z \otimes I), \quad (4.10)$$

$$(I \otimes I \otimes X)(I \otimes I \otimes I)(I \otimes |-_{-\frac{\pi}{4}}\rangle\langle-_{-\frac{\pi}{4}}| \otimes I)(|+\rangle\langle+| \otimes I \otimes I)(I \otimes {}^{C}Z)({}^{C}Z \otimes I), \quad (4.11)$$

$$(I \otimes I \otimes I)(I \otimes I \otimes Z)(I \otimes |+_{+\frac{\pi}{4}}\rangle\langle+_{+\frac{\pi}{4}}| \otimes I)(|-\rangle\langle-| \otimes I \otimes I)(I \otimes {}^{C}Z)({}^{C}Z \otimes I), \quad (4.12)$$

$$(I \otimes I \otimes X)(I \otimes I \otimes Z)(I \otimes |-_{+\frac{\pi}{4}}\rangle\langle-_{+\frac{\pi}{4}}| \otimes I)(|-\rangle\langle-| \otimes I \otimes I)(I \otimes {}^{C}Z)({}^{C}Z \otimes I), \quad (4.13)$$

with the outcomes 00, 01, 10 and 11 respectively. Moreover, they are applied on the whole quantum register of the three qubits of the computation space, and therefore they are 8×8 matrices. The input vector is a tensor product of the three qubits with qubit 1 in an arbitrary state $|\psi\rangle$ and the other two qubits in $|+\rangle$ states. The output quantum register is also separable into a tensor product of the qubits because we measure qubits 1 and 2, which collapse into the respective base states, and qubit 3 is in state $R_x(\frac{\pi}{4})|\psi\rangle$.

In section 5.3.3.3 we produce an algorithm to transform matrices of a command sequence into a unitary matrix of the corresponding operator. It is apparent from the example that when we employ a predefined pattern in an agent's event sequence, we are applying the operator directly on a given qubit, i.e., the input and output is the very same qubit. But when we use the commands, the desired output is a different qubit from the input one.

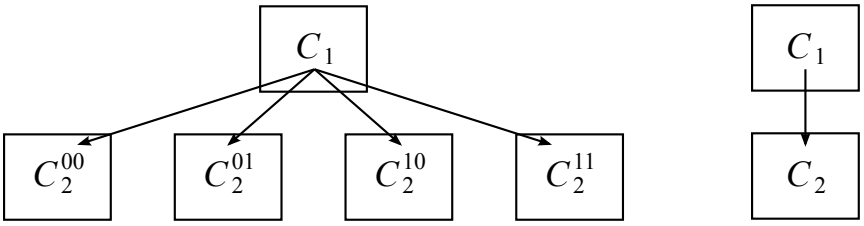

Figure 4.2: The epistemic equivalence relation of an agent and computational paths in the Rx network. Model of the command sequence is on the left, model of the pattern is on the right.

This does not matter, unless we are specifically interested in the states of other involved qubits or in the outcomes of the inner measurements. Otherwise we can use both interchangeably because, although the equivalence classes of an agent are different as shown in figure 4.2, the state of the output qubit is identical in all configurations in the second level, and so the truth value of any modal formula concerning the qubit is the same in both models. This is the reason why we require the patterns to be deterministic.

4.2.4 Formulae and Groups Modules

These two modules are closely related and they have a very similar format to the corresponding ISPL sections. The `Groups` module is used only when the modeller wants to verify a formula involving group modalities. A group includes one or more agents and is formally defined as

$$\begin{aligned} \text{groups} &::= \text{groups group} \mid \text{group} \\ \text{group} &::= \mathit{ID} = \{ \text{ members } \} \ ; \\ \text{members} &::= \text{members} \ , \ \mathit{agentID} \mid \mathit{agentID} \end{aligned}$$

where *ID* is an identifier of the group and *agentID* is an identifier of an agent who is a member of the group. In our example, *Alice* is a member of the *Imperial* group.

```
1 Imperial = {Alice};
```

Listing 4.5: The `Groups` module of the DMC file for the x-rotation.

The full grammar of the `Formulae` module is rather long and we omit it here but if the reader is interested we give the complete grammar of the language, including this module, in appendix C. Our DMC supports logical operators that are commonly used to reason about time and epistemic properties in interpreted systems. These are logical connectives (`and`, `or`, `!`, `->`), quantifiers over paths (`A`, `E`, `<ID>`), path-specific quantifiers (`X`, `F`, `G`, `U`) and epistemic modal operators (`K`, `O`, `GK`, `GCK`, `DK`).

The operators act upon the primitive propositions. We introduce them and their semantics in section 4.1.3.2. Currently supported primitives have the following syntax,

$$\begin{aligned} \text{atom} ::= \ & \mathit{agent} \ (\ \mathit{bit} \ ) = \mathit{value} \mid \mathit{agent} \ (\ \mathit{bit} \ ) = \mathit{agent} \ (\ \mathit{bit} \ ) \mid \mathit{agent\ has\ qubit} \mid \\ & \mathit{agent} \ \text{lop} \ \mathit{PC} \mid \mathit{qubit} = \mathit{qubit} \mid \mathit{qubit} = \mathit{init} \ (\ \mathit{qubit} \ ) \mid \mathit{qubit} \mid \\ & \mathit{qubit} = \{ \text{ coordinates } \} \end{aligned}$$

where *agent*, *bit* and *qubit* are respective identifiers, *value* is a value 0 or 1 of the bit and *PC* is the program counter of the agent. The atoms are always defined inside curly brackets.

We now finalise the example. Listing 4.6 displays the last snippet of the code. Assume that we would like to find out if the pattern *Rx* behaves as expected and if all members of *Imperal* group know it.

```
1 AG ({Alice >1} -> GK(Imperial,
2     {1 = {
3         (cos(pi/8), 0          ),
4         (0        , -sin(pi/8))
5     }});
```

Listing 4.6: The `Formulae` module of the DMC file for the x-rotation.

We have two atomic propositions. The first holds after *Alice* executes her first event, in this case *Rx* pattern. The second atomic proposition, specified on lines 2 to 5, is true if the qubit *1* has a particular quantum state, which we explain below.

When *Alice* applies the $R_x(\frac{\pi}{4})$ operator on her qubit with the initial state $|0\rangle$, it evolves to a new state $\cos(\frac{\pi}{8})|0\rangle - i\sin(\frac{\pi}{8})|1\rangle$. The notation can be seen on lines 3 and 4. Altogether the formula states that everybody in *Imperial* group know that qubit *1* is in state $(\frac{1}{2}e^{-i\frac{\pi}{8}} + \frac{1}{2}e^{i\frac{\pi}{8}})|0\rangle + (\frac{1}{2}e^{-i\frac{\pi}{8}} - \frac{1}{2}e^{i\frac{\pi}{8}})|1\rangle$ after *Alice* applies the $R_x(\frac{\pi}{4})$ operator on the qubit and we want to verify that the formula holds globally on all paths.

Chapter 5

Implementation of the Compiler

In this chapter, we describe implementation of the source-to-source compiler `dmc2ispl`, which translates a protocol specified in the standardised DMC input format into the corresponding ISPL code. Following compiler engineering practices [11], we divide the system into three main components to facilitate modularity and extensibility. We justify the selection of implementation language and state the uses of external tools. Subsequently we present the architecture of the system, which consist of three fundamental modules for parsing and validating the input file, generating the reachable quantum state space, and generating the ISPL output file.

5.1 Development Tools

When we chose implementation language and the development tools for our compiler, we looked for efficiency, available documentation, maximally consistent code-base and reasonable associated learning curve.

Each stage of the compiling process requires a different tool and we needed a programming language well suited to tying together the various parts of the compiler. Two natural choices were `C++` and `Java` because both can interface with almost any other language and support vast array of libraries. Each has its advantage over the other but with efficiency in mind we have chosen the former. For the lexical analysis and parsing of our standardised input format we have investigated several compiler construction tools `Spirit Parser Framework`, `ANTLR`, and `flex/GNU Bison` which not only provide a metasyntax for formal definition of programming languages but also automatically generate the code for the parser engine.

`Spirit Parser Framework` would be a relatively good choice, since the expression templates allows for approximation of the syntax in Extended Backus–Naur Form entirely in `C++`. However, it has no static checking of the grammar, and so excessive lookahead is required, which slows down the parsing, and is recommended for small to medium-size parsers. The main difference between `ANTLR`, a modern LL[1] parser generator, and `flex/Bison`, a combination of a fast lexical analyser generator and a general-purpose LALR[2] parser generator, is in the way they interpret grammars. LL approach is simpler with a better error recovery whilst LALR is more general. Both have comparable parsing speed. However, `ANTLR` does not support `C++` directly as its target language. It generates C library which is link compatible with `C++`. Also, `flex/Bison` are the only tools that output fully autonomous source code. For these reasons we have chosen them.

[1] "LL stands for Left-to-right Left-derivative" parsing and refers to a class of parsers that analyze text using a top-down approach.

[2] LALR stands for "LookAhead Left-to-right Right-derivative" parsing and uses optimised tables to determine whether a rule is complete.

Initially, we aimed to implement the mathematical part of the system in MATLAB especially because its Symbolic Math Toolbox provides tools for solving and manipulating symbolic expressions, which would be necessary for computing general quantum states. But a standalone application compiled with MATLAB Compiler is still interpreted by the runtime engine. This means that not only it requires MATLAB Compiler Runtime to be installed on a machine without MATLAB itself, but also slows down the compilation process.

Due to the breadth of internal support for technical mathematical computing comparable to MATLAB, we have chosen to implement the system in GNU Octave because it allows us to use its mathematical libraries directly in the application. Although script files and built-in functions still need the interpreter to be initialized first, all functionality we require does not need the interpreter.

In order to build a standalone application, we first compile the source code with g++ into object modules as usual. The executable is then built using mkoctfile linker with the command line option --link-stand-alone. The application is dynamically linked at runtime against the Octave libraries and any relevant support libraries, the most important of which, SuiteSparse, offers a powerful collection for computations involving sparse matrices.

To summarise, we have written the compiler in C++ with use of flex [34] version 2.5.35, GNU Bison [21] version 2.4.2 and GNU Octave [22] version 3.2.4. On top of the advantages mentioned above, these choices keep the implementation language consistent across the whole of the code-base.

5.2 Parser

The front end of dmc2isp compiler parses a DMC file containing a program written in the input format described in section 4.2 over three phases. First, it reads and syntactically validates the supplied file. Then it generates an internal representation of the specified quantum multi-agent system. And finally it checks the parsed input file for semantic errors and inconsistency.

The compiler's parsing technology is implemented in flex and GNU Bison, and is distributed across the four following different files:

- d2i-scanner.ll is the flex source file and it provides rules for lexical analysis. Lexical analysis is the lowest level of the translation process. A lexical analyzer, or scanner, converts an incoming stream of characters into an outgoing stream of tokens. In this file we define literals such as reserved keywords, identifiers for variables, numbers (integers and floats), operators, comments and whitespace.

- d2i-parser.yy is the Bison source file. The parser section of the compiler translates the input stream of tokens into a series of semantic actions and builds internal representation derived from the grammar of the language. In this file we specify a context free grammar using the Backus-Naur Form (BNF) metasyntax. BNF is a formal mathematical way to define the grammar of a language, so that there is no ambiguity. Bison's input language is essentially a machine-readable BNF, which makes it very convenient to use.

- d2i-driver.cc contains the d2i_driver class, which brings the scanner and the parser together. It facilitates a pure interface between the components.

- d2i.cc provides all data structures for the internal representation of the contents of the input file. These are classes that represent quantum MAS entities such as agents, qubits, registers, events and formulae. It also contains validation routines for context-sensitive consistency check of the source code.

	Data Type	Identifier	Description
Data Members	`string`	*id*	Unique identifier of the agent
	`string`	*defLocation*	Agent's declaration location in source code
	`int`	*index*	Agent's index in the vector of agents
	`int`	*programCounter*	Length of the agent's event sequence
	`map<string>*`	*variables*	Classical variables Γ of the agent
	`vector<qubit*>*`	*initOwnQubits*	Qubits initially owned by the agent
	`vector<qubit*>*`	*ownQubits*	Qubits in agent's possession at a time step
	`vector<qubit*>*`	*initKnownQubits*	Qubits a priori known to the agent
	`vector<qubit*>*`	*knownQubits*	Qubits known to the agent at a time step
	`vector<operation*>*`	*operations*	Agent's event sequence
	`vector<string>*`	*actions*	Set of agent's action in ISPL
	`vector<string>*`	*protocol*	Set of agents's protocol functions
	`vector<string>*`	*evolution*	Set of agent's evolution functions
	`vector<string>*`	*evolutionPCOnly*	Evolutions that change only agent's `pc`

Table 5.1: Data members of `agent` class.

In the heart of the scanner generated by `Flex` is the scanning function `yylex()`, which assembles the tokens by matching defined regular expressions against stream of input characters. It is able to deal effectively with ambiguous expressions by always choosing the longest matching string in the input stream. The input stream is fed into the lexical analyser and parser then receives the token sequence from the scanner as the return values of this function.

`Bison` generates a bottom-up parser. It starts with the string of terminal tokens and builds the parse tree from the leaves upward to the top. It shifts the tokens onto the parser stack along with their semantic values. When the tokens (terminals) and groupings (non-terminals) on top of the stack match a grammar rule, they are reduced according to the rule and a single new grouping replaces them on the stack. The parser continues until it reduces the entire input down into a single grouping. The parser looks one token ahead when deciding whether to shift or reduce. There are two potential problems with the approach. The first occurs when either shifting or reduction is the valid operation, the other happens when two grammar rules are applicable in the same situation. We have carefully designed the grammar of our input language and resolved all these conflicts which would lead to ambiguity during parsing of the input.

Each time the parser recognises a match for a grammar rule it attaches a semantic action to the rule (otherwise it would just accept or reject the input stream). The purpose of the action is to process tokens and non-terminals. Each token and non-terminal has the semantic value, which has the information about its meaning. The specification of the action in `C++` code is placed within the body of the rule. The semantic actions gradually build the internal representation of the input by instantiation of the relevant data structure classes.

The `Flex` output is a `C++` source file containing the scanning function and tables for matching tokens. Similarly, the `Bison` output is a `C++` source file called a Bison parser that parses language described by the grammar. The following command line statements invoke the generators

```
flex -od2i-scanner.cc d2i-scanner.ll
bison --defines=d2i-parser.hh d2i-parser.yy -o d2i-parser.cc
```

The scanner and the parser communicates via the driver object. It creates an instance of the parser and connects it to the lexical analyser. The bridge is done by declaration of the scanning function prototype in the `YY_DECL` macro inside the driver's header file. This way, each

	Data Type	Identifier	Description
Data Members	`int`	*id*	Unique identifier of the qubit
	`string`	*defLocation*	Qubit's declaration location in source code
	`bool`	*explicitlyDefined*	If `true` user supplies the initial state
	`agent*`	*initOwner*	Initial owner of the qubit
	`agent*`	*owner*	Owner of the qubit at a given time step
	`register*`	*initReg*	Initial entangled system of qubit, `null` if isolated
	`register*`	*currentReg*	Current entangled system of qubit, `null` if isolated
	`qState*`	*initState*	Initial quantum state of the qubit
	`set<string>*`	*reachableStates*	Set of names of all states qubit attains in protocol
Functions	Signature	`int setInitState(vector<pair<double,double>*>* s)`	
	Description	Sets initial state of qubit given coordinates, returns 0 if not normalised.	
	Signature	`bool compareQubits(const qubit* q1, const qubit* q2)`	
	Description	Returns `true` if identifier of the first qubit is less or equal to the second's.	
	Signature	`ostream& operator<<(ostream &out, qubit* q)`	
	Description	Overloads `ostream` insertion operator and formats the qubit for printing.	

Table 5.2: Data members and member functions of `qubit` class.

time the parser requests a new token, it calls the lexical analyser contained within the driver. The driver additionally performs several auxiliary tasks such as opening the file for parsing and initialising the scanning phase. It also contains error-reporting functions, which register located error messages and set the error flag.

Once the internal representation is successfully built, i.e. the input is parsed without any syntactical errors, the compiler goes through the data structure objects, checks their consistency and prepares them for the next phase. The main classes that represents the quantum multi-agent systems are `agent` (see table 5.1 for details on its data members), `qubit` (table 5.2), `register` (table 5.3) and `operation` (table 5.4). These structures are intuitive and contain all the data necessary for a successful translation.

The following procedures validate the internal representation, so that the reachable quantum state space analyser does not crash and the generated ISPL code corresponds to the DMC input.

- The function `classicalVariablesCheck()` checks if all classical input bits and signal are properly defined and that no agent's event depends on uninitialised value of a variable.
- The procedure `finaliseQubits()` assigns initial owners to all qubits and prepares implicitly declared qubits in $|+\rangle$ state. Returns 1 if an agent operates on a qubit which is not in her sort or if a qubit in the input sort of an agent is not defined explicitly.
- The function `finaliseCommunicationOperations()` pairs the send/receive classical and quantum communication events. It returns 1 if there is a problem such as unpaired events, sending of uninitialised values or invalid agent identifiers.
- The method `synchroniseProgramCounters()` assigns a time step to all events in agents' event sequences and generates the global sequence of events. Returns 1 if there are cyclic communication events that would lead to gridlock in the network.
- The procedure `processDefinitions()` verifies if declarations of patterns defined in the `Definition` module conform to the specifications from section 4.2.3.

	Data Type	Identifier	Description
Data	`string`	*id*	Unique identifier of the register
	`string`	*defLocation*	Register's declaration location in source code
	`int`	*globalStateIdx*	Index of states of the register in the global state
	`qState*`	*initState*	Initial state of the register, `null` if undefined
	`vector<qubit*>*`	*qubits*	Qubit members of the register
	`set<string>*`	*reachableStates*	Set of names of states register attains in protocol
Functions	Signature	`int setInitState(vector<pair<double,double>*>* s)`	
	Description	Sets initial state of register given coordinates, returns 0 if not normalised.	
	Signature	`string getEntangledId(vector<qubit*>* q)`	
	Description	Returns the identifier of a register containing given qubits.	
	Signature	`ostream& operator<<(ostream &out, register* r)`	
	Description	Overloads `ostream` insertion operator, formats the register for printing.	

Table 5.3: Data members and member functions of `register` class.

	Data Type	Identifier	Description
Data	`string`	*type*	Type of the operation event
	`string`	*defLocation*	Operation's declaration location in source code
	`agent*`	*theAgent*	The performer of the operation
	`int`	*programCntr*	Position of operation in agent's event sequence
	`int`	*globalProgramCntr*	The time step of the operation
	`set<string>*`	*actions*	Set of associated ISPL actions
Member Functions	Signature	`virtual vector<string>* setActions() =0`	
	Description	Creates ISPL actions associated with the operation and returns them.	
	Signature	`virtual vector<string>* getProtocol() =0`	
	Description	Returns the ISPL protocol function of the agent for the operation.	
	Signature	`virtual vector<string>* getEvolution() =0`	
	Description	Returns the ISPL evolution function of the agent for the operation.	
	Signature	`virtual operation* getOtherHalf() =0`	
	Description	Returns the opposite operation of communication events, `null` otherwise.	
	Signature	`vector<string>* getActions()`	
	Description	Returns ISPL actions associated with the operation.	
	Signature	`ostream& operator<<(ostream &out, operation* o)`	
	Description	Overloads `ostream` insertion operator, formats the operation for printing.	

Table 5.4: Data members and member functions of abstract `operation` class.

5.3 Quantum State Space Analyser

The entire analyser is implemented in file `d2i-octave.cc` and utilises several classes from the `Octave` library, notably its own complex data structure `Complex` and matrix data structures `ComplexMatrix`, `SparseMatrix` and `SparseComplexMatrix`. We make some general remarks about implementation of this phase of compilation and then we describe its main parts in more detail.

Quantum states of a n-qubit register are complex column vectors in $\mathbb{C}^{2^n}$ Hilbert space. We implement them as $2^n \times 1$ matrices of type `ComplexMatrix` rather then vectors due to easier manipulation. Thus, when we talk about matrices representing quantum states, it should be

remembered that they are in fact column vectors. The postulates of Quantum Mechanics require that the time evolution to a quantum state of a n-qubit system is realised via some $2^n \times 2^n$ unitary operator.

Apparently, even a small system involves matrices of huge dimensions. Luckily these matrices are populated primarily with zeros because our operators act on two qubits at most. This allows us to use sparse matrices for operators. The relevant data structures are `SparseComplexMatrix` (e.g. for measurement projections on a rotated basis) and `SparseMatrix` if the elements are real numbers (e.g. operators I, X, Z). Manual [22] explains that `Octave` stores sparse matrices in a compressed column format, a very efficient format for arithmetic operations and matrix-vector products, which we use extensively throughout the code.

The main purpose of the analyser is to compute the reachable state space, to enumerate encountered states, and to generate evolution functions of the global quantum system. We use a map from complex column vectors to names to uniquely enumerate the quantum states. To do this properly, we need standardisation of quantum states on two levels.

First, we have to recognise states that differ only in the global phase and eliminate the global phase factor. We discuss reasons why in section 4.1.2.2 but to do it practically, we introduce a standard form of a quantum state. We say that the state $|\psi\rangle$ is in the standard form if its first non-zero coordinate is a positive real number. For example, the state $|\psi\rangle = -\frac{i}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle$ is transformed to its standard form $|\psi\rangle = \frac{1}{\sqrt{2}}|01\rangle - \frac{i}{\sqrt{2}}|10\rangle$ by removing the global phase factor $e^{i\frac{\pi}{2}}$. Incidentally, if both these states are encountered in the state space they get the same name. On top of that, we round both real and imaginary parts of coordinates to five significant digits when comparing vectors to avoid problems connected with floating-point arithmetic.

Second standardisation concerns quantum registers. We implement registers as totally ordered sets of qubit references and the positions of qubits in the register matters. The same register with different order of qubits has generally different vector representation of the state it is in. We would have two different names for the same quantum state. To avoid this, we keep qubits in a register sorted in order of their increasing references.

We remind the reader that we sometimes refer to a single qubit as a register and use terms quantum system and quantum register interchangeably. Also, note that our numbering begins with zero, i.e. the index of the first element is 0. This is convenient in many computations, as well as consistent with `C++` and `Octave` libraries.

5.3.1 State Space Generator

We first generate sequence of quantum events that agents apply at each time step. Because `cX`, `cZ`, `Me` and patterns depend on classical variables, we consider actions for any combination of the variables. This leads to additional branching on top of two different measurement outcomes and each global quantum state may have several successors.

The *EventSequence* structure $\mathcal{E}$ is passed to the function `generateQuantumEventSequence` (see algorithm 5.1), which processes it and returns *QuantumEventSequence* structure $\mathcal{Q}$. This new structure consists of gpc[3] sets of sets, $level_i$. Each of these sets has n elements, where n is a number of all possible combinations of quantum operators, which agents execute at a given time step i with regards to values of relevant classical variables. The set ops_k corresponds to the kth such combination.

The *QuantumEventSequence* structure $\mathcal{Q}$ is then passed to the `walkReachableStateSpace` function (see algorithm 5.2), which generates reachable quantum state space of the protocol and returns the set of statements *evol*. The ISPL code generator later uses these statements to

[3]In this context *gpc* is the total number of time steps in the protocol.

Algorithm 5.1 `generateQuantumEventSequence`(EventSequence $\mathcal{E}$)

1: **for** $1 \leq i \leq \mathcal{E}.gpc$ **do**
2: $n \leftarrow 1$
3: **for** $0 \leq j < |\mathcal{E}.level_i|$ **do**
4: $n \leftarrow n * |\mathcal{E}.level_i.events_j.actions|$
5: **end for**
6: **for** $0 \leq j < |\mathcal{E}.level_i|$ **do**
7: **for** $0 \leq k < n$ **do**
8: $\mathcal{Q}.level_i.ops_k$.`insert`($\mathcal{E}.level_i.events_j.actions_{(k/n)\%|\mathcal{E}.level_i.events_j.actions|}$)
9: **end for**
10: **end for**
11: **end for**
12: **return** $\mathcal{Q}$

Algorithm 5.2 `walkReachableStateSpace`(QuantumEventSequence $\mathcal{Q}$)

1: $g \leftarrow$ Stack(`initialGlobalState`)
2: **while not** g.`empty` **do**
3: $\mathcal{G} \leftarrow g$.`pop`
4: **for** $|\mathcal{Q}.level_{\mathcal{G}.gpc}| > i \geq 0$ **do**
5: $\mathcal{H} \leftarrow$`computeNewGlobalState`($\mathcal{G}$, $\mathcal{Q}.level_{\mathcal{G}.gpc}.ops_i$)
6: $e \leftarrow$ `getEvolution`($\mathcal{G}$, $\mathcal{H}$, $\mathcal{Q}.level_{\mathcal{G}.gpc}.ops_i$)
7: $evol$.`push`(e)
8: **if** $\mathcal{G}.gpc < \mathcal{Q}.gpc$ **then**
9: g.`push`($\mathcal{H}$)
10: **end if**
11: **end for**
12: **end while**
13: **return** $evol$

produce the evolution function of the Environment agent, as well as Environment's actions and protocol if an impossible state is encountered.

At first, we push the initial global quantum state to the stack g of *GlobalState* structures. *GlobalState* structure has three members, integer *gpc* indicating the time step, to which the global state belongs, a Boolean flag *imp*, which indicate that a measurement outcome is impossible and a set of individual qubits and entangled quantum registers *regs*. A register is also a structure with members *state*, *qubits* and *pos*. The first member is a matrix representing the current local state of the register, if defined. The second is a set of qubits of the register and the last is a set of their positions. States of all registers, if defined, make up the state of the whole system.

We then apply all possible combinations of quantum operators, $\mathcal{Q}.level_{\mathcal{G}.gpc}.ops_i$, that agents execute in a given time step, and compute all the successors $\mathcal{H}$ of the global state $\mathcal{G}$. We determine evolution statements e from the differences between the global states. We keep adding new global states $\mathcal{H}$ to g until we reach the end of the quantum event sequence $\mathcal{Q}.gpc$. The whole process continues until all reachable quantum global states are dealt with.

The subroutine `computeNewGlobalState` returns a successor $\mathcal{H}$ of a *GlobalState* $\mathcal{G}$ given the specific set of operators *op*. We assume that the operators are applied sequentially by the agents, as we discuss in section 4.1.2.1, even though all the operations happen at the same time step. The evolution of the global state depends on a type of the operator executed. We describe functions responsible for the computations in the next section.

Algorithm 5.3 `computeNewGlobalState`(GlobalState $\mathcal{G}$, Set op)

```
1: H ← G, H.gpc ← H.gpc + 1
2: for 0 ≤ i < |op| do
3:    if op_i = En then
4:       H ← entanglement(H, op_i)
5:    else if op_i = Me then
6:       H ← measurement(H, op_i)
7:    else if (op_i = cX or op_i = cZ or op_i = Pa) then
8:       H ← singleQubitOp(H, op_i)
9:    end if
10: end for
11: return H
```

5.3.2 Quantum Commands

5.3.2.1 Entanglement

Entanglement means application of ${}^{C}Z$ gate on the qubits involved. These qubits may be in well-defined local states, part of an entangled system, or combination of both. We distinguish two cases. In the first, described by algorithm 5.4, the qubits are already part of the same entangled register. This is rather straightforward situation because we just update the state of the register, unless the operation disentangles it. We cover disentanglement and separation of quantum systems in section 5.3.3.2.

First, we need to build the matrix U representing the operator. ${}^{C}Z$ behaves such that when the control qubit is $|1\rangle$, the Z operation is performed on the second qubit, i.e. it negates the $|1\rangle$ qubit but leaves the $|0\rangle$ qubit unchanged. The Hilbert state space $\mathcal{H}$ of n-qubit system has dimension 2^n and its standard basis is a set of vectors $\mathcal{B} = \{B \otimes^n | B \in \{|0\rangle, |1\rangle\}\}$. So, if the register consists of n qubits and the operator acts on qubits q and r then U is a $2^n \times 2^n$ diagonal matrix with -1 in all positions corresponding to basis vectors with constituent q and r qubits equal to $|1\rangle$. All other diagonal elements of the matrix are 1 because the operator does not change states of these qubits.

For example, for $n = 2$ we have four basis vectors $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. The last one is the tensor product of $|1\rangle$ and $|1\rangle$, and so U is a 4×4 matrix with the fourth diagonal element equal to -1. This gives the usual ${}^{C}Z$ matrix. For $n = 3$ and first and third active qubits, the relevant basis vectors are $|101\rangle = |1\rangle \otimes |0\rangle \otimes |1\rangle$ and $|111\rangle = |1\rangle \otimes |1\rangle \otimes |1\rangle$, and so U is a 8×8 matrix with the sixth and eighth diagonal elements equal to -1.

$$n = 2, q = 1, r = 2 : U_{diag} = [1\ 1\ 1\ \text{-}1]$$
$$n = 3, q = 1, r = 3 : U_{diag} = [1\ 1\ 1\ 1\ 1\ \text{-}1\ 1\ \text{-}1]$$

The notation of basis vectors $\mathcal{B} = \{|\mathbf{x}\rangle | \mathbf{x} \in \{0,1\}^n\}$ is very convenient because it enumerates the vectors as binary numbers. We can easily identify values v_1 and v_2 of digits in positions[4] q and r of the vector i using formulae

$$v_1 = \lfloor \frac{i}{2^{n-q-1}} \rfloor \bmod 2 \qquad\qquad v_2 = \lfloor \frac{i}{2^{n-r-1}} \rfloor \bmod 2$$

where $i \in \{0, \ldots, 2^n - 1\}$ and $q, r \in \{0, \ldots, n-1\}$. The relevant vectors have both v_1 and v_2 equal 1. In the previous examples, if $n = 2$, $q = 0$, $r = 1$ then $v_1 = 1$ for $i \in \{2, 3\}$ and $v_2 = 1$

[4]Remeber that our numbering begins with zero, i.e. the position of the first qubit is 0.

Algorithm 5.4 `entanglement`(GlobalState $\mathcal{G}$, Operator $\mathcal{O}$)

$q \leftarrow \mathcal{O}.qubit1$, $r \leftarrow \mathcal{O}.qubit2$, $R \leftarrow \mathcal{G}.regs_q.state$, $U \leftarrow \text{Zero}(R.rows, R.rows)$, $\mathcal{H} \leftarrow \mathcal{G}$
$denom1 \leftarrow 2^{\log_2(R.rows) - G.regs_q.pos_q - 1}$, $denom2 \leftarrow 2^{\log_2(R.rows) - G.regs_q.pos_p - 1}$
for $0 \leq i < R.rows$ **do**
  if $((i/denom1)\%2 = 1$ **and** $(i/denom2)\%2 = 1)$ **then**
    $U_{i,i} = -1$
  else
    $U_{i,i} = 1$
  end if
end for
$R \leftarrow U * R$
if `isSeparable`(R, X, Y) **then**
  $\mathcal{H}.regs_q.state \leftarrow \emptyset$, $\mathcal{H}.regs_X.state \leftarrow X$, $\mathcal{H}.regs_Y.state \leftarrow Y$
else
  $\mathcal{H}.regs_q.state \leftarrow R$
end if
return $\mathcal{H}$

Algorithm 5.5 `entanglement`(GlobalState $\mathcal{G}$, Operator $\mathcal{O}$)

$q \leftarrow \mathcal{O}.qubit1$, $r \leftarrow \mathcal{O}.qubit2$, $S \leftarrow \mathcal{G}.regs_q.state$, $T \leftarrow \mathcal{G}.regs_r.state$, $\mathcal{H} \leftarrow \mathcal{G}$
$R \leftarrow$ `kron`(S, T)
for all i **in** $\mathcal{H}.regs_r.qubits$ **do**
  $pos_{old} \leftarrow \mathcal{H}.regs_r.pos_i$, $pos_{new} \leftarrow \mathcal{H}.regs_R.pos_i$
  `shiftQubit`(R, pos_{old}, pos_{new})
end for
$\mathcal{H}.regs_q.state \leftarrow \emptyset$, $\mathcal{H}.regs_r.state \leftarrow \emptyset$, $\mathcal{H}.regs_R.state \leftarrow R$
$\mathcal{H} \leftarrow$ `entanglement`($\mathcal{H}$, $\mathcal{O}$)
return $\mathcal{H}$

for $i \in \{1, 3\}$. Thus the fourth vector, $i = 3$, is the one we are looking for. Similarly, if $n = 3$, $q = 0$, $r = 2$ then $v_1 = 1$ for $i \in \{4, 5, 6, 7\}$ and $v_2 = 1$ for $i \in \{1, 3, 5, 7\}$ and the sixth $i = 5$ and eighth $i = 7$ vectors are the ones.

Once we construct the matrix U, the new state of the register is obtained from the current state R by matrix multiplication UR. After the operation, we check if the state UR is entangled. If it can be separated the quantum system is broken down into two non-separable subsystems, represented by matrices X and Y, and the state of the current register is no longer defined.

In the second case (see algorithm 5.5), the qubits are in two different registers or they may be just ordinary qubits in a well-defined local state. When the ${}^{C}Z$ operator is performed on the qubits, their respective registers merge into a new quantum system containing all qubits from both. Since we keep qubits of the register in the order of their global position, the merging poses a problem. State space of the quantum systems are $\mathcal{H}_1$ and $\mathcal{H}_2$ respectively, and so the state space $\mathcal{H}$ of the compound system equals $\mathcal{H}_1 \otimes \mathcal{H}_2$. This is fine but when we consider some concrete states $|\varphi\rangle \in \mathcal{H}_1$ and $|\theta\rangle \in \mathcal{H}_2$ then $|\varphi\rangle \otimes |\theta\rangle = |\psi\rangle \in \mathcal{H}$ and the qubits are arranged in the compound system such that qubits from the first register come first and qubits form the other register come second. Obviously, they may not be in the proper order.

We need to sort the qubits. First, we find positions of the qubits from the second register in the compound system and then we shift them one by one to their proper locations. The qubits from the first register are placed in the correct positions automatically because the `shiftQubit`

Algorithm 5.6 `shiftQubit`(Matrix R, int $from$, int to)

```
1: if |R| = 4 then
2:    R ← U * R
3: else
4:    while from < to do
5:       if from = 0 then
6:          T ← kron(U, I)
7:       else if from = 1 then
8:          T ← kron(I, U)
9:       else
10:         T ← kron(I, I)
11:         for 2 ≤ i < from do
12:            T ← kron(T, I)
13:         end for
14:         T ← kron(T, U)
15:      end if
16:      for from + 1 < i < |R| do
17:         T ← kron(T, I)
18:      end for
19:      R ← T * R, from ← from + 1
20:   end while
21: end if
```

function, shown in algorithm 5.6, shifts a qubit by swapping it iteratively with its neighbours. In fact, we perform an optimised bubble sort on the merged quantum register. Once this is done, we proceed as previously because qubits q and r are now part of the same quantum system.

The `shiftQubit` takes a matrix representation R of a state of a quantum system and two indices of current, *from*, and desired, *to*, location of a qubit. The function then iteratively swaps the qubit with its lower-indexed neighbour, updating *from* location each cycle, until the qubit reaches its destination.

Every iteration we need to construct a new swapping operator T because the position of the swapped qubits changes. However, it is quite straightforward for quantum systems of two qubits since the operator T consists of the swap gate U itself,

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

which swaps the two qubts. For the larger registers we need to tensor U with the appropriate number of identity operators I. For instance, if we want to swap qubits $q = 1$ and $q = 2$ of a 5-qubit register we need to use operator $T = I \otimes U \otimes I \otimes I$. The new state of the system is then TR.

Although we are not technically swapping qubits themselves but their states. This, however, does not matter since the result, quantum state of the register, is the same. Consider 2-qubit system $\{q, r\}$ with respective states of the qubits $\{|\varphi\rangle, |\theta\rangle\}$. Swapping qubits results in $\{r, q\}$ and $\{|\theta\rangle, |\varphi\rangle\}$, whilst swapping states leads to $\{q, r\}$ and $\{|\theta\rangle, |\varphi\rangle\}$. In both cases the system is in state $|\theta\rangle = |\theta\rangle \otimes |\varphi\rangle$. Moreover, we always refer to qubits by their indices in the set representing the quantum register, therefore their actual names are insignificant.

5.3.2.2 Measurement

Measuring is an act of carrying out an observation on a given quantum system. The property that can be measured is called an observable. Observable has an associated hermitian operator Ω. The eigenvalues λ_i of Ω are the only possible values observable can take as a result of measurement and its eigenvectors $|e_i\rangle$ form a basis of the state space. Operator Ω on n-dimensional Hilbert space $\mathbb{C}^n$ can be uniquely represented as a linear combination

$$\Omega = \sum_{i=1}^{n} \lambda_i P_i,$$

where $\lambda_i \in \mathbb{R}$ is an eigenvalue of Ω and P_i is an orthogonal projection onto the eigenspace generated by the eigenvector $|e_i\rangle$. In bra-ket notation this projection is represented by the outer product $P_i = |e_i\rangle \langle e_i|$.

Probability to observe λ_k in a state $|\psi\rangle$ is $|\alpha_k|^2 = |\langle e_k | \psi \rangle|^2 = \langle \psi | P_k | \psi \rangle$. We now can decompose the state $|\psi\rangle$ according to the observable Ω

$$|\psi\rangle = \sum_{i=1}^{n} \alpha_i |e_i\rangle = \sum_{i=1}^{n} \langle e_i | \psi \rangle |e_i\rangle = \sum_{i=1}^{n} |e_i\rangle \langle e_i | \psi \rangle = \sum_{i=1}^{n} P_i |\psi\rangle . \tag{5.1}$$

Therefore, when we perform a measurement of observable Ω the state $|\psi\rangle$ collapses to the state $|e_k\rangle$ and the measurement instrument displays λ_k with the probability $|\alpha_k|^2$.

Important for our purposes is the fact that evolution of a quantum system on measurement with outcome λ_k can be computed using projection P_k. The measurement command in DMC is defined as a projection on (possibly rotated) diagonal basis[5] $\{|+_\gamma\rangle, |-_\gamma\rangle\}$. As we explain in section 4.1.2.1, we divide it into two operations, one for each outcome. The matrices representing the projections are

$$M_+ = |+_\gamma\rangle \langle +_\gamma| = \frac{1}{2} \begin{bmatrix} 1 & e^{i\gamma} \\ e^{i\gamma} & 1 \end{bmatrix}, \qquad M_- = |-_\gamma\rangle \langle -_\gamma| = \frac{1}{2} \begin{bmatrix} 1 & -e^{i\gamma} \\ -e^{i\gamma} & 1 \end{bmatrix},$$

where γ is the angle of measurement and may depend on signals $s, t \in \mathbb{Z}_2$, in which case the angle is modified to $(-1)^s\gamma + t\pi$. The operation of measuring 0 correlates to M_+ when the measured qubit collapses to state $|+_\gamma\rangle$. Likewise, the operation of measuring 1 corresponds to M_- and the qubit collapses to state $|-_\gamma\rangle$. Both operators are certainly not unitary because their matrices are singular.

The algorithm 5.7 illustrates the `measurement` function. If we want to apply M_+ or M_- operator on a qubit which belongs to a system of several qubits we need to obtain the operator T with the proper dimension. T is the tensor product of the projections with identity operators because nothing acts on the rest of the qubits.

Matrix multiplication of operator T with the state R of the quantum register, however, does not produce the evolved state of the system. The equation 5.1 implies that $M_+ |\psi\rangle = \alpha_+ |+_\gamma\rangle$ and $M_- |\psi\rangle = \alpha_- |-_\gamma\rangle$ with α_+ and α_- being the respective probability amplitudes. These can potentially be equal to zero, meaning that the operation results in an impossible state. Indeed, this happens with $M_+ |-_\gamma\rangle$ and $M_- |+_\gamma\rangle$ because the outcome of measuring the basis vector cannot ever be the eigenvalue corresponding to another basis vector. If this situation arises and TR is the zero matrix, then the Boolean flag of *GlobalState* structure $\mathcal{H}$ indicates this fact and the `measurement` function runs again with the opposite measurement operator.

[5] We change the notation of measurement angle from α to γ here because we want to distinguish it from probability amplitude α_i

Algorithm 5.7 `measurement`(GlobalState $\mathcal{G}$, Operator $\mathcal{O}$)

```
1: q ← O.qubit, M ← O.mtrx, R ← G.regs_q.state, H ← G
2: if |R| = 2 then
3:    R ← M * R
4:    if R = Zero(R.rows, 1) then
5:       H.imp ← true, H ← measurement(H, O.opp)
6:    else
7:       R ←normalise(R), H.regs_q.state ← R
8:    end if
9: else
10:   if q = 0 then
11:      T ← kron(M, I)
12:   else if q = 1 then
13:      T ← kron(I, M)
14:   else
15:      T ← kron(I, I)
16:      for 2 ≤ i < q do
17:         T ← kron(T, I)
18:      end for
19:      T ← kron(T, M)
20:   end if
21:   for q < i < |R| do
22:      T ← kron(T, I)
23:   end for
24:   R ← T * R
25:   if R = Zero(R.rows, 1) then
26:      H.imp ← true, H ← measurement(H, O.opp)
27:   else
28:      R ←normalise(R), separateStatesOnMeasurement(R, O.basis, q)
29:      if isSeparable(R, X, Y) then
30:         H.regs_q.state ← O.basis, H.regs_X.state ← X, H.regs_Y.state ← Y
31:      else
32:         H.regs_q.state ← O.basis, H.regs_R.state ← R
33:      end if
34:   end if
35: end if
36: return H
```

Even if the operation is possible, the resulting TR matrix has to be renormalised due to the probability amplitude. At this point, unless the projection acts on a single qubit in a well-defined local state, the register is not entangled because the measured qubit collapses to the appropriate basis state. The function `separateStatesOnMeasurement` (see algorithm 5.8) isolates the qubit from the rest of the quantum system. It requires the matrix R representing the state of the register, the relevant measurement *basis* state and the index of the measured qubit q.

Because the measured qubit generally lies somewhere "inside" the register, it is not obvious that R is the tensor product of *basis* and some unknown matrix[6] R', i.e. $R \neq basis \otimes R'$

[6]We reiterate that even though we write "matrix", we mean "column vector" when discussing quantum states of dimension n because they are implemented as $n \times 1$ matrices. On the other hand, operators are matrices in

Algorithm 5.8 `separateStatesOnMeasurement`(Matrix R, Matrix B, int q)

1: $X \leftarrow \text{Matrix}(R.rows/2, R.rows/2)$, $Y \leftarrow \text{Matrix}(R.rows/2, 1)$
2: $denom \leftarrow 2^{\log_2(R.rows)-q-1}$, $idx \leftarrow 0$
3: **for** $0 \leq i < R.rows$ **do**
4: **if** $(i/denom)\%2 = 0$ **then**
5: $X_{idx,idx} \leftarrow B_{0,0}$, $Y_{idx,0} \leftarrow R_{i,0}$, $idx \leftarrow idx + 1$
6: **end if**
7: **end for**
8: $R \leftarrow X$.`solve`(Y)

nor $R \neq R' \otimes basis$. However, we utilise the fact that a rearrangement of qubits, TR, where T is a suitable shift operator, makes the decomposition of the tensor product possible, i.e. $TR = basis \otimes R'$. Although shifting the collapsed qubit would be needlessly inefficient and we employ more subtle approach.

The shift operator T can be though of as a series of n swap gates, $T = \prod_{i=1}^{n} S_i$, where each S_i swaps the qubit by one position. In case of splitting the register after measurement, the measured qubit needs to be shifted to the front. One way of constructing a unitary operator is by analysing its behaviour towards basis. Operators map basis to some other basis. In particular, T maps a standard basis vector to another standard basis vector. This means that each row and each column has exactly one non-zero element equal to 1. Moreover, the element can be located.

As we show in the previous section, the basis vectors are enumerated with binary numbers. Shifting a qubit q to the beginning of a register corresponds to moving the qth digit to the front. For example, relocating the third qubit in the 3-qubit system maps the second basis vector $|001\rangle$ to the fifth basis vector $|100\rangle$, therefore $T_{4,1} = 1$. The column index l of the element in row k is determined by formula

$$l = 2d\lfloor\frac{k - 2^{n-1}\lfloor\frac{k}{2^{n-1}}\rfloor}{d}\rfloor + d\lfloor\frac{k}{2^{n-1}}\rfloor + (k \bmod d),$$

where n is the number of qubits in the system, q is the position of the measured qubit and $d = 2^{n-q-1}$. Expanding $basis \otimes R' = TR$ in terms of individual elements leads a to system of 2^n linear equations in 2^{n-1} variables R'_j

$$basis_i * R'_j = \sum_{l=0}^{2^n-1} T_{k,l} * R_l = R_m,$$

where $i \in \{0, 1\}$, $j \in \{0, \ldots, 2^{n-1}\}$, $k = i2^{n-1} + j$ and m is the column index of the unique non-zero element in row k of the operator T.

Half of the equations can be omitted because $basis_0 * R'_j = \frac{1}{\sqrt{2}} * R'_j = R_{m_0}$ gives the same R_j as $basis_1 * R'_j = \frac{e^{i\alpha}}{\sqrt{2}} * R'_j = R_{m_1}$, otherwise there would be no solution, which is impossible since the measurement disentangles the state. `Octave` linear equation solver then produces the state R' of the reduced quantum register.

However, this new state may not be entangled at all. Assume that the measured qubit binds together two otherwise independent quantum systems. After the measurement, the systems split up again. For this reason the `isSeparable` function, covered in section 5.3.3.2 carries out the entanglement check on the new state.

both senses.

Algorithm 5.9 `singleQubitOp`(GlobalState $\mathcal{G}$, Operator $\mathcal{O}$)

1: $q \leftarrow \mathcal{O}.qubit$, $U \leftarrow \mathcal{O}.mtrx$, $R \leftarrow \mathcal{G}.regs_q.state$, $\mathcal{H} \leftarrow \mathcal{G}$
2: **if** $|R| = 2$ **then**
3: $\quad \mathcal{H}.regs_q.state \leftarrow U * R$
4: **else**
5: $\quad$ **if** $q = 0$ **then**
6: $\quad\quad T \leftarrow$ `kron`(U, I)
7: $\quad$ **else if** $q = 1$ **then**
8: $\quad\quad T \leftarrow$ `kron`(I, U)
9: $\quad$ **else**
10: $\quad\quad$ `kron`(I, I, T)
11: $\quad\quad$ **for** $2 \leq i < q$ **do**
12: $\quad\quad\quad T \leftarrow$ `kron`(T, I)
13: $\quad\quad$ **end for**
14: $\quad\quad T \leftarrow$ `kron`(T, U)
15: $\quad$ **end if**
16: $\quad$ **for** $q < i < |R|$ **do**
17: $\quad\quad T \leftarrow$ `kron`(T, I)
18: $\quad$ **end for**
19: $\quad \mathcal{H}.regs_q.state \leftarrow T * R$
20: **end if**
21: **return** $\mathcal{H}$

5.3.2.3 Corrections and Patterns

All single qubit unitary operators T may be constructed and applied the same way by tensoring the operator U with correct number of identity operators I. The only difference is the purpose of the U operator with $U \in \{X, Z\} \cup \mathcal{P}$ depending on the chosen operation. $\mathcal{P}$ contains all operators defined by patterns in the `Definitions` section, X and Z are basic corrections represented by Pauli matrices

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

We can conveniently implement all operations in one function `singleQubitOp` (algorithm 5.9 and just change the definition of the fundamental building bloc U. Matrix multiplication of the finished operator T with the state R of the quantum system then results in a desired new quantum state, which is returned by the function to the main loop.

5.3.3 Auxiliary Functions

We now describe implementation of three important functions that are used across the `Quantum State Space Analyser`.

5.3.3.1 Kronecker Tensor Product

The Kronecker tensor product `kron` is implemented in `Octave` as a dynamically linked function. Which means it is not part of the `Octave` library and requires invocation of the interpreter. This is rather inconvenient, and so we include the function[7] directly in our source code. The

[7] The algorithms were developed by Paul Kienzle as part of the `Octave` source code.

Algorithm 5.10 `kron`(Matrix A, Matrix B)

1: $C \leftarrow$ Matrix($A.rows * B.rows, A.columns * B.columns$)
2: $c \leftarrow 0$, $r \leftarrow 0$
3: **for** $0 \leq i < A.columns$ **do**
4: **for** $0 \leq j < A.rows$ **do**
5: **for** $0 \leq k < B.columns$ **do**
6: **for** $0 \leq l < B.rows$ **do**
7: $C_{r+l,c+k} \leftarrow A_{j,i} * B_{l,k}$
8: **end for**
9: **end for**
10: $r \leftarrow r + B.rows$
11: **end for**
12: $c \leftarrow c + B.columns$
13: **end for**
14: **return** C

Algorithm 5.11 `kron`(SparseMatrix A, SparseMatrix B)

1: $C \leftarrow$ SparseMatrix($A.rows * B.rows, A.columns * B.columns, A.nzmax * B.nzmax$)
2: $C.cidx_0 \leftarrow 0$, $n \leftarrow 0$
3: **for** $0 \leq i < A.columns$ **do**
4: **for** $0 \leq j < B.columns$ **do**
5: **for** $A.cidx_i \leq k < A.cidx_{i+1}$ **do**
6: **for** $B.cidx_j \leq l < B.cidx_{j+1}$ **do**
7: $C.data_n \leftarrow A.data_k * B.data_l$
8: $C.ridx_n \leftarrow A.ridx_k * B.rows + B.ridx_l$
9: $n \leftarrow n + 1$
10: **end for**
11: **end for**
12: $C.cidx_{i*B.columns+j+1} \leftarrow n$
13: **end for**
14: **end for**
15: **return** C

tensor product of ordinary matrices is computed in a different way than tensor product of sparse matrices.

The algorithm 5.10 shows the basic version. It is a fairly straightforward realisation of the operator. Tensoring input matrices A and B with respective dimensions $m \times n$ and $o \times p$ gives a $mo \times np$ block matrix C. Because we apply the operation extensively throughout the application on matrices with huge dimensions, it is essential to adhere to column-major order of storing matrices in `Octave`, when accessing the elements of the matrices.

The situation with sparse matrices is slightly more complicated and algorithm 5.11 describes the modified function. `Octave` stores sparse matrices in a compressed column format. The non-zero elements of a matrix are stored in three vectors *cidx*, *ridx* and *data*, representing the column indexing, row indexing and data respectively. The first vector stores information on the number of non-zero elements in each column of the matrix, the second holds their row index in the matrix and the third contains the data itself.

Algorithm 5.12 `isSeparable`(Matrix A, Matrix B, Matrix C)

```
1: iter ← (A.rows/2) − 1
2: for 0 ≤ i < iter do
3:    M ← getCoefficients(A)
4:    if M = null then
5:       continue
6:    end if
7:    B ← solve(M)
8:    C ← solve(A, B, iter)
9:    return true
10: end for
11: return false
```

5.3.3.2 Separation of Quantum Registers

We need an inverse operation to the tensor product of vectors since `En` and `Me` commands may evolve an entangled state of a quantum system into a separable state. The former arises because ${}^{C}Z$ is a self-adjoint operator, i.e. ${}^{C}Z{}^{C}Z = I$. Suppose an entangled quantum state $|\psi\rangle$ is such that ${}^{C}Z(|\varphi\rangle \otimes |\theta\rangle) = |\psi\rangle$. Then ${}^{C}Z\,|\psi\rangle = {}^{C}Z{}^{C}Z(|\varphi\rangle \otimes |\theta\rangle) = (|\varphi\rangle \otimes |\theta\rangle)$. The latter results from the fact that measurement projections are not represented by unitary, often not even invertible, matrices and after separation of the collapsed qubit, the quantum register may not be entangled. For instance, consider the following system of three entangled qubits

$$\frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle - |011\rangle + |100\rangle + |101\rangle - |110\rangle + |111\rangle)$$

$$\xrightarrow{M_1^0} \begin{cases} \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \\ \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes \frac{1}{\sqrt{2}}(|10\rangle - |11\rangle) = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes |1\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \end{cases}$$

After measurement the first qubit in $\{|+\rangle, |-\rangle\}$ basis, the remaining two qubits are either in $\frac{1}{\sqrt{2}}(|00\rangle + |01\rangle)$ or $\frac{1}{\sqrt{2}}(|10\rangle - |11\rangle)$ state, neither of which is entangled.

We do not need to check whether the new state is separable after executing `cX`, `cZ` and pattern commands because single qubit unitary operators cannot disentangle a quantum system. Suppose this is possible and an entangled quantum state $|\psi\rangle$ and an operator U are such that $U\,|\psi\rangle$ is separable into $|\varphi\rangle \otimes |\theta\rangle$. Then the conjugate transpose matrix $U^\dagger$ also represents a single qubit unitary operator and $U^\dagger U\,|\psi\rangle = |\psi\rangle$. Because $U^\dagger$ acts only on one qubit, we have either $U^\dagger\,|\varphi\rangle \otimes |\theta\rangle$ or $|\varphi\rangle \otimes U^\dagger\,|\theta\rangle$ by distributivity of the tensor pruduct, and so $|\psi\rangle$ is separable. But this contradicts with the supposition that $|\psi\rangle$ is entangled.

However, decomposition of a quantum state into its constituent substates is not a simple operation, partly because it may not be possible, and partly because we swap the qubits around to keep them sorted, as we explain in section 5.3.2.1, and so the vector representing the state may not be separable into the tensor product, even though the state itself is not entangled.

We devise a brute force method for decomposition of vectors into the Kronecker tensor product of two vectors. If a quantum system consists of n qubits then there are $2^{n-1} - 1$ possible pairs of compatible subsystems. Obviously, this number grows extremely fast and it may not be feasible to try to decompose the vector, especially when the quantum state space is huge and many such decomposition checks are necessary (although the compiler keeps track of previous results and does not recompute them).

For this reason we provide the command line option `-Soff`, which disables separation check of states and the function returns always *false*. However, the user must be aware of the fact that in this case only measurements break up quantum registers and properties of individual qubits cannot be verified if they are part of the register, even though they are in well-defined local states.

The challenge is similar to separation of a state on measurement, which we discuss in section 5.3.2.2, but this time both states are unknown. This makes the problem non-linear. However, if the solution exists then it is unique up to a global phase.

The main loop is presented in algorithm 5.12. Matrix A represents the quantum state to be decomposed, matrices B and C are dummy. The function returns *true* if A is separable, in which case matrices B and C will represent the constituent substates. The function returns *false* if the state is entangled. The three matrices are complex column vectors as usual.

The algorithm goes through all $2^{n-1} - 1$ combinations in the worst case when the state is entangled. We encode the register of qubits as a binary number such that qubits with assigned 1 are in the first subsystem and those assigned 0 are in the second one. Combinations are then enumerated with the first qubit always in the first subsytem. For example, register *1*, *2*, *3* can possibly be separated into (*1*; *2*, *3*), (*1*, *3*; *2*) and (*1*, *2*; *3*) with respective enumeration 100, 101 and 110.

In algorithm 5.13 we construct the matrix of coefficients M, which is the core of our method. If this matrix is consistent then the decomposition is possible and we can determine the two vectors. Now, suppose that $B \otimes C = A$ and B represents a state of a register consisting of o qubits, C represents a state of a system containing p qubits and A is a state of a system with $n = o + p$ qubits. Then M is a $2^o \times 2^p$ matrix of which elements A'_k are such that

$$\begin{array}{cccc}
B_0 * C_0 = A'_0 & B_0 * C_1 = A'_1 & \dots & B_0 * C_{2^p-1} = A'_{2^p-1} \\
B_1 * C_0 = A'_{2^p} & B_1 * C_1 = A'_{2^p+1} & \dots & B_1 * C_{2^p-1} = A'_{2^{p+1}-1} \\
\dots & \dots & \dots & \dots \\
B_{2^o-1} * C_0 = A'_{2^{o+p-1}} & B_{2^o-1} * C_1 = A'_{2^{o+p-1}+1} & \dots & B_{2^o-1} * C_{2^p-1} = A'_{2^n-1}
\end{array}$$

for every $A'_k \in TA$, $k \in \{0, \dots, 2^{n-1}\}$, where T is some shift operator as defined in section 5.3.2.2. This gives a system of 2^n non-linear equations in $2^o + 2^p$ variables.

First we consider situation where $A'_k = 0$ for some $k \in \{0, \dots, 2^{n-1}\}$. This implies that if $B_i * C_j = A'_k$ then B_i, or C_j, or both equal 0, and so either the whole row i, or column j, and possibly both have to be 0. If this is not the case then the system of equations has no solution and the algorithm tries another combination of registers or returns *false* if there is not any.

Then we find the last row with a non-zero element A'_k. There is always at least one because A corresponds to some quantum state, and is therefore normalised, and T is unitary. Suppose such a row belongs to $B_{i\star}$, then $B_{i\star}$ does not equal 0 and we can express all C_j as

$$C_0 = \frac{A'_{k\star}}{B_{i\star}}, \qquad C_1 = \frac{A'_{k\star+1}}{B_{i\star}}, \qquad \dots, \qquad C_{2^p-1} = \frac{A'_{k\star+2^p-1}}{B_{i\star}}.$$

Furthermore, all other non-zero coordinates B_i can be expressed in terms of C_j as

$$B_i = \frac{A'_k}{C_0}, \qquad B_i = \frac{A'_{k+1}}{C_1}, \qquad \dots, \qquad B_i = \frac{A'_{k+2^p-1}}{C_{2^p-1}},$$

for all non-zero C_j. Combining the two sets of equations gives

$$B_i = \frac{A'_k}{A'_{k\star}} B_{i\star} = \frac{A'_{k+1}}{A'_{k\star+1}} B_{i\star} = \dots = \frac{A'_{k+2^p-1}}{A'_{k\star+2^p-1}} B_{i\star}$$

Algorithm 5.13 `getCoefficients`(Matrix A, int $iter$)

1: $sizea \leftarrow \log_2(A.rows)$, $sizeb \leftarrow 1$, $pos_0 \leftarrow 1$, $denoms_0 \leftarrow 2^{sizea-1}$
2: **for** $1 \leq i < sizea$ **do**
3: $denom \leftarrow 2^{sizea-i-1}$, $pos_i \leftarrow (iter/denom)\%2$, $sizeb \leftarrow sizeb + pos_i$
4: **if** $pos_i = 1$ **then**
5: $denoms_i \leftarrow denom$
6: **else**
7: $denoms_i \leftarrow 0$
8: **end if**
9: **end for**
10: $sizec \leftarrow sizea - sizeb$
11: $M \leftarrow \text{Matrix}(2^{sizeb}, 2^{sizec})$
12: $nzrow \leftarrow M.rows$
13: **for** $0 \leq i < A.rows$ **do**
14: $row \leftarrow 0$, $idx \leftarrow 0$
15: **for** $0 \leq j < sizea$ **do**
16: **if** $pos_j = 1$ **then**
17: $row \leftarrow row + ((i/denoms_j)\%2) * 2^{sizeb-idx-1}$, $idx \leftarrow idx + 1$
18: **end if**
19: **end for**
20: $M_{row,columns_{row}} \leftarrow A_{i,0}$
21: $columns_{row} \leftarrow columns_{row} + 1$
22: **end for**
23: **repeat**
24: $nzrow \leftarrow nzrow - 1$, $zeroes \leftarrow M_{nzrow,-}$
25: **until not** $zeroes = \text{Zero}(M.columns)$
26: **for** $0 \leq i < nzrow$ **do**
27: **for** $0 \leq j < M.columns$ **do**
28: **if** ($zeroes_j = 0$ **and not** $M_{i,j} = 0$) **then**
29: **return** `null`
30: **else**
31: $M_{i,j} \leftarrow M_{i,j}/M_{nzrow,j}$
32: **if** $M_{i,j} = 0$ **then**
33: $flags_i \leftarrow$ **true**
34: **end if**
35: **end if**
36: **end for**
37: **end for**
38: **for** $0 \leq i < nzrow$ **do**
39: **for** $0 \leq j < M.columns$ **do**
40: **if** ($flags_i$ **and not** $M_{i,j} = 0$) **then**
41: **return** `null`
42: **else if not** ($M_{i,j} = 0$ **or** $M_{i,j} = M_{i,0}$) **then**
43: **return** `null`
44: **end if**
45: **end for**
46: **end for**
47: **return** M

Algorithm 5.14 `solve`(Matrix M)

$B \leftarrow \text{Matrix}(M.rows, 1)$, $nzrow \leftarrow M.rows$, $nzelems \leftarrow \text{Zero}(M.rows)$, $sum \leftarrow 1$
repeat
  $nzrow \leftarrow nzrow - 1$, $zeroes \leftarrow M_{nzrow,-}$, $B_{nzrow,0} \leftarrow 0$
until not $zeroes = \text{Zero}(M.columns)$
for $0 \leq i < nzrow$ **do**
  while ($M_{i,nzelems_i} = 0$ **and** $nzelems_i < M.columns$) **do**
    $nzelems_i \leftarrow nzelems_i + 1$
  end while
  $sum \leftarrow sum + |M_{i,nzelems_i}|^2$
end for
$B_{nzrow,0} \leftarrow \sqrt{1/sum}$
for $0 \leq i < nzrow$ **do**
  $B_{i,0} \leftarrow M_{i,nzelems_i} * B_{nzrow,0}$
end for
return B

for all non-zero B_i. Apparently, all fractions $\frac{A'_l}{A'_{l\star}}$, $l \in \{k, k+2^p-1\}$, $l\star \in \{k\star, k\star+2^p-1\}$, such that $A'_{l\star} \neq 0$ (and consequently $A'_l \neq 0$), must be equal, otherwise the system of equation has no solution.

To construct M for a particular iteration *iter*, we convert decimal representation of the iteration to its binary equivalent. The number of qubits o in the register represented by B is then the number of binary digits 1 plus 1 (because the front qubit is always in the first register) and the number of qubits p in the second register is the number of binary digits 0. We can determine a linear map $A \mapsto A'$ without computing the shift operator T. The following formulae assign for all indices $k \in \{0, \ldots, 2^{n-1}\}$ a unique value to every index $i \in \{0, \ldots, 2^{o-1}\}$ and $j \in \{0, \ldots, 2^{p-1}\}$

$$i = \sum_{l=0}^{n-1} (\lfloor \frac{iter + 2^{n-1}}{2^{n-l-1}} \rfloor \bmod 2) * (\lfloor \frac{k}{2^{n-l-1}} \rfloor \bmod 2) * 2^{o - \sum_{m=0}^{l} (\lfloor \frac{iter+2^{n-1}}{2^{n-m-1}} \rfloor \bmod 2)},$$

$$j = \sum_{l=0}^{n-1} (1 - (\lfloor \frac{iter + 2^{n-1}}{2^{n-l-1}} \rfloor \bmod 2)) * (\lfloor \frac{k}{2^{n-l-1}} \rfloor \bmod 2) * 2^{p - \sum_{m=0}^{l} (1 - (\lfloor \frac{iter+2^{n-1}}{2^{n-m-1}} \rfloor \bmod 2))},$$

such that $A_k \mapsto A'_{i*p+j}$ and $B_i * C_j = A'_{i*p+j}$.

With matrix M in place, the first function `solve` solves the equations for matrix B. So far, the system has infinitely many solutions but additional two equations, which result from the condition that matrices B and C are normal, tackle the problem

$$\sum_{i=0}^{o-1} |B_i|^2 = 1, \qquad \sum_{i=0}^{p-1} |C_i|^2 = 1.$$

For all non-zero B_i we have $B_i = \frac{A'_{k_i}}{A'_{k\star_i}} B_{i\star}$, and so by substitution

$$\sum_{i=0}^{o-1} |B_i|^2 = \sum_{i=0}^{o-1} |\frac{A'_{k_i}}{A'_{k\star_i}} B_{i\star}|^2 = |B_{i\star}|^2 \sum_{i=0}^{o-1} |\frac{A'_{k_i}}{A'_{k\star_i}}|^2 = 1.$$

Algorithm 5.15 `solve`(Matrix A, Matrix B, int $iter$)

1: $C \leftarrow$ Matrix($A.rows/B.rows$), $nzelem \leftarrow 0$, $sizea \leftarrow \log_2(A.rows)$, $column \leftarrow 0$
2: $pos_0 \leftarrow 1$, $denoms_0 \leftarrow 2^{sizea-1}$
3: **while** $B_{nzelem,0} = 0$ **do**
4: $nzelem \leftarrow nzelem + 1$
5: **end while**
6: **for** $1 \leq i < sizea$ **do**
7: $denom \leftarrow 2^{sizea-i-1}$, $pos_i \leftarrow (iter/denom)\%2$
8: **if** $pos_i = 1$ **then**
9: $denoms_i \leftarrow denom$
10: **else**
11: $denoms_i \leftarrow 0$
12: **end if**
13: **end for**
14: **for** $0 \leq i < A.rows$ **do**
15: $row \leftarrow 0$, $idx \leftarrow 0$
16: **for** $0 \leq j < sizea$ **do**
17: **if** $pos_j = 1$ **then**
18: $row \leftarrow row + ((i/denoms_j)\%2) * 2^{sizeb-idx-1}$, $idx \leftarrow idx + 1$
19: **end if**
20: **end for**
21: **if** $row = nzelem$ **then**
22: $C_{column,0} \leftarrow A_{i,0}/B_{nzelem,0}$, $column \leftarrow column + 1$
23: **end if**
24: **end for**
25: **return** C

Thus, we can qualify the modulus of $B_{i\star}$

$$|B_{i\star}| = \frac{1}{\sqrt{\sum_{i=0}^{o-1} |\frac{A'_{k_i}}{A'_{k\star_i}}|^2}}.$$

There is still an infinite number of solutions to the equation, however all the solution belong to the same equivalence class. The general solution can be expressed in form $b_{i\star}(\cos\gamma + i\sin\gamma)$, where $b_{i\star} = |B_{i\star}|$ is a positive real number and $\gamma \in \{0, 2\pi\}$ is a parameter. Because $e^{i\gamma}$ corresponds to the global phase factor and because quantum states are invariant to it, we assume the solution $B_{i\star} = b_{i\star}$. The values of the non-zero coordinates B_i are then $B_i = \frac{A'_{k_i}}{A'_{k\star_i}} b_{i\star}$.

The other `solve` function (see algorithm 5.15) then determines matrix C representing the quantum state of the second system. It simply substitutes the value of $B_{i\star}$ back into the system of equations $C_j = \frac{A'_{k\star_j}}{B_{i\star}}$, giving coordinates $C_j = \frac{A'_{k\star_j}}{b_{i\star}}$.

5.3.3.3 Reduction of Pattern Matrices

Whenever the user declares an operator as a pattern in `Definitions` section, `getPatternMatrix` function computes its matrix representation U. Each *Pattern* is a structure with four members. A set of *Operation* structures *op*, an integer *qc*, identifying a number of qubits in the pattern computation space, and integers *input* and *output* denoting positions of input and output qubits.

Algorithm 5.16 `getPatternMatrix`(Pattern $\mathcal{P}$)

1: $i \leftarrow 0$, $M \leftarrow$ Identity$(2^{\mathcal{P}.qc}, 2^{\mathcal{P}.qc})$
2: $s \leftarrow$ Stack$(i, M, variables)$
3: **while not** s.`empty` **do**
4: $i, M, variables \leftarrow s$.`pop`
5: **if** $\mathcal{P}.op_i =$ `En` **then**
6: $M \leftarrow$ `getEntanglementMatrix`$(\mathcal{P}.op_i.qubit1, \mathcal{P}.op_i.qubit2, \mathcal{P}.qc) * M$
7: **else if** $\mathcal{P}.op_i =$ `Me` **then**
8: $variablesN \leftarrow variables$
9: $M \leftarrow$ `getMeasurementMatrix`$(\mathcal{P}.op_i.qubit, \mathcal{P}.op_i.basis, \mathcal{P}.qc, 0) * M$
10: $variables[\mathcal{P}.op_i.signal] \leftarrow 0$
11: $N \leftarrow$ `getMeasurementMatrix`$(\mathcal{P}.op_i.qubit, \mathcal{P}.op_i.basis, \mathcal{P}.qc, 1) * M$
12: $variablesN[\mathcal{P}.op_i.signal] \leftarrow 1$
13: **else if** $\mathcal{P}.op_i =$ `cX` **then**
14: **if** $variables[\mathcal{P}.op_i.signal] = 1$ **then**
15: $M \leftarrow$ `getCorrectionXMatrix`$(\mathcal{P}.op_i.qubit, \mathcal{P}.qc) * M$
16: **end if**
17: **else if** $\mathcal{P}.op_i =$ `cZ` **then**
18: **if** $variables[\mathcal{P}.op_i.signal] = 1$ **then**
19: $M \leftarrow$ `getCorrectionZMatrix`$(\mathcal{P}.op_i.qubit, \mathcal{P}.qc) * M$
20: **end if**
21: **end if**
22: $i \leftarrow i + 1$
23: **if** $i < \mathcal{P}.pc$ **then**
24: s.`push`$(i, M, variables)$
25: s.`push`$(i, N, variablesN)$
26: **else**
27: $matrices$.`push`(M)
28: $matrices$.`push`(N)
29: **end if**
30: **end while**
31: $U \leftarrow$ `extractPatternMatrix`$(\mathcal{P}, matrices))$
32: **return** U

First, we build all matrices that characterise a trace-preserving map realised by the pattern. There are 2^m such matrices for m measurement commands and the size of the computation space qc sets their dimensions to $2^{qc} \times 2^{qc}$. Matrices are described by a unique string of *variables* representing the measurement outcomes. For example, we define Rx pattern in section 4.2.3 and equations 5.2 to 5.3 outline the matrices with their respective strings of measurement outcomes.

The matrices are constructed iteratively as algorithm 5.16 shows. We initialise the intermediate matrix M to the $2^{qc} \times 2^{qc}$ identity matrix and the assembly begins from the last command and progresses backwards (from right to left). The operator of the corresponding quantum command is computed by tensoring it with appropriate number of identity operators as before. The intermediate matrix M is then multiplied from the left with the operator and passed to the next iteration.

Each measurement branches the run and the value representing the outcome is assign to appropriate signal variable. Other measurements dependent on the signal as well as all corrections are then executed according to its value.

The next step involves isolating the 2×2 matrix of the operator defined by the pattern. Function `extractPatternMatrix`, responsible for this action (see algorithm 5.17), either returns the matrix or throws an exception if the defined pattern is not deterministic and/or not single qubit unitary. The compiler computes all matrices of defined patterns prior to generation of the quantum state space and stores them for later use, so the eventual errors are discovered early on.

The basic idea is that the output states O evolves from the input state I via the operator M such that $MI = \delta O$. Parameter δ is the product of probability amplitudes resulting from projecting I on measurements inside the pattern. Every qubit in I, apart from the input qubit, has default $|+\rangle$ state. All but the output qubit in O collapse to $|+_\alpha\rangle$ or $|-_\alpha\rangle$ state depending on measurement angles and outcomes. The input and output qubits are in some well-defined general local states $\iota_0 |0\rangle + \iota_1 |1\rangle$ and $\omega_0 |0\rangle + \omega_1 |1\rangle$ respectively.

Because all qubits are in well-defined local states, both matrices I and O can be expressed as Kronecker tensor products of individual qubits. Therefore kth row, $k \in \{0, 2^{qc-1}\}$, of δO can be expressed in terms of I and M

$$O_k = (\frac{1}{\sqrt{2}})^{qc-1}(\sum_{l=0}^{2^{qc}-1}(1 - (\lfloor\frac{l}{2^{qc-input-1}}\rfloor \bmod 2)) * M_{k,l}\iota_0 + (\lfloor\frac{l}{2^{qc-input-1}}\rfloor \bmod 2) * M_{k,l}\iota_1). \tag{5.2}$$

Also, we know states of the measured qubits in O because *variables* registers all measurement outcomes, and so an element O_k in terms of coordinates of the output qubit is

$$O_k = ((1 - (\lfloor\frac{k}{2^{qc-output-1}}\rfloor \bmod 2))\omega_0 + (\lfloor\frac{k}{2^{qc-output-1}}\rfloor \bmod 2)\omega_1) \prod_{l=0, l\neq output}^{qc} (-1)^{o_{m_l}} \frac{1}{\sqrt{2}} e^{i\alpha_l}, \tag{5.3}$$

where i is the imaginary unit, o_{m_l} is an outcome of measurement m_l and α_l is either 0 or the angle α_{m_l} of the measurement defined as

$$\alpha_l = (\lfloor\frac{k}{2^{qc-l-1}} \bmod 2)\alpha_{m_l}.$$

Due to the requirement for the pattern $\mathcal{P}$ to realise some single qubit unitary operator U, the state of the output qubit q_o in O must be identical to the state of the input qubit q_i after direct application of the operator U on it. We get two equations

$$U_{0,0}\iota_0 + U_{0,1}\iota_1 = \omega_0, \qquad U_{1,0}\iota_0 + U_{1,1}\iota_1 = \omega_1.$$

Now, there are 2^{qc} equations for ω_0 and ω_1 from MI but only two from Uq_i. This means that if $\mathcal{P}$ realises U then M has only two linearly independent rows. To get away with $e^{i\alpha_l}$ term in equation 5.3 we choose k such that $\alpha_l = 0$ for all l, i.e.

$$k = 0 \qquad \text{and} \qquad k = 2^{qc-output-1}.$$

Clearly, the two are linearly independent since the former involves ω_0 and the latter ω_1, and so we can extract U' from the matrix M as follows

$$U'_{0,0} = \sum_{l=0}^{2^{qc}-1}(1 - (\lfloor\frac{l}{2^{qc-input-1}}\rfloor \bmod 2)) * M_{0,l}, \quad U'_{0,1} = \sum_{l=0}^{2^{qc}-1}(\lfloor\frac{l}{2^{qc-input-1}}\rfloor \bmod 2) * M_{0,l},$$
$$U'_{1,0} = \sum_{l=0}^{2^{qc}-1}(1 - (\lfloor\frac{l}{2^{qc-input-1}}\rfloor \bmod 2)) * M_{k,l}, \quad U'_{1,1} = \sum_{l=0}^{2^{qc}-1}(\lfloor\frac{l}{2^{qc-input-1}}\rfloor \bmod 2) * M_{k,l},$$

Algorithm 5.17 `extractPatternMatrix`(Pattern $\mathcal{P}$, Stack *matrices*)

1: $dim \leftarrow 2^{\mathcal{P}.qc}$, $denom \leftarrow 2^{\mathcal{P}.qc-\mathcal{P}.input}$, $idx \leftarrow 2^{\mathcal{P}.qc-\mathcal{P}.output}$
2: $U \leftarrow \text{Matrix}(2,2)$, $S \leftarrow \text{Matrix}(dim, 2)$, $T \leftarrow \text{Matrix}(2,2)$
3: **for** $0 \leq i < dim$ **do**
4: $\quad flags_i \leftarrow (i/denom)\%2$
5: **end for**
6: **while not** *matrices*.`empty` **do**
7: $\quad M \leftarrow matrices$.`pop`
8: $\quad$ **for** $0 \leq i < dim$ **do**
9: $\quad\quad$ **for** $0 \leq j < dim$ **do**
10: $\quad\quad\quad$ **if** $flags_j = 0$ **then**
11: $\quad\quad\quad\quad S_{i,0} \leftarrow S_{i,0} + M_{i,j}$
12: $\quad\quad\quad$ **else**
13: $\quad\quad\quad\quad S_{i,1} \leftarrow S_{i,1} + M_{i,j}$
14: $\quad\quad\quad$ **end if**
15: $\quad\quad$ **end for**
16: $\quad$ **end for**
17: $\quad S \leftarrow \sqrt{2}^{\mathcal{P}.qc-1} * S$
18: $\quad T_{0,0} \leftarrow S_{0,0}$, $T_{0,1} \leftarrow S_{0,1}$, $T_{1,0} \leftarrow S_{idx,0}$, $T_{1,1} \leftarrow S_{idx,1}$
19: $\quad$ **if not** $T * T$.`hermitian` $= \text{Identity}(2,2)$ **then**
20: $\quad\quad$ **error** "Pattern does not realise unitary operator!"
21: $\quad$ **end if**
22: $\quad$ **if not** $T = U$ **then**
23: $\quad\quad$ **error** "Pattern is not deterministic!"
24: $\quad$ **else**
25: $\quad\quad U \leftarrow T$
26: $\quad$ **end if**
27: **end while**
28: **return** U

where $k = 2^{qc-output-1}$. The final matrix U must be normalised since probability amplitudes δ of measurement projections skew the matrix M, and so $U = \frac{1}{\delta}U'$ unless $\delta = 0$, in which case $\mathcal{P}$ does not realise a unitary operator. Matrix U still need not to be unitary even if $\delta \neq 0$ and we have to check $UU^\dagger = U^\dagger U = I$.

The `extractPatternMatrix` function repeats this procedure for every M generated by its parent function. All extracted operators U must be identical up to the global phase factor $e^{i\gamma}$, more precisely $U_i = e^{i\gamma}U_j$ for all $M_i, M_j \in matrices$ and some $\gamma \in \{0, 2\pi\}$. Otherwise the pattern $\mathcal{P}$ is non-deterministic and measurements are not sufficiently offset by appropriate corrections.

5.4 ISPL Code Generator

Once the analyser processes the reachable quantum state space, the code generator converts the internal representation of the quantum multi-agent system into the valid ISPL code. It goes through the data structure objects of the representation and maps them onto the ISPL. In this section, we describe the backend of the compiler which is realised with relevant functions from `d2i.cc` source file.

First, we check if there are some flags indicating that a measurement action leads to the impossible state. In the positive case, we need to treat the corresponding evolution functions separately for both *Environment* and the normal agents involved. The procedure `processImpossibleStates()` extracts all flagged statements from the set of evolution functions. All these evolution statements have associated the complete global state of the system at the point of the measurement because in general, there may be more measurement events at one time step as well as many different global quantum states corresponding to the time step. The function sorts all these global states with respect to local states of the relevant qubits and/or registers (i.e. the ones that change at the time step), removes all other local states from the global state and then merge the identical cases. Each of these structures together with the time step then comprises a condition in a protocol function of *Environment* agent. Each protocol function has associated unique action of the *Environment* that signals an impossible quantum evolution of the system in the given time step and state.

The next preliminary function `processAgentsEventSequence()` populates *actions*, *protocol* and *evolution* data members (see table 5.1) of all instantiated `agent` objects by iterating through their event sequences. The sequences are represented as vectors of `operation` objects. The `operation` class is a superclass for the individual event classes, namely `cSend`, `cReceive`, `qSend`, `qReceive`, `entanglement`, `xCorrection`, `zCorrection`, `measurement` and `pattern`, and provides a common interface to these. Each event type implements the pure virtual functions declared in the superclass (see table 5.4), which generates the appropriate statements of ISPL code.

Finally, the function `generateISPLCode()` writes the translated ISPL program into the file. The function consists of the following subroutines.

- `generateEnvironment()` produces *Environment* agent. It creates its set of actions and protocol functions if there are any as well as enumeration types corresponding to all qubits and quantum registers in the system and their attainable states. From the set of evolution statements returned by the analyser, it generates evolution functions of *Environment*. This is generally the largest section of the ISPL specification.

- `generateAgents()` iterates through all `agent` objects and writes out the relevant data members to corresponding sections of the agent definition in ISPL.

- `generateEvaluation()` creates `Evaluation` section of ISPL and generates all the atomic propositions with their associated Boolean formulae.

- `generateInitStates()` iterates through all `agent` objects and writes out the initial values of their classical variables. For `qubit` and `register` objects it produces their initial quantum states enumerations.

- `generateGroups()` creates `Groups` section of ISPL only if there are some groups defined. Otherwise does nothing.

- `generateFormulae()` writes out all statements from the set of formulae to the relevant ISPL section. All primitive propositions are replaced by their associated Boolean formula.

Chapter 6

Evaluation

In this chapter, we consider three different quantum security protocols to supplement the evaluation of the implemented system. For each case study, we describe the operational semantics of the underlying DMC network and inspect epistemic properties over the set of possible configurations of the network. We then translate the quantum security protocol specified in our standardised input format into ISPL using dmc2ispl compiler. Finally, we verify the generated code via MCMAS model checker.

At the end of the chapter, we conduct performance analysis, highlight several features and discuss limitations of the approach. Our test environment consists of `GNU Octave` (version 3.2.4) and `MCMAS` (version 1.0.0) software running on a 32-bit Fedora 12 (Constantine) Linux machine with a 2.26GHz Intel® Core™2 Duo P8400 processor and 2.9GiB RAM.

6.1 Case Study A: Quantum Teleportation

Quantum Teleportation was first presented in [6]. It is possibly the simplest protocol to express in DMC because it can be obtained by composition of two Hadamard patterns as we discuss at the end of section 2.3.1.5.

The goal of the quantum teleportation is to transmit a qubit from one party to another with the aid of an entangled pair of qubits and classical resources. In the first step of the protocol, Alice measures her qubits. Next, she sends the measurement outcomes to Bob, after which he applies corrections to his qubit dependent on these outcomes. The result is that Bob's qubit ends up in the same state as Alice's input qubit.

6.1.1 Semantics of Quantum Teleportation Network

First, we have a look at the operational semantics of the quantum teleportation in DMC. We also present the proof from [13] that the following network definition implements the protocol

$$\mathcal{N}_{TP} = \mathbf{A} : \{1,2\}.[(\mathrm{c}!s_2s_1).M_{12}^{0,0}] \mid \mathbf{B} : \{3\}.[X_3^{x_2}Z_3^{x_1}.(\mathrm{c}?x_2x_1)] \parallel E_{23}$$

To derive the operational semantics of the above network we use the small-step semantics given in section 2.3.2.2. The small-step transitions for distributed computations describe how the network evolves over different time steps. The only possibility for the first step is that **A** executes the local measurement $M_{12}^{0,0}$, which requires a local quantum input $|\psi\rangle$ from **A**. The rule 2.7 with $\Gamma_a = \emptyset[s_2s_1 \mapsto j_2j_1]$ derives

$$\frac{|\psi\rangle_1 \otimes E_{23}, M_{12}^{0,0} \longrightarrow_{1/4} X^{j_2}Z^{j_1}|\psi\rangle_3, \Gamma_a}{E_{23}, \emptyset, \mathbf{a} \Longrightarrow_{1/4} X^{j_2}Z^{j_1}|\psi\rangle_3, \Gamma_a, \mathbf{A}.(\mathrm{c}!s_2s_1)},$$

where for each of the values of $j_2 j_1$ the transition occurs with probability of $\frac{1}{4}$. The next step is a classical rendez-vous between both agents. The Rule 2.8 with $\Gamma_b = \emptyset[x_2 x_1 \mapsto j_2 j_1]$ derives:

$$\frac{\Gamma_a(s_2 s_1) = j_2 j_1}{X^{j_2} Z^{j_1} |\psi\rangle_3 \vdash (\Gamma_a, \mathbf{A}.(\mathrm{c}!s_2 s_1) \mid \emptyset, \mathbf{B} : \{3\}.[X_3^{x_2} Z_3^{x_1}.(\mathrm{c}?x_2 x_1)] \Longrightarrow \Gamma_a, \mathbf{A} \mid \Gamma_b, \mathbf{B} : \{3\}.X_3^{x_2} Z_3^{x_1})}.$$

The last step of the computation is the execution of a local pattern (rule 2.7) by agent **B**, as follows:

$$\frac{X^{j_2} Z^{j_1} |\psi\rangle_3, X_3^{x_2} Z_3^{x_1} \longrightarrow_1 |\psi\rangle_3, \Gamma_b}{X^{j_2} Z^{j_1} |\psi\rangle_3, \Gamma_b, \mathbf{B} : \{3\}.X_3^{x_2} Z_3^{x_1} \Longrightarrow |\psi\rangle_3, \Gamma_b, \mathbf{B} : \{3\}}$$

These reductions fire within the context of 2.10, which is omitted for simplicity. The only probabilistic transition is the first one. However, the four branches lead to identical final system state and agents specifications. Adding the probabilities as required, and since there are no classical inputs and outputs, for any quantum input $|\psi\rangle$ the operational semantics is

$$[\![\mathcal{N}]\!]_{TP} : (\{1,2\}, \{3\}) \to (-, \{3\}).\, |\psi\rangle \Longrightarrow |\psi\rangle .$$

This shows that the network indeed implements teleportation of a quantum state between two qubits of the agents.

6.1.2 Temporal and Epistemic Properties of Quantum Teleportation Network

In section 2.3.3 we cover a formal logic framework built on top of DMC. The authors evaluated the quantum teleportation in the paper [16]. We repeat the analysis here to show how a knowledge-based perspective ties together with the network's semantics. We remind the reader that we obtain the configurations in the previous section by following the rules for the small-step semantics.

In the case of quantum teleportation, we have probabilistic transition system due to the Bell measurement. This results in a branching structure. Also, configurations are parametrised by the quantum input $|\psi\rangle$ of the agent **A**. This feature expresses the fact that $|\psi\rangle$ is an unknown quantum state, i.e., neither **A** nor **B** knows anything about it, and so all possible quantum inputs are considered equivalent by the agents. The configurations are labelled by measurement outcomes obtained in the first step of the computation.

$$\begin{aligned}
C_1(|\psi\rangle) &= |\psi\rangle\, E_{23}; \emptyset, \mathbf{A} : \{1,2\}.[(\mathrm{c}!s_2 s_1).M_{12}^{0,0}] \mid \emptyset, \mathbf{B} : \{3\}.[X_3^{x_2} Z_3^{x_1}.(\mathrm{c}?x_2 x_1)], \\
C_2^{j_1 j_2}(|\psi\rangle) &= X^{j_2} Z^{j_1} |\psi\rangle\, ; [s_1 s_2 \mapsto j_1 j_2], \mathbf{A}.(\mathrm{c}!s_2 s_1) \mid \emptyset, \mathbf{B} : \{3\}.[X_3^{x_2} Z_3^{x_1}.(\mathrm{c}?x_2 x_1)], \\
C_3^{j_1 j_2}(|\psi\rangle) &= X^{j_2} Z^{j_1} |\psi\rangle\, ; [s_1 s_2 \mapsto j_1 j_2], \mathbf{A} \mid [x_1 x_2 \mapsto j_1 j_2], \mathbf{B} : \{3\}.X_3^{x_2} Z_3^{x_1}, \\
C_4^{j_1 j_2}(|\psi\rangle) &= |\psi\rangle\, , [s_1 s_2 \mapsto j_1 j_2], \mathbf{A} \mid [x_1 x_2 \mapsto j_1 j_2], \mathbf{B} : \{3\}.
\end{aligned}$$

The figure 6.1 gives a graphical representation of the possible configurations in $\mathcal{C}_{\mathcal{N}_{TP}}$. The temporal accessibility relation between them are denoted with arrows and equivalence classes are represented with boxes for agent **A** and dashed boxes for agent **B**.

We see that configuration $C_1(|\psi\rangle)$ is in its own equivalence class for agent **A**. However, this is in fact a set $\{C_1(|\psi\rangle), |\psi\rangle \in \mathbb{C}^2\}$ of equivalent configurations because **A** does not know $|\psi\rangle$ as stated before. After the measurement **A** distinguishes four possible configurations horizontally at each time step because her local state is different for each measurement outcome. Vertically, that is with respect to the evolution of time, configurations at the first two steps differ because her event sequence has changed but configurations at the third and fourth level are equivalent for **A**, since local operations of agent **B** are not observable by **A**.

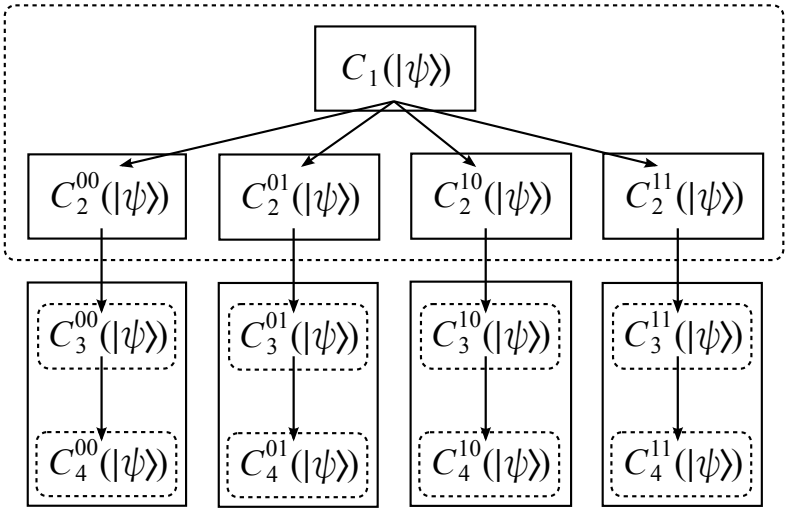

Figure 6.1: The epistemic accessibility relations of agents **A** and **B** in the TP network.

The possibility relation for **B** is different. She considers all configurations at the first two steps equivalent because **B** cannot see what **A** is doing. After she receives the measurement outcomes, **B** distinguishes configurations horizontally via her local state $[x_1x_2 \mapsto j_1j_2]$ and vertically by the change in her event sequence.

The possibility relations of the agents allow us to define the interpretation of facts over this model and to verify various features of the protocol.

$$C_1(|\psi\rangle), \mathcal{N}_{TP} \vDash A\Diamond(q_3 = init(q_1)), \tag{6.1}$$
$$C_1(|\psi\rangle), \mathcal{N}_{TP} \vDash A\Diamond K_{\mathbf{B}}(q_3 = init(q_1)), \tag{6.2}$$
$$C_1(|\psi\rangle), \mathcal{N}_{TP} \vDash \neg E\Diamond K_{\mathbf{A}}(q_3 = init(q_1)), \tag{6.3}$$
$$C_1(|\psi\rangle), \mathcal{N}_{TP} \vDash \neg E\Diamond K_{\mathbf{A}}(q_3 = |\psi\rangle) \wedge \neg E\Diamond K_{\mathbf{B}}(q_3 = |\psi\rangle). \tag{6.4}$$

Firstly, equation 6.1 states that the $\mathcal{N}_{TP}$ network is correct, since the state of the **B**'s qubit will eventually be equal to the initial state of **A**'s qubit. The other three formulae deal with epistemic properties of the network. Equation 6.2 expresses the fact that **B** eventually knows the third qubit is equal to the initial state of the first qubit, equation 6.3 asserts that **A** never knows this fact. And the equation 6.4 states that neither of the agents knows the actual quantum state of the third qubit at any point of the computation.

We can easly see from the temporal and epistemic accessibility relations that the first two formulae are true in the model. The reason that the last formula holds is that configurations $C_1(|\psi\rangle)$ are considered equivalent for all input states $|\psi\rangle$ by both agents, and so they cannot conclude anything about its properties.

However, formula 6.3 is not true in the model, because even though **A** cannot distinguish $C_3^{00}(|\psi\rangle)$ from $C_4^{00}(|\psi\rangle)$, primitive formula $q_3 = init(q_1)$ holds in both configurations for the reason that **B** does not apply any corrections for measurement outcomes $[s_1s_2 \mapsto 00]$, and so the quantum state of the system is invariant along this path. Authors of the paper [16] overlooked this fact but our `dmc2ispl` compiler together with `MCMAS` verify the formula correctly.

6.1.3 Automatic Verification of Quantum Teleportation

In order to verify these properties automatically in `MCMAS`, we need to translate the network specification from DMC into ISPL. This is done by our `dmc2ispl` compiler, which reads

```
1  -- AGENTS
2  Alice: {1, 2},
3         {},
4         {},
5         {c!(Bob,s2), c!(Bob,s1), Me(2,0,-,-,s2), Me(1,0,-,-,s1), En(1,2)};
6
7  Bob: {3},
8       {},
9       {},
10      {cX(3,s2), cZ(3,s1), c?(Alice,s2), c?(Alice,s1)};
11
12 -- QUBITS
13 1: ?;
14 2, 3: {(0.5,0), (0.5,0), (0.5,0), (-0.5,0)};
15
16 -- FORMULAE
17 AG ({Bob >4} -> {3 = init(1)});
18 AG ({Bob >4} -> K(Bob, {3 = init(1)}));
19 EF ({Bob >4} -> K(Alice, {3 = init(1)}));
20 !AG ({Bob >4} -> K(Alice, {3 = init(1)}));
21 !EF K (Alice, {3}) and !EF K (Bob, {3});
```

Listing 6.1: teleport.dmc

a dedicated modular programming language derived from DMC. We can formulate the teleportation protocol in our input format by the description given in listing 6.2.

There are few notable differences from the original DMC specification. Firstly, the initial quantum state of the entangled pair must be defined explicitly in the code. In the case of quantum teleportation, the shared entangled resource is in state $\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle - \frac{1}{2}|11\rangle$. Also, the fact that the first qubit is in an arbitrary state must be explicitly indicated by '?', otherwise the compiler would assume that the qubit is initialised in $|+\rangle$ state.

Not all the features of our input language are used in this example. For instance, neither of the agent have any prior knowledge of any of the qubit's quantum state, nor they have any classical input bits. Also, the definition of patterns and groups sections of the code are empty.

We verify five specifications of the Quantum Teleportation. The first formula (line 17) corresponds to equation 6.1 and states that after Bob executes all his local operations the state of qubit 3 will always be the same as the initial state of qubit 1. The formula on line 18 matches equation 6.2 and claims that Bob will always know that the states are equal. The next formula (line 19) states that Alice may know that the states are the same, which is the opposite of what equation 6.3 claims. However formula on line 20 asserts that she will not always know that. The last formula (line 21) corresponds to equation 6.4 and states that neither agent actually knows the actual state of qubit 3.

We then compile the DMC source file with our `dmc2ispl` tool. The translated ISPL code is displayed in listing D.1 in the appendix D. We point out several interesting features of the translation. First note, that there are 10 unique well-defined local states of qubits and registers throughout all possible executions of the protocol. We learn this information by inspecting the definitions of variables of *Environment* agent on beginning on line 2.

From the value of the global program counter on line 9 we know that there are 8 configurations in the sequence. When we illustrate the accessibility relations of agent in the previous section, we used only 4. The reason there is a difference is that agents perform only one local event per time step in our automated approach but the whole pattern in the configurations derived by hand. The manual configuration trees are cruder. We explain this issue in more detail in

```
************************************************************************
                        MCMAS v1.0.0

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check http://www-lai.doc.ic.ac.uk/mcmas/ for the latest release.
 Report bugs to <mcmas@imperial.ac.uk>
************************************************************************

teleport.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building OBDD for initial states...
Building reachable state space...
Checking formulae...
Verifying properties...
  Formula number 1: (AG (f0 -> f1)), is TRUE in the model
  Formula number 2: (AG (f0 -> K(Bob, f1))), is TRUE in the model
  Formula number 3: (EF (f0 -> K(Alice, f1))), is TRUE in the model
  Formula number 4: (! (AG (f0 -> K(Alice, f1)))), is TRUE in the model
  Formula number 5: ((! (EF K(Alice, f2))) && (! (EF K(Bob, f3)))), is TRUE in
      the model
done, 5 formulae successfully read and checked
execution time = 0
number of reachable states = 104
BDD memory in use = 5016164
```

Listing 6.2: `MCMAS` output for the Quantum Teleportation protocol.

section 4.1.2.1. The set of action of *Environment* agent on line 11 is empty, indicating that no impossible states were encountered during the generation of the reachable quantum state space. The atomic propositions in formulae are replaced by Boolean variables f_i and defined separately in the `Evaluation` section beginning on line 93.

The actual verification of the protocol is done by invocation of `MCMAS` with default options. The listing 6.2 contains the output of the modal checker with the evaluated formulae. All of them are true in the model as expected. This shows that equation 6.3 is evaluated wrongly in the paper [16].

6.2 Case Study B: Quantum Key Distribution

We introduce a version of Quantum Key Distribution protocol in section 2.1.5, however we adhere to a modification of the QKD protocol by Ekert [19] throughout this chapter.

The Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is distributed among agents Alice and Bob. The Bell state has the property that when measured in arbitrary but identical bases by the agents, both of them are guaranteed to obtain identical measurement outcomes. Although no-one can predict which of both outcomes, 0 or 1, Alice and Bob obtain. Therefore the agents randomly measure their qubits in $\{|0\rangle, |1\rangle\}$ or $\{|+\rangle, |-\rangle\}$ basis and tell each other via the classical channel, which basis they used. Alice and Bob then only keep their measurement outcomes if they measured in the identical basis. Only in this case it is guaranteed that their measurement outcomes are equal. These values are then kept as part of a secret key shared between the agents. A secret

key of adequate length is established by iterating this protocol as many times as required.

6.2.1 Semantics of QKD Network

We now describe the small-step semantics of the $\mathcal{N}_{QKD}$ network. The full specification of one step of the protocol is defined in the reference paper [12] as follows

$$\mathcal{N}_{QKD} = \mathbf{A}(a) : \{1\}.[(\mathrm{c}!a)(\mathrm{c}?b).M_1^0\mathcal{H}^a] \mid \mathbf{B}(b) : \{2\}.[(\mathrm{c}?a)(\mathrm{c}!b).M_2^0\mathcal{H}^b] \parallel |\Phi^+\rangle_{12},$$

The agents $\mathbf{A}$ and $\mathbf{B}$ are parametrised by Boolean variables a, respective b. Values of the variables represent the random choice of the measurement basis. We can express this random measurement in DMC by Hadamard patterns conditioned on the values of a and b. This is reflected in the names, $\mathbf{A}(a)$ and $\mathbf{B}(b)$, of the agents. Also note, that using Hadamard pattern $\mathcal{H}$ in this manner is a simplification of notation because in DMC we cannot express compound quantum commands like that.

From the initial configuration we go through the protocol step by step applying the formal semantics for networks to newly derived configurations. Agents execute their local quantum events concurrently in the firs step. Therefore we have to employ the rule 2.7 twice in the context of rule 2.10. For agent $\mathbf{A}$ we derive

$$\frac{|\Phi^+\rangle_{12}, M_1^0\mathcal{H}_1^a \longrightarrow_{1/2} \mathbf{0}, \emptyset[s_1 \mapsto j_1]}{|\Phi^+\rangle_{12}, \emptyset, \mathbf{A}(a) : \{1\}.[(\mathrm{c}!a)(\mathrm{c}?b).M_1^0\mathcal{H}_1^a] \Longrightarrow_{1/2} \mathbf{0}, \emptyset[s_1 \mapsto j_1], \mathbf{A}(a).[(\mathrm{c}!a)(\mathrm{c}?b)]},$$

and similarly for agent $\mathbf{B}$ we have

$$\frac{|\Phi^+\rangle_{12}, M_2^0\mathcal{H}_2^b \longrightarrow_{1/2} \mathbf{0}, \emptyset[s_2 \mapsto j_2]}{|\Phi^+\rangle_{12}, \emptyset, \mathbf{B}(b) : \{2\}.[(\mathrm{c}?a)(\mathrm{c}!b).M_2^0\mathcal{H}_2^b] \Longrightarrow_{1/2} \mathbf{0}, \emptyset[s_2 \mapsto j_2], \mathbf{B}(b).[(\mathrm{c}?a)(\mathrm{c}!b)]}.$$

Both agents have carried out their local quantum operations and obtained measurement outcomes j_1 and j_2. The network quantum state is now the null state $\mathbf{0}$ and agents' types disappear because measurements are destructive in patterns. There is branching due to varying measurement outcomes. In the next and final step of the computation Alice and Bob exchange the measurement basis and add the values of b, respectively a, to their local states. For classical communication we apply rule 2.8 as follows

$$\frac{\Gamma_a(\bar{a}) = a, \Gamma_b(\bar{b}) = b}{\mathbf{0} \vdash (\Gamma_a, \mathbf{A}(a).[(\mathrm{c}!a)(\mathrm{c}?b)] \mid \Gamma_b, \mathbf{B}(b).[(\mathrm{c}?a)(\mathrm{c}!b)] \Longrightarrow \Gamma_a[\bar{b} \mapsto b], \mathbf{A}(a) \mid \Gamma_b[\bar{a} \mapsto a], \mathbf{B}(b))},$$

where $\bar{a}$ and $\bar{b}$ are variable names.

6.2.2 Temporal and Epistemic Properties of QKD Network

After construction of all the configurations that may potentially occur during the execution of QKD we can evaluate properties of the protocol from a knowledge-based perspective. To summarise, there are the following configurations

$$C_1(a,b) = |\Phi^+\rangle_{12}; \emptyset, \mathbf{A}(a) : \{1\}.[(\mathrm{c}!a)(\mathrm{c}?b).M_1^0\mathcal{H}_1^a] \mid \emptyset, \mathbf{B}(b) : \{2\}.[(\mathrm{c}?a)(\mathrm{c}!b).M_2^0\mathcal{H}_2^b]$$
$$C_2(a,b) = \mathbf{0}; [s_1 \mapsto j_1], \mathbf{A}(a).[(\mathrm{c}!a)(\mathrm{c}?b)] \mid [s_2 \mapsto j_2], \mathbf{B}(b).[(\mathrm{c}?a)(\mathrm{c}!b)]$$
$$C_3(a,b) = \mathbf{0}; [s_1 \mapsto j_1, \bar{b} \mapsto b], \mathbf{A}(a) \mid [s_2 \mapsto j_2, , \bar{a} \mapsto a], \mathbf{B}(b)$$

Schematically we have two possible structures of configuration trees for the protocol, two of each type. These are represented in figure 6.2 for $a = b$ and $a \neq b$ respectively. We do not

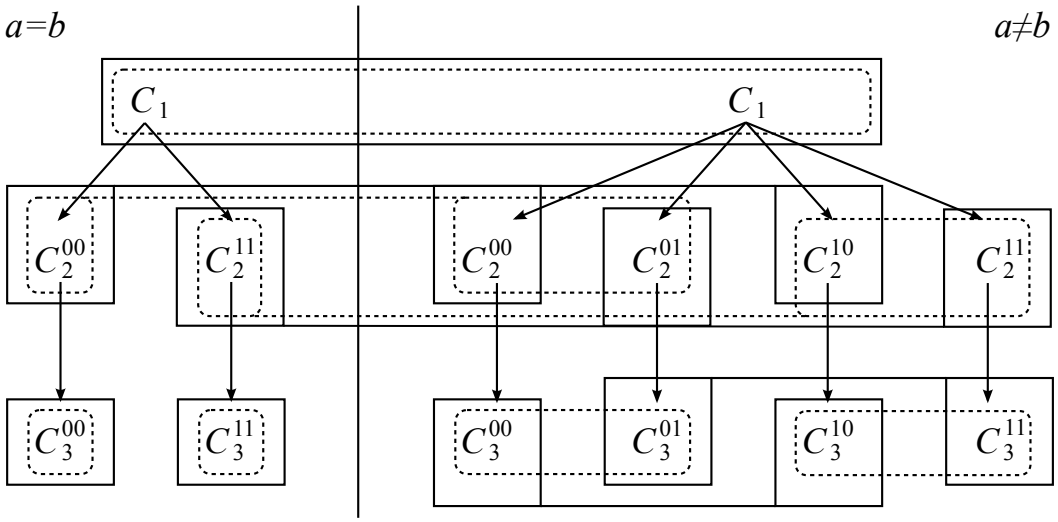

Figure 6.2: The epistemic accessibility relations of agents **A** and **B** in the QKD network.

explicitly write the dependency of each configuration on the parameters for conciseness. In the figure dotted boxes depict the equivalence classes for Alice and boxes picture equivalence classes for Bob.

Configurations across both trees with fixed a are in the same equivalence class for Alice, and similarly for Bob with fixed b. Vertically, each level is recognised by both agents because they perform a local event. Horizontally, different measurement outcomes discriminate between the configurations. Configuration at the second level in which Alice, respectively Bob, has a particular measurement outcome are equivalent across both trees for the same value of a, respectively b. At the final level they exchange the values of a and b and can distinguish in which configuration tree they are in. We are now able to formally prove the following statements

$$a = b \Rightarrow C_1(a,b), \mathcal{N}_{QKD} \vDash A\Diamond(K_{\mathbf{A}}(s_1 = s_2) \wedge K_{\mathbf{B}}(s_1 = s_2), \tag{6.5}$$

$$a \neq b \Rightarrow C_1(a,b), \mathcal{N}_{QKD} \vDash \neg E\Diamond(K_{\mathbf{A}}(s_1 = s_2) \vee K_{\mathbf{B}}(s_1 = s_2)). \tag{6.6}$$

Equation 6.5 asserts that if $a = b$ then the knowledge of both agents is identical at the final point. This is true since in the last level of the left tree, Agent and Bob have identical single-configuration equivalence classes. This means that in this case one bit of a secret key is successfully established. The equation 6.6 states that if $a \neq b$ the agents cannot be ever sure that this bit of a key has the same value for both of them. This formula holds because they cannot distinguish configurations with different measurement outcome of the other agent in the right configuration tree.

6.2.3 Automatic Verification of QKD

In listing 6.3, we illustrate specification of the Quantum Key Distribution protocol using our input format. This time we utilise the `Definition` feature of our language because we need to express that execution of Hadamard pattern $\mathcal{H}$ is dependent on the parameter of the agent. Therefore we define the pattern on line 2 and the agent later make use of the pattern in their respective event sequences on lines 8 and 13. The parameters a and b themselves are defined in the agents' classical input sets on lines 7 and 12. The symbol question mark symbol (?) denotes that the variables may have both values initially, and so we have four distinct initial configurations.

```
1  -- DEFINITIONS
2  #Ha: {1}, {2}, {cX(2,s1), Me(1,0,-,-,s1), En(1,2)};
3
4  -- AGENTS
5  Alice: {1},
6         {},
7         {a: ?},
8         {c!(Bob,a), c?(Bob,b), Me(1,0,-,-,s1), Ha(1,a)};
9
10 Bob: {2},
11       {},
12       {b: ?},
13       {c?(Alice,a), c!(Alice,b), Me(2,0,-,-,s2), Ha(2,b)};
14
15 -- QUBITS
16 1, 2: {(sqrt(0.5),0), (0,0), (0,0), (sqrt(0.5),0)};
17
18 -- FORMULAE
19 AG (({Alice(a) = Bob(b)} and {Bob >4}) ->
20      (K(Alice, {Alice(s1) = Bob(s2)}) and K(Bob, {Alice(s1) = Bob(s2)})));
21 AG (!{Alice(a) = Bob(b)} ->
22      (!K(Alice, {Alice(s1) = Bob(s2)}) and !K(Bob, {Alice(s1) = Bob(s2)})));
```

Listing 6.3: qkd.dmc

The shared entangled resource $|\Phi^+\rangle_{12}$ is explicitly defined on line 16 and each agent have one qubit of the pair in their input sort. We verify two temporal epistemic specifications. The first formula (line 19) corresponds to equation 6.5 and states that after the execution of the protocol both agents will know the established bit of the key if they have measured in the same basis. The second formula (line 21) matches equation 6.6 and claim that if the agent have measured in disparate basis they will never be sure if the measurement outcomes agree.

When we translate the DMC input file into its ISPL counterpart with `dmc2ispl` compiler we obtain the specification of the protocol described in listing D.2 in appendix D. In this case there are only 6 distinct well-defined quantum states. The entangled resource (line 6) attains two possible states. The first one corresponds to the case that both agents measure in the same basis and the other to the case when each agent measures in a different basis[1].

The Bob's qubit (line 5) has four reachable states as the enumeration value of its associated variable but in the evolution section (line 14) the two states are never assigned to it. This is because these states are intermediate states in which the qubit may exist just before it is measured by Bob. It happens when one of the agent applies the Hadamard pattern. Even though the measurements occur in the same time step, Alice acts first and her measurement forces the other qubit through entanglement to one of these states. But Bob then immediately measures it and the qubit collapses to one of the base state.

Other implication of measuring the Bell state $|\Phi^+\rangle_{12}$ is that Bob's measurement outcome depends on the outcome of Alice. Therefore if the register E_{12} is still in this state, the *Environment* agent signals it its action defined on line 11. In this case the assignment to Bob's variable s_2 in the evolution function (line 77) is subject to the foregoing Alice's action.

The second phase of the verification process is then done via `MCMAS` as before. Again, both formulae are true in the model as shown in listing 6.4 of the model checker's output.

[1]Note that, strictly speaking, both agents measures in the same $\{|+\rangle, |-\rangle\}$ basis and we mean that one of the agents has applied the Hadamard pattern before the measurement when we say that the measurement basis differ.

```
************************************************************************
                        MCMAS v1.0.0

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check http://www-lai.doc.ic.ac.uk/mcmas/ for the latest release.
 Report bugs to <mcmas@imperial.ac.uk>
************************************************************************

qkd.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building OBDD for initial states...
Building reachable state space...
Checking formulae...
Verifying properties...
  Formula number 1: (AG ((f0 && f1) -> (K(Alice, f2) && K(Bob, f2)))), is TRUE
      in the model
  Formula number 2: (AG ((! f0) -> ((! K(Alice, f2)) && (! K(Bob, f2))))), is
      TRUE in the model
done, 2 formulae successfully read and checked
execution time = 0
number of reachable states = 348
BDD memory in use = 5068580
```

Listing 6.4: `MCMAS` output for the QKD protocl.

6.3 Case Study C: Superdense Coding

The last case study is Superdense Coding protocol, presented in paper [7]. The specification in DMC is the longest of the three protocols because it requires, among others, $^{\mathrm{C}}$NOT pattern.

The aim of Superdense Coding is to transmit two classical bits from one party to the other with the aid of one entangled qubit pair. In the first step of the protocol Alice transforms her half of the entangled pair in a different way for each of the four possible classical inputs x_1 and x_2. Then she sends the qubit to Bob, who then measures the entangled pair. At the end of the protocol the measurement outcomes s_1 and s_2 are equal to Alice's inputs.

6.3.1 Semantics of Superdense Coding Network

As usual, we start with the semantics of the protocol. In the reference paper [16], the Superdense Coding network is defined as follows

$$\mathcal{N}_{SC} = \mathbf{A} : \{1\}.[(\mathrm{qc}!1).X_1^{x_2} Z_1^{x_1}] \mid \mathbf{B} : \{2\}.[M_{12}^{0,0}.(\mathrm{qc}?1)] \parallel |\Phi^+\rangle_{12},$$

however this is a simplification because by definition in the paper, $M_{12}^{0,0}$ is a Bell measurement on qubits 1 and 2, and not measurement in $\{|+\rangle, |-\rangle\}$ basis as we would expect. But since our compiler follows the rules of DMC strictly, we need to decompose the measurement event. Bell

basis [42] consists of the following four maximally entangled pairs of qubits

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle), \qquad |\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle),$$
$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle), \qquad |\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle),$$

and can be formed from the $\{|{+}{+}\rangle, |{+}{-}\rangle, |{-}{+}\rangle, |{-}{-}\rangle\}$ basis by combination of Hadamard and $^{\mathrm{C}}$NOT operators

$$U = (I \otimes H) * {}^{C}NOT = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}.$$

To decompose the measurement event $M_{12}^{0,0}$, we need to perform a reverse operation and then measure the two qubits of the pair separately. Since U is the unitary operator, $UU^\dagger = I$ and $U^\dagger = ((I \otimes H) * {}^{C}NOT)^\dagger = ({}^{C}NOT)^\dagger * (I \otimes H)^\dagger = {}^{C}NOT * (I^\dagger \otimes H^\dagger) = {}^{C}NOT * (I \otimes H)$ is the reverse operator. The definition of ${}^{C}\mathcal{X}$ ($^{\mathrm{C}}$NOT) pattern is given in the paper [15] as

$${}^{C}\mathcal{X} ::= (\{1,2,3,4\}, \{1,2\}, \{1,4\}, X_4^{s_3} Z_4^{s_2} Z_1^{s_2} M_3^0 M_2^0 E_{13} E_{23} E_{34}),$$

and in combination with Hadamard pattern, defined in section 2.3.1.1 and applied on qubit 4 $\mathcal{H} ::= (\{4,5\}, \{4\}, \{5\}, X_5^{s_4} M_4^0 E_{45})$, we obtain the decomposition pattern[2]

$$\mathcal{D} ::= (\{1,2,3,4,5\}, \{1,2\}, \{1,5\}, X_5^{s_4} M_4^0 E_{45} X_4^{s_3} Z_4^{s_2} Z_1^{s_2} M_3^0 M_2^0 E_{13} E_{23} E_{34}).$$

Since the output of the pattern consists of qubits 1 and 5, the measurement event changes to $M_{15}^{0,0}$ where the zeroes denote the angle of measurement as usual. Putting everything together, we can now rewrite the specification of the Superdense Coding network as follows

$$\mathcal{N}_{SC} = \mathbf{A} : \{1\}.[(\mathrm{qc}!1) X_1^{x_2} Z_1^{x_1}] \mid \mathbf{B} : \{2\}.[M_{15}^{0,0} \mathcal{D}.(\mathrm{qc}?1)] \parallel |\Phi^+\rangle_{12}.$$

To derive the operational semantics of the network we once again use the four small-step rules. In the first step agent **A** prepares the Bell state according to her input classical bits $\Gamma_a = [x_2 x_1 \mapsto j_2 j_1]$. From the rule 2.7 we derive:

$$\frac{|\Phi^+\rangle_{12} \bigotimes_{i=3}^{5} |+\rangle_i, \Gamma_a, X_1^{x_2} Z_1^{x_1} \longrightarrow_1 X^{j_2} Z^{j_1} E_{12} \bigotimes_{i=3}^{5} |+\rangle_i, \Gamma_a}{|\Phi^+\rangle_{12} \bigotimes_{i=3}^{5} |+\rangle_i, \Gamma_a, \mathbf{a} \Longrightarrow_1 X^{j_2} Z^{j_1} |\Phi^+\rangle_{12} \bigotimes_{i=3}^{5}, \Gamma_a, \mathbf{A}.(\mathrm{qc}!1)},$$

where for each of the values of $j_2 j_1$ the transition occurs with probability of 1 because an action without measurement is always deterministic. In the next step agent **A** sends agent **B** her qubit of the entangled pair. The rule 2.9 gives

$$\overline{X^{j_2} Z^{j_1} |\Phi^+\rangle_{12} \bigotimes_{i=3}^{5} \vdash (\Gamma_a, \mathbf{A} : \{1\}.(\mathrm{qc}!1) \mid \emptyset, \mathbf{B} : q_1.[M_{15}^{0,0} \mathcal{D}(\mathrm{qc}?1)] \Longrightarrow \Gamma_a, \mathbf{A} \mid \emptyset, \mathbf{B} : q_2.M_{15}^{0,0} \mathcal{D})}$$

with $q_1 = \{2, \ldots, 5\}$ and $q_2 = \{1, \ldots, 5\}$.

In the last step of the computation agent **B** performs the computation pattern $\mathcal{D}$ together with the local measurements $M_{15}^{0,0}$ and we infer the final configuration of the network from the rule 2.7 as follows

$$\frac{X^{j_2} Z^{j_1} |\Phi^+\rangle_{12} \bigotimes_{i=3}^{5}, M_{15}^{0,0} \mathcal{D} \longrightarrow_1 \mathbf{0}, \emptyset[s_2 s_1 \mapsto j_2 j_1]}{X^{j_2} Z^{j_1} |\Phi^+\rangle_{12} \bigotimes_{i=3}^{5}, \emptyset, \mathbf{B} : q_2.M_{15}^{0,0} \mathcal{D} \Longrightarrow \mathbf{0}, \emptyset[s_2 s_1 \mapsto j_2 j_1], \mathbf{B}}$$

[2]Note, that we do not rewrite the pattern to the standard EMC form.

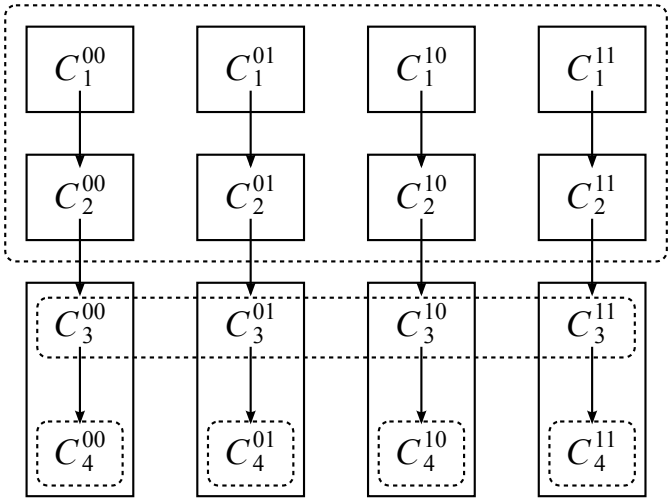

Figure 6.3: The epistemic equivalence relation of agents **A** and **B** in the SC network.

We again leave out the meta-rule 2.10 for simplicity. There are no probabilistic transitions in this network even though agent **B** measures all five qubits. There are two reasons for this. Firstly, computation pattern $\mathcal{D}$ is composed of deterministic patterns $\mathcal{H}$ and ${}^{C}\mathcal{X}$, therefore is deterministic also. And secondly, we are measuring a Bell state in Bell basis, which means that after the decomposition qubits 1 and 5 are either in $|+\rangle$ or $|-\rangle$ state and their measurements are deterministic. Since there are no quantum outputs, the operational semantics for any classical input $[x_2x_1 \mapsto j_2j_1]$ is

$$[\![\mathcal{N}]\!]_{SC} : (\{1\}, \{2 \ldots 5\}) \to (-,-).\, |\Phi^+\rangle\,, [x_2x_1 \mapsto j_2j_1] \mid \emptyset \Longrightarrow \mathbf{0}, [x_2x_1 \mapsto j_2j_1] \mid [s_2s_1 \mapsto j_2j_1].$$

6.3.2 Temporal and Epistemic Properties of Superdense Coding Network

The analysis of temporal and epistemic properties of Superdense Coding in the context of DMC has been done in the paper [16] and we will closely follow it although our specification of the network is slightly different as we explain above. We have the following configurations in the $\mathcal{C}_{\mathcal{N}_{SC}}$ network

$$\begin{aligned}
C_1^{j_1j_2} &= |\Phi^+\rangle_{12}\,; [x_1x_2 \mapsto j_1j_2], \mathbf{A} : \{1\}.[(\text{qc!1})X_1^{x_2}Z_1^{x_1}] \mid \emptyset, \mathbf{B} : \{2\}.[M_{15}^{0,0}\mathcal{D}(\text{qc?1})],\\
C_2^{j_1j_2} &= X_1^{x_2}Z_1^{x_1}\,|\Phi^+\rangle_{12}\,; [x_1x_2 \mapsto j_1j_2], \mathbf{A} : \{1\}.(\text{qc!1}) \mid \emptyset, \mathbf{B} : \{2\}.[M_{15}^{0,0}\mathcal{D}(\text{qc?1})],\\
C_3^{j_1j_2} &= X_1^{x_2}Z_1^{x_1}\,|\Phi^+\rangle_{12}\,; [x_1x_2 \mapsto j_1j_2], \mathbf{A} \mid \emptyset, \mathbf{B} : \{1,2\}.M_{15}^{0,0}\mathcal{D},\\
C_4^{j_1j_2} &= \mathbf{0}, [x_1x_2 \mapsto j_1j_2]; \mathbf{A} \mid [s_1s_2 \mapsto j_1j_2], \mathbf{B},
\end{aligned}$$

where j_1j_2 is equal to the input values 00, 01, 10 or 11. The equivalence relation of both agents for configurations in $\mathcal{C}_{\mathcal{N}_{SC}}$ is represented in figure 6.3, with arrows for computation paths, boxes for **A**'s equivalence classes and dashed boxes for **B**'s equivalence classes as before. Note that the $\mathcal{N}_{SC}$ network is correct, since

$$\forall j_1, j_2 : C_1^{j_1j_2}, \mathcal{N}_{SC} \vDash A\Diamond(s_1s_2 = j_1j_2).$$

A distinguishes four possible configurations at each time step horizontally because Γ_a is different for each input value. Vertically, that is with respect to the evolution of time, configurations differ whenever **A**'s event sequence changes. All configurations occurring at

```
1  -- AGENTS
2  Alice: {1},
3         {},
4         {x1: ?, x2: ?},
5         {qc!(Bob,1), cX(1,x2), cZ(1,x1)};
6
7  Bob: {2},
8       {},
9       {},
10      {Me(5,0,-,-,s5), Me(1,0,-,-,s1),
11       cX(5,s4), Me(4,0,-,-,s4), En(4,5), --Hadamard gate
12       cX(4,s3), cZ(4,s2), cZ(1,s2), Me(3,0,-,-,s3), Me(2,0,-,-,s2),
13                 En(1,3), En(2,3), En(3,4), --controlled X gate
14       qc?(Alice,1)
15      };
16
17 -- QUBITS
18 1, 2: {(sqrt(0.5),0), (0,0), (0,0), (sqrt(0.5),0)};
19
20 -- FORMULAE
21 AG ({Bob >14} -> ({Alice(x1) = Bob(s1)} and {Alice(x2) = Bob(s5)}));
22 AG ({Bob >14} -> K(Bob, {Alice(x1) = Bob(s1)} and {Alice(x2) = Bob(s5)}));
23 AG ({Bob <14} -> !K(Bob, {Alice(x1) = Bob(s1)} and {Alice(x2) = Bob(s5)}));
24 AG (!K(Alice, K(Bob, {Alice(x1) = Bob(s1)} and {Alice(x2) = Bob(s5)})));
```

Listing 6.5: superdc.dmc

the first two steps are considered equivalent by **B** because neither its local state nor event sequence change. In the third step the event sequence of **B** changes but not its local state and so configurations C_3 are equivalent to each other but different from the previous ones. In the last step all configurations C_4 are unique because both, her local state and event sequence, change when **B** finds out **A**' input values.

The possibility relations of both agents allow for derivation of several epistemic statements:

$$\forall j_1, j_2 : C_1^{j_1 j_2}, \mathcal{N}_{SC} \vDash A\Box K_{\mathbf{A}}(x_1 x_2 = j_1 j_2), \tag{6.7}$$

$$\forall j_1, j_2, t < 4 : C_t^{j_1 j_2}, \mathcal{N}_{SC} \vDash \neg K_{\mathbf{B}}(s_1 s_2 = j_1 j_2), \tag{6.8}$$

$$\forall j_1, j_2 : C_1^{j_1 j_2}, \mathcal{N}_{SC} \vDash A\Diamond K_{\mathbf{B}}(s_1 s_2 = j_1 j_2), \tag{6.9}$$

$$\forall j_1, j_2 : C_1^{j_1 j_2}, \mathcal{N}_{SC} \vDash \neg E\Diamond K_{\mathbf{A}} K_{\mathbf{B}}(s_1 s_2 = j_1 j_2). \tag{6.10}$$

Statement 6.7 trivially means that **A** always knows its input values, in fact, agents always know their own input values in any protocol. On the other hand, statement 6.8 says that **B** does not know the inputs in the first three steps, but, according to statement 6.9, **B** will eventually know it. Last statement 6.10 expresses that **A** never knows that **B** eventually knows the input values because **A** cannot know when **B** applies its local measurement.

6.3.3 Automatic Verification of Superdense Coding

The specification of the Superdense Coding in our input version of DMC is given in listing 6.5. There are few interesting features in the code. Alice has two classical input variables x_1 and x_2 defined in line 4. Both variables have two possible initial values, 0 and 1, and so there are four distinct initial states. This represents the four values that Alice wants to send to Bob.

In the previous protocol, the Hadamard pattern is declared as a macro in the `Definition` module but this time it is part of Bob's event sequence (line 11). A more complex pattern is the

```
************************************************************************
                        MCMAS v1.0.0

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check http://www-lai.doc.ic.ac.uk/mcmas/ for the latest release.
 Report bugs to <mcmas@imperial.ac.uk>
************************************************************************

superdc.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building OBDD for initial states...
Building reachable state space...
Checking formulae...
Verifying properties...
  Formula number 1: (AG (f0 -> (f1 && f2))), is TRUE in the model
  Formula number 2: (AG (f0 -> K(Bob, (f1 && f2)))), is TRUE in the model
  Formula number 3: (AG (f3 -> (! K(Bob, (f1 && f2))))), is TRUE in the model
  Formula number 4: (AG (! K(Alice, K(Bob, (f1 && f2))))), is TRUE in the model
done, 4 formulae successfully read and checked
execution time = 0
number of reachable states = 2192
BDD memory in use = 5502532
```

Listing 6.6: `MCMAS` output for the Superdense Coding protocol.

controlled-X, or controlled-NOT, defined on line 12. Together these operations comprise the longest event sequence of an agent across the three protocols. The shared entangled resource is once again the $|\Phi^+\rangle_{12}$ Bell state. The formulae to be verified correspond to respective equations 6.7, 6.8, 6.9 and 6.10. The formula on line 21 states that at the end of the protocol the values of classical bits are always equal, next formula (line 22) asserts that Bob always knows it, however formula on line 23 claims that Bob will never know it until the last step and the last formula (line 24) states that Alice never knows that Bob knows.

The automatically generated ISPL code can be found in listing D.3 (appendix D). We again highlight several stand out features. In the definition of qubits and register in `Vars` section of *Environment* agent (line 2) the quantum states have enumeration values up to 19 but some states are missing (e.g. *state7*). This is because these states are not entangled and they are immediately separated into the substates, and so the register never lasts beyond the time step it was created. For instance when Bob performs entanglement operation `e4x5_10` on qubits 4 and 5 (line 2) the new register E_{45} is not in the entangled state and it is immediately decomposed back to the individual qubits. Similar example is on line 23, where Bob applies entanglement event on qubit 1 and 3 that are part of the entangled register E_{1234} in state *state19*. In this case the new state of the register is disentangled, however no new intermediate register appears.

In this case, the *Environment* agent has defined four actions (line 14 indicating that some measurements lead to impossible states. This is expected because there are no probabilistic transitions in this network and so all measurement outcomes are determined by the way Alice prepares her qubit.

The protocol specification in ISPL is then verified by `MCMAS` as usual. Again, all four formulae are true in the model as we expected.

6.4 Experimental Performance

In this section we evaluate the compiler's performance and discuss the experimental results obtained while verifying the three protocols as well as running some artificially set up tests.

Table 6.1 reports the results for Quantum Teleportation, QKD and Superdense Coding protocols. The verification of a protocol is divided into two phases. First, we translate its DMC specification into the corresponding ISPL code with `dmc2ispl` compiler. And then we run the protocol through `MCMAS` model checker. The number of reachable states in the second and third columns differ because the compiler analyses only the quantum state space but the model checker considers both classical and quantum state space together. The last four columns indicate memory usage and time required for translation and verification respectively.

Protocol	Reachable States		Memory (kB)		Time (s)	
	`dmc2ispl`	`MCMAS`	`dmc2ispl`	`MCMAS`	`dmc2ispl`	`MCMAS`
Teleportation	40	108	7184	6068	0.015	0.066
QKD	53	348	7240	6119	0.016	0.014
Superdense Coding	4239	2192	8132	6279	0.112	0.407

Table 6.1: Verification results for Quantum Teleportation, QKD and Superdense Coding.

As we can see from the table, all three protocols have relatively small reachable state space and both tools deal with them without demanding resource requirements. However, these requirements raise quickly with the increasing number of events and register sizes. To inspect the limits of our compiler, we have devised several tests.

Firstly, we examine how different order of entanglements in an agent's event sequence has different demands on resources. The order does not matter physically because entanglement operators are represented by diagonal matrices and matrix multiplication of these matrices is commutative (e.g. $E_{12}E_{23} = E_{23}E_{12}$). However, the order does matter with respect to performance of the compiler. There are two reasons for this. One is that compiler remembers the outcome of an operation for given states of qubits, and so if entanglement events act on qubits with the same state, nothing is computing. The other reason is that the qubits are sorted in the register as we explain at the beginning of section 5.3. So, if the sequence of entanglements is such that many calls to the `shiftQubit` subroutine is required, the translation process slows down.

Table 6.2 illustrates entanglement of n qubits in three different orders with the separation check switched off. The *Sequential* order means execution of `En(i+1, i)`, $i \in \{1, \ldots, n-1\}$, in sequence. The other two result in the same quantum state of the register. The first, *Unsorted*, is completely random and some shifting is required. This makes it the slowest method. The

Register Size n	Memory (kB)			Time (s)		
	Unsorted	Sorted	Sequential	Unsorted	Sorted	Sequential
10	6964	6940	6968	0.009	0.008	0.010
12	7176	7160	7280	0.022	0.010	0.012
14	7664	7688	8340	0.068	0.010	0.024
16	8180	8180	10144	0.404	0.028	0.067
18	11156	11152	16164	2.089	0.144	0.255
20	23500	23636	40752	10.342	0.536	1.132

Table 6.2: Different order of entanglement events.

second, *Sorted*, is such that we first entangle qubits 1 to $\frac{n}{2}$ and $\frac{n}{2}+1$ to n and then we "connect" the two register by entangling qubits $\frac{n}{2}$ and $\frac{n}{2}+1$. This method is faster and requires less memory then the sequential one. It is apparent that the user must be careful with specification of entanglement events.

Secondly, we scrutinise the subroutine for separation of quantum registers. The brute force function `isSeparable` is called after every entanglement and measurement event unless the user switches it off with `-Soff` command line option. To see how the function performs, we sequentially execute entanglement events `En(i+1, i)` for $i \in \{1, \ldots, n-1\}$. We remind the reader that entanglement is defined as application of controlled-Z operator on the pair of qubits. This means that the pair may not be entangled after the operation, for example if one of them is in $|0\rangle$ state. But there are many more possible reasons why the function is desirable, we cover this topic in section 5.3.3.2.

Table 6.3 gives the result for three different set ups. Under the header *Entangled* there are the memory and time requirements in case the register cannot be separated after entanglement. Conversely, the header *Isolated* denotes cases where no entanglement event actually entangles the pair. The last header *Off* shows result for cases where the function is disabled, in which case doesn't matter if the register is separable or not.

Register	Memory (kB)			Time (s)		
Size n	Entangled	Isolated	Off	Entangled	Isolated	Off
10	7036	6888	6964	0.03	0.008	0.009
11	7160	6896	7072	1.25	0.009	0.012
12	7308	6904	7296	5.24	0.008	0.010
13	7596	6904	7568	22.27	0.010	0.012
14	8176	6904	8336	94.95	0.010	0.018
15	8948	6908	8728	402.25	0.009	0.029
16	9720	6904	10140	1715.68	0.009	0.064
17	12028	6908	12068	7354.13	0.010	0.132
18	N/A	6904	16164	N/A	0.009	0.248
19	N/A	6916	24364	N/A	0.011	0.514
20	N/A	6912	40744	N/A	0.010	1.021

Table 6.3: Performance of `isSeparable` function in three different scenarios.

From the result we conclude that it is very important for the modeller to choose the appropriate option because this feature can be extremely demanding, or not even feasible in some cases but it can improve performance in others. Moreover, if it is disabled the modeller may not be able to verify some properties of an individual qubit even though it is in a well-defined local state because the qubit is still part of the register which has not been separated.

We now consider an experimental protocol which tests the bound of what we can verify. This quantum MAS consists of only one agent Alice with empty input sorts. Her event sequence is defined as follows

$$\texttt{cX}(n,\texttt{s}_{n-1})\ldots\texttt{cX}(n,\texttt{s}_1)\texttt{Me}(n-1,(n-1)*\texttt{pi}/n,-,-,\texttt{s}_{n-1})\ldots\texttt{Me}(1,1*\texttt{pi}/n,-,-,\texttt{s}_1)$$
$$\texttt{E}(n,n-1)\ldots\texttt{E}(2,1)$$

In this scenario, called "EMC" protocol, Alice first serially entangles her qubits, then measures all but the last qubit and then applies X correction on the qubit for every measurement outcome. The measurement angle changes for each measurement and some entanglement events may not lead to an entangled register. The protocol is tested against the following temporal epistemic specification

```
AG {Alice(s1) = 0} -> K(Alice, {Alice(s1) = 0});
```

which means that if the measurement outcome of the first qubit is 0 then Alice knows it.

We present results for various number of qubits n in tables 6.4 (standard) and 6.5 (with disabled entanglement check). We postpone discussion of the results after introducing another, similar protocol, called "EM". It differs from the previous one in the event sequence of Alice. This time, she measures all qubits and there are no corrections afterwards. Furthermore, the entanglement events are such that the registers cannot be separated at any point, and so we disable the option. Table 6.6 shows the performance and the following statement defines the specification of the event sequence

$$\texttt{Me(}n\texttt{,}n\texttt{*pi/}n\texttt{,-,-,s}_n\texttt{)...Me(1,1*pi/}n\texttt{,-,-,s}_1\texttt{)E(}n\texttt{,}n\texttt{-1)...E(2,1)}$$

The first column indicates the number of qubits n in the protocol. Second column shows the size of the corresponding generated ISPL file. The rest record the number of actual reachable states, memory usage and time required for both phases of the verification process.

Register Size n	ISPL Size (kB)	Reachable States		Memory (kB)		Time (s)	
		`dmc2ispl`	`MCMAS`	`dmc2ispl`	`MCMAS`	`dmc2ispl`	`MCMAS`
10	11.0	524296	33044	7208	6556	5.49	1.524
11	13.9	2097161	51476	7236	6540	23.57	2.043
12	13.7	8388618	387112	7240	6579	101.34	2.993
13	13.5	33554443	746792	7256	6702	524.37	3.716
14	15.9	134217740	1455660	7264	7534	1819.93	4.443
15	19.4	536870925	2843180	7308	7686	7776.72	6.892

Table 6.4: Performance of the experimental "EMC" protocol with default settings.

Register Size n	ISPL Size (MB)	Reachable States		Memory (kB)		Time (s)	
		`dmc2ispl`	`MCMAS`	`dmc2ispl`	`MCMAS`	`dmc2ispl`	`MCMAS`
10	0.5	524296	32765	9236	17229	5.52	17.86
11	1.1	2097161	51197	11536	18037	23.57	39.60
12	1.4	8388618	368613	13712	37636	99.49	123.51
13	2.7	33554443	728893	19636	54679	425.00	275.07
14	5.5	134217740	1440250	31952	149199	1794.59	3460.98
15	11.4	536870925	2838520	52180	189948	7660.86	9638.52

Table 6.5: Performance of the experimental "EMC" protocol with `-Soff` option.

Register Size n	ISPL Size (MB)	Reachable States		Memory (kB)		Time (s)	
		`dmc2ispl`	`MCMAS`	`dmc2ispl`	`MCMAS`	`dmc2ispl`	`MCMAS`
15	3.9	65549	753661	38284	61894	4.29	849.99
16	7.4	131086	1179640	68536	110348	8.90	3102.95
17	14.6	262159	9175010	127716	231162	18.34	14052.25
18	29.0	524304	N/A	245108	N/A	38.22	N/A
19	58.0	1048593	N/A	479988	N/A	80.92	N/A
20	115.8	2097170	N/A	949860	N/A	166.26	N/A

Table 6.6: Performance of the experimental "EM" protocol.

When we compare tables 6.4 and 6.5 we can see that the separation check does not have influence on the number of reachable states. This is because the global quantum state does not change, only the way we describe the state differs. Moreover, the state of the last qubit is the same in both methods because all other qubits are measured by the end of the protocol. The separated registers are smaller and require less memory, although it takes time to perform the check. The separation of non-entangled states of register is beneficial if there are such states because it leads to a smaller generated ISPL code, i.e., the evolution function of *Environment* agent contains fewer statements. `MCMAS` then processes such a file much faster even though the size of the state space is about the same in both cases.

On the other hand, when the states cannot be separated, as in the "EM" protocol, there is nothing we can do about it. Because all measurements have unique measurement angle, each measurement results to a unique global state. In case of $n = 20$, even though the measured qubit collapses to one of the 40 base states, the rest of the register evolves via entanglement with this qubit to one of the 2^{20} unique states of the reachable state space. It is obvious, that the number of unique quantum local states is more important than the number of reachable global quantum states for verification in `MCMAS` modal checker whilst it is the other way round for `dmc2ispl` compiler.

On top of that, operators on large quantum registers have huge dimensions, it takes almost 1GB of memory to measure 20-qubit entangled system even using sparse matrices. It is easier to create these registers because entanglements are represented by real diagonal matrices but measurements have associated complex matrices with greater number of non-zero elements than that. However, there are many ways how to improve performance because the compilation process is not very optimised at the moment. We discuss these and other improvements in the next section.

A comparison to existing approaches is rather limited because, as far as we know, `dmc2ispl` is the only fully automatic method for verification of epistemic properties in the context of distributed quantum systems. However, the following two toolkits appear in the literature.

The probabilistic symbolic model checker PRISM [26] has been used in [23] for automated analysis of quantum information protocols. The technique is confined to protocols that involve only restricted set of Clifford group operators (Hadamard gate, phase gate and controlled-NOT gate) and operators derivable from these (e.g. Pauli gates X, Y and Z). The method works on similar principle as ours. First, it identifies the finite set of states, which arise by applying the operations to input states, then enumerate the states and verify the model in PRISM. However, determining which states belong to this set is done manually, and so the verification is not fully automatic. The authors illustrate the technique by modelling Superdense Coding, Quantum Teleportation, and Quantum Error Correction and verifying their basic correctness properties.

The authors have later built the Quantum Model Checker (QMC) [24, 33], with its own modelling language QMCLANG. As far as we know, this is the only other dedicated verification tool of any kind. The model checker supports specifications in quantum computational temporal logic, which is an extension of the classical CTL. The quantum circuits are again restricted to the set of Clifford group operators. This loss of expressive power is balanced out by efficient computation of quantum state space using stabilizer formalism. However this gain in simulation component is lost by inefficient verification of protocol properties. A direct efficiency comparison to this toolkit is problematic due to different input languages and different support of specifications.

Chapter 7

Conclusion

The goal of this project has been the development of techniques and tools for the automatic verification of quantum multi-agent systems via model checking.

At first, a translation map from DMC to ISPL has been designed, so that `MCMAS` model checker can verify protocols specified in this framework. Then the source-to-source compiler `dmc2ispl`, which realises this translation, has been developed in `C++`. The DMC formalism has been suitably adapted to be used as input language for the compiler. This standardised modular language closely follows the original DMC but also reflects the structure of ISPL.

The toolkit computes the reachable quantum state space of a system by simulating the quantum circuits defined via commands in agents' event sequences and enumerates all encountered local states of qubits and quantum registers. Because of universality of underlying MC (see section 2.3.1.4) the expressive power in terms of available quantum operations is limitless. In other words, any quantum circuit can be simulated and verified as long as enough resources are available. The corresponding ISPL program is generated from this information together with the classical parts of the system specification in the DMC input file.

Three quantum protocols, namely Quantum Teleportation [6], Quantum Key Distribution [19] and Superdense Coding [7], have been translated and their temporal epistemic properties successfully checked in `MCMAS`. Verification of these protocols is very fast and takes less than 1 second on a standard PC. Experimental protocols "EM" and "EMC" have shown that quantum protocols of up to 20 entangled qubits and with over 5×10^8 reachable quantum states can be realistically translated, however the number of reachable global states (conjunction of quantum state of the system and classical local states of all the agents) needs to be around 10^7 for verification phase.

7.1 Further Work

There is a lot of scope for future development of the `dmc2ispl` compiler. The possible improvements can be divided into two categories, features and performance.

Very important extension of the current version from the first category would be generalisation to the mixed states and general pure states. As we discuss in section 4.1.2.3, these are not supported at the moment. The former is necessary for verification of a real quantum hardware which is subject to decoherence, and the latter would allow for symbolic manipulation of general input qubits.

A nice extension of `MCMAS` would be introduction of probabilities, so that we can model probabilistic transitions more naturally as well as reason about likelihood of counterexamples and witnesses.

It is a relatively minor point but better error recovery in syntactical analysis of the input file would enhanced the user experience. The current version of the compiler has rather unforgiving parser regarding syntax errors.

There are many possible improvements in the performance category. A huge boost can be achieved by computing the reachable quantum state space with taking agents' local states into consideration. Currently, any conditional event (corrections, patterns and dependent measurements) have unnecessary branching factor. But if we compute the action only for the actual value of the variable, on which the event depends, only measurements will have a branching factor of 2. This would lead to much smaller state space and more succinct generated ISPL code.

Other possible optimisation is implementation of the signal shifting event from the extended MC. Then, for example in the "EMC" protocol from section 6.4, we would be able to merge all corrections X from the Alice's event sequence into one and the value of the merged signal, on which this single correction depends, equals to the sum of all current signals modulo 2. We would effectively cut off the large number of global quantum states at the bottom of the tree.

We can also extend the `Definition` module to patterns which operates on arbitrary number of qubits, not just one qubit, as well as allow for non-deterministic patterns. This would enable agents to do complex operations on qubits in just one time step which would again lead to considerably smaller state spaces. Although the trade-off here is the inability to verify properties of the system inside the pattern.

Apart from pruning the quantum state space, there are ways to speed up the generation of the reachable state space itself. General quantum circuits cannot be simulated on classical computers since they require an exponentially large amount of memory and time. After all, if they could be simulated then the Shor's algorithm [37] for integer factorization would render current public-key cryptography obsolete. However there are techniques and sophisticated software that allow for polynomial or even linear simulation of certain groups of quantum circuits. Currently, we compute the evolution of the global quantum state using `Octave` and algorithms we describe in section 5.3. By employing these tools, we could significantly decrease computational time required for certain computations in comparison with our rather generic simulation methods.

Especially `QuIDDPro` [41] seems to be very promising quantum circuit simulator. According to the benchmarks, the toolkit easily succeeds where we suffer from the exponential blow-up in size of the matrices and it handles up to 40 qubits with reasonable demand for resources. There are techniques in the manual [40] which can efficiently compute the equivalence of states up to global phase, an important feature for enumeration of the states. Also, a `C++` library and API are available to include the tool directly in applications.

Ultimately, the compiler and the model checker could be merged to form a dedicated tool for verification of quantum protocols. At the moment, the quantum state space is needlessly traversed twice. Changes in the compiler could be made such that instead of enumerating the states and translating the DMC code it would encode the global states into OBDDs directly and then the algorithms developed for `MCMAS` would take over. Other additional improvements, such as parallel computation, maybe also implemented.

Bibliography

[1] Alexandru Baltag and Sonja Smets. The logic of quantum programs. In *QPL2004: Proceedings of the 2nd International Workshop on Quantum Programming Languages*, Turku Center for Computer Science, 2004.

[2] Alexandru Baltag and Sonja Smets. Complete axiomatizations for quantum actions. *International Journal of Theoretical Physics*, 44(12), December 2005.

[3] Alexandru Baltag and Sonja Smets. Lqp: the dynamic logic of quantum information. *Mathematical Structures in Computer Science*, 16(3), June 2006.

[4] Alexandru Baltag and Sonja Smets. A dynamic-logical perspective on quantum behavior. *Studia Logica*, 89(2), July 2008.

[5] Charles H. Bennett. Quantum cryptography using any two nonorthogonal states. *Physical Review Letters*, 68(21), May 1992.

[6] 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), March 1993.

[7] Charles H. Bennett and Stephen J. Wiesner. Communication via one- and two-particle operators on einstein-podolsky-rosen states. *Physical Review Letters*, 69(20), November 1992.

[8] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, New York, NY, USA, 1999. ACM.

[9] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35, 1986.

[10] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), September 1994.

[11] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2004.

[12] Vincent Danos and Ellie D'Hondt. Classical knowledge for quantum cryptographic reasoning. *Electronic Notes in Theoretical Computer Science*, 192(3), November 2008.

[13] Vincent Danos, Ellie D'Hondt, Elham Kashefi, and Prakas Panangaden. Distributed measurement-based quantum computation. *Electronic Notes in Theoretical Computer Science*, 170, March 2007.

[14] Vincent Danos, Elham Kashefi, and Prakash Panangaden. Parsimonious and robust realizations of unitary maps in the one-way model. *Physical Review Letters*, 72(6), December 2005.

[15] Vincent Danos, Elham Kashefi, and Prakash Panangaden. The measurement calculus. *Journal of the ACM*, 54(2), April 2007.

[16] Ellie D'Hondt and Prakash Panangaden. Reasoning about quantum knowledge. *Lecture Notes in Computer Science*, 3821, December 2005.

[17] Ellie D'Hondt and Mehrnoosh Sadrzadeh. Classical knowledge for quantum security. *ArXiv e-prints: 0808.3574*, August 2008.

[18] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[19] Artur K. Ekert. Quantum cryptography based on bell's theorem. *Physical Review Letters*, 67(6), August 1991.

[20] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, USA, 2003.

[21] Free Software Foundation. Gnu bison. `http://www.gnu.org/software/bison/`, 2010.

[22] Free Software Foundation. Gnu octave. `http://www.gnu.org/software/octave/`, 2010.

[23] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. Probabilistic model–checking of quantum protocols. 2005.

[24] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. Qmc: A model checker for quantum systems. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008.

[25] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems (2nd edition)*. Cambridge University Press, Cambridge, England, UK, 2004.

[26] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism. `http://www.prismmodelchecker.org/`, 2010.

[27] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas: a model checker for the verification of multi-agent systems. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*. Springer, 2009.

[28] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas: a model checker for multi-agents systems. `http://www-lai.doc.ic.ac.uk/mcmas/`, 2010.

[29] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. *MCMAS v1.0.0: User Manual*. Imperial College London, 2010.

[30] Alessio Lomuscio and Franco Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, New York, NY, USA, 2006. ACM.

[31] Graham Markall. Epistemic verification of quantum protocols. ISO, Department of Computing, Imperial College London, April 2009.

[32] T. Rudolph E. Schenck H. Weinfurter V. Vedral M. Aspelmeyer P. Walther, K. J. Resch and A. Zeilinger. Experimental one-way quantum computing. *Nature*, 434, March 2005.

[33] Nikolaos Papanikolaou. Quantum model checker (qmc). `http://www.dcs.warwick.ac.uk/~nikos/quantummodelcheckerqmc/index.html`, 2010.

[34] The Flex Project. flex: The fast lexical analyzer. `http://flex.sourceforge.net/`, 2010.

[35] Franco Raimondi. *Model Checking Multi-Agent Systems.* PhD thesis, University College London, 2006.

[36] Robert Raussendorf and Hans J. Briegel. A one-way quantum computer. *Physical Review Letters*, 81(22), May 2001.

[37] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *SFCS '94: Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, 1994. IEEE Computer Society.

[38] Fabio Somenzi. Cudd: Cu decision diagram package release 2.4.2. `http://vlsi.colorado.edu/~fabio/CUDD/`, 2010.

[39] Ron van der Meyden and Manas Patra. Knowledge in quantum systems. In *TARK '03: Proceedings of the 9th conference on Theoretical aspects of rationality and knowledge*, New York, NY, USA, 2003. ACM.

[40] George F. Viamontes, Igor L. Markov, and John P. Hayes. *QuiDDpro User's Guide Version 3.1.* University of Michigan, 2007.

[41] George F. Viamontes, Igor L. Markov, and John P. Hayes. Quiddpro: High-performance quantum circuit simulation. `http://vlsicad.eecs.umich.edu/Quantum/qp/index.html`, 2010.

[42] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum Computing for Computer Scientists.* Cambridge University Press, New York, NY, USA, 2008.

Appendices

Appendix A

Background on Quantum Computing

Quantum computing [42] is at the intersection of computer science, mathematics and physics. This chapter gives a brief introduction to the basic concepts of this fascinating field.

- **Qubit** is a unit of information describing a two-dimensional quantum system. Whereas a bit can be either in state $|0\rangle$ or in state $|1\rangle$, superposition allows a qubit to be in both states simultaneously. Qubits are described by state vectors. These are normalised column vectors in a two-dimensional complex vector space $\mathbb{C}^2$. The states of the bits have the following representation

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

  Because these are two different orthogonal representations, they form a canonical basis of $\mathbb{C}^2$. Thus, any vector from this space can be written as a linear combination of the two vectors. This gives the following representation of an arbitrary state vector $|\psi\rangle$

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \alpha\,|0\rangle + \beta\,|1\rangle\,,$$

  where α and β are normalised complex numbers with $|\alpha|^2 + |\beta|^2 = 1$. $|\alpha|^2$ is to be interpreted as the probability that after measuring the qubit, it will be found in state $|0\rangle$. $|\beta|^2$ is to be interpreted as the probability that after measuring the qubit, it will be found in state $|1\rangle$. Whenever a qubit is measured, it automatically collapses to a bit.

- **Quantum register** is a quantum system composed of two or more qubits. A state space of this compound system is the tensor product of the state space of its constituents. A state of a two-qubit system is an element of $\mathbb{C}^2 \otimes \mathbb{C}^2$, denoted as $(\mathbb{C}^2)^{\otimes 2}$ and isomorphic to $\mathbb{C}^4$. A standard computational basis of the system are vectors $|00\rangle = |0\rangle \otimes |0\rangle$, $|01\rangle = |0\rangle \otimes |1\rangle$, $|10\rangle = |1\rangle \otimes |0\rangle$, $|11\rangle = |1\rangle \otimes |1\rangle$ and a general state can be written as

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \alpha \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \beta \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \gamma \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \delta \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle\,,$$

  where α, β, γ and δ are again probability amplitudes, squared norms of which represent the probabilities that after measuring the system, it will be found in one of the four base states.

- **Separable state** of a quantum register is a state that can be broken into tensor product of states from the constituent subsystems. For example, elements of the standard two-qubit system basis could be rewritten as the tensor product of $|0\rangle$ or $|1\rangle$. But not every generic state can be rewritten as such. **Entangled state** is unbreakable. For example, if the system is in the state

$$|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{\sqrt{2}}|0\rangle \otimes |0\rangle + \frac{1}{\sqrt{2}}|1\rangle \otimes |1\rangle , \tag{A.1}$$

  it can be represented only as a linear combination of tensor products and neither of the two constituent qubits is in a well-defined local state. However, the individual states of the two qubits are closely related to one another. If the first qubit in equation A.1 is measured, it has 50-50 chance of being found in the position $|0\rangle$ or in $|1\rangle$. Suppose that it is found in state $|0\rangle$ then there is no chance that the other qubit will be found in position $|1\rangle$, it will be forced to state $|0\rangle$. This behaviour is independent of their actual distance in space, a measurement's outcome for one qubit will always determine the measurement's outcome for the other one.

- **Quantum gate** is an operator that acts on qubits. Such operators have to follow the dynamics of quantum operations and are represented by unitary matrices. A matrix U is unitary if its inverse U^{-1} is equal to its conjugate transpose $U^{\dagger}$, more formally $UU^{\dagger} = U^{\dagger}U = I$. Applying a gate to a qubit can be represented by matrix multiplication. For example, Hadamard gate transforms qubit in state $|0\rangle$ to superposition of states $|0\rangle$ and $|1\rangle$

$$|\psi\rangle = H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle .$$

  Gates, such as Hadamard gate and Pauli X, Y and Z gates, can operate on a single qubit

$$X : \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \mapsto \begin{bmatrix} \beta \\ \alpha \end{bmatrix}, Y : \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \mapsto \begin{bmatrix} -\beta i \\ \alpha i \end{bmatrix} \text{ and } Z : \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \mapsto \begin{bmatrix} \alpha \\ -\beta \end{bmatrix},$$

  or, such as controlled-NOT gate and Toffoli gate, on quantum registers

$${}^{C}NOT : |x, y\rangle \mapsto |x, x \oplus y\rangle , Toffoli : |x, y, z\rangle \mapsto |x, y, z \oplus (x \wedge y)\rangle ,$$

  where x, y, z are individual qubits and $\oplus$ is the binary exclusive or. Quantum operators can be combined to form more complex ones. If matrix A corresponds to performing an operation and matrix B corresponds to performing another operation, then the matrix $B*A$ corresponds to performing the operations sequentially and the matrix $A \otimes B$ concurrently. Such sequential and parallel combinations of quantum gates are called quantum circuits.

- **Measurement** operation is a special type of gate. This is not unitary or, in general, even reversible. It is performed when we want to measure a qubit and the qubit is destroyed by the operation, i.e. it collapses to a bit. Measurements in basis are represented by orthogonal projections. A matrix P is an orthogonal projection if $P = P^{\dagger} = P^2$.

Appendix B

Background on Interpreted Systems

Interpreted systems [20] have been put forward as basic models of multi-agent systems and for reasoning about knowledge in such systems. In this chapter we describe the idea and the underlying temporal epistemic logic.

- **Global state** of a system with n agents is defined as $(n+1)$-tuple of the form $(l_e, l_1, \ldots, l_n)$, where l_e is the state of the environment and l_i is the local state of agent i. A global state describes the system at a given point in time. The set of global states $\mathcal{G}$ is a subset of the Cartesian product of its constituents $\mathcal{G} \subseteq L_e \times L_1 \times \cdots \times L_n$, where L_e is a set of possible states for the environment and L_i is a set of possible local states for agent i. A run r over $\mathcal{G}$ is a function from the time domain to $\mathcal{G}$. The global state at the point (r, m), where r is a run and m is time, is described by $r(m) = (l_e, l_1, \ldots, l_n)$. This gives a complete description of how the system evolves over time. A system $\mathcal{R}$ over $\mathcal{G}$ is a set of runs over $\mathcal{G}$ and (r, m) is a point in system $\mathcal{R}$ if $r \in \mathcal{R}$.

- **Interpreted system** $\mathcal{I}$ consists of pair $(\mathcal{R}, \pi)$, where $\mathcal{R}$ is a system over a set $\mathcal{G}$ of global states and π is an interpretation for the propositions in Φ over $\mathcal{G}$, which assigns truth values to the primitive propositions at the global states. Thus, for every $p \in \Phi$ and state $s \in \mathcal{G}$, $\pi(s)(p) \in \{\textit{true, false}\}$. Also, π induces an interpretation over the points of $\mathcal{R}$, where $\pi(r, m)$ is taken as $\pi(r(m))$. An interpreted system $\mathcal{I}$ can be associated with a Kripke structure $M = (S, \pi, \sim_1, \ldots, \sim_n)$, where S consists of the points in $\mathcal{I}$ and $\sim_1, \ldots, \sim_n$ are some binary relations on S. Thus, the $\models$ relation can be inductively defined

$$\begin{array}{lll} (\mathcal{I}, r, m) \models p \text{ (for } p \in \Phi) & \text{iff} & \pi(r, m)(p) = \textit{true}, \\ (\mathcal{I}, r, m) \models \neg\varphi & \text{iff} & (\mathcal{I}, r, m) \not\models \varphi, \\ (\mathcal{I}, r, m) \models \varphi \wedge \psi & \text{iff} & (\mathcal{I}, r, m) \models \varphi \text{ and } (\mathcal{I}, r, m) \models \psi. \end{array}$$

- **Epistemic logic** is a logic for reasoning about knowledge. In interpreted systems the epistemic relation $\sim_i$ in M has the following meaning: Two points (r, m) and (r', m') in the interpreted system $\mathcal{I}$ are *indistinguishable* to agent i, written as $(r, m) \sim_i (r', m')$, if i has the same local state in both $r(m)$ and $r'(m')$. Intuitively, agent i considers a state $r'(m')$ possible in a state $r(m)$ if $r(m)$ and $r'(m')$ are indistinguishable to i, i.e. $l_i = l'_i$. It is obvious that $\sim_i$ is an equivalence relation on points since it is reflexive, symmetric and transitive. The language is then augmented by modal operators $K_1, \ldots, K_n$. The statement K_i is read "agent i *knows* φ" and has the semantics that agent i knows φ in a state $r(m)$ of the interpreted system $\mathcal{I}$ if and only if φ is true at all states that i considers possible in $r(m)$, more formally

$$(\mathcal{I}, r, m) \models K_i\varphi \quad \text{iff} \quad (\mathcal{I}, r', m') \models \varphi \text{ for all } (r', m') \text{ such that } (r, m) \sim_i (r', m').$$

The language may be extended with notions of *common knowledge* and *distributed knowledge*. The corresponding modal operators E_G ("everyone in the group G knows"), C_G ("it is common knowledge among the agents in G") D_G ("it is distributed knowledge among the agents in G") are defined

$$\begin{array}{lll}(\mathcal{I},r,m) \vDash E_G\varphi & \text{iff} & (\mathcal{I},r,m) \vDash K_i\varphi \text{ for all } i \in G,\\ (\mathcal{I},r,m) \vDash C_G\varphi & \text{iff} & (\mathcal{I},r,m) \vDash E_G^k\varphi \text{ for all } k \geq 1,\\ (\mathcal{I},r,m) \vDash D_G\varphi & \text{iff} & (\mathcal{I},r',m') \vDash \varphi \text{ for all } (r',m') \text{ such that } (r',m') \in \cap_{i\in G} \sim_i,\end{array}$$

where $E_G^k\varphi$ abbreviates $E_G E_G^{k-1}\varphi$ (in particular, $E_G^1\varphi$ means just $E_G\varphi$).

- **Temporal logic** is a logical formalism for time. In the following definitions, time is assumed to be linear, discrete and infinite but, in general, it can be continuous and its flow branching or circular. Possible temporal operators used for reasoning about events that happen along a run are defined as follows: $\Box$ ("*always*"), $\Diamond$ ("*eventually*"), $\bigcirc$ ("*next time*") and U ("*until*"). Formal semantics of these modalities are

$$\begin{array}{lll}(\mathcal{I},r,m) \vDash \Box\varphi & \text{iff} & (\mathcal{I},r,m') \vDash \varphi \text{ for all } m' \geq m,\\ (\mathcal{I},r,m) \vDash \Diamond\varphi & \text{iff} & (\mathcal{I},r,m') \vDash \varphi \text{ for some } m' \geq m,\\ (\mathcal{I},r,m) \vDash \bigcirc\varphi & \text{iff} & (\mathcal{I},r,m+1) \vDash \varphi,\\ (\mathcal{I},r,m) \vDash \varphi U\psi & \text{iff} & (\mathcal{I},r,m') \vDash \psi \text{ for some } m' \geq m \text{ and}\\ & & \quad (\mathcal{I},r,m'') \vDash \varphi \text{ for all } m'' \text{ with } m \leq m'' < m'.\end{array}$$

By combining knowledge and temporal operators, assertions about evolution of knowledge in the system can be made. For example, $\Diamond K_i\varphi$ means "agent i will eventually know φ".

Appendix C

Complete Grammar

DMC Programming Language

dmc ::= definitions tail | tail
tail ::= agents middle formulae | agents formulae
middle ::= qubits groups | qubits | groups

Definitions Module

definitions ::= definitions definition | definition
definition ::= # defname : { **INT** } , { **INT** } , { plist } ;
defname ::= **ID**
plist ::= plist , pattern | pattern
pattern ::= **En** (**INT** , **INT**) | **cX** (**INT** , **ID**) | **cZ** (**INT** , **ID**) |
Me (**INT** , expr , signal , signal , **ID**)

Agents Module

agents ::= agents agent | agent
agent ::= name : ownqubits , knownqubits , cinputs , events ;
name ::= **ID**
ownqubits ::= { qlist } | { }
qlist ::= qlist , **INT** | **INT**
knownqubits ::= { nqlist } | { }
nqlist ::= nqlist , **INT** | **INT**
cinputs ::= { clist } | { }
clist ::= clist , **ID** : value | **ID** : value
events ::= { elist } | { }
elist ::= elist , event | event
event ::= **c!** (**ID** , **ID**) | **c?** (**ID** , **ID**) | **qc!** (**ID** , **INT**) | **qc?** (**ID** , **INT**) |
En (**INT** , **INT**) | **cX** (**INT** , **ID**) | **cZ** (**INT** , **ID**) |
Me (**INT** , expr , signal , signal , **ID**) | **ID** (**INT** , **ID**)

Qubits Module

qubits ::= qubits qubit | qubit
qubit ::= purequbits : { qstate } ; | purequbits : ? ;
purequbits ::= purequbits , **INT** | **INT**
qstate ::= qstate , complex | complex

Groups Module

groups ::= groups group | group
group ::= **ID** = { members } ;
members ::= members , **ID** | **ID**

Formulae Module

formulae ::= formulae formula ; | formula ;
formula ::= (formula) | formula **and** formula | formula **or** formula | ! formula |
formula -> formula | **AG** formula | **EG** formula | **AX** formula |
EX formula | **AF** formula | **EF** formula | **A** untilprefix | **E** untilprefix |
K epistemicprefix | **GK** gepistemicprefix | **GCK** gepistemicprefix |
DK dgepistemicprefix | atlprefix **F** formula | atlprefix **X** formula |
atlprefix **G** formula | **O** epistemicprefix | atlprefix untilprefix | { atom }
untilprefix ::= (formula **UNTIL** formula)
epistemicprefix ::= (**ID** , formula)
gepistemicprefix ::= (**ID** , formula)
dgepistemicprefix ::= (**ID** , formula)
atlprefix ::= < **ID** >
atom ::= **ID** (**ID**) = **INT** | **ID** (**ID**) = **ID** (**ID**) | **ID has INT** |
ID lop **INT** | **INT** = **INT** | **INT** = **init** (**INT**) | **INT** |
INT = { qstate }

Auxiliary

value ::= **INT** | ?
signal ::= **ID** | -
complex ::= (expr , expr)
expr ::= **INT** | **DOUBLE** | **ID** | **sin** (expr) | **cos** (expr) | **exp** (expr) |
ln (expr) | **sqrt** (expr) | expr + expr | expr - expr | expr * expr |
expr / expr | - expr | expr ^ expr | (expr)
lop ::= = | < | >

ID ::= [a-zA-Z][a-zA-Z_0-9]* **INT** ::= [0-9]+ **DOUBLE** ::= [\-]?[0-9]+[\.][0-9]+

Appendix D

Compiled ISPL Codes

Quantum Teleportation

```
Agent Environment
  Vars:
    q1i : {state2}; q2i : {undefined}; q3i : {undefined};
    q1 : {undefined, state2, state4, state6};
    q2 : {undefined, state4, state6};
    q3 : {undefined, state10, state2, state8, state9};
    e2_3 : {undefined, state1, state5, state7};
    e1_2_3 : {undefined, state3};
    gc : 1..8;
  end Vars
  Actions = {};
  Protocol:
  end Protocol
  Evolution:
    gc = gc + 1 and q1 = state4 and e2_3 = state5 and e1_2_3 = undefined if
        e1_2_3 = state3 and Alice.Action = m1m_2;
    gc = gc + 1 and q1 = state6 and e2_3 = state7 and e1_2_3 = undefined if
        e1_2_3 = state3 and Alice.Action = m1p_2;
    gc = gc + 1 and q1 = undefined and e2_3 = undefined and e1_2_3 = state3
        if q1 = state2 and e2_3 = state1 and Alice.Action = e1x2_1;
    gc = gc + 1 and q2 = state4 and q3 = state8 and e2_3 = undefined if e2_3
        = state7 and Alice.Action = m2m_3;
    gc = gc + 1 and q2 = state4 and q3 = state9 and e2_3 = undefined if e2_3
        = state5 and Alice.Action = m2m_3;
    gc = gc + 1 and q2 = state6 and q3 = state10 and e2_3 = undefined if e2_3
         = state5 and Alice.Action = m2p_3;
    gc = gc + 1 and q2 = state6 and q3 = state2 and e2_3 = undefined if e2_3
        = state7 and Alice.Action = m2p_3;
    gc = gc + 1 and q3 = state10 if q3 = state2 and Bob.Action = z3_4;
    gc = gc + 1 and q3 = state10 if q3 = state9 and Bob.Action = x3_3;
    gc = gc + 1 and q3 = state2 if q3 = state10 and Bob.Action = z3_4;
    gc = gc + 1 and q3 = state2 if q3 = state8 and Bob.Action = x3_3;
    gc = gc + 1 and q3 = state8 if q3 = state2 and Bob.Action = x3_3;
    gc = gc + 1 and q3 = state8 if q3 = state9 and Bob.Action = z3_4;
    gc = gc + 1 and q3 = state9 if q3 = state10 and Bob.Action = x3_3;
    gc = gc + 1 and q3 = state9 if q3 = state8 and Bob.Action = z3_4;
    gc = gc + 1 if Alice.Action = snd_Bob_s10 or Alice.Action = snd_Bob_s11
        or Alice.Action = snd_Bob_s20 or Alice.Action = snd_Bob_s21 or Bob.
        Action = skip;
  end Evolution
end Agent
```

```
Agent Alice
  Vars:
    s2 : {undefined, zero, one};
    s1 : {undefined, zero, one};
    q1 : 0..2;
    q2 : 0..2;
    q3 : 0..2;
    pc : 1..6;
  end Vars
  Actions = {e1x2_1, m1p_2, m1m_2, m2p_3, m2m_3, snd_Bob_s10, snd_Bob_s11,
      snd_Bob_s20, snd_Bob_s21, none};
  Protocol:
    pc = 1 : {e1x2_1};
    pc = 2 : {m1p_2, m1m_2};
    pc = 3 : {m2p_3, m2m_3};
    pc = 4 and s1 = zero : {snd_Bob_s10};
    pc = 4 and s1 = one : {snd_Bob_s11};
    pc = 5 and s2 = zero : {snd_Bob_s20};
    pc = 5 and s2 = one : {snd_Bob_s21};
    Other: {none};
  end Protocol
  Evolution:
    q1 = 1 and q2 = 1 and pc = pc + 1 if Action = e1x2_1;
    s1 = zero and q1 = 2 and pc = pc + 1 if Action = m1p_2;
    s1 = one and q1 = 2 and pc = pc + 1 if Action = m1m_2;
    s2 = zero and q2 = 2 and pc = pc + 1 if Action = m2p_3;
    s2 = one and q2 = 2 and pc = pc + 1 if Action = m2m_3;
    pc = pc + 1 if (Action = snd_Bob_s10 or Action = snd_Bob_s11) and Bob.
        Action = rcv_Alice_s1;
    pc = pc + 1 if (Action = snd_Bob_s20 or Action = snd_Bob_s21) and Bob.
        Action = rcv_Alice_s2;
  end Evolution
end Agent

Agent Bob
  Vars:
    s1 : {undefined, zero, one};
    s2 : {undefined, zero, one};
    q1 : 0..2;
    q2 : 0..2;
    q3 : 0..2;
    pc : 1..5;
  end Vars
  Actions = {rcv_Alice_s1, rcv_Alice_s2, x3_3, skip, z3_4, none};
  Protocol:
    pc = 1 : {rcv_Alice_s1};
    pc = 2 : {rcv_Alice_s2};
    pc = 3 and s2 = zero : {skip};
    pc = 3 and s2 = one : {x3_3};
    pc = 4 and s1 = zero : {skip};
    pc = 4 and s1 = one : {z3_4};
    Other: {none};
  end Protocol
  Evolution:
    s1 = zero and pc = pc + 1 if Action = rcv_Alice_s1 and Alice.Action =
        snd_Bob_s10;
    s1 = one and pc = pc + 1 if Action = rcv_Alice_s1 and Alice.Action =
        snd_Bob_s11;
```

```
      s2 = zero and pc = pc + 1 if Action = rcv_Alice_s2 and Alice.Action =
          snd_Bob_s20;
      s2 = one and pc = pc + 1 if Action = rcv_Alice_s2 and Alice.Action =
          snd_Bob_s21;
      pc = pc + 1 if Action = x3_3 or Action = skip or Action = z3_4;
  end Evolution
end Agent

Evaluation
  f0 if Bob.pc > 4;
  f1 if Environment.q3 = Environment.q1i and !Environment.q3 = undefined;
  f2 if Alice.q3 = 2;
  f3 if Bob.q3 = 2;
end Evaluation

InitStates
  Environment.q1 = state2 and Environment.q2 = undefined and Environment.q3 =
      undefined and
  Environment.e2_3 = state1 and Environment.e1_2_3 = undefined and
      Environment.gc = 1 and
  Alice.s2 = undefined and Alice.s1 = undefined and Alice.q1 = 1 and Alice.q2
      = 1 and Alice.q3 = 0 and Alice.pc = 1 and
  Bob.s1 = undefined and Bob.s2 = undefined and Bob.q1 = 0 and Bob.q2 = 0 and
      Bob.q3 = 1 and Bob.pc = 1;
end InitStates

Formulae
  AG ((f0) -> (f1));
  AG ((f0) -> K (Bob, (f1)));
  EF ((f0) -> K (Alice, (f1)));
  !AG ((f0) -> K (Alice, (f1)));
  !EF K (Alice, (f2)) and !EF K (Bob, (f3));
end Formulae
```

Listing D.1: teleport.ispl

Quantum Key Distribution

```
Agent Environment
  Vars:
    q1i : {undefined}; q2i : {undefined};
    q1 : {undefined, state3, state4};
    q2 : {undefined, state3, state4, state5, state6};
    e1_2 : {undefined, state1, state2};
    gc : 1..5;
  end Vars
  Actions = {env1, none};
  Protocol:
    gc = 2 and e1_2 = state1 : {env1};
    Other: {none};
  end Protocol
  Evolution:
    gc = gc + 1 and e1_2 = state2 if e1_2 = state1 and Alice.Action = Ha1_1
        and Bob.Action = skip;
    gc = gc + 1 and e1_2 = state2 if e1_2 = state1 and Alice.Action = skip
        and Bob.Action = Ha2_1;
    gc = gc + 1 and q1 = state3 and q2 = state3 and e1_2 = undefined if e1_2
        = state1 and Alice.Action = m1m_2 and Bob.Action = m2m_2;
    gc = gc + 1 and q1 = state3 and q2 = state3 and e1_2 = undefined if e1_2
        = state1 and Alice.Action = m1m_2 and Bob.Action = m2p_2;
```

```
      gc = gc + 1 and q1 = state3 and q2 = state3 and e1_2 = undefined if e1_2
          = state2 and Alice.Action = m1m_2 and Bob.Action = m2m_2;
      gc = gc + 1 and q1 = state3 and q2 = state4 and e1_2 = undefined if e1_2
          = state2 and Alice.Action = m1m_2 and Bob.Action = m2p_2;
      gc = gc + 1 and q1 = state4 and q2 = state3 and e1_2 = undefined if e1_2
          = state2 and Alice.Action = m1p_2 and Bob.Action = m2m_2;
      gc = gc + 1 and q1 = state4 and q2 = state4 and e1_2 = undefined if e1_2
          = state1 and Alice.Action = m1p_2 and Bob.Action = m2m_2;
      gc = gc + 1 and q1 = state4 and q2 = state4 and e1_2 = undefined if e1_2
          = state1 and Alice.Action = m1p_2 and Bob.Action = m2p_2;
      gc = gc + 1 and q1 = state4 and q2 = state4 and e1_2 = undefined if e1_2
          = state2 and Alice.Action = m1p_2 and Bob.Action = m2p_2;
      gc = gc + 1 if (e1_2 = state1 and Alice.Action = Ha1_1 and Bob.Action =
          Ha2_1) or Alice.Action = skip and Bob.Action = skip or Alice.Action =
           snd_Bob_a0 or Alice.Action = snd_Bob_a1 or Bob.Action = snd_Alice_b0
           or Bob.Action = snd_Alice_b1;
  end Evolution
end Agent

Agent Alice
  Vars:
    a : {zero, one};
    b : {undefined, zero, one};
    s1 : {undefined, zero, one};
    q1 : 0..2;
    q2 : 0..2;
    pc : 1..5;
  end Vars
  Actions = {Ha1_1, skip, m1p_2, m1m_2, rcv_Bob_b, snd_Bob_a0, snd_Bob_a1,
      none};
  Protocol:
    pc = 1 and a = zero : {skip};
    pc = 1 and a = one : {Ha1_1};
    pc = 2 : {m1p_2, m1m_2};
    pc = 3 : {rcv_Bob_b};
    pc = 4 and a = zero : {snd_Bob_a0};
    pc = 4 and a = one : {snd_Bob_a1};
    Other: {none};
  end Protocol
  Evolution:
    s1 = zero and q1 = 2 and pc = pc + 1 if Action = m1p_2;
    s1 = one and q1 = 2 and pc = pc + 1 if Action = m1m_2;
    b = zero and pc = pc + 1 if Action = rcv_Bob_b and Bob.Action =
        snd_Alice_b0;
    b = one and pc = pc + 1 if Action = rcv_Bob_b and Bob.Action =
        snd_Alice_b1;
    pc = pc + 1 if (Action = snd_Bob_a0 or Action = snd_Bob_a1) and Bob.
        Action = rcv_Alice_a;
    pc = pc + 1 if Action = Ha1_1 or Action = skip;
  end Evolution
end Agent

Agent Bob
  Vars:
    b : {zero, one};
    a : {undefined, zero, one};
    s2 : {undefined, zero, one};
    q1 : 0..2;
    q2 : 0..2;
    pc : 1..5;
```

```
    end Vars
    Actions = {Ha2_1, skip, m2p_2, m2m_2, snd_Alice_b0, snd_Alice_b1,
        rcv_Alice_a, none};
    Protocol:
      pc = 1 and b = zero : {skip};
      pc = 1 and b = one : {Ha2_1};
      pc = 2 : {m2p_2, m2m_2};
      pc = 3 and b = zero : {snd_Alice_b0};
      pc = 3 and b = one : {snd_Alice_b1};
      pc = 4 : {rcv_Alice_a};
      Other: {none};
    end Protocol
    Evolution:
      s2 = zero and q2 = 2 and pc = pc + 1 if (Action = m2p_2 and !((
          Environment.Action = env1 and Alice.Action = m1m_2))) or (Action =
          m2m_2 and ((Environment.Action = env1 and Alice.Action = m1p_2)));
      s2 = one and q2 = 2 and pc = pc + 1 if (Action = m2m_2 and !((Environment
          .Action = env1 and Alice.Action = m1p_2))) or (Action = m2p_2 and ((
          Environment.Action = env1 and Alice.Action = m1m_2)));
      pc = pc + 1 if (Action = snd_Alice_b0 or Action = snd_Alice_b1) and Alice
          .Action = rcv_Bob_b;
      a = zero and pc = pc + 1 if Action = rcv_Alice_a and Alice.Action =
          snd_Bob_a0;
      a = one and pc = pc + 1 if Action = rcv_Alice_a and Alice.Action =
          snd_Bob_a1;
      pc = pc + 1 if Action = Ha2_1 or Action = skip;
    end Evolution
end Agent

Evaluation
  f0 if Alice.a = Bob.b;
  f1 if Bob.pc > 4;
  f2 if Alice.s1 = Bob.s2 and !Alice.s1 = undefined;
end Evaluation

InitStates
  Environment.q1 = undefined and Environment.q2 = undefined and
  Environment.e1_2 = state1 and Environment.gc = 1 and
  Alice.b = undefined and Alice.s1 = undefined and Alice.q1 = 1 and Alice.q2
      = 0 and Alice.pc = 1 and
  Bob.a = undefined and Bob.s2 = undefined and Bob.q1 = 0 and Bob.q2 = 1 and
      Bob.pc = 1;
end InitStates

Formulae
  AG (((f0) and (f1)) -> (K (Alice, (f2)) and K (Bob, (f2))));
  AG (!(f0) -> (!K (Alice, (f2)) and !K (Bob, (f2))));
end Formulae
```

Listing D.2: qkd.ispl

Superdense Coding

```
Agent Environment
  Vars:
    q1i : {undefined}; q2i : {undefined}; q3i : {state2}; q4i : {state2}; q5i
        : {state2};
    q1 : {undefined, state2, state9};
    q2 : {undefined, state2, state9};
    q3 : {undefined, state2, state9};
```

```
    q4 : {undefined, state10, state11, state2, state9};
    q5 : {undefined, state2, state9};
    e1_2 : {undefined, state1, state16, state3, state4};
    e3_4 : {undefined, state5, state8};
    e1_2_3_4 : {undefined, state14, state17, state19, state6};
    gc : 1..17;
  end Vars
  Actions = {env1, env2, env3, env4, none};
  Protocol:
    gc = 15 and q1 = state2 : {env1};
    gc = 15 and q1 = state9 : {env2};
    gc = 16 and q5 = state2 : {env3};
    gc = 16 and q5 = state9 : {env4};
    Other: {none};
  end Protocol
  Evolution:
    gc = gc + 1 and e1_2 = state1 and e3_4 = state5 and e1_2_3_4 = undefined
        if e1_2_3_4 = state19 and Bob.Action = e1x3_4;
    gc = gc + 1 and e1_2 = state16 and e3_4 = state8 and e1_2_3_4 = undefined
         if e1_2_3_4 = state17 and Bob.Action = e1x3_4;
    gc = gc + 1 and e1_2 = state16 if e1_2 = state1 and Alice.Action = x1_2;
    gc = gc + 1 and e1_2 = state3 and e3_4 = state5 and e1_2_3_4 = undefined
        if e1_2_3_4 = state14 and Bob.Action = e1x3_4;
    gc = gc + 1 and e1_2 = state3 if e1_2 = state1 and Alice.Action = z1_1;
    gc = gc + 1 and e1_2 = state4 and e3_4 = state8 and e1_2_3_4 = undefined
        if e1_2_3_4 = state6 and Bob.Action = e1x3_4;
    gc = gc + 1 and e1_2 = state4 if e1_2 = state3 and Alice.Action = x1_2;
    gc = gc + 1 and e1_2 = undefined and e3_4 = undefined and e1_2_3_4 =
        state14 if e1_2 = state3 and e3_4 = state5 and Bob.Action = e2x3_3;
    gc = gc + 1 and e1_2 = undefined and e3_4 = undefined and e1_2_3_4 =
        state17 if e1_2 = state16 and e3_4 = state5 and Bob.Action = e2x3_3;
    gc = gc + 1 and e1_2 = undefined and e3_4 = undefined and e1_2_3_4 =
        state19 if e1_2 = state1 and e3_4 = state5 and Bob.Action = e2x3_3;
    gc = gc + 1 and e1_2 = undefined and e3_4 = undefined and e1_2_3_4 =
        state6 if e1_2 = state4 and e3_4 = state5 and Bob.Action = e2x3_3;
    gc = gc + 1 and q1 = state2 and q2 = state2 and e1_2 = undefined if e1_2
        = state1 and Bob.Action = m2p_5;
    gc = gc + 1 and q1 = state2 and q2 = state2 and e1_2 = undefined if e1_2
        = state16 and Bob.Action = m2p_5;
    gc = gc + 1 and q1 = state2 and q2 = state9 and e1_2 = undefined if e1_2
        = state3 and Bob.Action = m2m_5;
    gc = gc + 1 and q1 = state2 and q2 = state9 and e1_2 = undefined if e1_2
        = state4 and Bob.Action = m2m_5;
    gc = gc + 1 and q1 = state2 if q1 = state9 and Bob.Action = z1_7;
    gc = gc + 1 and q1 = state9 and q2 = state2 and e1_2 = undefined if e1_2
        = state3 and Bob.Action = m2p_5;
    gc = gc + 1 and q1 = state9 and q2 = state2 and e1_2 = undefined if e1_2
        = state4 and Bob.Action = m2p_5;
    gc = gc + 1 and q1 = state9 and q2 = state9 and e1_2 = undefined if e1_2
        = state1 and Bob.Action = m2m_5;
    gc = gc + 1 and q1 = state9 and q2 = state9 and e1_2 = undefined if e1_2
        = state16 and Bob.Action = m2m_5;
    gc = gc + 1 and q1 = state9 if q1 = state2 and Bob.Action = z1_7;
    gc = gc + 1 and q3 = state2 and q4 = state10 and e3_4 = undefined if e3_4
         = state5 and Bob.Action = m3p_6;
    gc = gc + 1 and q3 = state2 and q4 = state11 and e3_4 = undefined if e3_4
         = state8 and Bob.Action = m3p_6;
    gc = gc + 1 and q3 = state9 and q4 = state10 and e3_4 = undefined if e3_4
         = state8 and Bob.Action = m3m_6;
```

```
      gc = gc + 1 and q3 = state9 and q4 = state11 and e3_4 = undefined if e3_4
          = state5 and Bob.Action = m3m_6;
      gc = gc + 1 and q3 = undefined and q4 = undefined and e3_4 = state5 if q3
          = state2 and q4 = state2 and Bob.Action = e3x4_2;
      gc = gc + 1 and q4 = state10 if q4 = state11 and Bob.Action = x4_9;
      gc = gc + 1 and q4 = state11 and q5 = state9 if q4 = state11 and q5 =
          state2 and Bob.Action = e4x5_10;
      gc = gc + 1 and q4 = state11 if q4 = state10 and Bob.Action = x4_9;
      gc = gc + 1 and q4 = state2 if q4 = state10 and Bob.Action = m4p_11;
      gc = gc + 1 and q4 = state2 if q4 = state11 and Bob.Action = m4p_11;
      gc = gc + 1 and q4 = state9 if q4 = state10 and Bob.Action = m4m_11;
      gc = gc + 1 and q4 = state9 if q4 = state11 and Bob.Action = m4m_11;
      gc = gc + 1 if (q1 = state2 and Bob.Action = m1m_13) or (q1 = state2 and
          Bob.Action = m1p_13) or (q1 = state9 and Bob.Action = m1m_13) or (q1
          = state9 and Bob.Action = m1p_13) or (q4 = state10 and Bob.Action =
          z4_8) or (q4 = state10 and q5 = state2 and Bob.Action = e4x5_10) or (
          q4 = state11 and Bob.Action = z4_8) or (q5 = state2 and Bob.Action =
          m5m_14) or (q5 = state2 and Bob.Action = m5p_14) or (q5 = state2 and
          Bob.Action = x5_12) or (q5 = state9 and Bob.Action = m5m_14) or (q5 =
           state9 and Bob.Action = m5p_14) or (q5 = state9 and Bob.Action =
          x5_12) or Alice.Action = qsnd_Bob_1 or Alice.Action = skip or Bob.
          Action = skip;
  end Evolution
end Agent

Agent Alice
  Vars:
    x1 : {zero, one};
    x2 : {zero, one};
    q1 : 0..2;
    q2 : 0..2;
    q3 : 0..2;
    q4 : 0..2;
    q5 : 0..2;
    pc : 1..4;
  end Vars
  Actions = {z1_1, skip, x1_2, qsnd_Bob_1, none};
  Protocol:
    pc = 1 and x1 = zero : {skip};
    pc = 1 and x1 = one : {z1_1};
    pc = 2 and x2 = zero : {skip};
    pc = 2 and x2 = one : {x1_2};
    pc = 3 : {qsnd_Bob_1};
    Other: {none};
  end Protocol
  Evolution:
    q1 = 0 and pc = pc + 1 if Action = qsnd_Bob_1 and Bob.Action =
        qrcv_Alice_1;
    pc = pc + 1 if Action = z1_1 or Action = skip or Action = x1_2;
  end Evolution
end Agent

Agent Bob
  Vars:
    s5 : {undefined, zero, one};
    s1 : {undefined, zero, one};
    s4 : {undefined, zero, one};
    s3 : {undefined, zero, one};
    s2 : {undefined, zero, one};
    q1 : 0..2;
```

```
      q2 : 0..2;
      q3 : 0..2;
      q4 : 0..2;
      q5 : 0..2;
      pc : 1..15;
    end Vars
    Actions = {qrcv_Alice_1, e3x4_2, e2x3_3, e1x3_4, m2p_5, m2m_5, m3p_6, m3m_6
        , z1_7, skip, z4_8, x4_9, e4x5_10, m4p_11, m4m_11, x5_12, m1p_13,
       m1m_13, m5p_14, m5m_14, none};
    Protocol:
      pc = 1 : {qrcv_Alice_1};
      pc = 2 : {e3x4_2};
      pc = 3 : {e2x3_3};
      pc = 4 : {e1x3_4};
      pc = 5 : {m2p_5, m2m_5};
      pc = 6 : {m3p_6, m3m_6};
      pc = 7 and s2 = zero : {skip};
      pc = 7 and s2 = one : {z1_7};
      pc = 8 and s2 = zero : {skip};
      pc = 8 and s2 = one : {z4_8};
      pc = 9 and s3 = zero : {skip};
      pc = 9 and s3 = one : {x4_9};
      pc = 10 : {e4x5_10};
      pc = 11 : {m4p_11, m4m_11};
      pc = 12 and s4 = zero : {skip};
      pc = 12 and s4 = one : {x5_12};
      pc = 13 : {m1p_13, m1m_13};
      pc = 14 : {m5p_14, m5m_14};
      Other: {none};
    end Protocol
    Evolution:
      q1 = 1 and pc = pc + 1 if Action = qrcv_Alice_1 and Alice.Action =
          qsnd_Bob_1;
      q3 = 1 and q4 = 1 and pc = pc + 1 if Action = e3x4_2;
      q2 = 1 and q3 = 1 and pc = pc + 1 if Action = e2x3_3;
      q1 = 1 and q3 = 1 and pc = pc + 1 if Action = e1x3_4;
      s2 = zero and q2 = 2 and pc = pc + 1 if Action = m2p_5;
      s2 = one and q2 = 2 and pc = pc + 1 if Action = m2m_5;
      s3 = zero and q3 = 2 and pc = pc + 1 if Action = m3p_6;
      s3 = one and q3 = 2 and pc = pc + 1 if Action = m3m_6;
      q4 = 1 and q5 = 1 and pc = pc + 1 if Action = e4x5_10;
      s4 = zero and q4 = 2 and pc = pc + 1 if Action = m4p_11;
      s4 = one and q4 = 2 and pc = pc + 1 if Action = m4m_11;
      s1 = zero and q1 = 2 and pc = pc + 1 if (Action = m1p_13 and !(
          Environment.Action = env2)) or (Action = m1m_13 and (Environment.
          Action = env1));
      s1 = one and q1 = 2 and pc = pc + 1 if (Action = m1m_13 and !(Environment
          .Action = env1)) or (Action = m1p_13 and (Environment.Action = env2))
          ;
      s5 = zero and q5 = 2 and pc = pc + 1 if (Action = m5p_14 and !(
          Environment.Action = env4)) or (Action = m5m_14 and (Environment.
          Action = env3));
      s5 = one and q5 = 2 and pc = pc + 1 if (Action = m5m_14 and !(Environment
          .Action = env3)) or (Action = m5p_14 and (Environment.Action = env4))
          ;
      pc = pc + 1 if Action = z1_7 or Action = skip or Action = z4_8 or Action
          = x4_9 or Action = x5_12;
    end Evolution
end Agent
```

```
Evaluation
  f0 if Bob.pc > 14;
  f1 if Alice.x1 = Bob.s1;
  f2 if Alice.x2 = Bob.s5;
  f3 if Bob.pc < 14;
end Evaluation

InitStates
  Environment.q1 = undefined and Environment.q2 = undefined and Environment.
      q3 = state2 and Environment.q4 = state2 and Environment.q5 = state2 and
  Environment.e1_2 = state1 and Environment.e3_4 = undefined and Environment.
      e1_2_3_4 = undefined and Environment.gc = 1 and
  Alice.q1 = 1 and Alice.q2 = 0 and Alice.q3 = 0 and Alice.q4 = 0 and Alice.
      q5 = 0 and Alice.pc = 1 and
  Bob.s5 = undefined and Bob.s1 = undefined and Bob.s4 = undefined and Bob.s3
       = undefined and Bob.s2 = undefined and Bob.q1 = 0 and Bob.q2 = 1 and
      Bob.q3 = 2 and Bob.q4 = 2 and Bob.q5 = 2 and Bob.pc = 1;
end InitStates

Formulae
  AG ((f0) -> ((f1) and (f2)));
  AG ((f0) -> K (Bob, (f1) and (f2)));
  AG ((f3) -> !K (Bob, (f1) and (f2)));
  AG (!K (Alice, K (Bob, (f1) and (f2))));
end Formulae
```

Listing D.3: superdc.ispl