

UNIVERSITÀ DEGLI STUDI DI UDINE

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE

DOTTORATO DI RICERCA IN INFORMATICA E SCIENZE MATEMATICHE E FISICHE

PH.D. THESIS

Entang λ e: a Framework from Quantum Programming to Quantum Model Checking

CANDIDATE

Linda Anticoli

SUPERVISOR

Prof. Carla Piazza

CO-SUPERVISOR

Prof. Paolo Zuliani
Newcastle University, UK

INSTITUTE CONTACTS

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Università degli Studi di Udine

Via delle Scienze, 206

33100 Udine — Italia

+39 0432 558400

<http://www.dimi.uniud.it/>

AUTHOR'S CONTACTS

Via delle Scienze 206

Udine — Italia

+39 347 8478138

linda.anticoli@uniud.it

To my parents.

Acknowledgements

First, I shall thank my supervisors Professors Carla Piazza and Paolo Zuliani for giving me helpful hints during these years.

A special thanks goes to Red, to whom I owe a great deal of gratitude, love and admiration and who always trusted and backed me.

Thank you Luca, you will always remain my best role model for a scientist, mentor, and teacher.

Thank you Edoardo, for supporting me throughout this Ph.D. boosting my self-esteem in the time of need.

A special thank goes to Agostino Dovie, who had the chance to show me how Academia truly works and which are the important things to focus on.

Finally, I want to thank my family and all the other friends who were right there for me during these years.

Abstract

The main purpose of this thesis is to analyse the state-of-art in the fields of quantum programming languages and model checking of quantum algorithms, and to propose improvements involving a possible integration of the two aforementioned areas, which has not previously been provided in the existing literature.

Using as a starting point the quantum programming language Quipper and the quantum model checking system QPMC, we developed a tool, called **Entangle**, for the translation of Quipper programs, whose semantics is given in terms of quantum circuits, into QPMC models, whose semantics is given in terms of superoperator weighted quantum Markov chains. **Entangle** provides an ad-hoc verification tool for Quipper code, which we used in order to simulate and formally analyze quantum protocols.

In order to have a more complete framework, we then investigated recursive quantum programs and we implemented a module allowing to translate tail-recursive Quipper code as well. In order to perform such a translation, we developed an extended version of **Entangle**, from a fragment of Quipper, called *Quip-E*, to the QPMC model checker. This framework allows the verification of both recursive and non-recursive quantum programs.

We tested the improved version of **Entangle** on several different quantum algorithms, including an implementation of the quantum-switch based on the Grover's diffusion operator, and the BB84 protocol for quantum key distribution. We also explored possible uses of **Entangle** in order to verify quantum properties such as *entanglement*.

As a future work we consider a possible improvement of **Entangle**, that we called *Quipks*, and which is currently under development. We suggest that this method will allow an abstract, more efficient approach in the properties evaluation of a quantum protocol.

The experience in designing a quantum model checker led us to the investigation of new data structures for the representation of multipartite entanglement, which we introduce the final part of the thesis. Such data structures has been proposed in terms of evolving hypergraphs, called Evolving Entangled Hypergraphs (EEHs), which we suggest can represent quantum protocols where entanglement is an emergent behaviour by encoding the information of both its spatial and the temporal (evolution) parts. As a future work, we plan to define an operational semantics in terms of EEHs, and to use them as models to perform quantum spatio-temporal model checking.

Contents

Contents	ix
List of Figures	xiii
List of Tables	xv
I Preliminaries	1
1 Introduction and Motivation	3
1.1 Structure of the Thesis	5
2 Quantum Computation and Information	7
2.1 General formalisms of Quantum Mechanics	8
2.1.1 The space \mathbb{C}^n	8
2.1.2 Operators in \mathbb{C}^n	10
2.1.3 Postulates of quantum mechanics	15
2.2 Quantum information theory	23
2.2.1 Quantum computation	23
2.2.2 Quantum circuit model	24
2.2.3 Quantum information	29
2.2.4 Entanglement in Bipartite Systems	31
2.3 Quantum algorithms and protocols	33
2.3.1 Deutsch's algorithm	34
2.3.2 Deutsch-Jozsa algorithm	35
2.3.3 Grover's search algorithm	37
2.3.4 Teleportation protocol	42
2.3.5 Quantum Key Distribution	44
3 Model Checking	47
3.1 Creation of the model	48
3.2 Properties specification	51
3.2.1 Temporal logics	51
3.3 Model checking	55
3.3.1 CTL Model Checking	55
3.3.2 Symbolic CTL Model Checking	57

3.3.3	Probabilistic CTL Model Checking	59
3.3.4	PRISM	60
II	State of the art	63
4	Quantum Programming and Quantum Model Checking	65
4.1	Quantum programming languages	65
4.2	Quipper	68
4.2.1	Execution steps	69
4.2.2	Parameter/input distinction	69
4.2.3	Circuit description language	69
4.2.4	Simulation	70
4.3	Quantum programs verification	71
4.4	QPMC	72
4.4.1	The QPMC model checker	75
4.4.2	QPMC module	77
4.4.3	QCTL	78
III	Contribution	81
5	Entangle	83
5.1	From Circuits to Quantum Markov Chains	83
5.1.1	Circuits	84
5.1.2	From Strong Normal Form Circuits to QMC's	87
5.1.3	Translation Algorithm	88
5.2	Implementation	89
5.2.1	Scalability of the swap algorithm	91
5.3	Extension to Tail-recursive Quantum Programs	92
5.3.1	<i>Quip-E</i> : a Quipper recursive fragment	92
5.4	Structural Operational Semantics for <i>Quip-E</i>	95
5.5	Translation of <code>trc_C</code> Programs	99
5.6	The tool Entangle	103
5.6.1	Quipper	103
5.6.2	Tree Block	104
5.6.3	QPMC Block	104
5.7	Experimental Results	106
5.7.1	Deutsch–Jozsa	106
5.7.2	Grover's Quantum Search	107
5.7.3	Quantum Switch	111
5.7.4	Teleportation	114
5.7.5	BB84 Protocol	116
5.7.6	BB84 Tests:	119
5.7.7	Entanglement Detection:	121
5.7.8	Multiple SWAP Optimisation	122
5.8	Summary	124

6	Entangled Evolving Hypergraphs	127
6.1	Classification of three-qubit Entanglement	129
6.2	Evolving Entangled Hypergraphs	130
6.3	Example: Modeling Quantum Protocols	133
6.3.1	Teleportation Protocol	134
6.3.2	QKD using W states	136
6.3.3	Technical improvements and implementations	137
6.4	Summary and Future Work	140
7	Conclusion and Future Work	143
7.1	Future Work	143
7.1.1	<i>Quipks</i> : improving Entanglement	143
7.1.2	Applying the Evolving Entangled Hypergraphs	144
7.2	Conclusion	144
A	QPMC code	147
	Bibliography	155

List of Figures

2.1	Circuit representation of the Hadamard gate	27
2.2	Circuit Representation of the Controlled-not Gate.	28
2.3	Circuit Representation of a Generic Controlled Gate U	29
2.4	Quantum Circuit for the Deutsch's Algorithm.	34
2.5	Quantum circuit for the Deutsch-Jozsa algorithm.	37
2.6	Quantum Circuit for the Grover's Algorithm.	38
2.7	Detail of the Grover Operator G	38
2.8	Geometric Interpretation of Grover's Oracle and Phase Shift Gates.	41
2.9	Quantum Circuit for the Teleportation Protocol.	43
2.10	Quantum Circuit for $ \Phi^+\rangle$	43
3.1	Outline of Model Checking Process.	48
3.2	Time Models.	52
3.3	CTL Example Trees.	53
3.4	Example of PCTL P operator for the formula $P_{\leq 0.5}[\mathbf{F}(g)]$	55
3.5	Examples of satisfying paths.	55
3.6	Example of OBDD Transformation Steps.	58
3.7	PRISM Model Creation.	60
3.8	PRISM Model Checking.	61
4.1	QMC associated to the Quantum Coin Tossing.	75
5.1	Deutsch circuit in Quipper	85
5.2	Deutsch circuit with labels	85
5.3	Example of a circuit in Normal Form	86
5.4	Example of a circuit in Strong Normal Form	86
5.5	QMC associated to the circuit of Figure 5.4.	88
5.6	Quantum circuit with intermediate states.	89
5.7	Quantum Markov Chain for the Quipper circuit	90
5.8	Test circuit of size 7	91
5.9	Scalability test.	91
5.10	Entangle GUI.	103
5.11	Quipper Block	105
5.12	QPMC Block	105
5.13	Grover's algorithm in Quipper	108
5.14	QMC associated to Grover's Algorithm	109
5.15	Quantum Switch without superposition of variables.	111

5.16	Quantum Switch with Superposition of Variables.	112
5.17	Quantum Switch Simulation.	113
5.18	QMC for the Teleportation Protocol.	115
5.19	High level representation of BB84 steps.	117
5.20	Random string generation.	117
5.21	BB84 Body.	118
5.22	Recursive BB84 Body.	119
5.23	High level representation of recursive BB84 steps.	120
5.24	QCTL formulas and results.	122
5.25	Composition of SWAP.	123
5.26	Comparison between <code>multiply</code> (naive method) and <code>single</code>	125
6.1	Classification of three-qubit entanglement in terms of entangled hypergraphs	131
6.2	Forbidden entangled hypergraph of 3-qubit entanglement	131
6.3	Example of an EEH in which decoherence occurs.	133
6.4	Example of an EEH in which entanglement creation occurs.	134
6.5	Teleportation Pseudocode and Circuit	135
6.6	EEH for an ideal teleportation protocol.	136
6.7	Teleportation Pseudocode and Circuit with Noise	137
6.8	EEHs for a teleportation protocol with noise.	138
6.9	QKD with W states pseudocode and EEH.	139
6.10	QKD with W states EEH noisy channel.	139
6.11	Graphical representation of a W state EH via incidence matrix	140

List of Tables

2.1	One-bit Boolean Functions.	34
4.1	Example of Property Specification in QCTL.	79
5.1	Operational Semantics of <i>Quip-E</i>	97
5.2	Deutsch-Jozsa Constant Verification.	107
5.3	Deutsch-Jozsa Balanced Verification.	107
5.4	Grover's Algorithm Verification.	110
5.5	Grover's Algorithm Verification.	110
5.6	Quantum Switch Verification.	113
5.7	Teleportation Protocol Verification.	115
5.8	QCTL example tests for recursive and non-recursive BB84 protocol. . .	120
6.1	Three-qubit GSD's coefficients that make nonzero concurrences and tangle	130

I

Preliminaries

1

Introduction and Motivation

Classical computation, in its infancies, was mainly involved with hardware, and any attempt to specify an early-computer algorithm was an intriguing sequence of hardware-related steps, e.g., by means of punching cards and/or writing low level machine code. During this period, computation was quite context-dependent, since the same program was not guaranteed to be portable on different machines. Moreover, the specification languages were difficult to manage and a prior knowledge of the physical details of the machine would be needed to write programs correctly. Later, the theory of computation allowed to abstract from the underlying physical model, by focusing more on the common characteristics that each machine had and providing a way to write programs and algorithms in a more intuitive way.

The specification of algorithms in human-readable form and their translation into machine executable code is one of the main goals of high-level programming languages. Nowadays we are able to develop and test computer programs without any knowledge of the underlying physical hardware, which becomes completely transparent. Programming languages, which are endowed with a set mathematical and logical rules, allow to design and implement algorithms which are completely independent from the computer where they run on; compilers are now responsible of the translation of the source code into a lower level, more performing version for any specific hardware architecture.

In the late 80's a new model of computation was proposed by Feynman [38], which investigated how to simulate a quantum system by means of a classical computer. In principle, any quantum system can, at the cost of an exponential complexity growing with the size of the quantum system, be simulated by a classical computer. Later, Feynman on the one hand and Deutsch [30] on the other, proposed, in different ways, the idea of a *quantum* computer. The first model of quantum computer was the quantum Turing machine, a direct generalisation of the classical one: the head states and the tape symbols are represented by orthogonal quantum states, the head position corresponds to a quantum observable with integer spectrum, and the transition function is a unitary transformation [93]. Later, it was introduced the quantum circuit model, i.e., a generalisation of the classical circuit model which has become a widely used notation for describing quantum algorithms. Similar to the classical case, even if quantum circuits

have a simple mathematical description, they could be very difficult to realise in practice without a deep knowledge of the essential features of the physical phenomena under consideration. The applications of quantum computers and the difficulty to separate the underlying architecture from the logical behaviour of the algorithms, justify the need for tools providing an abstraction from a low-level description of quantum “programs” and protocols, by allowing programmers to use a quantum computer without knowing the laws of quantum physics. A similar tool might be regarded as a *quantum programming language*. There have been many attempts to develop a quantum programming language, usually relying on the quantum circuit model: we can cite, [94], Quipper [44] and QCL [65], among other examples of quantum programming languages.

Classically, the introduction of high-level formalisms allows to define and automatically verify formal properties of algorithms by abstracting away from low-level physical details. In order to assess the correctness of a given algorithm, formal verification techniques have been introduced; they are an important tool for the validation and verification of programs in classical computer science. Experimental verification (i.e., testing) is not as precise as formal methods, since an algorithm might be tested on several cases, but there is no assurance that each possible error is avoided—in particular if the tests are not well developed. By using formal verification techniques such as Hoare logics and *model checking*, among others, it is possible to test the properties of an algorithm by evaluating all possible cases, since we unfold all its possible computation paths. In the context of quantum computation, which is based on the counter-intuitive laws of quantum physics, the possibility of testing quantum algorithms becomes very important, since in their specification might be present errors difficult to find without a thorough knowledge of the restrictions imposed by the underlying model. In particular, protocols for quantum cryptography, which are deeply investigated at the moment due to their applications on the secure transmission of information, require certifications of correctness in order to be used.

The main aim of this thesis is to provide a tool allowing to specify and test quantum algorithms and protocols. Indeed, at present quantum algorithms specification and their formal verification are two deeply investigated fields which provide on the one hand a class of programming languages and, on the other hand some verification tools, but there is a lack of an integrated framework allowing to perform both the tasks. Due to this lack, we isolated two suitable tools: the functional programming language Quipper [84] and the quantum model checking system QPMC [35], and we decided to use them as a starting point for the development of a framework providing both a high-level programming style and a formal verification tool. In details, Quipper is a quantum programming language based on the Haskell functional programming language, that allows to build quantum circuits by describing them in a simple programming style and provides the possibility to simulate the circuit. QPMC is a model checker for quantum protocols that uses an extension of PCTL, a probabilistic temporal logic, to verify properties of quantum protocols. Quipper has been used to program a set of non-trivial quantum algorithms, it is supported by a community and provides a high-level programming environment based on Haskell. Unfortunately, Quipper lacks of a built-in formal verification tool. On the other hand, QPMC supports formal verification but it is based on a low-level specification language. Hence, we decided to build a tool, i.e., **Entangle**, which acts as a bridge between them by isolating a subclass of Quipper, that

we called *Quip-E*, and translating it into the QPMC formalism, providing an ad-hoc verification framework to *Quip-E* programmers. The development of the aforementioned translation framework has been a first step in the direction of providing a complete programming and verification framework for quantum computing. For this reason we are currently working on improvements from both the quantum programming language and the quantum model-checker perspective.

1.1 Structure of the Thesis

The thesis is structured into three parts; the first one is devoted to introduce the quantum formalism and the model checking one. Since this thesis is the output of an interdisciplinary work between computer science and physics, we will provide the suitable instruments for understanding what follows, both from a computer science and a physics perspective. The second part is devoted to the analysis of the state of the art in the fields of quantum programming and quantum model checking, and provides a thorough description of Quipper and QPMC. The third part is devoted to the presentation of my contributions in the aforementioned fields, together with a parallel work on the representation of tripartite entanglement, which is still in its infancies.

The Chapters are organised as follows:

Chapter 2 provides a short introduction to the quantum formalism used throughout the thesis, by presenting an essential mathematical description of some of the concepts of quantum mechanics. First, we recall some basic linear algebra, which will be used in the formulation of the *postulates of quantum mechanics*. Then, we will briefly introduce the notion of quantum information theory, quantum information and entanglement. The Chapter ends with a description of the main quantum algorithms and protocols used in the thesis.

Chapter 3 presents at an high-level the concept of model checking, a formal verification technique used to assess the correctness of algorithms. We will define all the three steps involved in the model checking process and the concept of *temporal logic* will be introduced.

Chapter 4 analyses the state-of-art in the fields of quantum programming languages, by briefly presenting some of the milestones. Then it focuses on Quipper, whose main features are presented, together with some examples. The second part of the Chapter is devoted to the analysis of the milestones in the field of quantum program verification techniques. The Chapter ends with an in depth presentation of QPMC, and of its semantics given in terms of *quantum Markov chains*.

Chapter 5 shows the framework, called **Entangle**, that we implemented in order to translate Quipper-like programs into QPMC structures. The first part is devoted to the formalisation of the main structures used, showing how to translate quantum circuits into quantum Markov chains. The second part of the Chapter extends the presented tool, by allowing the translation of *tail-recursive* quantum programs. In this section we isolated a Quipper recursive fragment, and we defined its syntax and operational semantics. In the third part we present some examples of how to use **Entangle** in order to implement and translate some known quantum algorithms and protocols in their recursive (when necessary) and non-recursive version: e.g., Deutsch-Jozsa, instances of Grover's search, teleportation and quantum key distribution. Some improvements, e.g.,

a faster algorithm for computing the matrix element of a permutation operator, and an optimised version of the framework (currently still in its infancies), are discussed at the end of the Chapter.

Chapter 6 proposes a new structure to represent entanglement in multipartite systems, namely the *evolving entangled hypergraphs*. This technique, which by now has been proposed only in the two and tripartite case, is based on the classification proposed in [42] and it investigates an alternative to QMCs when dealing with entangled systems. Nevertheless, we emphasise that this approach is still under investigation. First, a classification of tripartite entangled states is given in terms of entangled hypergraphs, which later in the Chapter will be used to evaluate whether entanglement arises or decays during a computation.

Chapter 7 ends the thesis.

2

Quantum Computation and Information

The subjects of quantum computation and quantum information are related to the study of information processing tasks achieved through quantum mechanical systems. For this reason, the laws of quantum mechanics are at the basis of quantum computation and quantum information. Born in the first years of 1900, quantum mechanics is a fundamental theory in physics aiming at the the description of nature at its smallest scales. Quantum mechanics has been formulated due to the difficulty to deal with observations which could not be explained by the laws of classical physics, and then it evolved into a mathematical formalism. Quantum mechanics is, at present, the most accurate and complete description of nature.

In this Chapter we explore the basic notions which are needed in order to understand in a satisfactory way the research contribution of the thesis, for this reason we won't delve into a more in-depth description of quantum mechanics, but we provide an introduction about quantum system and list the notation and formalisms used to describe it.

Section 2.1 begins with an high level description of the basic formalisms of quantum mechanics. Then, we can find a review of linear algebra and the *Dirac* notation. The final part of this Section describes the basic postulates of quantum mechanics. Section 2.2.1 explores the notion of Quantum Computation, quantum bits (herein *qubits*) and quantum gates and circuits. Section 2.2.3 briefly describes what Quantum Information is, together with the notion of *entanglement* and some applications, i.e., quantum cryptography and teleportation. The Chapter ends with a short presentation of the main quantum algorithms used in the following part of this work.

Similar to the classical case, in quantum mechanics we should provide a notion of *state* of a quantum system, which will evolve according to certain rules, of *observable* and of *measurement*. We will describe two formalisms of quantum mechanics that provide a description for the aforementioned notions, namely the *state vector* formalism and the *density operator* one. The second one enriches the first by allowing to treat also a larger class of quantum states and operations.

2.1 General formalisms of Quantum Mechanics

Before we deal with formalisms and rules governing the quantum world we shall give a light review of basic linear algebra that will be used in the context of quantum computation and information; in this way even with no prior knowledge of quantum mechanics, the reader can understand what follows.

The following definitions are presented using the Dirac (or *bra-ket*) notation, widely used in physics to describe quantum systems.

We used [52, 64, 60] as references, which we suggest also for a more in-depth study.

2.1.1 The space \mathbb{C}^n

The aim of this Section is to describe the properties of \mathbb{C}^n , i.e., the set n -tuples of complex numbers. It will represent the prototypical finite-dimensional Hilbert space associated to the quantum systems that we will consider in the following.

$$\mathbb{C}^n ::= \left\{ \begin{pmatrix} \psi_1 \\ \vdots \\ \psi_n \end{pmatrix} \mid \psi_i \in \mathbb{C}, i = 1, \dots, n \right\} \quad (2.1)$$

Each element of such space can be represented by means of a more compact notation, known as *Dirac* notation, e.g.:

$$|\psi\rangle ::= \begin{pmatrix} \psi_1 \\ \vdots \\ \psi_n \end{pmatrix}$$

where $|\psi\rangle$ is called *ket*.

To each ket $|\psi\rangle$ it is uniquely associated an object $\langle\psi|$, called *bra*, represented by an one-row matrix as follows:

$$\langle\psi| ::= (\psi_1^*, \dots, \psi_n^*)$$

where ψ_i^* are the complex conjugates of the numbers ψ_i .¹

\mathbb{C}^n is proved to be a *complex linear space* with respect to the two following composition laws, a vector sum and a scalar multiplication with a complex number α , which are defined as follows:

$$\begin{aligned} |\psi\rangle + |\varphi\rangle &= \begin{pmatrix} \psi_1 \\ \vdots \\ \psi_n \end{pmatrix} + \begin{pmatrix} \varphi_1 \\ \vdots \\ \varphi_n \end{pmatrix} ::= \begin{pmatrix} \psi_1 + \varphi_1 \\ \vdots \\ \psi_n + \varphi_n \end{pmatrix} \\ \alpha|\psi\rangle &= \alpha \begin{pmatrix} \psi_1 \\ \vdots \\ \psi_n \end{pmatrix} ::= \begin{pmatrix} \alpha\psi_1 \\ \vdots \\ \alpha\psi_n \end{pmatrix} \end{aligned} \quad (2.2)$$

¹Given a complex number $z = x + iy$, its complex conjugate is $z^* = x - iy$

Definition 2.1.1 (Inner product space). Given the complex linear space \mathbb{C}^n , we can define an *inner* product as a function, denoted as $\langle \cdot | \cdot \rangle : \mathbb{C}^n \times \mathbb{C}^n \rightarrow \mathbb{C}$, mapping two vectors $|\psi\rangle$ and $|\varphi\rangle \in \mathbb{C}^n$, to a complex number, as follows:

$$\langle \psi | \varphi \rangle = (\psi_1^*, \dots, \psi_n^*) \begin{pmatrix} \varphi_1 \\ \vdots \\ \varphi_n \end{pmatrix} ::= \sum_j \psi_j^* \varphi_j$$

Such inner product can be proven to satisfy the following properties:

1. Linearity on the second argument:

$$\langle \psi | (\alpha|\phi\rangle + \beta|\chi\rangle) \rangle = \alpha\langle \psi | \phi \rangle + \beta\langle \psi | \chi \rangle$$

where $\alpha, \beta \in \mathbb{C}$ and $|\phi\rangle, |\chi\rangle \in \mathbb{C}^n$;

2. $\langle \psi | \varphi \rangle = \langle \varphi | \psi \rangle^*$

3. $\langle \psi | \psi \rangle \geq 0$ with equality iff $|\psi\rangle = \vec{0}$
where $\vec{0}$ is the null vector in \mathbb{C}^n , i.e., the vector whose entries are all zeroes.

Two vectors $|\psi\rangle$ and $|\phi\rangle \in \mathbb{C}^n$ such that $\langle \psi | \phi \rangle = 0$ are said to be *orthogonal*.

A complex linear space \mathbb{C}^n endowed with an inner product is known as a *inner product space*.

The inner product space \mathbb{C}^n can be equipped with a *norm* induced by the inner product as follows:

$$\| |\psi\rangle \| ::= \sqrt{\langle \psi | \psi \rangle} \quad (2.3)$$

It is a map $\| \cdot \| : \mathbb{C}^n \rightarrow \mathbb{R}$, which satisfies the following properties:

1. $\| |\psi\rangle \| = 0 \iff |\psi\rangle = \vec{0}$
2. $\| \alpha|\psi\rangle \| = |\alpha| \| |\psi\rangle \| \quad \forall \alpha \in \mathbb{C}, \forall |\psi\rangle \in \mathbb{C}^n$
3. $\| |\psi\rangle + |\varphi\rangle \| \leq \| |\psi\rangle \| + \| |\varphi\rangle \| \quad \forall |\psi\rangle, |\varphi\rangle \in \mathbb{C}^n$

A vector whose norm equals 1, i.e., $\| |\psi\rangle \| = 1$, is said to be *normalised*. Moreover, a set of vectors $\{ |\phi_1\rangle, \dots, |\phi_k\rangle \}$ is an *orthonormal set* if $\langle \phi_i | \phi_j \rangle = \delta_{ij}$ for $i, j = 1, \dots, k$.²

A central notion in the formalism of quantum mechanics is the one of *Hilbert space*. In the case of finite-dimensional linear vector spaces an Hilbert space is exactly the same thing as an inner product space and the two terms will be interchangeable. On the contrary, in the infinite-dimensional case an Hilbert space is an inner product space which satisfies two additional constraints, namely completeness and separability. However, since in quantum computation and quantum information only finite-dimensional inner products spaces are considered, we won't define such constraints.

² δ_{ij} , also known as Kronecker delta, is defined as follows:

$$\delta_{ij} \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

The dimension of an inner product space can be increased by means of an operation, called *tensor product*, which can be intuitively be regarded as a composition of two inner product spaces and it is described in the following.

It is possible to combine two Hilbert spaces \mathbb{C}^n and \mathbb{C}^m in order to obtain a larger Hilbert space $\mathbb{C}^n \otimes \mathbb{C}^m$ by means of the operation \otimes called *tensor product*. The full definition of the tensor product operation is quite complex but for our purposes we will limit to say that, given the vectors $|\psi\rangle \in \mathbb{C}^n$ and $|\varphi\rangle \in \mathbb{C}^m$ their tensor product $|\psi\rangle \otimes |\varphi\rangle$ is a vector of the space $\mathbb{C}^n \otimes \mathbb{C}^m$. The tensor product satisfies the following properties:

1. $|\psi\rangle \otimes (\alpha|\varphi\rangle + \beta|\chi\rangle) = \alpha(|\psi\rangle \otimes |\varphi\rangle) + \beta(|\psi\rangle \otimes |\chi\rangle)$
2. $(\alpha|\varphi\rangle + \beta|\chi\rangle) \otimes |\psi\rangle = \alpha(|\varphi\rangle \otimes |\psi\rangle) + \beta(|\chi\rangle \otimes |\psi\rangle)$

Moreover, in the tensor product Hilbert space $\mathbb{C}^n \otimes \mathbb{C}^m$ an inner product can be defined, by resorting to the inner products in \mathbb{C}^n and \mathbb{C}^m respectively, as:

$$\langle \psi_1 | \otimes \langle \psi_2 | \langle \varphi_1 | \otimes \langle \varphi_2 |_{\mathbb{C}^n \otimes \mathbb{C}^m} ::= \langle \psi_1 | \varphi_1 \rangle_{\mathbb{C}^n} \cdot \langle \psi_2 | \varphi_2 \rangle_{\mathbb{C}^m} \quad (2.4)$$

The tensor product Hilbert space $\mathbb{C}^n \otimes \mathbb{C}^m$ can be proven to be isomorphic to the Hilbert space \mathbb{C}^{nm} .

2.1.2 Operators in \mathbb{C}^n

Definition 2.1.2 (Linear Operator). Let \mathbb{C}^n be a Hilbert space. A map $\hat{L} : \mathbb{C}^n \rightarrow \mathbb{C}^n$ is a *linear operator* if it acts linearly on the input, i.e.:

$$\hat{L}(\alpha|\psi\rangle + \beta|\varphi\rangle) ::= \alpha\hat{L}|\psi\rangle + \beta\hat{L}|\varphi\rangle \quad (2.5)$$

$$\forall \alpha, \beta \in \mathbb{C} \text{ and } \forall |\psi\rangle, |\varphi\rangle \in \mathbb{C}^n$$

An example of a trivial linear operator is the *identity* operator, denoted by \mathbb{I} and defined as :

$$\mathbb{I}|\psi\rangle = |\psi\rangle \quad \forall |\psi\rangle \in \mathcal{H} \quad (2.6)$$

Linear operators \hat{L}_1 and \hat{L}_2 can be summed, multiplied by a scalar and composed in order to obtain other linear operators such as $\hat{L}_1 + \hat{L}_2$, $\alpha\hat{L}_1$ and $\hat{L}_1\hat{L}_2$ as follows:

$$(\hat{L}_1 + \hat{L}_2)|\psi\rangle ::= \hat{L}_1|\psi\rangle + \hat{L}_2|\psi\rangle \quad \forall |\psi\rangle \in \mathbb{C}^n \quad (2.7)$$

$$\alpha\hat{L}_1|\psi\rangle ::= \alpha(\hat{L}_1|\psi\rangle) \quad \forall |\psi\rangle \in \mathbb{C}^n \quad \forall \alpha \in \mathbb{C} \quad (2.8)$$

$$(\hat{L}_2\hat{L}_1)|\psi\rangle ::= \hat{L}_2(\hat{L}_1|\psi\rangle) \quad \forall |\psi\rangle \in \mathbb{C}^n \quad (2.9)$$

Definition 2.1.3 (Eigenvalue equation). Given a linear operator \hat{L} , any nonzero vector $|\ell\rangle \in \mathbb{C}^n$ is an *eigenstate* of \hat{L} with *eigenvalue* $\lambda \in \mathbb{C}$ if the following equation is satisfied:

$$\hat{L}|\ell\rangle = \lambda|\ell\rangle \quad (2.10)$$

An eigenvalue λ is said to be *n-degenerate* if there exist n linearly independent eigenvectors $\{|\ell_1\rangle, \dots, |\ell_n\rangle\}$ associated to it. It can be proven that any linear combination with complex coefficients c_i of the eigenvectors $|\ell_i\rangle$ is an eigenvector of \hat{L} with the same eigenvalue λ . In fact, due to the linearity of \hat{L} :

$$\hat{L}\left(\sum_{i=1}^n c_i|\ell_i\rangle\right) = \sum_{i=1}^n c_i\hat{L}|\ell_i\rangle = \sum_{i=1}^n c_i\lambda|\ell_i\rangle = \lambda\left(\sum_{i=1}^n c_i|\ell_i\rangle\right) \quad (2.11)$$

Therefore, the eigenvectors associated to a given eigenvalue λ form a linear subspace called *eigenspace* associated to λ .

Matrix representation of a linear operator: Linear operators in \hat{L} on \mathbb{C}^n can be represented in terms of square matrices. There exists a one-to-one correspondence between linear operators in \mathbb{C}^n and the set of $n \times n$ complex matrices $M_n(\mathbb{C})$.

A linear operator \hat{L} has a matrix representation L which depends on the choice of basis set for the space \mathbb{C}^n . In fact, given an orthonormal basis $\{|\phi_i\rangle\} \in \mathbb{C}^n$, the matrix associated to the operator \hat{L} is:

$$L = \begin{pmatrix} L_{11} & \dots & L_{1n} \\ \vdots & \ddots & \vdots \\ L_{n1} & \dots & L_{nn} \end{pmatrix} \quad \text{such that} \quad L_{ij} ::= \langle \phi_i | \hat{L} | \phi_j \rangle \quad (2.12)$$

The action of a linear operator \hat{L} onto an arbitrary $|\psi\rangle \in \mathbb{C}^n$ turns out to be equivalent to the usual multiplication of a matrix by a column vector:

$$L|\psi\rangle = \begin{pmatrix} L_{11} & \dots & L_{1n} \\ \vdots & \ddots & \vdots \\ L_{n1} & \dots & L_{nn} \end{pmatrix} \begin{pmatrix} \psi_1 \\ \vdots \\ \psi_n \end{pmatrix} ::= \begin{pmatrix} \sum_j L_{1j}\psi_j \\ \vdots \\ \sum_j L_{nj}\psi_j \end{pmatrix} \quad (2.13)$$

Any operator is uniquely determined by its matrix elements.

Tensor product of linear operators: If \hat{A} and \hat{B} are linear operators acting on the Hilbert spaces \mathbb{C}^n and \mathbb{C}^m respectively, the tensor product linear operator $\hat{A} \otimes \hat{B}$, acting on $\mathbb{C}^n \otimes \mathbb{C}^m$, is first defined on product vectors by:

$$(\hat{A} \otimes \hat{B})|\psi\rangle \otimes |\varphi\rangle ::= \hat{A}|\psi\rangle \otimes \hat{B}|\varphi\rangle \quad \forall |\psi\rangle \in \mathbb{C}^n, \forall |\varphi\rangle \in \mathbb{C}^m \quad (2.14)$$

and then extended to sum of tensor product vectors in a linear way. The matrix representation of $\hat{A} \otimes \hat{B}$ can also be obtained by using the matrix representations of \hat{A} and

\hat{B} ; in fact:

$$A \otimes B = \begin{pmatrix} A_{11}B & A_{12}B & \dots & A_{1n}B \\ A_{21}B & A_{22}B & \dots & A_{2n}B \\ \vdots & & \ddots & \vdots \\ A_{n1}\hat{B} & A_{n2}\hat{B} & \dots & A_{nn}\hat{B} \end{pmatrix} \quad (2.15)$$

where $A_{ij}B$ is the matrix obtained by multiplying each entry of the matrix B by the complex number A_{ij} .

Adjoint operators: Given a linear operator $\hat{L} : \mathbb{C}^n \rightarrow \mathbb{C}^n$, the adjoint (or Hermitian conjugate) of \hat{L} is the operator $\hat{L}^\dagger : \mathbb{C}^n \rightarrow \mathbb{C}^n$ defined by the following condition on its matrix elements:

$$\langle \psi | \hat{L}^\dagger | \phi \rangle = \langle \phi | \hat{L} | \psi \rangle^* \quad (2.16)$$

for all $|\psi\rangle, |\phi\rangle \in \mathbb{C}^n$. The adjoint operator \hat{L}^\dagger has matrix elements $(L^\dagger)_{ij} = L_{ji}^*$. In fact, given an orthonormal basis set $\{|\phi_i\rangle\}$:

$$(L^\dagger)_{ij} = \langle \phi_i | \hat{L}^\dagger | \phi_j \rangle = \langle \phi_j | \hat{L} | \phi_i \rangle^* = L_{ji}^* \quad (2.17)$$

The operation of taking the adjoint of an operator satisfies the following properties:

- $(\hat{L}_1 + \hat{L}_2)^\dagger = \hat{L}_1^\dagger + \hat{L}_2^\dagger$;
- $(\hat{L}^\dagger)^\dagger = \hat{L}$;
- $(\alpha \hat{L}^\dagger) = \alpha^* \hat{L}^\dagger$;
- $(\hat{L}_1 \hat{L}_2)^\dagger = \hat{L}_2^\dagger \hat{L}_1^\dagger$.

Hermitian operators A linear operator $\hat{L} : \mathbb{C}^n \rightarrow \mathbb{C}^n$, is *Hermitian*, or self adjoint, if it coincides with its adjoint, i.e.:

$$\hat{L} = \hat{L}^\dagger \quad (2.18)$$

as a consequence, the matrix elements of an Hermitian operator \hat{L} possess the following property:

$$L_{ij}^* = L_{ji} \quad (2.19)$$

$$\langle \psi | \hat{L} | \phi \rangle = \langle \phi | \hat{L} | \psi \rangle^* \quad (2.20)$$

for all $|\psi\rangle, |\phi\rangle \in \mathbb{C}^n$.

Unitary operators

Definition 2.1.4 (Inverse operator). Let \hat{L} be an operator in \mathbb{C}^n , we denote with $\mathcal{R} = \{\hat{L}|\psi\rangle : |\psi\rangle \in \mathbb{C}^n\}$ the *range* of \hat{L} . The operator \hat{L}^{-1} on \mathcal{R} is an inverse operator if the following holds:

$$\hat{L}\hat{L}^{-1}|\psi\rangle = |\psi\rangle \quad \forall |\psi\rangle \in \mathcal{R} \quad (2.21)$$

$$\hat{L}^{-1}\hat{L}|\psi\rangle = |\psi\rangle \quad \forall |\psi\rangle \in \mathcal{H} \quad (2.22)$$

from which follows that $\hat{L}\hat{L}^{-1} = \hat{L}^{-1}\hat{L} = \mathbb{I}$.

Definition 2.1.5 (Unitary operator). An operator \hat{U} on \mathcal{H} is a unitary operator if its inverse is equal to its adjoint, i.e., $\hat{U}^{-1} = \hat{U}^\dagger$:

$$\hat{U}^\dagger\hat{U} = \hat{U}\hat{U}^\dagger = \mathbb{I} \quad (2.23)$$

Projectors In order to introduce the projection operators we need to define two preliminary objects, i.e., *orthogonal complement* and *orthogonal projection*.

Definition 2.1.6 (Orthogonal complement). Let S be a subspace of an Hilbert space \mathbb{C}^n . A vector $|\psi\rangle \in \mathbb{C}^n$ is orthogonal to S if, for all $|\omega\rangle \in S$, $\langle\psi|\omega\rangle = 0$. The space $S^\perp = \{|\psi\rangle \in \mathbb{C}^n | \langle\psi|\omega\rangle = 0 \quad \forall |\omega\rangle \in S\}$ is called orthogonal complement of S .

Two linear subspaces S_1 and S_2 are orthogonal, i.e., $S_1 \perp S_2$ if the following condition holds:

$$\langle\psi|\omega\rangle = 0 \quad \forall |\psi\rangle \in S_1 \quad \forall |\omega\rangle \in S_2 \quad (2.24)$$

Theorem 2.1.1 (Orthogonal projection). Let S be a linear subspace of \mathcal{H} , then each vector $|\chi\rangle \in \mathcal{H}$ can be *uniquely* decomposed into $|\chi\rangle = |\psi\rangle + |\omega\rangle$, with $|\psi\rangle \in S$ and $|\omega\rangle \in S^\perp$.

Definition 2.1.7 (Projection operators). Let S be a linear subspace of \mathbb{C}^n , and $|\chi\rangle \in \mathbb{C}^n$, $|\psi\rangle \in S$ and $|\omega\rangle \in S^\perp$ respectively. Given $|\chi\rangle = |\psi\rangle + |\omega\rangle$, a linear operator $\hat{P}_S : \mathbb{C}^n \rightarrow \mathbb{C}^n$ is a projection operator on S , and it is defined as follows:

$$\hat{P}_S|\chi\rangle ::= |\psi\rangle \quad (2.25)$$

Given a complete orthonormal basis set $\{|\psi_i\rangle\}$ for S , the projection operator on S is defined as:

$$\begin{aligned} \hat{P}_S|\chi\rangle &::= \sum_{i=1}^{\dim(S)} \langle\psi_i|\chi\rangle |\psi_i\rangle \\ &= \sum_{i=1}^{\dim(S)} |\psi_i\rangle \langle\psi_i| |\chi\rangle \end{aligned} \quad (2.26)$$

From Eq.2.26 follows that $\hat{P}_S = \sum_{i=1}^{\dim(S)} |\psi_i\rangle \langle\psi_i|$.

The projection operator onto the subspace S^\perp is $\hat{P}_{S^\perp} ::= \mathbb{I} - \hat{P}_S$.

Theorem 2.1.2. An operator \hat{P} is a projection operator if and only if the following conditions hold:

- $\hat{P}^2 = \hat{P}$ idempotent;
- $\hat{P}^\dagger = \hat{P}$ hermitian.

The only eigenvalues of a projection operator are 0 and 1.

Outer Product: a useful way to represent linear operators is known as the outer product representation. Given two vectors $|\psi\rangle, |\varphi\rangle \in \mathbb{C}^n$, we define $|\psi\rangle\langle\varphi|$ to be the linear operator whose action on a generic vector $|\chi\rangle \in \mathbb{C}^n$ is defined as follows:

$$(|\psi\rangle\langle\varphi|)|\chi\rangle ::= |\psi\rangle\langle\varphi|\chi\rangle = \langle\varphi|\chi\rangle|\psi\rangle \quad (2.27)$$

i.e., the result of the operator $|\psi\rangle\langle\varphi|$ acting on $|\chi\rangle$ is equal the multiplication of the vector $|\psi\rangle$ by the complex number $\langle\varphi|\chi\rangle$.

The linear combinations of outer product operators is given in the usual way, such that Eq.2.27 becomes:

$$\sum_i \alpha_i (|\psi_i\rangle\langle\varphi_i|)|\chi\rangle ::= \sum_i \alpha_i |\psi_i\rangle\langle\varphi_i|\chi\rangle \quad (2.28)$$

Given an orthonormal basis set $\{|i\rangle\} \in \mathbb{C}^n$, such that an arbitrary vector $|\psi\rangle \in \mathbb{C}^n$ can be written as $|\psi\rangle = \sum_i \alpha_i |i\rangle$ with $\alpha_i \in \mathbb{C}$ and $\langle i|\psi\rangle = \alpha_i$:

$$\left(\sum_i |i\rangle\langle i|\right)|\psi\rangle = \sum_i |i\rangle\langle i|\psi\rangle = \sum_i \alpha_i |i\rangle = |\psi\rangle \quad (2.29)$$

As a consequence, we can state the *completeness relation* as follows:

$$\sum_i |i\rangle\langle i| = \mathbb{I} \quad (2.30)$$

Given two vectors $|\psi\rangle, |\varphi\rangle \in \mathbb{C}^n$, the matrix representation of the outer product can be given as follows:

$$|\psi\rangle\langle\varphi| = \begin{pmatrix} \psi_1 \\ \vdots \\ \psi_n \end{pmatrix} (\varphi_1^*, \dots, \varphi_n^*) = \begin{pmatrix} \psi_1\varphi_1^* & \psi_1\varphi_2^* & \dots & \psi_1\varphi_n^* \\ \psi_2\varphi_1^* & \psi_2\varphi_2^* & \dots & \psi_2\varphi_n^* \\ \vdots & \ddots & \ddots & \vdots \\ \psi_n\varphi_1^* & \psi_n\varphi_2^* & \dots & \psi_n\varphi_n^* \end{pmatrix} \quad (2.31)$$

Spectral decomposition for hermitian operators In this paragraph we recall two theorems about hermitian operators which will be used in the following.

Theorem 2.1.3. The eigenvalues of an hermitian operator \hat{L} are real numbers. Eigenstates corresponding to two different eigenvalues are orthogonal.

Theorem 2.1.4 (Spectral decomposition). Let \hat{L} be an hermitian operator over \mathbb{C}^n satisfying the eigenvalue equation $\hat{L}|\ell_i\rangle = \lambda_i|\ell_i\rangle$. The set of all the eigenstates of \hat{L} forms

an orthonormal basis set for \mathbb{C}^n . Thus any vector $|\psi\rangle \in \mathbb{C}^n$ can be expanded as follows:

$$|\psi\rangle = \sum_i \langle \ell_i | \psi \rangle |\ell_i\rangle \quad (2.32)$$

It can also be proved that the hermitian operator \hat{L} possesses a spectral representation in terms of the projectors $\hat{P}_i = |\ell_i\rangle\langle\ell_i|$:

$$\hat{L} = \sum_i \lambda_i \hat{P}_i \quad (2.33)$$

2.1.3 Postulates of quantum mechanics

In this Section we will provide the mathematical laws by which the following questions, by now considered just in an intuitive way, find an answer:

- How is described the state of a quantum system?
- Given this state, how can we predict the results of the measurement of a physical quantity?
- How does the state of the system evolve in time?

Such laws, called *postulates of quantum mechanics* will be summarised in the following, giving a definition for the *state space*, *observables*, *evolution* and *measurement* of a quantum system.

Postulates in the State Vector Formalism

Postulate 1 (States). *Each physical system \mathcal{S} is associated to an Hilbert space \mathcal{H} , and each state of a physical system is completely described by a normalised vector $|\psi\rangle \in \mathcal{H}$, i.e., $\| |\psi\rangle \| = 1$.*

Comment. A physical system is the part of nature we want to investigate and a state represents the most complete mathematical description of its properties.

The normalisation condition we impose to the state vectors is related, as we will see, to the peculiar statistical interpretation of the results of measurement processes in quantum mechanics.

Postulate 2 (Observables). *Each physical observable O is associated to an hermitian operator \hat{O} on the Hilbert space \mathcal{H} , and the only possible outcomes of a measurement of the observable are the (real) eigenvalues ω_i of the corresponding hermitian operator. If the state of the system is described by the vector $|\psi\rangle$, then the probability that a measurement of O gives the result ω_i is:*

$$\mathcal{P}_{|\psi\rangle}(O = \omega_i) = |\langle o_i | \psi \rangle|^2 \quad (2.34)$$

where $|o_i\rangle$ is the normalised eigenstate of \hat{O} with respect to the eigenvalue ω_i .

Comment. An observable is any property of a physical system that can be measured, such as position, momentum, energy, and spin. The measurement of an observable is a physical process yielding in output a real number, which represents the value possessed by that observable after the measurement has been performed. On the contrary of what happens in classical mechanics, the result of a measurement process of an observable in quantum mechanics is generally unpredictable, and we can only determine the probability of one of its possible outcomes. The only situation in which the prediction is certain is the one in which the physical system is described by an eigenstate of the observable, i.e.:

$$\mathcal{P}_{|\psi\rangle}(O = \omega_i) = 1 \quad \Leftrightarrow \quad |\psi\rangle \equiv |o_i\rangle \quad (2.35)$$

We note that in stating the Postulate we have limited ourselves, for simplicity, to considering a nondegenerate observable, where there is only one eigenvector associated to each of its eigenvalues and the probability in Eq. 2.34 can be rewritten as:

$$\mathcal{P}_{|\psi\rangle}(O = \omega_i) = |\langle o_i|\psi\rangle|^2 = \langle o_i|\psi\rangle^* \langle o_i|\psi\rangle = \langle \psi|o_i\rangle \langle o_i|\psi\rangle = \langle \psi|\hat{P}_{\omega_i}|\psi\rangle \quad (2.36)$$

where \hat{P}_{ω_i} is the projector operator onto the one-dimensional manifold spanned by $|o_i\rangle$. Hence, quantum mechanical probabilities of the measurement outcomes of an observable can always be expressed as mean values of projection operators.

Postulate 3 (Evolution equation). *The evolution of a closed quantum system is described by a unitary transformation, that is, if $|\psi(t_0)\rangle \in \mathcal{H}$ is the state of a physical system at time t_0 , then the state vector at any later time $t > t_0$ is:*

$$|\psi(t)\rangle = \hat{U} |\psi(t_0)\rangle \quad (2.37)$$

where \hat{U} is a unitary operator which depends on t_0 and t .

Comment. The quantum mechanical evolution of a state is linear, deterministic and reversible. It is linear because it has to preserve the linear structure of \mathcal{H} , so that linear combinations of states evolves to linear combinations of the evolved states:

$$\hat{U}(\alpha|\psi_1(t_0)\rangle + \beta|\psi_2(t_0)\rangle) = \alpha\hat{U}|\psi_1(t_0)\rangle + \beta\hat{U}|\psi_2(t_0)\rangle = \alpha|\psi_1(t)\rangle + \beta|\psi_2(t)\rangle \quad (2.38)$$

It is deterministic because, given the initial state $|\psi(t_0)\rangle$ and a unitary operator \hat{U} there exist a unique state vector $|\psi(t)\rangle$ of Eq. 2.37, which describes the system at each time $t > t_0$.

Finally, it is reversible because, given the state $|\psi(t)\rangle$, one can always retrieve the initial state $|\psi(t_0)\rangle$ before the evolution induced by \hat{U} occurred; in fact, $|\psi(t_0)\rangle = \hat{U}^{-1}|\psi(t)\rangle$.

Postulate 4 (State reduction). *Suppose that $|\psi(t)\rangle$ is the state vector associated to the physical system at a certain time t , when a measurement of the observable O is performed and the outcome ω_i is obtained. Then, the state vector immediately after the measurement occurred is the eigenstate $|o_i\rangle$ associated to the eigenvalue ω_i .*

$$|\psi(t)\rangle \xrightarrow{O=\omega_i} |o_i\rangle \quad (2.39)$$

Comment. The state reduction represents another kind of physical evolution besides the unitary evolution described in the Postulate 3, which takes place when a measurement of an observable is performed onto a physical system. Such an evolution is nonlinear, nondeterministic and irreversible.

It is nonlinear because a linear combination of state vectors is mapped into a state which is not, in general, the linear combination of the evolved initial states.

It is nondeterministic because the resulting state, after the measurement has taken place, could be any of the eigenstates of the observable. The only situation in which such evolution is deterministic is in the case in which the state of the system is already an eigenstate of the measured observable. In this case no modification of the state takes place.

Finally, it is irreversible since, through state reduction, all states non-orthogonal to $|o_i\rangle$ could have produced the resulting eigenstate $|o_i\rangle$.

The state reduction of Eq. 2.39 could be equivalently described by using the following mapping, using the projection operator $\hat{P}_{\omega_i} = |o_i\rangle\langle o_i|$:

$$|\psi(t)\rangle \xrightarrow{O=\omega_i} \frac{\hat{P}_{\omega_i}|\psi(t)\rangle}{\|\hat{P}_{\omega_i}|\psi(t)\rangle\|} \quad (2.40)$$

In fact:

$$\frac{\hat{P}_{\omega_i}|\psi(t)\rangle}{\|\hat{P}_{\omega_i}|\psi(t)\rangle\|} = \frac{|o_i\rangle\langle o_i|\psi(t)\rangle}{\sqrt{|\langle o_i|\psi(t)\rangle|^2}} = \frac{|o_i\rangle\langle o_i|\psi(t)\rangle}{|\langle o_i|\psi(t)\rangle|} = e^{i\alpha(t)}|o_i\rangle \quad (2.41)$$

which is completely equivalent to the state $|o_i\rangle$ given by Eq. 2.39, apart from the physically irrelevant phase factor $e^{i\alpha(t)}$.

Postulates in the Density Matrix Formalism

In this Section we will discuss another possible formalism, which is mathematically equivalent to the state vector approach and that is more convenient when dealing with quantum systems whose state is not completely known.

Let us consider a quantum system which, due to a classical lack of information, can be in one of m , not necessarily orthogonal, possible states $|\psi_1\rangle, \dots, |\psi_m\rangle$ with respective classical statistical weights p_1, \dots, p_m that quantify our ignorance about the real state of the system, which can be one out of m states. In this situation, the state of the quantum system is not described by a state vector but by an operator, known as density operator which is defined as follows:

$$\rho ::= \sum_{i=1}^m p_i |\psi_i\rangle\langle\psi_i| \quad (2.42)$$

where the $|\psi_i\rangle$ are normalised states, and $\sum_{i=1}^{\infty} p_i = 1$. ρ can be diagonalised, hence Eq. 2.42 becomes:

$$\rho ::= \sum_{i=1}^m r_i |r_i\rangle\langle r_i| \quad (2.43)$$

where the r_i and $|r_i\rangle$ are, respectively, the eigenvalues and eigenstates of ρ .

Given an orthonormal basis $\{|i\rangle\}$, the matrix representation of ρ , also known as density matrix, is given by:

$$\rho_{ij} = \langle i|\rho|j\rangle \quad (2.44)$$

A quantum system whose state is exactly known, because it is described by a unique state vector $|\psi\rangle$ is said to be a *pure state*. In this situation, the density operator of Eq. 2.42 reduces to:

$$\rho = |\psi\rangle\langle\psi| \quad (2.45)$$

and ρ turns out to be the projection operator associated to the one-dimensional manifold spanned by $|\psi\rangle$. Otherwise, when the density operator contains more than one state, the state is said to be a *mixed state*.

A density operator ρ , as defined by Eq. 2.42 can be proven to satisfy the following properties:

1. ρ is an hermitian, i.e., $\rho = \rho^\dagger$;
2. ρ has trace equal to 1, i.e., $\text{tr}(\rho) = \sum_{i=1}^n \rho_{ii} = 1$;
3. ρ is a positive operator, i.e., $\forall |\psi\rangle \in \mathcal{H} \quad \langle\psi|\rho|\psi\rangle \geq 0$.

Properties 2) and 3) completely characterise the class of density operators; in fact, we can, equivalently to Eq. 2.42, define a density operator to be a positive operator whose trace is equal to 1.

Now, having introduced the notion of density operator with the aim of describing systems whose state is not completely known, we can reformulate the postulates of quantum mechanics.

Postulate 1 (States). *Each physical system \mathcal{S} is associated to an Hilbert space \mathcal{H} , and each state of a physical system is completely described by a density operator.*

The set D of all the density operators over \mathcal{H} is a *convex* set, i.e., given $\rho_1, \rho_2 \in D$ it is possible to build another density operator $\rho_\lambda \in D$ by means of a convex sum $\rho_\lambda = \lambda\rho_1 + (1 - \lambda)\rho_2$. It can be proven that a pure state cannot be represented as a convex sum of density operators. Let us suppose that a pure state $\rho = |\psi\rangle\langle\psi|$ can be decomposed as a convex sum:

$$\rho = \lambda\rho_1 + (1 - \lambda)\rho_2$$

For each vector $|\psi_\perp\rangle$ such that $\langle\psi|\psi_\perp\rangle = 0$, we have that the convex sum can be lifted to the mean value of ρ as follows:

$$\langle\psi_\perp|\rho|\psi_\perp\rangle = \lambda\langle\psi_\perp|\rho_1|\psi_\perp\rangle + (1 - \lambda)\langle\psi_\perp|\rho_2|\psi_\perp\rangle \quad (2.46)$$

Since ρ is pure, $\langle\psi_\perp|\rho|\psi_\perp\rangle = 0$, but the right part of the equation $\lambda\langle\psi_\perp|\rho_1|\psi_\perp\rangle + (1 - \lambda)\langle\psi_\perp|\rho_2|\psi_\perp\rangle = 0$ if and only if $\rho = \rho_1 = \rho_2$.

Pure states are called *extremal elements* of the set D , since they belong to its boundary, and each element inside D can be decomposed (not uniquely) as convex sum of extremal points. In the centre of the set D over a n -dimensional Hilbert space, there is the totally mixed state $\rho_{mix} = \frac{1}{n}\mathbb{I}$.

Postulate 2 (Observables). *Quantum measurements are described by a collection $\{M_k\}$ of measurement operators satisfying the completeness equation:*

$$\sum_k M_k^\dagger M_k = \mathbb{I} \quad (2.47)$$

The index k refers to the possible measurement outcomes. If the state of the system is described by the density operator ρ , then the probability that the outcome associated to the index k occurs in a measurement is:

$$\mathcal{P}_k = \text{tr}(M_k^\dagger M_k \rho) \quad (2.48)$$

Comment. A collection of measurement operators satisfying the completeness equation 2.48 represents a generalisation of the measurement process which has been described in the Postulate 2 of the previous Section. When the measurement operators $\{M_k\}$ are the projection operators \hat{P}_{ω_k} onto the one-dimensional manifolds associated to the eigenvalues ω_k of an hermitian operator O , Eq. 2.48 takes a simpler form:

$$\text{tr}(M_k^\dagger M_k \rho) = \text{tr}(P_{\omega_k}^\dagger P_{\omega_k} \rho) = \text{tr}(P_{\omega_k} \rho) = \text{tr}(P_{\omega_k} \sum_i p_i |\psi_i\rangle\langle\psi_i|) = \sum_i p_i |\langle\omega_k|\psi_i\rangle|^2 \quad (2.49)$$

When the state is described by the density operator ρ , there are two kind of uncertainties related to a measurement outcome: a classical one, associated to the probabilities p_i and due to our epistemic ignorance about the state vector associated to the system, and a quantum mechanical one, related to the probabilities $|\langle\omega_k|\psi_i\rangle|^2$ which are intrinsic to the quantum description of nature.

Postulate 3 (Evolution). *The evolution of a closed quantum system is described by unitary transformation, that is, if $\rho(t_0)$ is the density operator associated to a physical system at time t_0 , then the density operator at any later time $t > t_0$ is:*

$$\rho(t) = U\rho(t_0)U^\dagger \quad (2.50)$$

where \hat{U} is a unitary operator which depends on t_0 and t .

Postulate 4 (State reduction). *Suppose that $\rho(t)$ is the density operator associated to the physical system at a certain time t , when a quantum measurement of a collection*

of measurement operators is performed, and the outcome associated to the index k is obtained. Then, the density operator immediately after the measurement occurred is

$$\frac{M_k \rho M_k^\dagger}{\text{tr}(M_k^\dagger M_k \rho)} \quad (2.51)$$

Superoperators

There is a general way for describing the evolution of a quantum system under various circumstances, including stochastic changes to quantum states. When dealing with *open* quantum systems—which can be informally be regarded as systems interacting with an external quantum system (i.e., the environment)—unitary transformations and measurements are not sufficient to describe the behaviour of the system, and we should allow a broader class of evolutions.

Therefore, the most general evolution operator mapping a quantum state, represented by the density operator $\rho(t_0)$ at time t_0 , to the evolved one $\rho(t)$ at time $t > t_0$ can be described as follows:

$$\rho(t) = \mathcal{E}(\rho(t_0)) \quad (2.52)$$

Since \mathcal{E} is an operator acting on operators, it is called a *superoperator*. In Eq. 2.50 we have already shown an example of superoperator, i.e., the unitary transformation $\mathcal{E}(\rho) = U\rho U^\dagger$.

A superoperator is a function \mathcal{E} mapping a density matrix $\rho(t_0)$ at time t_0 to a density matrix $\rho(t)$ at time $t_1 > t_0$ and it is defined by the following properties:

- (1) $0 \leq \text{tr}(\mathcal{E}(\rho)) \leq 1$ for any state ρ ;
- (2) $\mathcal{E}\left(\sum_i p_i \rho_i\right) = \sum_i p_i \mathcal{E}(\rho_i)$, where $\sum_i p_i = 1$;
- (3) \mathcal{E} is completely positive.

Property (1) tells us that a superoperator should always preserve the trace of the density operator in the case in which the evolution is deterministic, otherwise, when dealing with measurements, it should not increase it.

Property (2) expresses the preservation of the convex structure of the state space.

Property (3), i.e., complete positivity, states that if the ρ is a density operator, then its evolved $\mathcal{E}(\rho)$ must be a density operator as well, which is not in general guaranteed by \mathcal{E} being positive. Indeed, positivity turns out to be a sufficient condition only in the case in which we are considering a quantum system not coupled with any other ancillary system (or in the case in which ρ is separable³); otherwise, let us consider a Hilbert space \mathcal{H} coupled with an m -dimensional ancillary system (denoted by \mathcal{H}_A) such that the resulting composite space is described by $\mathcal{H} \otimes \mathcal{H}_A$. Let now $\rho^{TOT} = \rho \otimes \rho_A$ be a state of this composite system. We denote with \mathcal{I} the superoperator associated to the identity operator. If \mathcal{E} is a *positive* superoperator, the superoperator $\mathcal{E} \otimes \mathcal{I}^m$ acting

³The notion of separability will be explored in the following Sections.

on the state of the composite system may result in a nonpositive operator, while if \mathcal{E} is *completely positive*, the positivity of the operator $(\mathcal{E} \otimes \mathcal{I}^m)(\rho^{TOT})$ is guaranteed to hold, for all $m \geq 0$.

Example 2.1.1 (Partial Transpose). Given a basis set $|i\rangle$ over a n -dimensional Hilbert space \mathcal{H} , a generic state ρ on \mathcal{H} has the following matrix representation: $\rho_{ij} = \langle i|\rho|j\rangle$. We now consider the *transposition* superoperator T which acts as follows: $T(\rho) = \rho^T$, where ρ^T is the transpose of ρ and has the following matrix representation: $\rho_{ji} = \langle j|\rho|i\rangle$. ρ^T is still a density operator (since the total transpose does not change the spectrum). Hence, the transposition superoperator T is *positive*, since it maps positive operators into positive operators.

$$\rho \geq 0 \longrightarrow \rho^T \geq 0$$

Let us now introduce a m -dimensional ancillary system \mathcal{H}_A , with orthonormal basis $|a\rangle$, which is coupled with the previously considered system such that $\mathcal{H} \otimes \mathcal{H}_A$. The matrix representation of an arbitrary state over the composite system is hence $\rho^{TOT} \rightarrow \rho_{ia,jb} = \langle i, a|\rho|j, b\rangle$. If we lift the operator T to $T \otimes \mathcal{I}^m$, then we obtain a partial transposition (on the first subsystem):

$$(T \otimes \mathcal{I}^m)\rho^{TOT} = \rho^{TOT_1^T} \rightarrow \langle j, a|\rho^{TOT}|i, b\rangle$$

which is not guaranteed to be positive.

As an example, let us now take $\mathcal{H} = \mathbb{C}^2$ and $\mathcal{H}_A = \mathbb{C}^2$, and a generic state $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ over the composite space $\mathcal{H} \otimes \mathcal{H}_A$, where $|00\rangle$ and $|11\rangle$ represent the tensor product of two basis vectors: ⁴ in both of them the first element belongs to \mathcal{H} and the second one belongs to the ancillary system \mathcal{H}_A .

The density operator associated to $|\psi\rangle$ is $\rho = |\psi\rangle\langle\psi|$ which can be rewritten as:

$$\rho = \frac{1}{2} \left(|00\rangle\langle 00| + |00\rangle\langle 11| + |11\rangle\langle 00| + |11\rangle\langle 11| \right)$$

whose matrix representation is given explicitly as follows:

$$\rho_{ia,jb} = \frac{1}{2} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

The application of the partial transpose map results in the following operator:

$$(T \otimes \mathcal{I})\rho = \rho^{T_1} = \frac{1}{2} \left(|00\rangle\langle 00| + |10\rangle\langle 01| + |01\rangle\langle 10| + |11\rangle\langle 11| \right)$$

⁴We will provide a more detailed explanation about the notation in Section 2.2.1.

whose matrix representation is given explicitly as follows:

$$\rho_{ja,ib} = \frac{1}{2} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since ρ is a density operator, its spectrum is nonnegative, while the spectrum of ρ^{T_1} has at least one negative eigenvalue (e.g., in this case it has three eigenvalues equal to $1/2$ and one equal to $-1/2$), hence ρ^{T_1} is not a density operator.

There is an alternative form to represent superoperators, known as *operator-sum representation* which, given a density operator ρ and a superoperator \mathcal{E} , is described as follows:

$$\mathcal{E}(\rho) = \sum_i E_i \rho E_i^\dagger \quad (2.53)$$

where $\{E_k\}$ are known as *operation elements* and they must satisfy the following *completeness relation*, arising from the property (1) of superoperators.

$$\sum_i E_i^\dagger E_i \leq \mathbb{I} \quad (2.54)$$

It can be proven that map \mathcal{E} is a superoperator, i.e., \mathcal{E} satisfies properties (1),(2) and (3), if and only if it can be written using the operator-sum representation as in Eq. 2.53.

In the following the will summarise some kinds of superoperators that will be used in this thesis:

Unitary Superoperators: Given a unitary operator U , a state $\rho(t_0)$ at time t_0 and its evolved $\rho(t)$ at time $t > t_0$; the corresponding superoperator \mathcal{U} can be written as follows:

$$\mathcal{U}(\rho(t)) = U\rho(t_0)U^\dagger \quad (2.55)$$

Measurement Superoperators: As seen in Postulate 2, quantum *measurement* is described by a collection $\{M_i\}$ of measurement operators. Given a state ρ , the superoperator \mathcal{M} associated to a measurement operator can be written as follows:

$$\mathcal{M}(\rho) = \sum_i \mathcal{M}_i = \sum_i M_i \rho M_i^\dagger \quad (2.56)$$

Positive operator valued measure

Different kind of measurement exist in quantum mechanics; we have described projective measurement and a larger class of measurement operators, giving the description of the state of the system after the measurement occurred.

The description of the post-measurement state of the system can sometimes be ignored, since we might be most interested in the probabilities of the measurement outcomes. In this case a mathematical tool, i.e., the positive operator valued measurement (a.k.a., POVM), is more suitable to describe it. In the following we will neglect the discussion about the most general kind of POVM since it considers a continuous family of measurement outcomes and we will only focus on a finite number of possible outcomes, over a finite-dimensional space. Before we define such objects, we introduce the concept of *measure* in the sense of mathematical probability and measure theory; a measure is an *additive set function*, i.e., a function $E(\cdot)$ whose argument is a set Ω (rather than a number, or a point in space) and for any two disjoint subsets of Ω A_i and A_j , with $i \neq j$, the following holds:

$$E\left(\bigcup_i A_i\right) = \sum_i E(A_i) \quad (2.57)$$

In particular, if E is a probability measure, given $A \subset \Omega$:

$$0 \leq E(A) \leq 1 \quad (2.58)$$

with $E(\Omega) = 1$.

A POVM is a measure $\hat{E}(\Omega)$; it is a positive operator over a Hilbert space \mathcal{H} , hermitian and normalised in the sense that:

$$\hat{E}(\Omega) = \mathbb{I} \quad (2.59)$$

where \mathbb{I} is the identity operator. We can informally regard at POVMs as maps from the state-space to the space of solutions. POVMs are a special case of the measurement formalism, which provide a way to study the measurement statistics, ignoring the description of the post-measurement state.

2.2 Quantum information theory

Quantum information theory investigates all the possible ways to transmit and manipulate information by using quantum mechanics. Abbreviated in QIT, it includes quantum computation which is a theoretical field of research that aims to use the laws of quantum theory to perform computational tasks.

2.2.1 Quantum computation

This Section is devoted to the summarisation of the main aspects of quantum computation. It is based on both quantum physics and classical computer science, and its goal is to use the laws of quantum mechanics to develop powerful algorithms and protocols [61]. As classical computation could be described by using different models such as circuits, Turing machines, Random access machine and Lambda calculus, also quantum computation, and the broader field of quantum information, could be described by different models including the quantum circuit (or network) model, adiabatic quantum

computation, the quantum Turing machines and measurement-based models such as teleportation-based approaches as well as the one-way quantum computer [20]. Since the features of these models differ significantly, some computational schemes may lend themselves more than others to understand certain aspects of quantum computation and to overcome challenges in their experimental realisation. One of the most used paradigm for quantum computation is the quantum circuit model which is a universal language developed to describe quantum computations [64].

2.2.2 Quantum circuit model

Classical circuits

A (deterministic) classical computer evaluates a function f : given n -bits of input it produces m -bits of output that are uniquely determined by the input; that is, it finds the value of the function

$$f : \{0, 1\}_n \longrightarrow \{0, 1\}_m \quad (2.60)$$

for a particular specified n -bit argument x . A function with an m -bit output is equivalent to m functions, each with a one-bit output, so we may just as well say that the basic task performed by a computer is the evaluation of

$$f : \{0, 1\}_n \longrightarrow \{0, 1\} \quad (2.61)$$

A function that takes an n -bit input to a one-bit output is called *Boolean*. We may think of such a function as a binary string of length 2^n , where each bit of the string is the output $f(x)$ for one of the 2^n possible values of the input x [76]. In this way we deal with 2^{2^n} bit strings. Boolean functions are usually implemented by means of logic gates. A partial list of such gates includes the AND, OR, NOT, NAND ones, but there are many others useful to build circuits in order to perform computations. The basic operations of these gates could be described with the aid of truth tables. In the following table we will see some examples for the AND, OR and NAND gates.

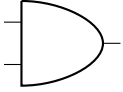
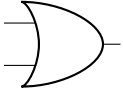
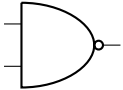
Each one of these gates inputs two bits and produces a single bit as output, hence, with the exception of the NOT gate which takes one bit in input (and produces in output the negation of the input bit), they are not invertible. All these gates could be simulated by reduced sets of other gates, known as *universal gates*, e.g. the NAND and NOR gates among others, by means of which the behaviour of all the other logic gates can be re-created.

Quantum bits

In devising a model of a quantum computer we will generalize the circuit model of classical computation. In the classical theory of computation the simplest unit of information is the *bit*, a two-state system encoding information by taking value 0 or 1. In quantum computation the most fundamental system is the quantum bit, or *qubit*.

Let us consider the Hilbert space \mathbb{C}^2 , and we denote as:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad (2.62)$$

AND		$A \wedge B = \mathcal{O}$	<table style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">A</th> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">B</th> <th style="border-bottom: 1px solid black;">\mathcal{O}</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	\mathcal{O}	0	0	0	0	1	0	1	0	0	1	1	1
A	B	\mathcal{O}																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$A \vee B = \mathcal{O}$	<table style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">A</th> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">B</th> <th style="border-bottom: 1px solid black;">\mathcal{O}</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	\mathcal{O}	0	0	0	0	1	1	1	0	1	1	1	1
A	B	\mathcal{O}																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NAND		$\overline{A \wedge B} = \mathcal{O}$	<table style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">A</th> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">B</th> <th style="border-bottom: 1px solid black;">\mathcal{O}</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	\mathcal{O}	0	0	1	0	1	1	1	0	1	1	1	0
A	B	\mathcal{O}																
0	0	1																
0	1	1																
1	0	1																
1	1	0																

an orthonormal basis set of vectors for it, i.e., $\langle 0|1\rangle = 0$ and $\| |0\rangle\| = \| |1\rangle\| = 1$.

Such states, which constitute the so-called *standard computational basis* for \mathbb{C}^2 represent the quantum analogue of the classical bits 0 and 1. They are the eigenstates with respect to the eigenvalues $+1$ and -1 respectively, of the hermitian operator σ_z :

$$\sigma_z|0\rangle = +1|0\rangle \quad \sigma_z|1\rangle = -1|1\rangle \tag{2.63}$$

A qubit is an arbitrary unit state vector $|\psi\rangle$ of the two dimensional Hilbert space \mathbb{C}^2 :

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.64}$$

where the complex coefficients $\alpha, \beta \in \mathbb{C}$ are constrained to satisfy the normalisation condition:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.65}$$

They are related to the probabilities of obtaining the outcomes of σ_z . More precisely, $|\alpha|^2$ is the probability of obtaining outcome $+1$, after a measurement of σ_z , when the system is described by the state $|\psi\rangle$ of Eq. 2.64. Equivalently, $|\beta|^2$ is the probability of obtaining outcome -1 , after a measurement of σ_z .

There are three remarkable differences between quantum and classical bits.

The first one follows directly from the definition of qubit; indeed, while a classical bit can take one value at a time, i.e., it can be 0 or 1 only, a qubit can exist in a linear superposition of $|0\rangle$ and $|1\rangle$. The physical meaning of such superposition of states is that we can acquire knowledge about the value of a qubit only by measuring the observable σ_z , which will yield the outcome associated to the bit 0 (eigenvalue $+1$ of

σ_z) or 1 (eigenvalue -1), with a specified probability, only after the measurement has been performed.

The second difference is that, classical bits can be observed and accessed without disturbing them as many times as we wish, while the information content of a qubit is destroyed by the state reduction evolution after the measurement. A qubit is not disturbed by a measurement of σ_z only when either α or β is equal to 0.

Finally, the third difference is that while a classical bit can be copied, in general an arbitrary qubit cannot be cloned.

The quantum counterpart of the classical strings of bits are the quantum registers, i.e., *multiple qubits*. They are vectors belonging to the Hilbert space $\mathbb{C}^2 \otimes \dots \otimes \mathbb{C}^2$, which is isomorphic to the 2^n -dimensional Hilbert space \mathbb{C}^{2^n} . A generic multiple qubit state is

$$|\psi_1\rangle|\psi_2\rangle \dots |\psi_n\rangle ::= |\psi_1\rangle \otimes |\psi_2\rangle \otimes \dots \otimes |\psi_n\rangle \quad (2.66)$$

where $|\psi_i\rangle$ with $i = 1, \dots, n$ is an arbitrary qubit.

The Hilbert space associated is the tensor product of n single qubit spaces, thus $\mathcal{H} = \mathbb{C}^2 \otimes \dots \otimes \mathbb{C}^2 = \mathbb{C}^{\otimes 2^n}$, with 2^n basis states and complex amplitudes. Even though this might seem a natural mathematical description, it has an enormous potential in term of computational power since, even for small n a quantum computer might operate in parallel on 2^n states at the same time. The drawback is that it also has to *store* the same number of complex amplitudes, which is severely limited by the available classical technologies.

Quantum Circuits

Quantum computation uses qubits as basic units of information and performs state changes on them. A quantum computer is abstracted by a quantum circuit, similar to a classical one, with the quantum counterpart of the classical logic gates manipulating the qubits. In the following there are summarised some of the main quantum gates that will be used throughout the thesis. The first thing that we should understand is how to use classical functions in the quantum circuit model; we should model them into *oracles*. The quantum logic gates will be *unitary transformations*, and hence will be invertible making the computation reversible.

A reversible computer evaluates an invertible function taking n bits to n bits

$$f : \{0, 1\}_n \longrightarrow \{0, 1\}_n \quad (2.67)$$

[76] and has a unique input for each output so we can run the computation backwards to recover the input from the output. An invertible function is a permutation of the 2^n n -bit strings and the important fact that permits us to move from a classical (irreversible) computation to a quantum (reversible) one is that [76] any irreversible computation can be packaged as an evaluation of an invertible function. As an example, for any:

$$f : \{0, 1\}_n \longrightarrow \{0, 1\} \quad (2.68)$$

we can build a function such that:

$$\hat{f} : \{0, 1\}_{n+1} \longrightarrow \{0, 1\}_{n+1} \quad (2.69)$$

such that

$$\hat{f}(x, y) = (x, y \oplus f(x)) \quad (2.70)$$

In general, quantum gates are represented by square matrices, i.e., the matrix representation of the unitary operators corresponding to the state transformation. In the quantum circuit model qubits are represented by wires, as well as in the classical circuit case. In a quantum computation, a finite number n of qubits (a quantum register) are initialised, then a finite number of unitary gates is applied to one, or more, of them. Finally, a measurement on all the qubits (or on a subset of them) is performed. The most common and the most used single-qubit gates are the *Pauli* operators σ_x, σ_y and σ_z , the Hadamard gate H , the phase shift gate S and the T gate (also known as $\pi/8$ gate), whose matrix representation in the orthonormal basis set $\{|0\rangle, |1\rangle\}$ is given as follows.

$$\begin{aligned} \sigma_0 &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \sigma_x &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ \sigma_y &= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} & \sigma_z &= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \end{aligned} \quad (2.71)$$

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix} \quad (2.72)$$

Gates are usually represented in the circuit by a square box labelled with the letter associated to the gate. In Fig. 2.1 we can see an example for the Hadamard gate.

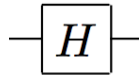


Figure 2.1: Circuit representation of the Hadamard gate

The effect of the gates in Eq. 2.71 and 2.72 applied to the computational basis vectors $\{|0\rangle, |1\rangle\}$ are summarized as follows:

$$\begin{aligned}
\sigma_x|0\rangle &= |1\rangle & \sigma_x|1\rangle &= |0\rangle \\
\sigma_y|0\rangle &= i|1\rangle & \sigma_y|1\rangle &= -i|0\rangle \\
\sigma_z|0\rangle &= |0\rangle & \sigma_z|1\rangle &= -|1\rangle \\
H|0\rangle &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} & H|1\rangle &= \frac{|0\rangle - |1\rangle}{\sqrt{2}}
\end{aligned} \tag{2.73}$$

Single qubit gates can be composed and applied to more than a qubit by taking the tensor product of the gates. If for example, during a computation we want to apply the unitary gate U to a state of two qubits we have to take the product $U \otimes U$. This can be generalised to a state of n qubits as follows:

$$U^{\otimes n} \tag{2.74}$$

Controlled operations The most basic two qubits quantum transformations are the controlled ones. By acting on two qubits at the same time, these gates give rise to interesting physical phenomena which will be investigated later in the Section.

The prototypical controlled-operation is the *controlled-not* gate (also referred as CNOT). Given in input two computational basis vectors $|\psi\rangle$ and $|\varphi\rangle$, which are known as the control and the target qubit respectively, the controlled-not gate acts as in Eq.(2.75).

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} \sigma_0 & 0 \\ 0 & \sigma_x \end{pmatrix} \tag{2.75}$$

$$CN(|\psi\rangle|\varphi\rangle) = |\psi\rangle|\varphi \oplus \psi\rangle \tag{2.76}$$

where \oplus represents the addition modulo 2, making the controlled-not gate the quantum counterpart of the classical *XOR* gate. The action can be summarized as follows: if the control qubit is set to 1, the target qubit is flipped, otherwise the state remains the same as before. A representation of this gate can be seen in Fig. 2.2.

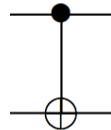


Figure 2.2: Circuit Representation of the Controlled-not Gate.

Any arbitrary unitary operator U can be controlled. A controlled- U operation is applied on two (or more) qubits with some of them acting as controls and (one or more) other as targets. If all the controls are set to $|0\rangle$, then U is applied to the target. This can be represented as in Fig. 2.3, where we focused on a two qubit case:

Following from the definition of quantum logic gate, we might ask whether there is a

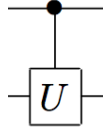


Figure 2.3: Circuit Representation of a Generic Controlled Gate U .

quantum counterpart of the classical notion of *universal* set of gates, i.e., a set of unitary operators acting on a set of qubits which can mimic the behaviour of any other quantum gate. In our case it is sufficient to state that the set composed by the controlled not gate, together with a generic single-qubit gate is sufficient to build any other unitary transformation on an n -qubits register.

2.2.3 Quantum information

Quantum information is a discipline which investigates all the possible ways to transmit and manipulate information by using quantum mechanics. Like classical information theory, it is really specialised in topics involving the field of quantum information processing tasks, rather than quantum algorithm design; thus it is interested in the most fundamental and physical description of what quantum information is and by which laws it is governed. According to Refs.([64, 76]), quantum information is mainly concerned with the following topics:

1. *Quantum channels* and transmission of quantum and classical information along them, with applications spanning from teleportation to quantum cryptography;
2. Identification of the static and dynamical *properties* of physical systems, which can be viewed as resources for outperforming classes of tasks;
3. Quantification and classification of *entanglement*.

All the mathematical details of the aforementioned tasks are not to be discussed in this thesis, but we will define the basic concepts that are used in following Chapters.

Definition 2.2.1 (Quantum channel). A quantum channel is a communication channel allowing to transmit quantum and classical information. It is realised by a completely positive trace-nonincreasing map, i.e., a superoperator.

In the realm of information theory one of the most important concepts is that of *entropy*, which is used to quantify the quantum and classical information content. In QIT, the quantum counterpart of the classical Shannon entropy is the von Neumann entropy, defined as follows:

Definition 2.2.2 (von Neumann entropy). The von Neumann entropy $S(\rho)$ of a given density matrix ρ is defined as follows:

$$S(\rho) = -\text{tr}(\rho \log \rho) = -\sum_i r_i \log r_i$$

where r_i are the (strictly) positive eigenvalues of ρ as in Eq. 2.43.

As a consequence, if ρ is a pure state all its eigenvalues are 0 but one, thus $S(\rho) = 0$; otherwise, in the case of a generic mixed state, the von Neumann entropy is larger than 0 and corresponds to the Shannon entropy of its spectrum.

Some of the main properties of $S(\rho)$ are listed in the following, while for a more in-depth description we address to [64, 76, 10]:

1. If $\rho = \sum_i \lambda_i |\psi_i\rangle\langle\psi_i|$ with non-orthogonal $|\psi_i\rangle$ (i.e. $\langle\psi_i|\psi_j\rangle \neq \delta_{ij}$) then it holds $S(\sum_i \lambda_i \rho_i) \geq \sum_i \lambda_i S(\rho_i)$ for all $\lambda_i \geq 0$ such that $\sum_i \lambda_i = 1$. Thus, a larger entropy S quantifies a larger ignorance about the state of the system.
2. Given a state ρ with n non-vanishing eigenvalues, it is always true that $S(\rho)$ is bounded as follows:

$$S(\rho) \leq \log n \quad (2.77)$$

with equality holding when the eigenvalues are equal. Thus a maximally mixed state has the maximum value of S . A consequence of this is that we can use entropy to discriminate whether a state is pure or not, since a pure state ρ_{pure} has vanishing entropy $S(\rho_{pure}) = 0$.

3. Given a state in a composite system $\rho \in \mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2$, in general the following equation holds:

$$S(\rho) \leq S(\rho_1) + S(\rho_2) \quad (2.78)$$

where ρ_1 and ρ_2 are the reduced density operators, obtained by using the *partial trace*, defined in Eq.(2.79).

Given a generic state of a bipartite system $\rho_{1,2} \in \mathcal{H}_1 \otimes \mathcal{H}_2$, it is possible to introduce the notion of **reduced density operators** ρ_1 and ρ_2 , which are obtained by tracing away the subsystem we are not interest in with the **partial trace** operation:

$$\rho_i = \text{tr}_j(\rho_{ij}) = \sum_n \langle\psi_n^{(j)}|\rho_{ij}|\psi_n^{(j)}\rangle$$

with $\{|\psi_n^{(j)}\rangle\} \in \mathcal{H}^j$. Hence:

$$\begin{aligned} \rho_1 &= \text{tr}_2(\rho_{1,2}) \\ \rho_2 &= \text{tr}_1(\rho_{1,2}) \end{aligned} \quad (2.79)$$

Equality in Eq.2.78 holds if $\rho = \rho_1 \otimes \rho_2$ is a product state, i.e., when the two subsystems are uncorrelated. We will explore the meaning of product state and correlation in the next paragraph.

Other properties for the von-Neumann entropy can be found in the literature; in this Section we will restrict to the ones which are more relevant for the work presented in this thesis.

2.2.4 Entanglement in Bipartite Systems

In this Section we introduce the concept of *entanglement*, which is related to the existence of quantum states belonging to composite quantum systems that cannot be represented in terms of product states. A composite system \mathcal{H} is represented by the tensor product of n component subsystems:

$$\mathcal{H} = \mathcal{H}_1 \otimes \cdots \otimes \mathcal{H}_n$$

The simplest composite system is the *bipartite* one, whose Hilbert space is the tensor product $\mathcal{H} = \mathbb{C}^n \otimes \mathbb{C}^m$.

Pure States: A bipartite pure state $|\psi\rangle \in \mathcal{H}$, is said to be *non-entangled* (or separable) if it can be expressed as a product state, i.e., it is always possible to factorise it as follows:

$$|\psi\rangle = |\xi\rangle \otimes |\chi\rangle \tag{2.80}$$

with $|\xi\rangle \in \mathbb{C}^n$ and $|\chi\rangle \in \mathbb{C}^m$. Let us fix a basis $\{|\psi_{1i}\rangle\}$ for \mathbb{C}^n and $\{|\psi_{2j}\rangle\}$ for \mathbb{C}^m , the general form of a bipartite pure state is $|\psi\rangle = \sum_{i,j} a_{ij} |\psi_{1i}\rangle \otimes |\psi_{2j}\rangle$, for which it is not immediate to ascertain whether it can be factorised in the sense of Eq. (2.80), or not. If such product form does not exist, then the state is said to be entangled.

Example 2.2.1. Let us consider, as an example, the following states in the standard basis $\{|0\rangle, |1\rangle\}$:

$$\begin{aligned} |\psi\rangle &= \frac{1}{\sqrt{2}}(|01\rangle + |00\rangle) \\ |\Phi^+\rangle &= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \end{aligned}$$

The first one can be factorized as $|\psi\rangle = |0\rangle \otimes \frac{1}{\sqrt{2}}(|1\rangle + |0\rangle)$. On the contrary, the second one is a genuine entangled state, since it does not exist complex coefficients $\alpha, \beta, \gamma, \delta$ such that:

$$\begin{aligned} |\Phi^+\rangle &= \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = (\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle) \\ &= \alpha\gamma|00\rangle + \beta\delta|11\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle \end{aligned}$$

In fact, in order to satisfy the previous equality $\alpha\gamma = \beta\delta = \frac{1}{\sqrt{2}}$ and $\alpha\delta = \beta\gamma = 0$ which are conditions which cannot be fulfilled simultaneously. Thus, $|\Phi^+\rangle$ cannot be factorized, i.e., it is an entangled state.

Separability for Pure States: For pure states in bipartite quantum systems there are many methods to assess if a state $|\psi\rangle$ is entangled or not. One of them is given by using the *Schmidt decomposition* of pure states.

Lemma 2.2.1 (Schmidt decomposition). Given a state $|\psi\rangle \in \mathcal{H} = \mathbb{C}^n \otimes \mathbb{C}^m$, it is always possible to find orthonormal states $|\psi_i\rangle \in \mathbb{C}^n$ and $|\varphi_j\rangle \in \mathbb{C}^m$ yielding the following

decomposition:

$$|\psi\rangle = \sum_{k=1}^{\min(n,m)} \sqrt{\lambda_k} |\psi_k\rangle \otimes |\varphi_k\rangle \quad (2.81)$$

where $\lambda_k \geq 0$ and $\sum_k \lambda_k = 1$.

The number of nonzero values $\sqrt{\lambda_k}$ is called the *Schmidt number* for the state $|\psi\rangle$. It can be used to determine whether an arbitrary, bipartite pure state is entangled or not; in fact, it can be proven that a state $|\psi\rangle$ is non-entangled if and only if its Schmidt number is equal to 1. In this case, this amounts to say that the Schmidt decomposition of the state $|\psi\rangle$ reduces to a single tensor product of states.

Another method to detect entanglement of pure states uses the von-Neumann entropy. Given a state $|\psi\rangle \in \mathbb{C}^n \times \mathbb{C}^m$, and its corresponding density operator $\rho = |\psi\rangle\langle\psi|$ it can be proved that ρ is separable if and only if both its entropy $S(\rho)$ and the entropy of its reduced density operators $S(\rho_1)$ and $S(\rho_2)$ is 0.

$$\rho \text{ separable} \Leftrightarrow S(\rho) = S(\rho_1) = S(\rho_2) = 0 \quad (2.82)$$

Instances of entangled pure states that are worth mentioning are the *Bell states*, whose description in the standard computational basis $\{|0\rangle, |1\rangle\}$ is given as follows:

$$\begin{aligned} |\Phi^+\rangle &= \frac{1}{\sqrt{2}}(|0\rangle|0\rangle + |1\rangle|1\rangle) \\ |\Phi^-\rangle &= \frac{1}{\sqrt{2}}(|0\rangle|0\rangle - |1\rangle|1\rangle) \\ |\Psi^+\rangle &= \frac{1}{\sqrt{2}}(|0\rangle|1\rangle + |1\rangle|0\rangle) \\ |\Psi^-\rangle &= \frac{1}{\sqrt{2}}(|0\rangle|1\rangle - |1\rangle|0\rangle) \end{aligned} \quad (2.83)$$

Mixed States: A density operator ρ in a bipartite system is called separable if and only if it can be written as convex sum of pure states, as follows:

$$\rho = \sum_{i,j} p_{ij} \rho_{1i} \otimes \rho_{2j}$$

which, through spectralisation, can be reduced to a convex combination of projectors:

$$= \sum_{k,l} \tilde{p}_{k\ell} |\xi_k\rangle\langle\xi_k| \otimes |\chi_\ell\rangle\langle\chi_\ell| \quad (2.84)$$

where $p_{ij} \geq 0$ and $\sum_{i,j} p_{ij} = 1$. States that cannot be written in such form as Eq. 2.84 are called non-separable, or equivalently, *entangled*.

Separability for Mixed States: Checking for separability in the case of a mixed state turns out to be much more difficult than for pure case. In fact, at present do not exist necessary and sufficient conditions to determine whether an arbitrary mixed state

ρ acting on $\mathbb{C}^n \otimes \mathbb{C}^m$, with arbitrary values of n and m , is separable or not. On the contrary, such conditions exist in the case of a two qubit mixed state, i.e., when the Hilbert space is $\mathbb{C}^2 \otimes \mathbb{C}^2$. One of such conditions exploits the *concurrence* $\mathcal{C}(\rho)$ which is defined as follows.

$$\mathcal{C}(\rho) ::= \max(0, \lambda_1 - \lambda_2 - \lambda_3 - \lambda_4) \quad (2.85)$$

where $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq \lambda_4$ are the eigenvalues of the hermitian operator R , defined in terms of the mixed state ρ :

$$R = \sqrt{\sqrt{\rho} \tilde{\rho} \sqrt{\rho}} \quad (2.86)$$

with $\tilde{\rho} = (\sigma_y \otimes \sigma_y) \rho^* (\sigma_y \otimes \sigma_y)$. It can be proven that concurrence $\mathcal{C}(\rho)$ for an arbitrary two-qubits mixed state ρ is strictly positive if and only if ρ is non-separable. Consequently, $\mathcal{C}(\rho)$ is null if and only if ρ is separable.

2.3 Quantum algorithms and protocols

In this Section we will summarise the main quantum algorithms and protocols that will be used in the following. We have already introduced the main ingredients of quantum computation and the concept of quantum circuit, which is the standard notation to model quantum computations by means of qubits and quantum logic gates. Now, we can define the concept of quantum algorithm, which is a sequence of unitary operators applied to a number n of quantum registers, i.e., arrays of qubits. The output will be produced by a measurement process onto the states of the standard computational basis; for this reason, quantum algorithms are probabilistic and the main purpose becomes the extraction of the desired result from a statistical distribution of possible outcomes.

The existing quantum algorithms are much more efficient than their classical counterpart, due to the following features of quantum mechanics:

- **Parallelism:** due to the linearity of both state space and operators, it is possible to compute all the possible values of a given function f using a single application of the corresponding unitary operator U_f :

$$U_f \left(\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle |0\rangle \right) = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle |f(x)\rangle \quad (2.87)$$

where the final state encodes all the possible values of the function $f(x)$.

- **Interference:** it is possible, after a suitable preparation of the initial state, to make the states *interfere* to delete the “wrong” ones, while increasing the probability that the states carrying the desired result are obtained when a measurement is performed.
- **Correlations:** the final states of a quantum computation can be entangled, thus they can exhibit non-local correlations between the outcomes of measurements

performed on different quantum registers.

2.3.1 Deutsch's algorithm

The Deutsch's algorithm is the simplest quantum algorithm, and it is used to highlight the aforementioned features of quantum computation. It tackles the following problem: given a black box U_f , also known as *oracle*, implementing an unknown one-bit Boolean function $f : \{0, 1\} \rightarrow \{0, 1\}$, determine whether f is *constant* or *balanced* by querying the oracle. A function is constant if it always produces the same bit in output, i.e., $f(0) = f(1)$, and it is balanced if, on different inputs, it provides different outputs, i.e., $f(0) \neq f(1)$. There are four possible one-bit Boolean functions, two of them are constant and the other two are balanced, as we will see in the Table 2.1 below.

Constant 1 Zero	Constant 2 One	Balanced 1 Not	Balanced 2 Identity
$f(0) = 0$	$f(0) = 1$	$f(0) = 1$	$f(0) = 0$
$f(1) = 0$	$f(1) = 1$	$f(1) = 0$	$f(1) = 1$

Table 2.1: One-bit Boolean Functions.

The problem could classically be solved by querying the oracle twice, and then evaluating $f(0) \oplus f(1)$, where \oplus represents the addition modulo 2. If the function is constant, then $f(0) \oplus f(1) = 0$, while if f is balanced, then $f(0) \oplus f(1) = 1$. On the contrary, we will show that a single query of the quantum oracle U_f is sufficient in order to determine with certainty whether the function is constant or balanced.

Since the function f is not always invertible, in order to implement it by a quantum unitary operator, we should use a method to make f invertible. The quantum oracle U_f , which computes the values of f should use an ancillary qubit. In this way, U_f acts on a four-dimensional space $\mathbb{C}^2 \otimes \mathbb{C}^2$ as follows:

$$U_f|x\rangle|y\rangle \equiv |x\rangle|y \oplus f(x)\rangle \quad (2.88)$$

for all $x, y \in \{0, 1\}$.

The quantum circuit implementing the Deutsch's algorithm is the one in Fig. 2.4, and it shows the compositions, in successive steps of the quantum gates used to solve the problem.

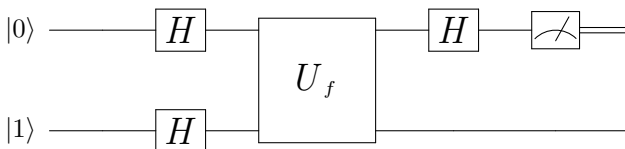


Figure 2.4: Quantum Circuit for the Deutsch's Algorithm.

In the following we summarise the computational steps of the algorithm:

1. preparation of the initial state: $|\psi_0\rangle = |0\rangle|1\rangle$;
2. application of the Hadamard gate on both registers; this gate puts the state in a linear superposition of the four basis states:

$$|\psi_0\rangle \xrightarrow{H \otimes H} |\psi_1\rangle = \frac{1}{\sqrt{2}}((|0\rangle + |1\rangle)(|0\rangle - |1\rangle)) \quad (2.89)$$

3. application of the unitary operator U_f which, exploiting the quantum parallelism, computes at the same time all the values of $f(x)$:

$$|\psi_1\rangle \xrightarrow{U_f(x)} |\psi_2\rangle = \sum_{x=0}^1 (-1)^{f(x)} |x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad (2.90)$$

4. application of the Hadamard gate on the first register (we recall that the function $f(x)$ is guaranteed to be one of the functions from Table 2.1):

$$\begin{aligned} |\psi_2\rangle \xrightarrow{H \otimes \mathbb{I}} |\psi_3\rangle &= \left[\frac{(-1)^{f(0)} + (-1)^{f(1)}}{2} |0\rangle + \frac{(-1)^{f(0)} + (-1)^{f(1)}}{2} |1\rangle \right] \\ &= \pm |f(0) \oplus f(1)\rangle \frac{1}{2\sqrt{2}}(|0\rangle - |1\rangle) \end{aligned} \quad (2.91)$$

In this way we use the interference between quantum states in order to determine with certainty whether the function is constant or balanced.

5. A measurement of σ_z is performed onto the first register: f is constant if and only if the eigenvalue corresponding to the eigenstate $|0\rangle$ is obtained; as a consequence, if the eigenvalue corresponding to the eigenstate $|1\rangle$ is obtained, then the function is balanced.

Hence, Deutsch's quantum algorithm has been able to determine with certainty, by querying the oracle only once, whether the function f is constant or balanced. This turns out to be a more efficient method, compared to its classical counterpart which needs two queries to determine the same result.

2.3.2 Deutsch-Jozsa algorithm

In the previous paragraph we presented the Deutsch's algorithm, which shows an example of how quantum algorithms can give some advantages over classical ones by using quantum effects such as parallelism and interference. In this part we will present the Deutsch-Jozsa algorithm, which generalizes the previous one to n -bits Boolean functions.

The problem is the following: suppose we are given a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$, which is promised to be one of two kinds: either $f(x)$ is *constant*, for all values of $x \in \{0, 1\}^n$ or else $f(x)$ is *balanced*, i.e., equal to 1 for exactly half of the possible inputs x , and 0 for the other half. We want to determine whether f is constant or balanced using the minimum number of queries to an oracle which computes the values of such function.

Classically this problem requires at least $2^{n-1} + 1$ queries of the oracle to determine with certainty whether the function is constant or balanced; in fact, in the worst case, we might have observed 2^{n-1} times the same bit, and we need one more value, which can be either equal or different from the previous ones, to assess the difference between the two kind of functions.

On the contrary, we will show that a single query to a quantum oracle U_f is sufficient in order to determine with certainty that the function is constant or balanced. As before, in order to make the function f invertible we should use a unitary operator U_f acting on two registers,

$$U_f|x\rangle|y\rangle \equiv |x\rangle|y \oplus f(x)\rangle \quad (2.92)$$

where $x \in \{0, 1\}^n$ are the entries of the query register and $y \in \{0, 1\}$ are those of the ancilla.

The steps of the algorithm are depicted in Fig. 2.5 and the execution steps are summarised as follows:

1. preparation of the initial state: $|\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle$;
2. application of $n + 1$ Hadamard gates on the qubits of the query and the ancilla registers;

$$|\psi_0\rangle \xrightarrow{H^{\otimes n+1}} |\psi_1\rangle = \frac{1}{\sqrt{2^n}}|x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad (2.93)$$

3. application of the unitary operator U_f which, exploiting the quantum parallelism, computes at the same time all the values of $f(x)$:

$$|\psi_1\rangle \xrightarrow{U_f(x)} |\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{f(x)}|x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad (2.94)$$

4. application of n Hadamard gates on the query register:

$$|\psi_2\rangle \xrightarrow{H^{\otimes n} \otimes \mathbb{I}} |\psi_3\rangle = \sum_{z \in \{0,1\}^n} \sum_{x \in \{0,1\}^n} \frac{(-1)^{\langle x|z \rangle + f(x)}}{2^n} |z\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad (2.95)$$

where $x, z \in \{0, 1\}$ and $\langle x|z \rangle = \bigoplus_{i=0}^n x_i z_i$.

5. as a last step, we perform a measurement onto the states of the query register only. The amplitude associated to the state $|z\rangle = |0\rangle^{\otimes n}$, is:

$$\frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1)^{f(x)} \quad (2.96)$$

Now, if the function is constant, then such an amplitude is equal to ± 1 while, if the function is balanced, it will be equal to 0, since the positive and negative contributions cancel. This amounts to say that the function is constant if and only if a measurement of σ_z performed on each qubit in the query register yields the

eigenvalue 0 for all qubits. As a consequence, the function is balanced if and only if at least one qubit in the query register yields a 1.

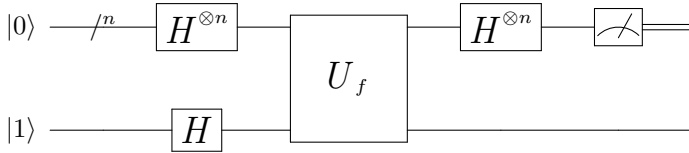


Figure 2.5: Quantum circuit for the Deutsch–Jozsa algorithm.

Hence, once again Deutsch–Jozsa quantum algorithm has been able to determine with certainty, by querying the oracle only once, the kind of a n -bit Boolean function f which was known to be only either constant or balanced. This turns out to be an exponentially faster method, compared to its classical counterpart which needs at worst $2^{n-1} + 1$ queries to determine the same result.

2.3.3 Grover’s search algorithm

The Grover’s search quantum algorithm is used to solve in a more efficient way the unstructured database search problem. The problem is the following: given an unstructured search space of size N , we have to find an object of that search space possessing a known property by querying an oracle, which is a black box with the ability to recognise whether an object has the desired property or not.

If there are M solutions, with $1 \leq M \leq N$, any classical algorithm requires $O(\frac{N}{M})$ (on average) oracle queries before determining at least one solution. On the contrary, we will exhibit a quantum algorithm which needs $O(\sqrt{\frac{N}{M}})$ queries of a quantum oracle to get the same result.

In order to simplify the notation, rather than searching the objects directly we focus on their index, i.e., a number $x = 0, \dots, N - 1$ labelling each object. For convenience we assume $N = 2^n$, so that the index referring to the objects possesses exactly n bits. Consequently, the search space is mathematically represented by the set of n -bit strings and the oracle, which recognises the objects, is associated to a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$:

$$f(x) ::= \begin{cases} 0 & x \notin S \\ 1 & x \in S \end{cases} \quad (2.97)$$

where S is the set of the M indices labelling the objects our search algorithm wants to determine.

In Fig.2.6 it is possible to see the quantum circuit associated to the Grover’s search quantum algorithm, while Fig.2.7 shows in detail the *Grover operator* G , which is composed by the quantum oracle U_f followed by a *phase-shift* operator, and will be analysed in depth in the following.

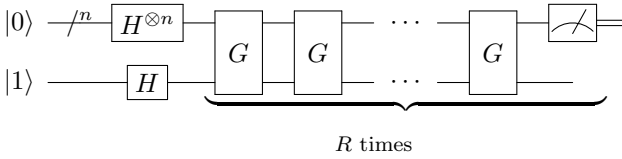


Figure 2.6: Quantum Circuit for the Grover’s Algorithm.

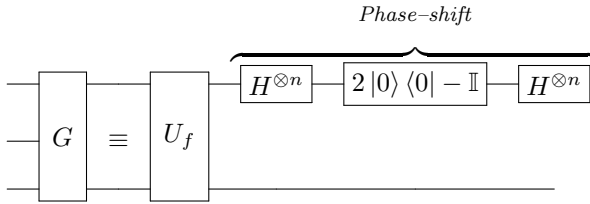


Figure 2.7: Detail of the Grover Operator G .

The quantum oracle implementing the function f is represented by a unitary operator U_f such that:

$$U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle \tag{2.98}$$

where $x \in \{0, 1\}^n$ is the index register, and $y \in \{0, 1\}$ is an ancilla bit, which is needed to make the function invertible. Given a state $|x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, we can prove that the operator U_f has the following behaviour:

$$U_f|x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = (-1)^{f(x)}|x\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.99}$$

The second register is left untouched by the oracle, hence from now on it can be omitted; by using this simplification the action of the oracle can be rewritten as follows:

$$U_f|x\rangle = (-1)^{f(x)}|x\rangle \tag{2.100}$$

i.e., the quantum oracle marks the solutions of the search problem by changing their phase.

Since the states, both the ones which are solutions and those which are not, after the application of the oracle have the same amplitude, we should reduce the probability that one of the undesired states may occur after a measurement. We define a *phase-shift* operator U_{phase} as follows:

$$U_{phase} ::= 2|0\rangle\langle 0| - \mathbb{I} \tag{2.101}$$

whose action on the state $|x\rangle$, for all $x \in \{0, 1\}^n$ is the following:

$$U_{phase}|x\rangle = -(-1)^{\delta_{x0}}|x\rangle \quad (2.102)$$

since:

$$U_{phase}|x\rangle = (2|0\rangle\langle 0| - \mathbb{I})|x\rangle = 2|0\rangle\langle 0|x\rangle - |x\rangle = 2|0\rangle\delta_{x0} - |x\rangle = \begin{cases} |0\rangle & \text{if } x = 0 \\ |1\rangle & \text{if } x \neq 0 \end{cases}$$

The operator U_{phase} adds in this way a phase (-1) to each vector of the computational basis but $|0\rangle$.

Given a uniform linear superposition of basis states, each one representing an element of the search space:

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \quad (2.103)$$

the following holds:

$$H^{\otimes n} U_{phase} H^{\otimes n} = 2|\psi\rangle\langle\psi| - \mathbb{I} \quad (2.104)$$

which can be proven as follows:

$$H^{\otimes n} (2|0\rangle\langle 0| - \mathbb{I}) H^{\otimes n} = H^{\otimes n} 2|0\rangle\langle 0| H^{\otimes n} - H^{\otimes n} H^{\otimes n} = 2|\psi\rangle\langle\psi| - \mathbb{I} \quad (2.105)$$

and for $H^{\otimes n}|0\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle = |\psi\rangle$.

The unitary *Grover operator* G defined as in Fig. 2.6 is therefore equal to:

$$G = (2|\psi\rangle\langle\psi| - \mathbb{I}) U_f \quad (2.106)$$

where $|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$.

Hence, the quantum search algorithm can be summarised as follows:

1. The first register is prepared in a linear superposition of basis states $|\psi\rangle$.
2. The state $|\psi\rangle$ is given in input to the Grover operator G , consisting of the following steps:
 - i. application of U_f ;
 - ii. application of the Hadamard gate $H^{\otimes n}$;
 - iii. application of a conditional phase shift gate which outputs every state, except $|0\rangle$, with a phase shift of (-1) ;
 - iv. application of of the Hadamard gate $H^{\otimes n}$.

The Grover operator, in order to provide with a probability close to 1 one of the desired solutions, must be iterated $R = \pi\sqrt{2^n}/4$ times.

3. Measurement of the final state.

By taking in input the state $|\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle$ the steps presented are implemented as follows:

$$\begin{aligned} |\psi_0\rangle &\xrightarrow{H^{\otimes n+1}} |s\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) \\ |s\rangle &\xrightarrow{G} G^R |s\rangle = |x_i\rangle \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) \quad \text{where } R = \pi\sqrt{2^n}/4 \\ &\xrightarrow{\text{Measure}} x_i \end{aligned} \quad (2.107)$$

where $f(x_i) = 1$ is one of the desired results.

Geometric Interpretation: Let's now analyse the Grover's algorithm by means of a geometrical visualisation. The Grover iteration can be visualised geometrically as a rotation in a two dimensional space spanned by $|\psi\rangle$ and the vector consisting of a linear superposition of all states belonging to S , which we recall is the set of solutions to the search problem, with $|S| = M$. We define two states as follows:

$$|s\rangle ::= \frac{1}{\sqrt{M}} \sum_{x \in S} |x\rangle \quad (2.108)$$

$$|s_\perp\rangle ::= \frac{1}{\sqrt{N-M}} \sum_{x \in S_\perp} |x\rangle \quad (2.109)$$

which are orthonormal, since $|S| = M$, $|S_\perp| = N - M$ and $N \cap M = \emptyset$. The initial state $|\psi\rangle$, which lays in the space spanned by $|s\rangle$ and $|s_\perp\rangle$, can be rewritten as follows:

$$|\psi\rangle = \sqrt{\frac{N-M}{N}} |s_\perp\rangle + \sqrt{\frac{M}{N}} |s\rangle \quad (2.110)$$

The effect of the operator G is a rotation, i.e., a reflection about $|s_\perp\rangle$, performed by U_f :

$$U_f(\alpha|s_\perp\rangle + \beta|s\rangle) = \alpha|s_\perp\rangle - \beta|s\rangle \quad (2.111)$$

which is followed by a reflection about the state $|\psi\rangle$ performed by $2|\psi\rangle\langle\psi| - \mathbb{I}$:

$$(2|\psi\rangle\langle\psi| - \mathbb{I})(\alpha|s\rangle + \beta|s_\perp\rangle) = 2|\psi\rangle\langle\psi|(\alpha|s\rangle + \beta|s_\perp\rangle) - \alpha|s\rangle + \beta|s_\perp\rangle = \alpha|s\rangle - \beta|s_\perp\rangle \quad (2.112)$$

since $\langle\psi|s_\perp\rangle = 0$.

Let's now introduce the angle θ such that $\cos \frac{\theta}{2} = \sqrt{\frac{N-M}{N}}$, then:

$$|\psi\rangle = \cos \frac{\theta}{2} |s_\perp\rangle + \sin \frac{\theta}{2} |s\rangle \quad (2.113)$$

It is possible to see in Fig.2.8, the operator G turns the state $|\psi\rangle$ to the following

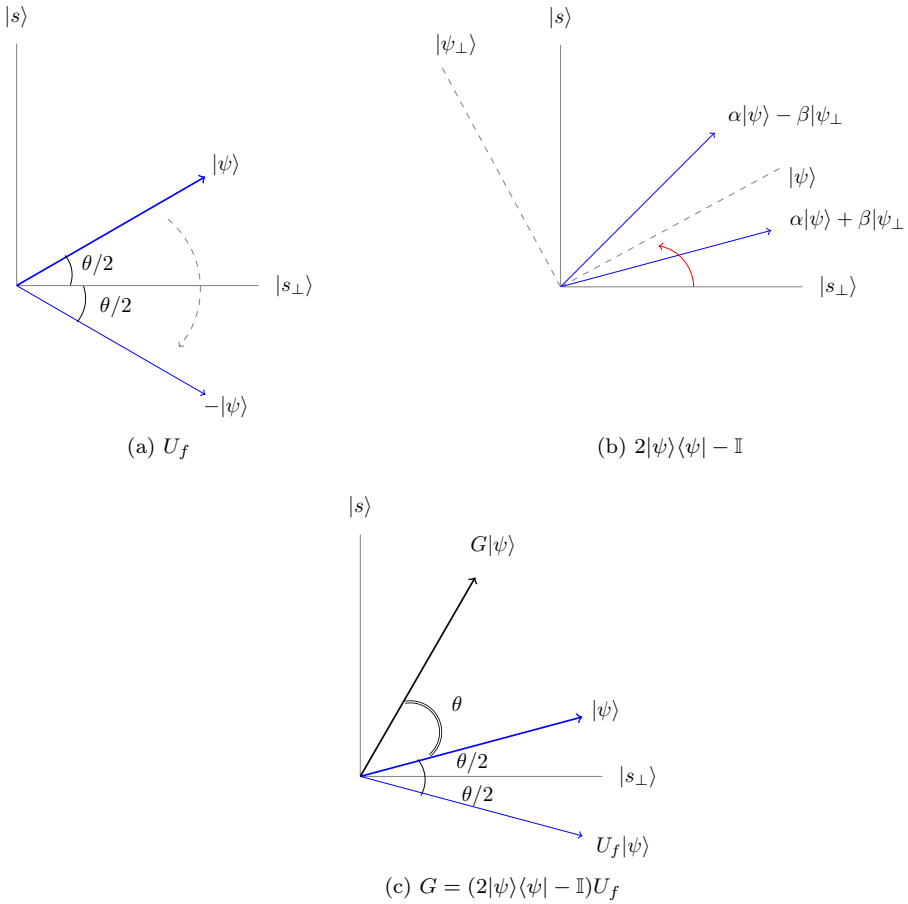


Figure 2.8: Geometric Interpretation of Grover's Oracle and Phase Shift Gates.

one:

$$G|\psi\rangle = \cos \frac{3\theta}{2}|s_{\perp}\rangle + \sin \frac{3\theta}{2}|s\rangle \quad (2.114)$$

from which follows that, after R applications of the Grover iteration:

$$G^R|\psi\rangle = \cos \frac{(2R+1)\theta}{2}|s_{\perp}\rangle + \sin \frac{(2R+1)\theta}{2}|s\rangle \quad (2.115)$$

for all $R \in \mathbb{N}$. Repeated applications of the Grover iteration bring the state $|\psi\rangle$ closer to $|s\rangle$, i.e., $\frac{\pi}{2}$. If the number of iterations has been properly chosen, a measure produces with a high probability one of the desired outcomes.

In order to determine the optimal number R of iterations of the Grover operator, bringing the initial state $|\psi\rangle$, to the desired state $|s\rangle$, we should rotate the initial state at most by $R\theta \leq \frac{\pi}{2}$, from which it follows:

$$\begin{aligned} R\theta &\leq \frac{\pi}{2} \\ R &\leq \frac{\pi}{2\theta} \end{aligned} \quad (2.116)$$

Now, since we know that $\sin x \leq x$ for $0 \leq x \leq \frac{\pi}{2}$, we can determine the lower bound of the angle θ as follows:

$$\frac{\theta}{2} \geq \sin\left(\frac{\theta}{2}\right) = \sqrt{\frac{M}{N}} \longrightarrow \frac{\theta}{2} \geq \sqrt{\frac{M}{N}} \longrightarrow \theta \geq 2\sqrt{\frac{M}{N}} \quad (2.117)$$

Then, we can use it and Eq. 2.110 to retrieve the upper bound for R .

$$R \leq \frac{\pi}{2\theta} = \frac{\pi}{2\left(2\sqrt{\frac{M}{N}}\right)} = \frac{\pi}{4}\sqrt{\frac{N}{M}} \quad (2.118)$$

Hence, the optimal number of iterations of G is $R \leq \frac{\pi}{4}\sqrt{\frac{N}{M}}$.

2.3.4 Teleportation protocol

Quantum teleportation is a protocol which allows to make a copy of an arbitrary, unknown quantum state at a distant location in absence of a quantum communication channel between the sender and the receiver. The protocol works successfully by exploiting entanglement. As an example, suppose that two parties, namely Alice and Bob, shared long ago an entangled state of two qubits. The first qubit belongs to Alice and the second to Bob, which are located spatially far away. Alice wants to deliver the qubit $|\psi\rangle$ to Bob but she does not know the state of the qubit, and she can only send classical information. Moreover, the laws of quantum mechanics do not allow Alice to observe the state without changing it. The technique used by Alice to send $|\psi\rangle$ to Bob can, intuitively, be summarised as follows:

1. Alice makes $|\psi\rangle$ interact with her half of the entangled pair; then she measures

both the two qubits in her possession obtaining a random classical outcome, i.e., a two-bit string $k \in \{00, 01, 10, 11\}$;

2. Alice communicates k to Bob which, according to the message received, applies a unitary operator on his part of the entangled pair; in this way the state $|\psi\rangle$ is completely reconstructed.

The quantum circuit implementing the teleportation protocol is depicted in Fig.2.9.

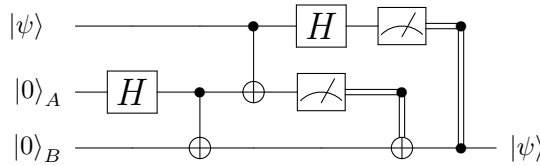


Figure 2.9: Quantum Circuit for the Teleportation Protocol.

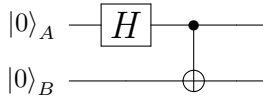


Figure 2.10: Quantum Circuit for $|\Phi^+\rangle$.

Given the state to be teleported $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the computational steps by which the protocol is performed are the following:

1. from an initial state of three qubits $|\psi_s\rangle = |\psi\rangle|0\rangle_A|0\rangle_B$, we create the Bell state $|\Phi^+\rangle$ by using the circuit in Fig.2.10:

$$\begin{aligned} |\psi_s\rangle &\xrightarrow{\text{Bell circuit}} |\psi_0\rangle = |\psi\rangle|\Psi^+\rangle = |\psi\rangle \frac{1}{\sqrt{2}} (|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B) \\ &= \frac{1}{\sqrt{2}} \left(\alpha|0\rangle (|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B) + \beta|1\rangle (|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B) \right) \end{aligned} \quad (2.119)$$

In this way, Alice's second qubit and Bob's one start in an entangled state.

2. Alice's qubits are sent through a CN gate:

$$|\psi_0\rangle \xrightarrow{CN} |\psi_1\rangle = \frac{1}{\sqrt{2}} \left(\alpha|0\rangle (|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B) + \beta|1\rangle (|1\rangle_A|0\rangle_B + |0\rangle_A|1\rangle_B) \right) \quad (2.120)$$

3. Alice applies an Hadamard transformation to the first qubit of the resulting state:

$$|\psi_1\rangle \xrightarrow{H} |\psi_2\rangle = \frac{1}{2} \left((\alpha(|0\rangle + |1\rangle)(|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B) + \beta(|0\rangle - |1\rangle)(|1\rangle_A|0\rangle_B + |0\rangle_A|1\rangle_B)) \right) \quad (2.121)$$

which results to be equal, by regrouping terms, to the following state:

$$|\psi_2\rangle = \frac{1}{2} \left(|0\rangle_A|0\rangle_B(\alpha|0\rangle + \beta|1\rangle) + |0\rangle_A|1\rangle_B(\alpha|1\rangle + \beta|0\rangle) + |1\rangle_A|0\rangle_B(\alpha|0\rangle - \beta|1\rangle) + |1\rangle_A|1\rangle_B(\alpha|1\rangle - \beta|0\rangle) \right) \quad (2.122)$$

4. Alice performs a measurement on the first and second qubits obtaining one of the following output bits:

$$\begin{aligned} 00 &\rightarrow |\psi_3\rangle = \alpha|0\rangle + \beta|1\rangle \\ 01 &\rightarrow |\psi_3\rangle = \alpha|1\rangle + \beta|0\rangle \\ 10 &\rightarrow |\psi_3\rangle = \alpha|0\rangle - \beta|1\rangle \\ 11 &\rightarrow |\psi_3\rangle = \alpha|1\rangle - \beta|0\rangle \end{aligned}$$

where $|\psi_3\rangle$ is the state of the third qubit, i.e., the Bob's part of the former entangled pair.

5. Alice communicates the classical output to Bob, which according to the bit string received applies a unitary operator on his state, retrieving in this way the original $|\psi\rangle$ state as follows:

$$\begin{aligned} 00 &\rightarrow \mathbb{I} |\psi_3\rangle = |\psi\rangle \\ 01 &\rightarrow \sigma_x |\psi_3\rangle = |\psi\rangle \\ 10 &\rightarrow \sigma_z |\psi_3\rangle = |\psi\rangle \\ 11 &\rightarrow \sigma_y |\psi_3\rangle = |\psi\rangle \end{aligned}$$

2.3.5 Quantum Key Distribution

A remarkable application of quantum information theory is now known as quantum cryptography: it exploits the features of quantum physics to encrypt private information by using techniques such as *quantum key distribution*. Quantum key distribution (herein QKD) consist in a set of protocols which rely on private key distribution, i.e., private key bits are shared between two parties over a public channel, in a provably secure way. The security of the shared key is guaranteed by the properties of quantum systems, since an eavesdropper cannot gain any information from the qubits transmitted from a party to another one without disturbing their state, and thus being detected.

In the following we will present a QKD protocol, which will be used later in this thesis. The first version of the protocol has been presented in [12].

BB84 protocol

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

1. In order to provide a secure communication, Alice can choose between four non-orthogonal states. She can choose between the standard computational basis $\{|0\rangle, |1\rangle\}$ and the $\{|+\rangle, |-\rangle\}$ one ⁵.
2. Alice *randomly* generates two classical n -bit strings $\mathbf{a} = [a_1, \dots, a_n]$ and $\mathbf{b} = [b_1, \dots, b_n]$;
3. Alice encodes the classical bit strings in a quantum state $|\Psi\rangle = \bigotimes_{i=1}^n |\psi_{a_i b_i}\rangle$, where the symbol \bigotimes denotes the tensor product of the arguments, and $a_i = \mathbf{a}[\mathbf{i}]$, $b_i = \mathbf{b}[\mathbf{i}]$, i.e., the string \mathbf{b} is used to determine in which basis each state will be encoded, hence:

$$\begin{aligned} |\psi_{00}\rangle &= |0\rangle & |\psi_{10}\rangle &= |1\rangle \\ |\psi_{01}\rangle &= |+\rangle & |\psi_{11}\rangle &= |-\rangle \end{aligned}$$

4. The state $\rho = |\Psi\rangle\langle\Psi|$ is transmitted along a quantum channel \mathcal{E} and it is received by Bob. Now, the state $\mathcal{E}(\rho)$ encodes also the effects of possible noise, or eavesdropping. Since no basis has yet been revealed by Alice, neither Bob nor an eavesdropper know how to measure the state in order to retrieve the message;
5. Bob generates *randomly* a classical n -bit string $\mathbf{b}' = [b'_1, \dots, b'_n]$ and uses it to determine the basis in which he measures $|\Psi\rangle$, obtaining the classical string $\mathbf{a}' = [a'_1, \dots, a'_n]$;
6. Alice and Bob compare on a public classical channel the strings \mathbf{b} and \mathbf{b}' by considering the elements in i -th position of both the strings, and if there is any difference between them, then the element of the strings \mathbf{a} and \mathbf{a}' in the same i -th position (i.e., a_i and a'_i) are considered not reliable, thus discarded. There remain $k \leq n$ bits resulting from measuring in the same basis; Alice and Bob both chose $k/2$ elements from \mathbf{a} and \mathbf{a}' respectively, and they publicly compare them. If they match (above a certain threshold), then the check is passed, otherwise the protocol fails because of the disturbance or eavesdropping occurred during the transmission.

This protocol relies on a property of quantum systems which states that any attempt to distinguish between two different and non-orthogonal quantum states $|\psi\rangle$ and $|\varphi\rangle$, i.e., $\langle\psi|\varphi\rangle \neq 0$, will result in the disturbance of at least one of the states. Let's assume that we want to produce a copy of either $|\psi\rangle$ or $|\varphi\rangle$, by means of a unitary transformation U and an ancilla qubit $|\chi\rangle$, as follows:

$$\begin{aligned} U(|\psi\rangle|\chi\rangle) &= |\psi\rangle|\xi\rangle \\ U(|\varphi\rangle|\chi\rangle) &= |\psi\rangle|\omega\rangle \end{aligned}$$

⁵Using the standard computational basis we have: $|0\rangle \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $|1\rangle \equiv \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, $|+\rangle \equiv 1/\sqrt{2}(|0\rangle + |1\rangle)$ and $|-\rangle \equiv 1/\sqrt{2}(|0\rangle - |1\rangle)$

Then, since $|\psi\rangle \neq |\varphi\rangle$, in order to gain information from them also $|\chi\rangle \neq |\omega\rangle$, but since inner products are preserved by U the following statement holds:

$$\langle\chi|\chi\rangle\langle\psi|\varphi\rangle = \langle\xi|\omega\rangle\langle\psi|\varphi\rangle$$

from which it follows that:

$$\langle\chi|\chi\rangle = \langle\xi|\omega\rangle$$

which implies that $|\xi\rangle = |\omega\rangle$. The protocol relies on this feature of quantum states and transmits non-orthogonal qubit states between two parties. In order to establish a threshold on any noise or eavesdropping occurring in their communication channel, they check for disturbance in their transmitted states.

Model Checking

In this Chapter we will summarise the main concepts about *model checking*, providing the basic terminology which is required to understand the following Part of the thesis. The following paragraphs are largely inspired by [32] and [7].

Model checking is an automatic technique allowing an exhaustive exploration of the state space of a concurrent system, and it is used in order to investigate whether an error state is reachable or not. It works by producing counterexamples. It has been designed to verify the design of sequential circuits and communication protocols. Nowadays, it shows many advantages on the classical verification techniques such as testing and simulation; the application of model checking it is not only concerned with hardware, but it also investigates whether software design may result in failures during the execution of a program. The process of model checking can be summarised in three phases, as follows:

1. **Creation of the model.** In this phase the program, or hardware design, is converted into an abstract model. The modelling phase relies on suitable data structures such as *Kripke* structures or labelled transition systems;
2. **Properties specification.** A list of the properties that the program/hardware is required to possess is given. Their specification is typically given in terms of *temporal logics*;
3. **Verification of the model.** The last phase is the verification one, which is performed in a completely automatic way by a model checking tool. In the case in which a failure occurs, an error record (called *error trace*) is provided.

Anyway, model checking shows some important drawbacks, for which a solution is currently under investigation. One of the main problems is the *state explosion* one; i.e., the fact that a finite-state system with many components, or with many relations between them, when modelled may result in a data structure with a huge number of states which can be difficult to be automatically verified, due to its complexity. Indeed, model checking explores the state space of the system to determine whether the properties specified are possessed or not, and if the state space is too large, this task

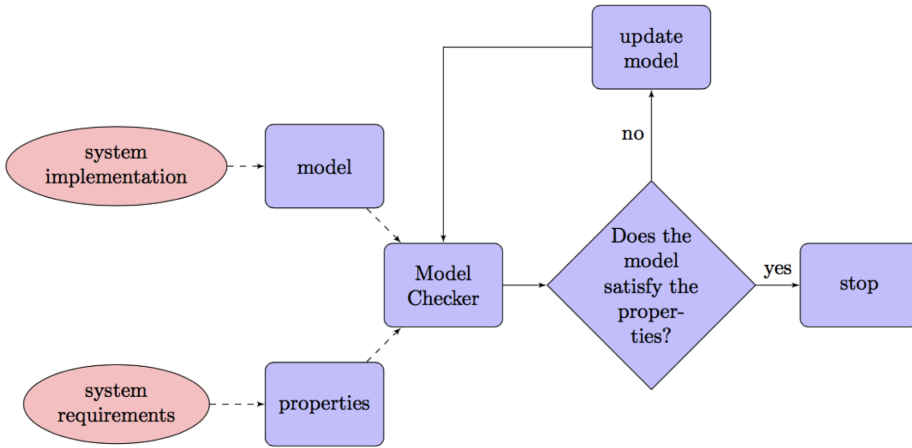


Figure 3.1: Outline of Model Checking Process.

might become really difficult to be performed in reasonable time. Another significant drawback is related to both an incorrect modelling of the system, or of its specification. These errors may result in false negatives in the error trace, and to a wrong identification of failures.

In the following Sections we will briefly explore how the three phases of modelling, specification and verification are realised and which methods are used to tackle the aforementioned drawbacks.

3.1 Creation of the model

The first step we mentioned in the introduction of the Chapter is the construction of the abstract model of the system we want to verify. In order to keep the verification process the least faulty and computationally complex as possible, the main challenge is to build a model allowing a suitable representation of the properties needed to assess the correctness of the system, by abstracting away all the non-relevant details. For example, as presented in [32], in the modelling of a Boolean circuit we might be more interested in logic gates and Boolean values instead of the actual voltage levels of the component of the circuit.

A model is composed by *states*; each state refers to an instantaneous description of the variables of the system at a given time. The causal change from a state to another one, in the model, is represented by a *transition*, i.e., the couple (s_i, s_j) of states in which s_i is the state before an action occurs and s_j is the resulting state. Within this setting, a *computation* is a sequence of transitions from a state to a new one. According to [78] two of the most used types of models are Kripke structures and labeled transition systems (herein KS and LTS respectively). KS are state-based, i.e., the states are labeled with a name, and LTS are event-based, thus in their case transitions are labeled, instead of states. Nevertheless, an equivalence between them has been given in [29], thus KS and LTS are interchangeable in many tasks. Moreover, since the work presented in this

thesis is mainly focused on KS-like structures, we will not give a formal definition of LTS, providing in the following the one of KS instead. We will use some first order logic concepts in their usual meaning, thus a basic knowledge of connectives (*and* \wedge , *or* \vee , *not* \neg), and existential and universal quantification (\exists , \forall) is assumed.

Definition 3.1.1 (Kripke structure). A Kripke structure K over a set of atomic propositions AP is a tuple (S, S_0, R, L) , where:

1. S is a set of states;
2. $S_0 \subseteq S$ is the set of initial states;
3. $R \subseteq S \times S$ is a total transition relation, i.e., $\forall s \in S, \exists t \in S$ such that $R(s, t)$. $R(s, s)$ holds, in the case s has no successor;
4. $L : S \rightarrow 2^{|AP|}$ is the labelling function.

A *path* in K from a state $s \in S$ is a sequence $\pi = s_0, s_1 \dots$ such that $s = s_0$ and $\forall i \geq 0, R(s_i, s_{i+1})$ holds. The size $|K|$ of a Kripke structure is defined to be the number of its states, i.e., $|K| = |S|$.

Due to the diversity of the systems to be modelled, we use a unifying formalism which allows to represent systems of any type, i.e., first order formulae.

By using this interpretation, a state in a KS is a valuation $s : V \rightarrow D$, where V is the set of variables of the system and D is the finite set of values that a variable in V can assume, also called *domain* of the interpretation. We describe the initial states of the system by using a first order formula \mathcal{S}_0 over the set V .

Transitions are specified by a transition relation from a set of *present state* variables V to a set of *next state* variables V' . Each variable $v \in V$ corresponds to a variable $v' \in V'$. A valuation on v and v' can be treated as a transition and, given R a transition relation, then $\mathcal{R}(V, V')$ is a formula representing it. Finally, we define a set of atomic propositions AP , whose elements have the form $v = d$, where $v \in V$ and $d \in D$. Such proposition is *true* in a state s if $s(v) = d$. If v is a variable over the Boolean domain, v indicates $s(v) = true$ and $\neg v$ corresponds to $s(v) = false$.

Definition 3.1.2 (First order representation). Given the formulae \mathcal{S}_0 and \mathcal{R} representing the system, it is possible to extract the associated Kripke structure.

1. S is the set of all valuations for V ;
2. S_0 is the set of valuation s_0 for V satisfying the formula \mathcal{S}_0 ;
3. let s and s' be two states, $R(s, s')$ holds if \mathcal{R} evaluates to *true* when to each $v \in V$ is assigned the value $s(v)$ and to each $v' \in V'$ the value $s'(v')$;
4. the labelling function is defined such that $L(s)$ is the set of all atomic propositions *true* in s . If $v \in \{true, false\}$, then $v \notin L(s) \Rightarrow s(v) = false$ and $v \in L(s) \Rightarrow s(v) = true$.

Since the transition relation of a KS is total, the relation R is extended to the case of states with no successors. Hence, also $R(s, s)$ holds.

It is important to recall that transitions must be atomic, in order to avoid failures in finding important errors, or to check for errors that will never happen in practice, which are respectively due to excessively coarse or fine grained transitions. In order to avoid the aforementioned errors, in the model no observable state can result from the execution of just a part of a transition.

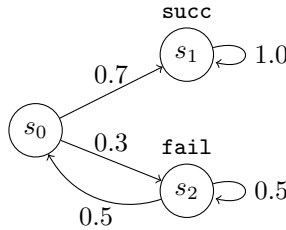
Another kind of structure used to model system which may exhibit probabilistic behaviours are the Discrete Time Markov chains.

Definition 3.1.3 (Discrete Time Markov Chain). A Discrete Time Markov chain (DTMC) K over a set of atomic propositions AP is a tuple (S, S_0, P, L) , where:

1. S is a countable set of states;
2. $S_0 \subseteq S$ is the initial state;
3. $P : S \times S \rightarrow [0, 1]$ is a transition probability function, i.e., $\forall s \in S, \sum_{s' \in S} P(s, s') = 1$;
4. $L : S \rightarrow 2^{|AP|}$ is the labelling function.

It is important to note that every state has at least one outgoing transition, and we add self loops to represent final states. A finite, or infinite, path in K is a sequence of states $\pi = s_0, s_1, \dots$ such that $P(s_i, s_{i+1}) > 0$ and it represent a possible computation of the system modelled by K .

Example 3.1.1. In this example we present a DTMC for a very simple toy protocol, which can be seen in the following:



The DTMC $K = (S, S_0, P, L)$ has the following form: $S = s_0, s_1, s_2$, $L(s_0) = \emptyset$, $L(s_1) = \{\text{succ}\}$, $L(s_2) = \{\text{fail}\}$, and the transition matrix P is:

$$P = \begin{pmatrix} 0 & 0.7 & 0.3 \\ 0 & 1.0 & 0 \\ 0.5 & 0 & 0.5 \end{pmatrix}$$

where each entry of $p_{ij} \in P$ represents $P(s_i, s_j)$.

Probability Spaces over Paths: in order to reason about the probability space over paths, it is useful to introduce the following notions.

Let Ω be a non-empty set; a σ -algebra on Ω is a family $\Sigma = \{A_i\}$, $A_i \subseteq \Omega$ closed under complementation, i.e., $A \in \Sigma \Rightarrow \Omega \setminus A \in \Sigma$ and countable union, i.e., $A_i \in \Sigma \Rightarrow \bigcup_i A_i \in \Sigma$, for $i \in \mathbb{N}$.

Definition 3.1.4 (Probability Space). A probability space is a tuple $(\Omega, \Sigma, \mathcal{P})$ where:

1. Ω is the sample space;
2. Σ is the set of events;
3. $\mathcal{P} : \Sigma \rightarrow \{0, 1\}$ is the probability measure.

Definition 3.1.5 (Probability Space over Paths). A probability space over paths in a DTMC K is a tuple $(\Omega, \Sigma, \mathcal{P})$ where:

1. $\Omega = Path(s)$, where $Path(s)$ is the set of all infinite paths over K , starting from the state s ;
2. $\Sigma_{Path(s)}$ is the set of events. Given the set $C(\hat{\pi}) = \{\pi \in Paths | \hat{\pi} \text{ is prefix of } \pi\}$, $\Sigma_{Path(s)}$ is a σ -algebra on the set $Paths$, i.e., $\Sigma_{Path(s)} = \{C\hat{\pi} | \hat{\pi} \text{ finite path} \in K\} \cup \{\emptyset\}$;
3. $\mathcal{P}_s : \Sigma_{Path(s)} \rightarrow \{0, 1\}$ is the probability measure. Given a finite path $\hat{\pi}$ of length n , its behaviour is the following:

$$\mathcal{P}_s(\hat{\pi}) ::= \begin{cases} 1 & \hat{\pi} = s \\ P(s, s_1)P(s_1, s_2) \dots P(s_{n-1}, s_n) & \text{otherwise} \end{cases} \quad (3.1)$$

3.2 Properties specification

Before a proper verification process is carried on, the properties of the system, which is now abstracted in a model, should be given in a formal way. In model checking, for both hardware and software, *modal logics* such as temporal and spatial logics are used to assert how the properties of the system evolve in time/space. We remark that an important issue in property specification is their completeness, since we may forget to specify relevant features of the system, making the verification only partial. In the following sections we will provide a description of two logics used to specify properties. Among these, we will focus in particular on *temporal* logics, since are the most widely used in both model checking and in the following parts of this thesis.

3.2.1 Temporal logics

Temporal logics, first introduced by Prior in his seminal work [77], can describe the behaviour of a system in terms of events ordered in time. Usually, logics relies on atomic propositions, connectives, and quantification operators to describe properties of a given state. When dealing with systems in which states evolve according to some measure of time (which can also be an external cause, as in reactive systems), we are not only interested in the specification of state properties, but also in the transition ones. For this reason, approaches similar to the Floyd–Hoare ones, concerned with the input–output semantic of a program, are not enough. We are now interested in the internal details of computation, thus in sequences of transitions between states, instead of focusing on a static description of the start and conclusion of execution of a system/program. Together with the usual propositional logic operators, temporal logics allow the use of

another class of operators, aiming to make valuations time-dependent, by associating a separate valuation with each point of a given flow of time \mathcal{T} .

Definition 3.2.1 (Time frame). A time frame $\mathcal{T} = (T, <)$ is a structure in which T is the set of *time points*, and $<$ a binary precedence relation such that, given two time points $s, t \in T$, if $s < t$ then s temporally precedes t .

It is worth mentioning that one can model temporal sequences in many ways; for example, in Fig. 3.2 we depicted a *linear* time model and a *branching* time model.

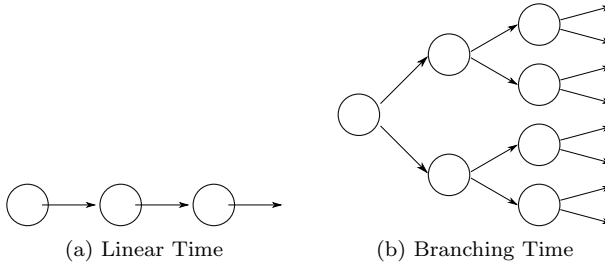


Figure 3.2: Time Models.

In linear time, every state has an unique successor and the computation is expressed by infinite sequences; the logic used as a specification language for this kind of time model is called linear-time temporal logic, also known as LTL, [75]. On the contrary, in the branching time setting every state can have several successors, generating an infinite computation tree; in this case the logic used is called Computation Tree Logic, abbreviated in CTL. Thus, in LTL, operators are provided for describing events along a single computation path, while in the branching-time logic CTL they quantify over the paths that are possible from a given state. It is worth mentioning a very expressive logic, CTL^* , which combines both branching-time and linear-time operators [25]. Since we won't focus on linear time logics, we will limit to the definition of the formalism of CTL.

A CTL formula formalises the properties of the computation tree obtained by choosing an initial state in S_0 of a structure K and unwinding the structure in an infinite tree according to its transition relation R and the labelling function L . CTL formulae are composed by the *path quantifiers* **A** (i.e., for all computation paths) and **E** (i.e., eventually, for some computation path), which roughly correspond to the universal and existential quantifiers \forall and \exists , and by the following temporal operators:

- **X** (i.e., next time), which requires that a property holds in the next state of the path;
- **F** (i.e., in the future), which requires that the property holds at some point along the path;
- **G** (i.e., globally) specifies a property that holds at every state in the path;

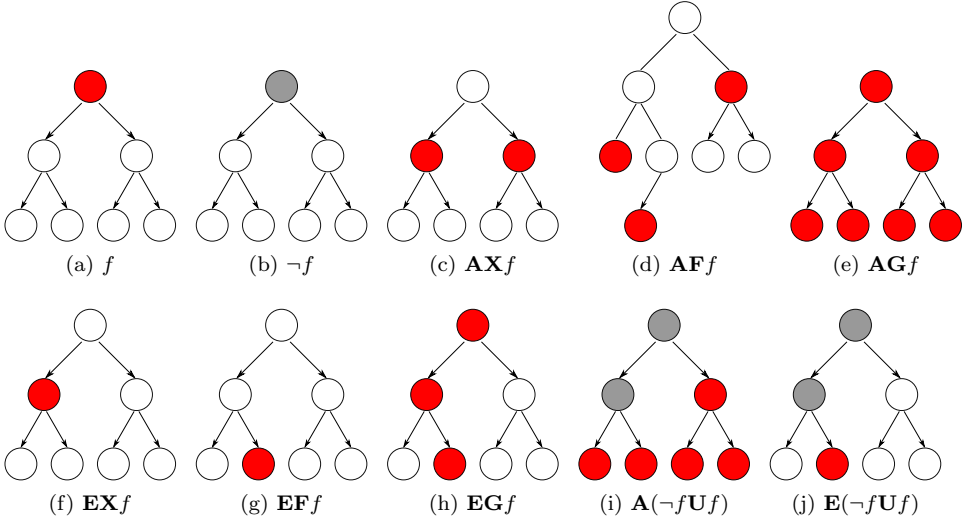


Figure 3.3: CTL Example Trees.

- **U** (i.e., until) is a binary operator, and it is used to combine two properties. It holds if there is a state in K where the second property holds, and at every preceding state along the path, the first property holds;
- **R** (i.e., release) is a binary operator which holds when the second property holds along the path, up to the first state, where the first property holds;

Some of the aforementioned formulae can be depicted as in Fig. 3.3, where $f = red$ and $\neg f = gray$, and we are supposed to start from the root of the structure.

The syntax of CTL is given in terms of *state* and *path* formulae; the former specify properties in a specific state while the latter along a specific path. Nevertheless, since it always starts with a path quantifier, a CTL formula is a state formula.

Definition 3.2.2 (CTL syntax). Given a set AP of atomic propositions and $a \in AP$, the following grammar holds:

$$\begin{array}{ll}
 \text{State formula} & f ::= a \mid \neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid \mathbf{E}g \mid \mathbf{A}g \\
 \text{Path formula} & g ::= f \mid \mathbf{X}f \mid \mathbf{F}f \mid \mathbf{G}f \mid f_1 \mathbf{U}f_2 \mid f_1 \mathbf{R}f_2
 \end{array}$$

The operators \vee , \neg , **X**, **U** and **E** are the minimal set to express any other CTL formula. Thus, given a KS K , a state s , a path $\pi = s_0, s_1, \dots$ and a *suffix* $\pi^i = s_i, \dots$, the semantics of CTL is given in Definition 3.2.3.

Definition 3.2.3 (CTL semantics). Given state formulae f_1 and f_2 and path formulae g_1 and g_2 respectively, the relation $(K, \tau) \models \phi$ (which is read “the state/path formula ϕ holds at state/along a path τ in K ”) is defined as follows:

$$\bullet \quad s \models a \quad \Leftrightarrow \quad a \in L(s);$$

- $s \models \neg f_1 \quad \Leftrightarrow \quad s \not\models f_1;$
- $s \models f_1 \vee f_2 \quad \Leftrightarrow \quad s \models f_1 \text{ or } s \models f_2;$
- $s \models \mathbf{E}g_1 \quad \Leftrightarrow \quad \exists \pi = s, \dots \text{ such that } \pi \models g_1;$
- $\pi \models f_1 \quad \Leftrightarrow \quad s \text{ is the first state of } \pi \text{ and } s \models f_1;$
- $\pi \models \mathbf{X}g_1 \quad \Leftrightarrow \quad \pi^1 \models g_1;$
- $\pi \models g_1 \mathbf{U}g_2 \quad \Leftrightarrow \quad \exists k \geq 0 \text{ such that } \pi^k \models f_2 \text{ and for } 0 \leq j < k \quad \pi^j \models f_1.$

All the temporal logics presented in this thesis will be extensions of CTL.

In the following we will briefly introduce a probabilistic temporal logic which will be used in this work.

PCTL In [47] the authors presented a *probabilistic* real time CTL (herein PCTL). Formulae in this logic allow us to express real-time and probability in systems and are built using atomic propositions, propositional connectives, and time and probability operators. The syntax of PCTL is given as follows:

Definition 3.2.4 (PCTL syntax). Given a set AP of atomic propositions, $a \in AP$, an integer $t = 0, \dots, \infty$, and a real number p , satisfying $0 \leq p \leq 1$, a PCTL formula can take the following forms:

$$\begin{array}{ll} \text{State formula} & f ::= a \mid \neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid P_{\sim p}(g) \\ \text{Path formula} & g ::= \mathbf{X}f \mid f_1 \mathbf{U}f_2 \mid f_1 \mathbf{U}^{\leq t} f_2 \end{array}$$

where $\sim \in \{<, \leq, >, \geq\}$, $P_{\sim p}(g)$ expresses that g holds for a path from a given state s with probability bounded by p , and the bounded until operator $f_1 \mathbf{U}^{\leq t} f_2$ means that after the formula f_1 holded for t time steps, the formula f_2 holds. From the syntax definition we note that a PCTL formula is always a state formula and path formulae only occur inside the probabilistic operator P .

PCTL formulae are interpreted over *discrete time Markov chains* and their semantic is given as follows.

Definition 3.2.5 (PCTL semantics). Given a DTMC K , state formulae f_1 and f_2 and a path formula g respectively, the relation $(K, \tau) \models \phi$ (which is read “the state/path formula ϕ holds at state/along a path τ in K ”) is defined as follows:

- $\pi \models f_1 \mathbf{U}^{\leq t} f_2 \quad \Leftrightarrow \quad \exists i \leq t \text{ such that } \pi^i \models f_2 \text{ and } \forall k \leq i \quad \pi^k \models f_1;$
- $s \models P_{\sim p}(g) \quad \Leftrightarrow \quad \mathcal{P}_{(s,g)} \sim p \text{ where } \mathcal{P}_{(s,g)}$
is the probability that, from a state s the formula g is true for an outgoing path.

Non-probabilistic PCTL formulae obey to the same semantics as CTL.

An example of operator P can be seen in Fig. 3.4 in which we ask if the probability, starting from the state s , that g is true for an outgoing path is constrained by 0.5.

Two examples of satisfying paths, related to the DTMC of Example 3.1.1, can be seen in Fig. 3.5.

In these paragraphs we briefly introduced the concept of temporal logics and their role in modelling properties of a system, which will later be verified.

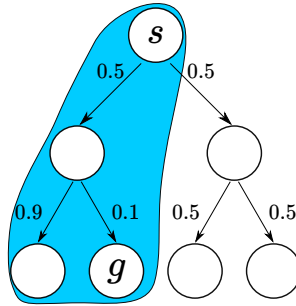


Figure 3.4: Example of PCTL P operator for the formula $P_{\leq 0.5}[\mathbf{F}(g)]$.

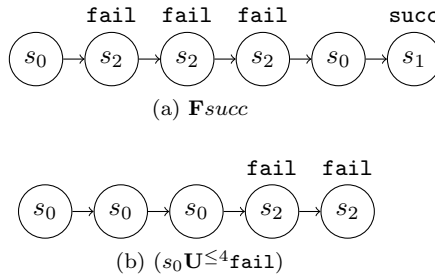


Figure 3.5: Examples of satisfying paths.

3.3 Model checking

The model checking problem can be described as follows: given a *structure* K , e.g., a KS or a DTMC, representing a finite state system, and a temporal formula f expressing the property that such system is supposed to match, we must find the set of all states $s \in S$ satisfying f , i.e.:

$$S_f = \{s \in S \mid (K, s) \models f\} \tag{3.2}$$

with the *initial* states of the structure belonging to S_f . Some model checking algorithms use an explicit representation of the structure K as a labelled, directed graph while others use symbolic techniques.

A brief summary of the main CTL model checking techniques is given in the following, using Kripke structures as models (this can be easily extended to other structures, e.g., LTSs or DTMCs). Since a more in-depth description of these approaches is out of the scope of this thesis, we refer to [32, 7] to a more complete explanation.

3.3.1 CTL Model Checking

Given a Kripke structure $K = (S, S_0, R, L)$, we aim at determining the set S_f of states satisfying a CTL formula f . This can be achieved by an algorithm which labels each

state s with the set $sat(s)$ of subformulae of f which are *true* in s . At first $sat(s) = L$, then the algorithm proceeds recursively as follows: at each i -th step, the subformulas of f with $i-1$ CTL nested operators are processed, by putting each of them into the set of labels of each state in which the subformula is *true*. At the end of the computation we obtain that $s \models f \iff f \in sat(s)$.

Now, we will show how the intermediate steps of the model checking algorithm work for the CTL formulae g ; since it can be proven that every CTL formula can be expressed in terms of $\neg, \vee, \mathbf{EX}, \mathbf{EU}$ and \mathbf{EG} , then g can be in one of the following configurations:

$g = \neg f$: the algorithm labels the states which are not labelled by f ;

$g = f_1 \vee f_2$: the algorithm labels any state which is labelled by f_1 , by f_2 or by both;

$g = \mathbf{EX}f$: the algorithm labels each state having one or more successors labelled by f ;

$g = \mathbf{E}[f_1 \mathbf{U} f_2]$: first the algorithm retrieves all the states labelled with f_2 , then it works backwards to find all the states reachable by a path with all the states labelled by f_1 ; this can be achieved by using the inverse of the transition relation R . The pseudocode describing a procedure to add g to $sats$ for the *until* case is described in the following:

Algorithm 1 Model Checking CTL Until

```

1: procedure EU( $f_1, f_2$ )
2:    $T \leftarrow \{s \mid f_2 \in sat(s)\}$ 
3:   for all  $s \in T$  do
4:      $sat(s) \leftarrow sat(s) \cup \{\mathbf{E}[f_1 \mathbf{U} f_2]\}$ 
5:     while  $T \neq \emptyset$  do
6:       choose  $s \in T$ 
7:        $T \leftarrow T \setminus \{s\}$ 
8:       for all  $t \mid R(t, s)$  do
9:         if  $\mathbf{E}[f_1 \mathbf{U} f_2] \notin sat(t)$  and  $f_1 \in sat(t)$  then
10:           $sat(t) \leftarrow sat(t) \cup \{\mathbf{E}[f_1 \mathbf{U} f_2]\}$ 
11:           $T \leftarrow T \cup \{t\}$ 
12:        end if
13:      end for
14:    end while
15:  end for
16: end procedure

```

By using this backward search, we add $\mathbf{E}[f_1 \mathbf{U} f_2]$ to $sat(s)$ for each $s \models \mathbf{E}[f_1 \mathbf{U} f_2]$, provided that $s \models f_1 \iff f_1 \in sat(s)$ and $s \models f_2 \iff f_2 \in sat(s)$. The whole procedure's cost is $O(|S| + |R|)$.

$g = \mathbf{EX}$: the algorithm first decomposes K into nontrivial strongly connected components, i.e., a subgraph C in which each node is reachable from every other node along a directed path. C is nontrivial if and only if it has more than one node or the

only node has a self loop. Then, given $K' = (S', S'_0, R', L')$ obtained from K , where $S' = S \setminus \{s | s \neq f\}$, $R' \in S' \times S'$ and L changes accordingly, the algorithm relies on Lemma 3.3.1:

Lemma 3.3.1. $s \models \mathbf{EG}f$ if and only if the following holds:

1. $s \in S'$
2. $\exists \pi \in K' | R'(s, t) \in C$

The proof of Lemma 3.3.1 can be seen in [32]. In the following we show the procedure to compute the set $sat(s)$ for $\mathbf{EG}f$.

Algorithm 2 Model Checking CTL Globally

```

1: procedure EG( $f$ )
2:    $S' \leftarrow \{s | f_1 \in sat(s)\}$ 
3:    $SCC \leftarrow \{C | C \text{ nontrivial SCC of } S'\}$ 
4:    $T \leftarrow \bigcup_{C \in SCC} \{s | s \in C\} \{s | f_1 \in sat(s)\}$ 
5:   for all  $s \in T$  do
6:      $sat(s) \leftarrow sat(s) \cup \{\mathbf{EG}f\}$ 
7:     while  $T \neq \emptyset$  do
8:       choose  $s \in T$ 
9:        $T \leftarrow T \setminus \{s\}$ 
10:      for all  $t \in S'$  and  $R(t, s)$  do
11:        if  $\mathbf{EG}f \notin sat(t)$  then
12:           $sat(t) \leftarrow sat(t) \cup \{\mathbf{EG}f\}$ 
13:           $T \leftarrow T \cup \{t\}$ 
14:        end if
15:      end for
16:    end while
17:  end for
18: end procedure

```

The first part of the algorithm, which computes the strongly connected components, takes $O(|S'| + |R'|)$. Then, as in the *until* case, we use the converse of R' to find all the states that can be reached by a path in which every state is labelled by f , which takes $O(|S| + |R|)$. Since we should run the algorithm for every subformula of f , the algorithm requires $O(|f|) \cdot O(|S| + |R|)$.

We recall that a remarkable feature of model checking is the one of finding *counterexamples* and *witnesses*, i.e., the model checker, to prove $\mathbf{A}f = false$, finds a computation path satisfying $\neg \mathbf{A}f = true$. Likewise, to check $\mathbf{E}f = true$, it will find a path in which the formula holds.

3.3.2 Symbolic CTL Model Checking

The CTL model checking procedure described assumes that the Kripke structure K has an explicit representation in which, for each state we can always know its predecessor and successor lists. This *enumerative* representation turns out to be not adequate for

very large KS, causing an explosion in the size of the model. To deal with this scenario, *symbolic* model checking algorithms for CTL have been presented. These algorithms rely on a binary encoding of the states, identifying both the subsets of the state space and the transition relation R with Boolean functions. Operations on Boolean functions can be implemented as graph algorithms operating on a particular data structure called *Ordered Binary Decision Diagrams* (herein OBDDs), in this way we don't need to build a full enumeration of the problem space (e.g., a Kripke structure). OBDDs represent Boolean functions as directed acyclic graphs; they are binary decision diagrams (BDDs) in which we impose a total ordering $<$ over the set of variables var and we require that for any node u in the graph, and either each nonterminal child v , their respective variables must be ordered $var(u) < var(v)$, moreover there shouldn't be any redundancy in the nodes of the diagram. In this way, we can build an OBDD from a BDD by means of a repeated application the following transformation rules, depicted as an example for the Boolean function $f = (x_2 \wedge x_3) \vee (x_1 \wedge x_3)$, in Fig. 3.6.

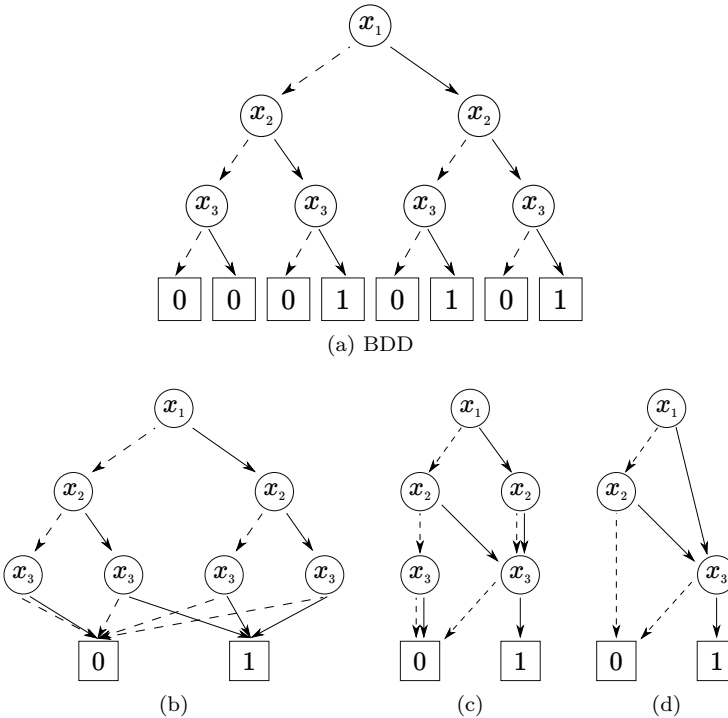


Figure 3.6: Example of OBDD Transformation Steps.

- Elimination of duplicate terminal nodes: all but one terminal node (for each label) are removed and the remaining edges are redirected into the remaining nodes.
- Elimination of duplicate internal nodes: if two nodes have the same variable set $var(u) = var(v)$, and are directed towards the same children, i.e., $low(u) = low(v)$

and $high(u) = high(v)$, then one of the two parent nodes is removed and the edges are redirected to the remaining one.

- Elimination of redundant tests: if an internal node v has the same children, i.e., $low(v) = high(v)$, then v is removed and its edges redirected toward $low(v)$.

We won't delve more into the theoretical details of OBDDs, which can be found in Refs. [21, 32]. OBDDs are useful to represent in a compact way relations over finite domains, thus also Kripke structures can be modelled in such way; in the following we briefly show how to build an OBDD from a KS.

OBDD Representation of KS: OBDDs are useful to represent in a compact way relations over finite domains, thus also Kripke structures can be modelled in such way; we briefly show how to build an OBDD from a KS. Let $K = (S, S_0, R, L)$ be a KS, in order to translate it in an OBDD we must describe S and R as follows:

- States $s \in S$ are described by means of vectors of Boolean values; each Boolean vector can be represented by a Boolean formula $f_S(s)$ which holds if the system is in the state s . The characteristic function $f_S(S) = \bigvee_{s \in S} f_S(s)$ representing S is an OBDD.
- In order to represent the transition relation $R(s, s')$ we need two Boolean vectors \hat{s} and \hat{s}' representing the initial and final state of the transition, respectively. We define the Boolean relation associated as $\hat{R}(\hat{s}, \hat{s}') ::= f_S(s) \wedge f_S(s')$ and the transition relation can be represented by means of its characteristic function $\bigvee_{s, s' \in R} \hat{R}(\hat{s}, \hat{s}') = \bigvee_{s, s' \in R} f_S(s) \wedge f_S(s')$.

Symbolic Model Checking Algorithm for CTL: The symbolic model checking algorithm relies on a procedure **Check** which takes in input a CTL formula g and returns in output an OBDD representing the states satisfying the formula. The algorithm can be defined recursively over the structure of CTL formulas. Since it relies on different methods, which are out of the scope of this thesis, we refrain from a more in-depth explanation, by addressing [32] as detailed reference.

3.3.3 Probabilistic CTL Model Checking

This section considers the automated verification of probabilistic systems. The structure used to model this kind of systems is a discrete time Markov chain, i.e., a Kripke structure in which each nondeterministic choice between successor states is replaced by probabilities. Given a state s , its successor s' is chosen from a probability distribution depending from the current state only, and not from any state belonging to the path previous to s . The temporal logic used is PCTL. We can briefly describe the PCTL model checking for DTMCs [47] in which the procedure takes in input a DTMC and a formula g , and gives in output the set $sat(g)$ of states satisfying g in a way similar to the CTL model checking algorithm presented above. Initially $sat(s) = L(s)$ for each $s \in S$. Then, a labelling procedure is performed, from the smallest subformula of g that has not yet been labelled, and ending to g itself. Composite formulae are labelled according to the labelling of their parts.

3.3.4 PRISM

PRISM (PRobabilistic Symbolic Model checker) is a symbolic model-checker for probabilistic systems [58], which allows model checking of formulae of the probabilistic temporal logic PCTL. PRISM takes in input a model written in a probabilistic variant of Reactive Modules [4], then it computes the set of reachable states. The model can be, among others, a DTMC which is used to test specifications in the logic PCTL. The tool then performs model checking to determine which states of the system satisfy each specification. For PCTL properties and DTMC models, PRISM implements the algorithms of [47, 18, 8]. It is also possible to export the transition matrix of the model, enabling analysis in other applications and visualisation of the model. Snapshots of the PRISM model checker, for both the model creation and model checking steps, can be seen in Figures. 3.7 and 3.8.

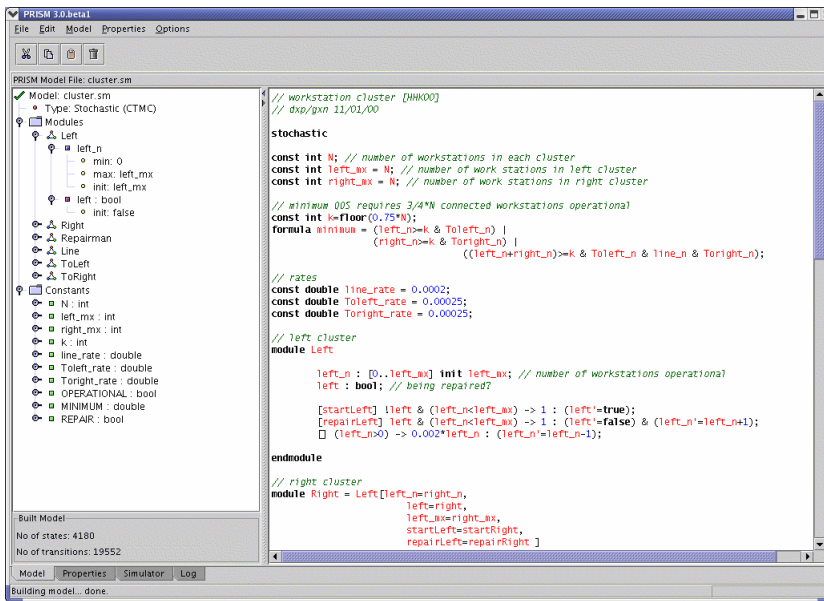


Figure 3.7: PRISM Model Creation.

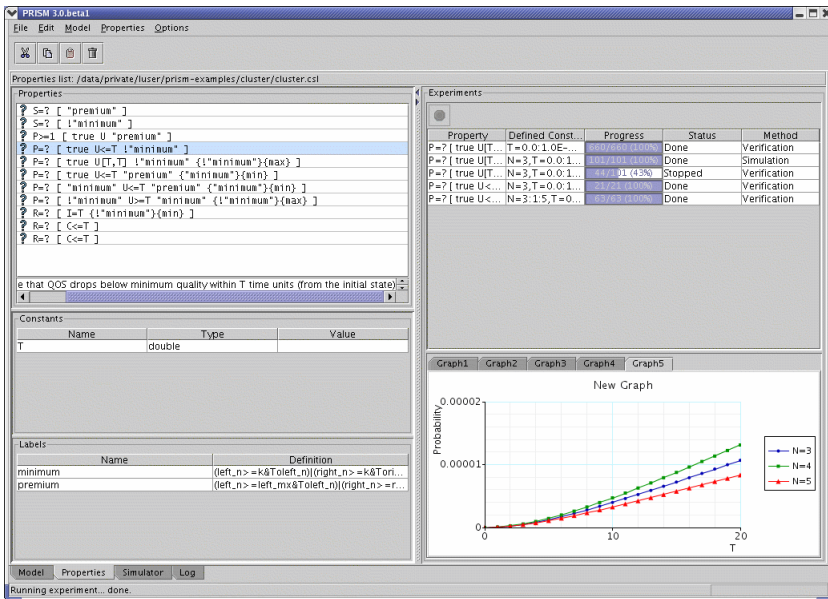


Figure 3.8: PRISM Model Checking.

II

State of the art

4

Quantum Programming and Quantum Model Checking

This Chapter is devoted to briefly summarise the state-of-the-art in the fields of quantum programming and the verification of quantum programs. The main aim of computer science is the study of computational models and algorithms, and their translation into computer programs, topics that have been applied to the quantum information theory setting as well. Nevertheless, quantum phenomena such as superposition, interference and entanglement do not always have a straightforward classical counterpart and exhibit completely new behaviours that do not arise in classical computation. For this reason, different quantum abstract languages and verification techniques have been proposed, aiming at modelling in the most accurate way the effects of quantum physics. After a brief summary of the state-of-the-art, we will focus in particular on a quantum programming language, i.e., Quipper, and on a quantum model checker, i.e., QPMC, since they have been used as a starting point in the presented work. For a more in-depth presentation of the models listed in this Chapter, we recommend [90, 80, 44, 66, 65, 40].

4.1 Quantum programming languages

One of the main purposes of programming languages is to abstract the computation from the physical low-level point of view, i.e., from a computation given in terms of Boolean circuits, logic gates, voltage levels and really long strings of bits, to a more abstract, *high level* description, which is more human readable and allows a more abstract problem solving process, rather than a context oriented one. The study of quantum algorithms, from a computational point of view, is mainly given in terms of quantum circuits, which are similar to the low-level circuit description of classical computation already mentioned. According to [44] each algorithm, or protocol, relies on an ad-hoc set of transformations to achieve its goal. The goal of quantum programming is to abstract from its low-level description, providing a more general way to treat quantum programs.

Quantum models of computation: Many different abstract computational models have been proposed. Among them, the most popular one is the quantum circuit model, which has been presented in Section 2.2.1 and consists of qubits, unitary gates acting on them, and final measurements. Throughout the years, different authors in [11, 16, 30] proposed the concept of *Quantum Turing Machines*, which as in the classical case consist of a tape, a finite control and a head. All the state transformation are unitary and measurements are not performed. The two models presented can be regarded as the quantum counterpart of their classical analogue, with the difference that states, in a Quantum Turing Machine, cannot be observed nor copied.

A third model, equivalent to the first two mentioned, is the *QRAM* one presented in [57], in which a quantum device is controlled by a classical one, and allows an interleaving of both unitary transformation and measurements. The quantum device in the QRAM contains n qubits, and it manipulates then according to the instructions sent by the classical device; in the end, the quantum device provides the output of the computation to the classical one which, according to the results, end the computation or re-initialises it.

Other models are the *measurement based quantum computation*, presented in [20] or the *teleport based one*, proposed in [63]; both of them rely on quantum measurement not only to extract information from a quantum state, but also to perform the computation itself on a particular entangled state called *cluster state*. Recently, *adiabatic* or *topological* quantum computation models have been proposed, in [34, 56] respectively, which are more physically rooted.

Imperative quantum programming languages: The quantum programming languages relying on the imperative paradigm are probably the earlier presented; this class of languages considers a program as a sequence of instructions which, step by step, updates a global state. The main imperative quantum programming languages are listed in the following:

- **Quantum pseudocode:** proposed by Knill in [57]. This approach is the first formalised language for the description of quantum algorithms. It is connected with the aforementioned QRAM model.
- **QCL:** the Quantum Computer Language has been proposed in [65] and it is an high-level language for the implementation and simulation of quantum programs, derived from classical procedural languages such as *C*, with classical data types similar to the ones found in *C*. The basic built-in quantum data type in QCL is `qreg` (i.e., quantum register), which can be interpreted as a an array of qubits and the QCL standard library provides the main quantum operators used in quantum algorithms, plus the possibility to have user-defined functions.
- **Q Language:** developed in [17] is an extension of the *C++* programming language. It provides classes for the basic quantum operations like `QHadamard`, `QFourier`, `QNot`, and `QSwap`, which are derived from the base class `Qop`. New operators can be defined using the *C++* class mechanism. Q Language provides a simulator.

- **qGCL:** the Quantum Guarded Command Language (qGCL) has been presented in [93] and it is based on Guarded Command Language created by Dijkstra. It is more similar to a specification language, since it is endowed with a formally defined operational semantics.

It can be described as a language for quantum programs specification.

Functional quantum programming languages: The quantum programming languages relying on the functional paradigm do not perform computation by updating a global state, but by the definition of a function which maps inputs to outputs. Functional programming languages for quantum computation have seen the light with an attempt to define a quantum extension of the lambda-calculus, which has been developed by van Tonder, and refined by Selinger and Valiron. The main functional quantum programming languages are listed in the following:

- **Quantum Lambda Calculus:** the first definition of quantum lambda calculus can be found in [59]. Then, in [86] the author provided an extension of lambda calculus suitable to prove correctness of quantum programs based on the Scheme programming language. This version does not incorporate any classical data type.
- **QPL:** the Quantum Programming Language was defined in [82]. Its extension cQPL incorporates elements for modelling quantum communication protocols. At the moment cQPL compiler generates $C++$ code for linking with the simulation library from QCL.
- **QFC:** in [82], the authors developed QFC which aims at representing quantum programs using the functional version of flow charts. The language offers also a text-based syntax and it is endowed with a safe type-system, and it can be compiled in the QRAM model.
- **QML:** in [3] the authors proposed a quantum programming language based on Haskell, endowed with an equational theory and with a quantum control.
- **Quipper:** [44], a functional programming languages embedded in Haskell. We will describe Quipper in details in Section 4.2, since it will be widely used throughout the thesis.
- **LIQUi)** and **Q#:** both developed by Microsoft; [87] is a functional programming language embedded in $F\sharp$, while $Q\sharp$ is a domain-specific language initially released to the public as part of the Quantum Development Kit <https://docs.microsoft.com/en-us/quantum/?view=qsharp-preview>.

The latter three languages, i.e., Quipper, LIQUi) and $Q\sharp$ are also the most recent ones.

Quantum compilers From a practical point of view, we witnessed many different examples of design and implementation of quantum languages and compilers. In [85] a *layered quantum software architecture* has been proposed, which maps a quantum program specified in an high-level language to a quantum device by means of an intermediate quantum assembly language. In [62] the authors proposed the Sequential

Quantum Random Access Memory machine (abbreviated in SQRAM), a semi-classical architecture based on Knill's QRAM, allowing to use quantum assembly code. Nagarajan and its team also developed a compiler for a subset of Selinger's QPL. Zuliani in [94] designed a compiler for qGCL in which the compilation step is realized by manipulating algebraically a qGCL program into a canonical form that can be executed by a target machine. A translation between the quantum extension of the while-language defined in [90] and a quantum extension of classical flowchart language was given in [92]. All of these studies are based on the popular circuit model of quantum computation. In [27] the authors presented a low-level language based on the measurement-based one-way quantum computer. Recently, quantum compilation has been intensively researched and series of compilation techniques has been developed through the recent projects of languages Quipper , LIQUi]), among others. Recently, *QA Prolog* and *QMASM* have been proposed in [69].

Categorical and concurrent approaches For the sake of completeness, here we summarise approaches which are different to the ones mentioned in the previous paragraphs. In particular, in [1] the authors proposed a categorical semantics for quantum programs, based on a categorical axiomatisation of quantum mechanics. This method is effective when addressing high-level description and verification of quantum communication protocols. In [1, 48] the authors proposed two examples of categorial quantum logics. In order to deal with concurrent quantum programs, *quantum process algebras*, e.g., the ones presented in [28, 71, 72, 91], have been proposed.

4.2 Quipper

In the context of this Thesis we will provide an in-depth description of the Quipper programming language, which will be used in the following Chapters.

Quipper is an embedded functional programming language for quantum computation [44]. Quipper is based on the Knill's QRAM model [57] of quantum computation. This model uses both a classical and a quantum device to perform a quantum computation. The classical device performs purely classical computations (control flow, test, loops). The quantum computer is a specialised device that is able to perform only two kinds of instruction: unitary operations and measurements; we can think about it as a sort of quantum *co-processor*.

As defined in [84], there is no control flow on the quantum co-processor, i.e., any loop or conditional branching has to come from the classical device controlling the co-processor. As a result, a quantum computation can be pictured as a linear circuit, representing the flow of elementary instructions sent to the co-processor.

However, in some algorithms the circuit is conditioned on the result of intermediate measurements and part of the circuit can depend on a measurement done at a previous stage. For this reason, there is the need for a scalable programming language, to accommodate such a dynamic representation of circuits [84]. In order to accomplish this requirement, Quipper's developers decided to write the language as a library for Haskell; a strongly typed functional classical programming language. Quipper has a collection of data types, combinators, and a library of functions within Haskell, together with an idiom, i.e., a preferred style of writing embedded programs [44]. Quipper provides many

higher order and overloaded operators, even though Haskell does not have linear type and dependent type features; in fact, to overcome this problem, Quipper checks linear and dependent types at run time rather than at compile time [83].

Quipper provides an extended circuit model of quantum computation. Usually, the quantum circuit model is concerned with qubits and unitary gates, while quipper provides a larger class of circuits allowing both quantum and classical wires and gates within a circuit.

Naturally, we assume that quantum instructions could be controlled by a classical wire whilst the vice versa is not possible.

In Quipper measurement is a gate that, probabilistically, turns a qubit in a classical bit, e.g., the qubit associated to the standard computational basis state $|0\rangle$ is transformed into the bit 0, and the qubit $|1\rangle$ is transformed into the bit 1.

4.2.1 Execution steps

Quipper programs have three different phases of execution [44]:

- **Compile-time:** the compile phase of a Quipper function is the same as an Haskell one. The input to this phase are source code and compile-time parameters. The output is executable object code.
- **Circuit-generation-time:** This takes place on a classical computer. The input to this phase are the executable object code generated by the previous phase, and the circuit parameters (for example, the size of registers, problem sizes, the size of time steps, error thresholds, etc.). The output is a representation of a quantum circuit.
- **Circuit-execution-time:** This takes place on a physical quantum computer. The input to this phase is a quantum circuit, and possibly some circuit inputs. The output consists of circuit outputs (for example, classical bits representing measurement results).

4.2.2 Parameter/input distinction

Parameters are values known at circuit generation time (for example the size of a circuit) and **inputs** are values known at circuit-execution time (i.e. the state of a qubit represented as a wire in the circuit). Quipper introduces *three* data types for qubits and bits:

1. `Bool` is a Boolean *parameter*,
2. `Bit` is a Boolean *input*,
3. `Qubit` is a qubit *input*.

4.2.3 Circuit description language

Quipper is above all a circuit description language. For this reason it uses the state vector formalism and its main purpose is to make circuit implementation easier even

when we are dealing with difficult and large-scale circuits. It provides operation for circuit manipulation; these operations include inversion, iteration, ancilla management and circuit transformations. The philosophy of the Quipper paradigm is that qubits are held in variables and gates are applied to them one at a time. Subroutines can be used to group gate-level operations together where the programmer finds it useful.

Quipper provides **block-structure** operation as:

```
with_controls :: Qubit -> Circ a -> Circ a
```

that lets a qubit control an entire block of gates, and

```
with_ancilla :: (Qubit -> Circ a) -> Circ a
```

that provides an ancilla qubit to a group of gates. It provides also other circuit operators that can be found in the Quipper Online Documentation at <http://www.mathstat.dal.ca/~selinger/quipper/doc/frames.html>.

A Quipper program is a function that receives as input some quantum data, performs state changes on it, and then outputs the changed quantum data. This is encapsulated in a Haskell monad called `Circ`.

The `Circ` monad is a `ReadWrite` monad, wrapped with an additional state. It encapsulates the type of quantum operations. For example, a quantum operation that inputs two `Qubits` and outputs a `Qubit` and a `Bit` has the following type:

```
(Qubit, Qubit) -> Circ (Qubit, Bit).
```

Usually functions are written in a `do` block. A `do` block starts with the `do` keyword which is followed by a series of expressions or operations. The starting point of a Quipper program is the `main` function. As we have mentioned before, the circuit generation time phase sends a circuit defined in the source code to the physical quantum device to execute it. The `print_simple` and `print_generic` functions are used to print the circuit in an available output format [83]. Another important remark about Quipper is that its host language Haskell provides many data structures like `Map`, `Set` etc, but Quipper mainly uses `list` and `tuple`.

Among others, Quipper provides two important features that are the automated generation of quantum oracles and extensible quantum data type.

4.2.4 Simulation

Quipper provides three different simulators for the execution of quantum algorithms on a classical computer (with an exponential slowdown):

- a classical simulator;
- a stabilizer simulator;
- a quantum simulator.

The quantum simulator takes in input a Quipper circuit producing function and uses a transformer to lazily simulate the resulting circuit (see `Quipper.Transformer` library for more details). The implementation of the quantum simulator makes use of a `State` monad, that is used to hold the quantum state. The simulator uses complex numbers as probability amplitudes, and a random number generator to simulate quantum randomness. Basis states are stored as a map.

As an example, let us now consider a simple instance of the Deutsch algorithm, in which we use a controlled-not gate as oracle. In Quipper, the steps of the algorithm can be written as follows:

```
deutschCirc :: (Qubit, Qubit) -> Circ Bit
deutschCirc (q0, q1) = do
  q0 <- qinit False
  q1 <- qinit True
  label (q0, q1) ("|0>", "|1>")
  map_hadamard_at (q0, q1)
  qnot_at q1 'controlled' q0
  c0 <- measure q0
  return c0
```

Quipper provides the main quantum gates used in the circuit model, e.g. Hadamard (`hadamard`), Pauli and phase-shift gates (`gate_X`, `gate_Y`, `gate_Z`, `gate_S`) among others, which can be written in both functional and imperative style. Quipper provides also *controlled* operators which, given two quantum registers, i.e., two lists of qubits q_0 (target) and q_1 (control), are written according to the following syntax: `q0 'controlled' q1`.

In addition, Quipper provides the opportunity to apply custom unitary matrices by importing some additional libraries, i.e., `Libraries.Synthesis.Matrix`, `QuipperLib.Synthesis` and `Libraries.Synthesis.Ring`. In the following, we show an example of a re-defined controlled-not gate.

```
customCN :: Matrix Four Four D0mega
customCN = matrix4x4 (1,0,0,0)
                  (0,1,0,0)
                  (0,0,0,1)
                  (0,0,1,0)
```

which allows to rewrite the code for the aforementioned instance of the Deutsch algorithm as follows:

```
deutschCirc :: (Qubit, Qubit) -> Circ Bit
deutschCirc (q0, q1) = do
  q0 <- qinit False
  q1 <- qinit True
  label (q0, q1) ("|0>", "|1>")
  map_hadamard_at (q0, q1)
  (q0,q1) <- customCN (q0,q1)
  c0 <- measure q0
  return c0
```

Quipper syntax is not restricted only to the application of unitary and measurement gates to a list of qubits, but provides a larger class of operations, thus we refer to the Quipper Online Documentation in <http://www.mathstat.dal.ca/~selinger/quipper/doc/frames.html> for a thorough description of the language.

4.3 Quantum programs verification

Analysis and verification of quantum programs has been recently explored from many points of view. In [89, 24] the authors investigated the possibility to apply Floyd-Hoare-style logics for the verification of quantum programs. Another possibility is to apply model checking techniques to the verification of quantum algorithms and protocols. The main obstacle arising from these approaches is that the set of all quantum states, traditionally regarded as the underlying state space of the model to be checked, is a

continuum so the techniques of classical model checking, since they require a finite state space cannot be applied directly to the quantum case [35]. Quantum model checking must find a solution for the two following problems:

1. the problem of defining a formal framework to suitably model both the program and the properties;
2. the problem of defining techniques allowing to perform the checks on a finite number of representative states only.

The structures proposed in literature to model a quantum program are either *Quantum Automata* or *Quantum Markov Chains*. The main difference between the two formalisms is that the first one represents actions in terms of unitary operators, while the second uses the more general formalism of superoperators, allowing to capture a wider set of dynamics. Since some classical key problems in model checking can be restricted to the problem of reachability of states on a structure, reachability analysis of quantum Markov chains provides a suitable basis for quantum model checking. Many definitions of quantum Markov chains have been investigated in literature, in particular by [46]. At present, some model checking techniques have been developed both for quantum programs and for quantum systems. In [39] the authors provided a model-checker to verify quantum communication protocols exploiting the probabilistic model checker PRISM [58]. This work tried to solve the aforementioned state problem by restricting the state space to a set of finitely describable states called *stabiliser states*, and the set of operations applied on them to the class of *Clifford group* (Hadamard, controlled-not and phase gate). In a later work, presented in [41], the authors presented theorised QMC, an automatic tool which verifies quantum protocols expressible in stabiliser formalism so that the state space can be encoded in a classical way. QMC uses the quantum computation tree logic developed by [68] to express the properties to be checked. Anyways, this approach, according to [35], does not work for general protocols since it is focused on the stabilizer formalism. Other model checking techniques, based on both quantum Markov chains and QCTL have been investigated in [36], in which the authors provided an automatic tool, QPMC, which will be described in the following section.

4.4 QPMC

In [35] the authors proposed the novel notion of *superoperator weighted Markov chain* in which the state space is taken classical (and usually can be finite), while all quantum effects are encoded in the superoperators labelling the transitions. This approach is suited for verification of *classical* properties for which only the measurement outcomes as well as the probabilities of obtaining them are relevant. Quantum effects caused by superposition, entanglement, etc., are merely employed to increase the efficiency or security of the protocol [35].

There are many advantages in the use of superoperator weighted Markov chains for model checking purpose:

1. they provide a way to check, once for all, that a property is verified and it holds for all input quantum states (so no input preparation is needed, we can consider

a generic state). For example, for the reachability problem we calculate the accumulated super-operator \mathcal{E} along all valid paths. The *reachability* probability when the program is executed on the input quantum state ρ is simply the trace $tr(\mathcal{E}(\rho))$ of $\mathcal{E}(\rho)$

2. As the state space is usually finite, techniques from *classical model checking* can be adapted to verification of quantum systems [35].

Before we formally define quantum Markov chains, we introduce the following objects.

Let $\mathcal{S}(\mathcal{H})$ be the set of superoperators over a Hilbert space \mathcal{H} . $\mathcal{S}^{\mathcal{I}}(\mathcal{H})$ is the set of trace-nonincreasing superoperators over \mathcal{H} , i.e.:

$$\mathcal{S}^{\mathcal{I}}(\mathcal{H}) = \{\mathcal{E} \in \mathcal{S}(\mathcal{H}) \mid 0_{\mathcal{H}} \lesssim \mathcal{E} \lesssim \mathcal{I}_{\mathcal{H}}\} \quad (4.1)$$

with $0_{\mathcal{H}}$ and $\mathcal{I}_{\mathcal{H}}$ identify the null and identity superoperators respectively. Since $\mathcal{E} \in \mathcal{S}^{\mathcal{I}}(\mathcal{H}) \Leftrightarrow tr(\mathcal{E}(\rho)) \in [0, 1] \forall \rho$ it is natural to regard the set $\mathcal{S}^{\mathcal{I}}(\mathcal{H})$ as the quantum counterpart the domain of traditional probabilities [35].

Let $\mathcal{E}, \mathcal{F} \in \mathcal{S}(\mathcal{H})$, we say that the relation $\mathcal{E} \lesssim \mathcal{F}$ holds if, for any quantum state ρ over \mathcal{H} , $tr(\mathcal{E}(\rho)) \leq tr(\mathcal{F}(\rho))$, the probability of performing \mathcal{E} is always less or equal to the probability of performing \mathcal{F} . The relation $\approx ::= \lesssim \cap \gtrsim$ is the equivalence one.

Since we are interested in computing the accumulated superoperators along the paths in the QMC, we define a *superoperator-valued-measure* (herein SVM), i.e., an instance of POVM in which positive operators are replaced by superoperators. Given a non-empty set Ω and a partition $\{A_i\}$ over it, an SVM $F(\Omega)$ is a map which satisfies the following conditions:

$$\begin{aligned} F(\Omega) &\approx \mathcal{I}_{\mathcal{H}} \\ F\left(\bigcup_i A_i\right) &\approx \sum_i F(A_i) \end{aligned} \quad (4.2)$$

Definition 4.4.1. (Quantum Markov Chain [35]) A superoperator weighted Markov chain K , also referred to as quantum Markov chain (herein QMC), over a Hilbert space \mathcal{H} is a tuple $K = (S, Q, AP, L)$, where:

1. S is a countable (finite) set of classical states;
2. $Q: S \times S \rightarrow \mathcal{S}^{\mathcal{I}}(\mathcal{H})$; is the transition matrix, where for each $s \in S$, the superoperator $\sum_{t \in S} Q(s, t)$ is trace-preserving;
3. AP is a finite set of atomic propositions
4. $L: S \rightarrow 2^{|AP|}$ is a labelling function.

A QMC could be seen as a discrete time Markov chain where classical probabilities are replaced with quantum probabilities, i.e., the entries of the transition matrix are replaced by superoperators. We denote with $\pi(i)$ the i -th state of a path π , and with $Path^K(s)$ the set of all paths in K starting from the state s . Conversely, we denote with $\hat{\pi}(i)$ the i -th state of a *finite* path $\hat{\pi}$, and with $Path^K_{fin}(s)$ the set of all finite paths in K starting from s . In order to perform model checking on QMC K , we have to determine

the accumulated superoperators along its paths by means of a SVM Q_s built in the following way:

1. We define a superoperator $\mathcal{Q}(\hat{\pi})$:

$$\mathcal{Q}(\hat{\pi}) ::= \begin{cases} \mathcal{I}_{\mathcal{H}} & n = 0 \\ \mathbf{Q}(s_{n-1}, s_n), \dots, \mathbf{Q}(s_{n-1}, s_n) & \text{otherwise} \end{cases} \quad (4.3)$$

for all $\hat{\pi} \in \text{Path}_{fin}^K(s)$.

2. We define the cylinder set of all infinite paths with prefix $\hat{\pi}$ as:

$$C(\hat{\pi}) ::= \{\pi \in \text{Path}^K(s) : \hat{\pi} \text{ is a prefix for } \pi\} \quad (4.4)$$

3. We define $\mathcal{S}^K(s) ::= \{C(\hat{\pi}) : \hat{\pi} \in \text{Path}_{fin}^K(s)\} \cup \{\emptyset\}$.

4. From the above definition we build an SVM $Q_s : \mathcal{S}^K(s) \rightarrow \mathcal{S}^{\mathcal{I}}(\mathcal{H})$ where:

$$\begin{aligned} Q_s(\emptyset) &= 0_{\mathcal{H}} \\ Q_s(C(\hat{\pi})) &= \mathcal{Q}(\hat{\pi}) \end{aligned} \quad (4.5)$$

Example 4.4.1. As an example, in order to show the behaviour of the objects presented above, let us consider a simple pseudocode implementing the most basic instance of quantum coin tossing: given a quantum state initialised to $|0\rangle$, as a first step it is put in an uniform superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ by the application of an Hadamard gate. Then, a projective measurement is performed onto the uniform superposition, holding an equal probability of ending in the state $|0\rangle$ or $|1\rangle$.

We can rewrite the states, the unitary operators and the projection ones by using the density matrix formalism as follows:

$$\begin{aligned} |0\rangle &\rightarrow |0\rangle\langle 0| = q_0 \\ H|0\rangle &\rightarrow \mathcal{E}_H(q_0) \\ M_0 H|0\rangle &\rightarrow \mathcal{M}_0(\mathcal{E}_H(q_0)) = |0\rangle\langle 0| \\ M_1 H|0\rangle &\rightarrow \mathcal{M}_1(\mathcal{E}_H(q_0)) = |1\rangle\langle 1| \end{aligned}$$

with $\sum_{i=0}^1 M_i = \mathbb{I}$ and \mathcal{E}_H , \mathcal{M}_0 and \mathcal{M}_1 are the superoperators associated to H , M_0 and M_1 respectively.

In the pseudocode we add a loop; according to the instructions, if the final state is $|1\rangle$ the algorithm starts again from the initial state otherwise it terminates.

$$\begin{aligned} s_0 &: q_0 = \mathcal{E}_H(q_0) \\ s_1 &: \mathbf{if} \mathcal{M}_0(q_0) \\ s_2 &: \mathbf{then} \text{ end else} \\ s_3 &: \text{restart} \end{aligned} \quad (4.6)$$

The QMC associated is shown in Fig. 4.1

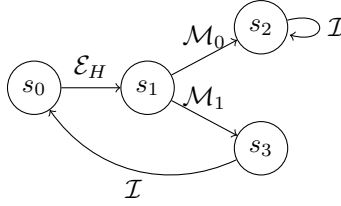


Figure 4.1: QMC associated to the Quantum Coin Tossing.

The resulting quantum Markov chain $K = (S, \mathbf{Q}, AP, L)$ is described by:

1. $S = \{s_i : 0 \leq i \leq 3\}$;
2. $AP = S$;
3. $L(s_i) = \{s_i\}$;
4. $\mathbf{Q}(s_i, s_j)$ is the following:

$$\mathbf{Q}(s_0, s_1) = \mathcal{E}_H, \quad \mathbf{Q}(s_1, s_2) = \mathcal{M}_0, \quad \mathbf{Q}(s_1, s_3) = \mathcal{M}_1, \quad \mathbf{Q}(s_2, s_2) = \mathcal{I}, \quad \mathbf{Q}(s_3, s_0) = \mathcal{I}.$$

Given a finite path $\hat{\pi} = s_0 s_1 s_3 s_0 s_1 s_2$, the accumulated superoperator along $\hat{\pi}$ is:

$$\mathbf{Q}(\hat{\pi}) = \mathbf{Q}(s_1, s_2) \mathbf{Q}(s_0, s_1) \mathbf{Q}(s_3, s_0) \mathbf{Q}(s_1, s_3) \mathbf{Q}(s_0, s_1) = \mathcal{M}_0 \mathcal{E}_H \mathcal{I} \mathcal{M}_1 \mathcal{E}_H$$

4.4.1 The QPMC model checker

QPMC is a model checker for quantum programs and protocols based on the density matrix formalism available in both web-based and off-line version¹. It is an extension to QMCs of the ISCASMC model checker, which has been enriched with the data structures for matrices and superoperators. Since QMCs show a different behaviour than classical discrete time Markov chains, some aspects must be taken into consideration; e.g., non-commutativity of superoperator multiplication.

The specification language is an extension of the guarded command language PRISM [58] allowing, in addition to the usual constants definable in PRISM, the specification of types `vector`, `matrix`, and `superoperator`. QPMC supports the bra-ket notation and inner, outer and tensor product can be written using it. For example $|v\rangle_n$ denotes a vector in \mathcal{H}^n .

QPMC provides the following predefined functions and commands:

Computational-basis states: $|0\rangle_n, \dots, |n-1\rangle_n \in \mathcal{H}^n$

i.e., n -dimensional vectors with all the entries being 0s but one, which is the number 1 appearing in the $i + 1$ -th position as follows:

¹At the time of writing, the web version is accessible at the address <http://iscasmc.ios.ac.cn/too/qmc>.

$$|i\rangle_n = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} \leftarrow i + 1 - th$$

Inner-product: $\langle 0|1\rangle_2$ stands for $\langle 0|1$

Outer-product: $|0\rangle_2 \langle 1|_2$ stands for $|0\rangle\langle 1|$

Tensor-product: $|0\rangle_2|1\rangle_2$ stands for $|0\rangle \otimes |1\rangle$

Amplitude-damping-operators: $E_0 E_1$

Unitary-matrices: PX, PY, PZ (Pauli matrices), HD (Hadamard), PS (phase-shift), measurement operators with respect to the standard computational basis, M_0 and M_1 , controlled-not matrix CN, swap matrix SW, Toffoli matrix TF and identity ID(n) with $n \in \mathbb{N}$.

Matrices can be defined directly in a Matlab-like syntax, e.g., `const matrix mymatrix = [1,2;3,4]` whose corresponding matrix is:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Superoperators can either be defined as their combination of matrices e.g., `<<[1,0; 0,0], [0.5,0.5; 0,0]>>` or directly as a single matrix, using the command `mf2so`.

Other commands:

- `conj(...)`: conjugation of an imaginary number;
- `ctran(...)` = `conj(tran(...))`: conjugation of a transposed matrix;
- `kron`: Kronecker operator;
- `log`: logarithm;
- `matrix`: for **const** definitions (`const matrix ...`);
- `trans(...)`: transpose a matrix;
- `vector`: for **const** definitions;
- `*`: not currently used, but would be used for entrywise multiplication if needed;
- `mf2so`: matrix to superoperator;

- superoperator: for `const << ... >>`;
- i, j : imaginary numbers, e.g. $2+4i, 2+4j$;
- `identity(...)`: identity matrix of a given dimension;
- `ID(...)`: identity matrix of a given dimension.

4.4.2 QPMC module

The model is specified in the *module* environment. It allows the initialisation of a state variable s , and of several guarded commands representing the system transitions. As in PRISM, each guarded command has a precondition, followed by the possible state updates. The only difference is that each update is associated with a superoperator instead of a probability. We always omit the identity superoperators along the updates [35].

Example 4.4.2. Let's consider the quantum coin flipping pseudocode in Example 4.4.1. To translate it into a QPMC model the following steps are needed:

1. declaration of the model type, i.e., QMC;
2. definition of the constants, i.e., in our case we define a vector $q_0 = |0\rangle\langle 0|$;
3. definition of the module in which, for each step we have the following syntax:
`guard → superoperator : next state.`

Thus, the resulting code would be:

```
qmc
const matrix q0= |0>_2 <0|_2;

module coinflipping
s : [0..3] init 0;
[] (s=0) -> <<H>> : (s'=1);
[] (s=1) -> <<M0>> : (s'=2) + <<M1>> : (s'=3);
[] (s=2) -> (s'=2);
[] (s=3) -> (s'=0);
endmodule
```

where the operator `+` corresponds to a nondeterministic choice.

We should note that the quantum state q_0 has not been used inside the module, which only shows the application of the desired superoperator given the classical state s_i of the QMC. The quantum state will be used in the following, in the properties specification phase.

4.4.3 QCTL

The aim of QPMC is to provide a formal framework where to define and analyse properties of quantum protocols. The properties to be verified over QMC are expressed using the quantum computation tree logic (QCTL), a temporal logic for reasoning about evolution of quantum systems introduced in [36] that is a natural extension of PCTL.

Definition 4.4.2 (QCTL syntax [36, 35]). Given a state formula f and a path formula g , a QCTL formula is defined over the following grammar:

$$\begin{aligned} f &::= a \mid \neg f \mid f_1 \wedge f_2 \mid \mathbb{Q}_{\sim\epsilon}[g] \\ g &::= X f \mid f_1 U^{\leq k} f_2 \mid f_1 U f_2 \end{aligned}$$

where $a \in AP$, $\sim \in \{\lesssim, \gtrsim, \approx\}$, $\mathcal{E} \in \mathcal{S}^{\mathcal{I}}(\mathcal{H})$, $k \in \mathbb{N}$.

The quantum operator formula $\mathbb{Q}_{\sim\epsilon}[g]$ is a more general case of the PCTL probabilistic operator $\mathbb{P}_{\sim p}[g]$ and it expresses a constraint on the probability that the paths from a certain state satisfy the formula g . Besides the logical operators presented in QCTL, QPMC supports an extended operator $Q =?[g]$ to calculate the matrix representation of the superoperator satisfying g . We recall that the matrix representation of a superoperator is given as in the following equation:

$$M_{\mathcal{E}} = \sum_i E_i \otimes E_i^* \quad (4.7)$$

where E_i are the operation elements of \mathcal{E} .

In addition, QPMC provides a function $qeval((Q =?[g]), \rho)$ to compute the density operator obtained from applying the resultant superoperator on a given density operator ρ , and $qprob((Q =?[g]), \rho) = tr(qeval((Q =?[g]), \rho))$ to calculate the probability of satisfying g , starting from the quantum state ρ [35].

QCTL formulae are interpreted over QMCs. The semantics of QCTL is given as follows:

Definition 4.4.3 (QCTL semantics). Given a QMC K , two state formulae f_1 and f_2 and a path formula g , the relation $K, \tau \models \phi$ (which is read “the state/path formula ϕ holds at state/along a path τ ”), for all states $s \in S$ and for all paths $\pi \in K$ is interpreted as follows:

- $s \models \mathbb{Q}_{\sim\epsilon}(g) \iff Q^K(s, g) \sim \mathcal{E}$
 where $Q^K(s, g) = Q_s(\{\pi \in Path^K(s) \mid \pi \models g\})$.

Definition 4.4.4 (QCTL satisfaction set). Given a state formula f and a path formula g , the satisfaction set $Sat(f) = \{s \in S : s \models f\}$ is defined as follows:

$$Sat(\mathbb{Q}_{\sim\epsilon}[g]) = \{s \in S : Q^K(s, g) \sim \mathcal{E}\} \quad (4.8)$$

All the other operators and satisfiability relations are the same as CTL and PCTL.

An example of property specification, related to the model presented in Example 4.4.2 is given in Table 4.1.

Example 4.4.3. The first property holds if $s_0 \models \mathbb{Q}_{\geq \frac{1}{2}} \mathcal{I}$, i.e., if starting from the state s_0 , the probability of reaching the state s_2 is greater or equal to $1/2$, while the others return the matrix representation of the density operator associated to the states s_i , with $i = 0, 1, 2, 3$, given the initial quantum state q_0 as declared in the module. We note that the final states, after the measurement operators have been applied, are not automatically normalised.

QCTL Formula	Output	Trace
$\mathbb{Q} \geq (1/2) \ [F \ (s=2)]$;	true	
$\text{qeval}(Q=? \ [F \ (s=0)], \ q_0)$;	$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$	1
$\text{qeval}(Q=? \ [F \ (s=1)], \ q_0)$;	$\frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	1
$\text{qeval}(Q=? \ [F \ (s=2)], \ q_0)$;	$\frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$	$\frac{1}{2}$
$\text{qeval}(Q=? \ [F \ (s=3)], \ q_0)$;	$\frac{1}{2} \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$	$\frac{1}{2}$

Table 4.1: Example of Property Specification in QCTL.

III

Contribution

5

Entangle

During the first part of my Ph.D, we have analysed the state-of-art in the fields of quantum programming languages and model checking of quantum algorithms and we developed a tool for the verification of Quipper programs into the model checking system QPMC. In this way, we provided an ad-hoc verification tool for Quipper based on QPMC, which allowed us to simulate and make formal analysis on quantum protocols using a high-level language. Due to the different representation of both states and dynamics used by the two systems (i.e., Quipper and QPMC) we faced different issues in programming this unifying framework. A first output to this implementation has been used to verify non-recursive instances of quantum algorithms. Then, due to the expressive power of the languages used in the framework, we decided to extend our framework in order to deal with *tail-recursive* quantum circuits, providing in this way a more complete tool.

The Chapter is organised as follows: first, we present an abstract algorithm for translating Quipper circuits into quantum Markov chains, i.e., QPMC models. Then, we provide a description of our first implementation of the translation algorithm, together with some examples we developed and tested. The final Sections are devoted to describe **Entangle**, i.e., the extension of our framework. **Entangle** allows to translate also tail-recursive Quipper programs, which have been tested using tail-recursive versions of quantum algorithms and protocols.

5.1 From Circuits to Quantum Markov Chains

In order to be able to define a mapping from Quipper to QPMC programs, in this Section we work at the semantic level. This means that we consider a quantum circuit generated by Quipper in order to define a correspondent QMC having an *equivalent* behavior.

5.1.1 Circuits

First, we need a formal definition of quantum circuits generated from Quipper. Even though Quipper supports also classical wires, here we focus on circuits over quantum ones. As in Quipper, we consider only measurements of one qubit at a time with respect to the standard computational basis. We assume the reader to be familiar with the classical notions of graphs and Boolean circuits. Given a node v of a directed graph we use the notations $In(v)$, and $Out(v)$ to denote the number of edges incoming and outgoing in v . A quantum circuit is an extension of a Boolean circuit, where operation gates are labeled with unitary operators. When a unitary operator is applied to k qubits it is necessary to know in which order the qubits are used; for this reason each edge of a quantum circuit has two integer labels associated.

Definition 5.1.1 (Quantum Circuit). A Quantum Circuit is a directed acyclic graph (herein DAG) $C = (V, E)$ whose nodes, also called *gates*, are of types *Qubit* (Q), *Unitary* (U), *Measurement* (M) and *Termination* (T) and satisfy the following conditions:

1. Q gates: each node v of type *Qubit* is an input node, i.e. $In(v) = 0$ and $Out(v) = 1$;
2. U gates: each node v of type *Unitary* is labelled with an integer $dim(v)$ and a square unitary matrix $U(v)$ of complex numbers of dimension $2^{dim(v)}$. Moreover, it holds that $In(v) = Out(v) = dim(v)$;
3. M gates: each node v of type *Measurement* is an output node, i.e. $In(v) = 1$ and $Out(v) = 0$;
4. T gates: each node v of type *Termination* is an output node, i.e. $In(v) = 1$ and $Out(v) = 0$.
5. Edges: each edge $e \in E$ is labelled with two integers $\mathcal{S}(e)$ and $\mathcal{T}(e)$ such that:
 - for each node u , the set of labels $\mathcal{T}(\cdot)$ of the edges ingoing in u is $\{1, \dots, In(u)\}$;
 - for each node u , the set of labels $\mathcal{S}(\cdot)$ of the edges outgoing from u is $\{1, \dots, Out(u)\}$.

A Quantum Circuit with k nodes of type *Qubit* is said to have *size* k .

Example 5.1.1. Let us consider the following Quipper function implementing Deutsch's algorithm.

```
deutsch :: (Qubit, Qubit) -> Circ Bit
deutsch (q1, q2) = do
  hadamard q1
  hadamard q2
  qnot_at q2 'controlled' q1
  hadamard q1
  measure q1
```

Quipper graphically represents the circuit as shown in Figure 5.1.

Our definition enriches the above representation with labels denoting the order in which the qubits are used, as depicted in Figure 5.2.

Definition 5.1.2 (Circuit Normal Form). A Quantum Circuit of size k is said to be in *Normal Form* if each *Unitary* node v in the circuit has $dim(v) = k$.

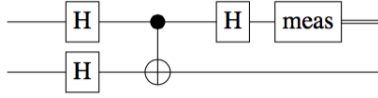


Figure 5.1: Deutsch circuit in Quipper

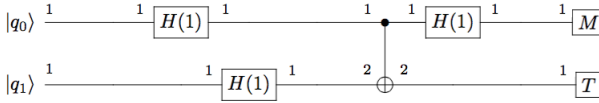


Figure 5.2: Deutsch circuit with labels

Definition 5.1.3 (Strong Normal Form). A Quantum Circuit C of size k is said to be in *Strong Normal Form* (herein SNF) if C is in Normal Form, for each edge $e \in E$ between two Unitary nodes $\mathcal{S}(e) = \mathcal{T}(e)$ holds and the first $h \leq k$ edges outgoing the last Unitary node enter into Measurement nodes.

A circuit C in SNF is completely specified by the tuple $\langle k, [U_1, \dots, U_n], h \rangle$ where k is the size of C , U_1, \dots, U_n are the Unitary operators in the order they occur in C , and h is the number of Measurement nodes.

In Figure 5.3 we can see that in a circuit in Normal Form the order of the labels on the edges is not preserved. This is due to the fact that many gates are applied to a permutation of the input qubits. On the contrary, a circuit in SNF requires a precise ordering of the input and output edges. We will see that in order to match this requirement, SWAP operators have to be added.

We now need a notion of equivalence between quantum circuits. This will allow us to move from a generic quantum circuit to a SNF circuit. Intuitively, two quantum circuits are equivalent if, for any k -tuple of initial values of the qubits, the values of the qubits before measurements/terminations are the same. Moreover, in order to be equivalent, two circuits need to give the same outputs with the same probabilities. Formally, let C be a Quantum Circuit of size k , we denote by $Sem(C)$ the pair of functions $(F(C), M(C))$ where:

- $F(C) : \mathcal{H}^k \rightarrow \mathcal{H}^k$ is the function which maps k qubits to the value they have just before the Measurement and Termination nodes;
- $M(C) : \mathcal{H}^k \times \{0, 1\}^h \rightarrow [0, 1]$ is the function such that $M(C)(|\psi\rangle, (b_1, \dots, b_n))$ is the probability of getting output $(b_1, \dots, b_n) \in \{0, 1\}^h$ on input $|\psi\rangle$.

Notice that if $C = \langle k, [U_1, \dots, U_n], h \rangle$ is a SNF circuit, then $F(C)(|\tau\rangle) = U_n \dots U_1 |\tau\rangle$.

Definition 5.1.4 (Quantum Circuit Equivalence). Given two Quantum Circuits C_1 and C_2 of size k , then C_1 and C_2 are equivalent, denoted by $C_1 \approx C_2$ if and only if $Sem(C_1) = Sem(C_2)$.

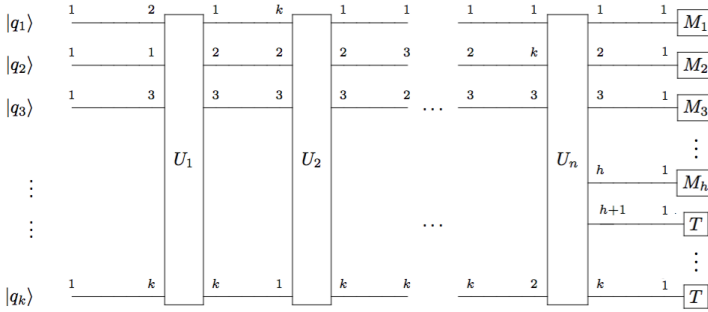


Figure 5.3: Example of a circuit in Normal Form

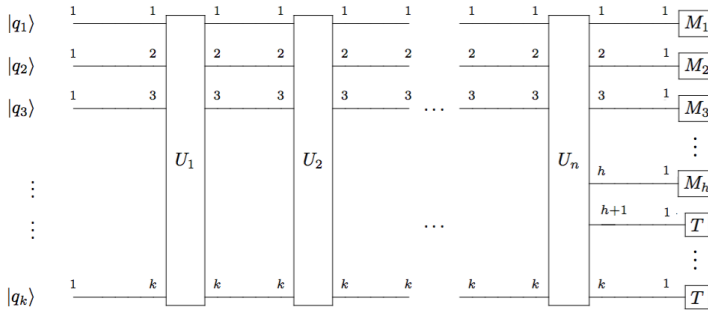


Figure 5.4: Example of a circuit in Strong Normal Form

Lemma 5.1.1. Every Quantum Circuit is equivalent to a circuit in Normal Form.

Proof. Let C be a Quantum Circuit of size k . C is a DAG so it admits a *topological ordering* of its nodes. Qubit nodes do not have any incoming edge so we choose an ordering in which the first nodes are all the ones of type Qubit. Measurement and Termination nodes do not have any outgoing edge so we choose them as final nodes in the ordering. The nodes in between initial and final nodes are only the one of type Unitary. We will proceed by induction on the number n of Unitary nodes $\{U_1, \dots, U_n\}$.

Base case: For $n = 1$ our circuit has only one Unitary gate. If $\dim(U_1) = k$ then the circuit is in Normal Form.

If $\dim(U_1) = h < k$ we replace U_1 with the node $\tilde{U}_1 = U_1 \otimes I$ where $\dim(I) = k - h$ and then we append the remaining $k - h$ edges.

Induction step: If $n > 1$ by induction we know that we can normalise the first $n - 1$ Unitary gates and we proceed as in the base case on the last one. \square

Lemma 5.1.2. Every Quantum Circuit in Normal Form is equivalent to a circuit in Strong Normal Form.

Proof. In order to prove our thesis we need a notion of generalised *SWAP* gate. The *SWAP* gate takes in input two qubits and swaps them, i.e., $SWAP|x, y\rangle = |y, x\rangle$. A *generalised SWAP* gate is an operator acting on k qubits that returns in output a permutation p_i of them. It is possible to build such operators by combining sequentially two-dimensional *SWAP* gates. Using the definition of generalised *SWAP* gates the proof is straightforward. Given a quantum circuit in Normal Form of size k , we obtain a circuit in SNC by opportunely swapping the Qubit indexes after the application of a unitary gate. \square

5.1.2 From Strong Normal Form Circuits to QMC's

We are now ready to define the QMC associated to a circuit in SNF. Intuitively, the states of the QMC correspond to the edges of the circuit, while the edges of the QMC connect subsequent states. Moreover, states without outgoing edges are added in the QMC to represent all the possible outputs of the circuit.

Definition 5.1.5 (QMC associated to a Circuit). Let C be a Quantum Circuit in SNF of size k with n Unitary nodes $\{U_1, \dots, U_n\}$ and h Measurement nodes, the QMC Q_C associated to C is defined as follows:

- the k -tuple of edges of C entering the Unitary node U_i is associated to the state s_i in Q_C ;
- the k -tuple of edges outgoing from the last Unitary node U_n is associated to the state s_{n+1} ;
- in Q_C there are 2^h states $t_0, t_1, \dots, t_{2^h-1}$;
- for each $i \in \{1, \dots, n\}$ there is an edge from s_i to s_{i+1} labelled with the superoperator $SO(U_i)$ associated to the Unitary gate U_i ;
- for each $i \in \{0, \dots, 2^h - 1\}$ there is an edge from s_{n+1} to t_i labelled with the superoperator $\widetilde{M}_i = M_i^h \otimes I^{k-h}$, where I^{k-h} is the identity matrix of size 2^{k-h} and M_i^h is a matrix of size 2^h having 1 in the $i + 1$ -th position and all 0's in the remaining.

In Figure 5.5 we can see the QMC associated to the example circuit shown in Figure 5.4.

Lemma 5.1.3. Given a quantum circuit C in SNF we can always build the QMC Q_C associated to C and it holds that:

1. $\forall |\tau\rangle \in \mathcal{H}, \forall i \in \{1, \dots, n\}$,

$$U_i|\tau\rangle = |\psi\rangle \quad \text{iff} \quad SO(U_i)|\tau\rangle\langle\tau|SO(U_i)^\dagger = |\psi\rangle\langle\psi|$$

2. $\forall |\tau\rangle \in \mathcal{H}$ if $F(C)(|\tau\rangle) = |\psi\rangle$ and $M(C)(|\tau\rangle, \{b_1, \dots, b_h\}) = p$, with $m = \text{bin}(b_1 \dots b_h)$ (i.e., the natural with binary expansion $b_1 \dots b_h$) then:

$$p = \text{tr}(\widetilde{M}_m|\psi\rangle\langle\psi|\widetilde{M}_m^\dagger)$$

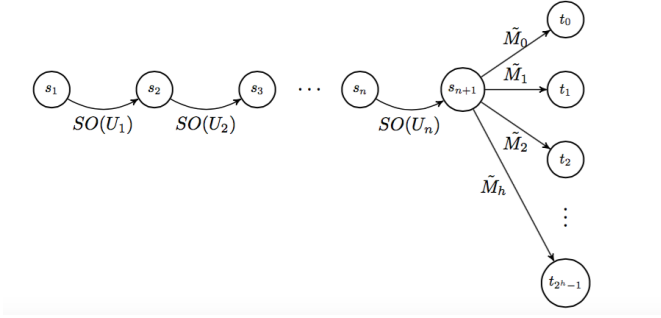


Figure 5.5: QMC associated to the circuit of Figure 5.4.

and

$$\widetilde{M}_m |\psi\rangle \langle \psi | \widetilde{M}_m^\dagger = |b_1, \dots, b_h \dots\rangle \langle b_1, \dots, b_h \dots|$$

Proof. It immediately follows from our definitions. \square

The above lemma states an equivalence between the semantics of a circuit C in SNF and its associated QMC. Thus, any Temporal Logic coherently defined on both formalisms can be equivalently model checked either on the circuit or on its associated QMC.

5.1.3 Translation Algorithm

The results described in the previous sections allow us to define an algorithm that maps a quantum circuit into an *equivalent* QMC. In particular, Algorithm Translate performs the following steps:

- it transforms a quantum circuit into a normal form circuit (see Lemma 5.1.1);
- it transforms a normal form circuit into a SNF circuit (see Lemma 5.1.2);
- it transforms a SNF circuit into its corresponding QMC (see Definition 5.1.5).

Hence, given a quantum circuit C the output of $\text{Translate}(C)$ is a QMC equivalent to C in the sense of Lemma 5.1.3.

The computational complexity of $\text{Translate}(C)$ depends on the number n of Unitary nodes occurring in C and on its size k . For each Unitary we need to perform a number of binary swaps which depends on k . Without any efficient strategy in the worst case we could perform $\Theta(k^2)$ binary swaps. Hence, $\text{Translate}(C)$ generates a QMC having $O(n * k^2)$ internal nodes. Each of this step requires the computation of a matrix of size 2^k . However, we can lower the complexity of the algorithm by directly implementing generalized swaps without relying on binary ones. Such optimization would generate a QMC having at most $O(n)$ internal nodes, requiring the computation of $O(n)$ swap matrices.

5.2 Implementation

In Section 5.1 we presented an abstract algorithm that translates a quantum circuit into a QMC. We now describe an implementation of the Translation Algorithm in which the input quantum circuit is a Quipper function in the `Circ` monad and the output QMC is a QPMC model.

Our implementation exploits the `Transformer` module of Quipper. `Transformer` is a library which provides functions for defining general purpose transformations on low-level circuits, and it works at data structure level. By using the `Transformer` module we can re-use Quipper's code, avoiding to implement the instructions again in an intermediate language.

The actual translation can be summarised in three steps. As a first step the circuit is decomposed; the gates in the quantum circuit are grouped together with their associated qubits, taking care that the execution order is preserved. The function implementing this decomposition return a tuple (*integer-list*) where the integer represents the index of the current time-step and the list contains the name of the gates acting in that interval, together with their associated qubits. Those indexes will become the values associated to the state variable s in the QMC. In this way we have an abstract representation of both the states and the transitions of the QMC.

Since the gates are represented by labels, i.e., without a proper numerical representation as required by QPMC, as a second step we calculate the matrix representation of the quantum gates. We implemented a set of user-defined functions useful to perform operations on matrices (e.g., the tensor product). It is important to note that, since we need a circuit in SNF, our code provides a set of functions that generate the required swaps using compositions of binary swaps realising a permutation by transposition. Then our algorithm takes the resulting matrix and associates it to the gate input qubits, while the identity matrix is associated the remaining ones. Finally, the qubits are moved back in their original positions. All the matrices are computed in MATLAB notation.

The last step is the conversion of the list of transitions into QPMC code. All these functions have been written in order to be kept as polymorphic as possible.

The following is a toy example of Quipper code given in input to our tool:

```
oneq :: Qubit -> Circ Qubit
oneq q = do
  hadamard_at q
  return q
```

In Figure 5.6 we show the intermediate result of our translation. In this example the time-step is the transition from $|\psi_0\rangle$ to $|\psi_1\rangle$

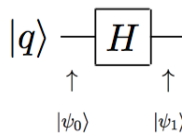


Figure 5.6: Quantum circuit with intermediate states.

We can see the resulting QMC in Figure 5.7, in which the $|\psi_i\rangle$ become state variables s_i and the operators in the stratum will label the transitions between states.

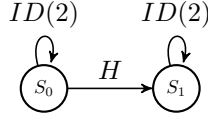


Figure 5.7: Quantum Markov Chain for the Quipper circuit

The resulting QPMC code is:

```
qmc
module oneq
const matrix A0 = [1/(sqrt(2)), 1/(sqrt(2));
1/(sqrt(2), -(1)/(sqrt(2))];
s: [0..1] init 0;
[] (s = 0) -> <<A0>> : (s' = 1)
[] (s = 1) -> (s' = 1)
endmodule
```

Example 5.2.1. In order to show a more realistic example, let us consider again the Quipper's function for the Deutsch's Algorithm presented in Example 5.1.1. It can be compiled in Quipper generating the circuit represented in Figure 5.1. The intermediate step of the translation produces the following code:

```
1: Gate "H" [] [2], Gate "H" [] [1]
2: Gate "not" [1] [2]
3: Gate "H" [] [1]
4: Measure 1
```

which allows to our implementation to convert the Deutsch's Quipper code into the QPMC model below.

```
qmc
const matrix A1 = [(1 / 2), (1 / 2), (1 / 2), (1 / 2); (1 / 2), (-1 / 2), (1 / 2), (-1 / 2)
; (1 / 2), (1 / 2), (-1 / 2), (-1 / 2); (1 / 2), (-1 / 2), (-1 / 2), (1 / 2)];
const matrix A2 = [1,0,0,0;0,1,0,0;0,0,0,1;0,0,1,0];
const matrix A3 = [(sqrt(2) / 2),0,(sqrt(2) / 2),0;0,(sqrt(2) / 2),0,(sqrt(2) / 2);(
sqrt(2) / 2),0,((-1 * sqrt(2)) / 2),0;0,(sqrt(2) / 2),0,((-1 * sqrt(2)) / 2)];
const matrix A4 = [1,0,0,0;0,1,0,0;0,0,0,0;0,0,0,0];
const matrix A5 = [0,0,0,0;0,0,0,0;0,0,1,0;0,0,0,1];
module test
s: [0..5] init 0;
[] (s = 0) -> <<A1>> : (s' = 1);
[] (s = 1) -> <<A2>> : (s' = 2);
[] (s = 2) -> <<A3>> : (s' = 3);
[] (s = 3) -> <<A4>> : (s' = 4) + <<A5>> : (s' = 5);
[] (s = 4) -> (s' = 4);
[] (s = 5) -> (s' = 5);
endmodule
```

Notice that, differently from what we wrote in our definition of QMC associated to a circuit, in the implementation we do not distinguish states s_i 's from states t_i 's in the generated QPMC model.

5.2.1 Scalability of the swap algorithm

We performed some scalability tests on an artificial example which requires a high number of swaps. Recall that, since we need the circuit to be translated in SNF, for each Unitary gate we need to perform a number of binary swaps depending on the number k of qubits used in the circuit. In this part of the experiment we focused on the execution time of our implementation, i.e., the time required to produce the QPMC model. The circuits given in input have been chosen to maximize the number of binary swaps required by our implementation. An example of such circuits of size 7 can be seen in Figure 5.8.

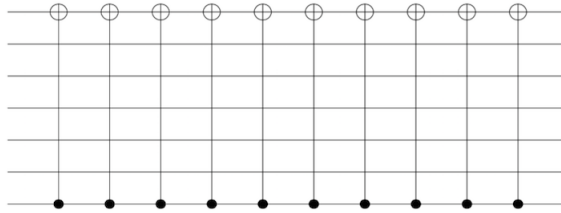


Figure 5.8: Test circuit of size 7

We decided to test circuits built using from 3 to 7 qubits. In the first version of `Entangle`, times were recorded using the `time` utility of the Bash shell on an early 2014 MacBook Air with a 1.4 GHz Intel Core i5 processor. For each size of the input, the program has been executed five times and the mean time has been computed. The results are shown in Figure 5.9. We can see that also for a circuit of size 8, when we have to generate swap matrices of size $2^7 \times 2^7$ our algorithm works in *reasonable* times.

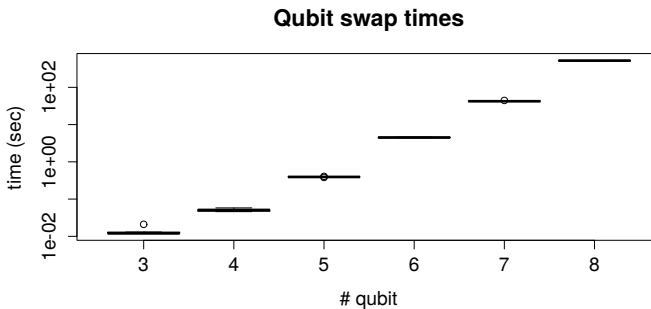


Figure 5.9: Scalability test.

5.3 Extension to Tail-recursive Quantum Programs

The first version of our tool provides a translation from a quantum circuit to a QMC. This translation is limited to circuits, where only quantum controls are used. On the one hand, since Quipper is an expressive language, it allows to write both classical and quantum *programs*, instead of simple circuits, by using control structures e.g., conditional branches and recursion. In this direction, our aim has been to extend the framework in order to allow the translation of quantum programs as well. On the other hand, QPMC allows to build QMCs where loops are present, while our tool only permits linear circuits in which no loops occur. To overcome the limitations of the first version of our translator we decided to extend it by allowing the translation of code in which both classical controls and loops are present. In particular, we want to write *quantum programs* in which the circuit is conditioned on the result of intermediary measurements [84]. In this way, we allowed the translation of programs in which there are loops, with the conditional branches depending on the measurements outcomes, i.e., tail-recursive quantum programs. In order to extend our framework to a larger class of programs, we restricted Quipper to an *ad hoc* sublanguage, called *Quip-E*, which allows the use of reset operators, unitary and measurement gates, and tail-recursion.

5.3.1 *Quip-E*: a Quipper recursive fragment

We briefly recall that in Quipper classical controls can be applied both outside and inside the `Circ` monad. In the extension of the work presented in [5] we were interested in classical operations and controls inside the `Circ` monad allowing to define also *recursive functions*. In Example 5.3.1 we show part of the code of a Quipper recursive version of the quantum Fourier transform in the `Circ` monad (see [43]).

Example 5.3.1. The function `qft'` computes the quantum Fourier transform of a list of qubits. If the list is empty, the circuit itself is empty. If the input is a list of one qubit, then the Hadamard gate is applied. The circuit for a list of $n + 1$ qubits applies the circuit for n qubits to the last n elements of the list, followed by a set of rotations over all $n + 1$ qubits.

```
qft' :: [Qubit] -> Circ [Qubit]
qft' [] = return []
qft' [x] = do
  hadamard x
  return [x]
qft' (x:xs) = do
  xs' <- qft' xs
  xs'' <- rotations x xs' (length xs')
  x' <- hadamard x
  return (x':xs'')
where ...
```

Quipper allows the use of Boolean operators and `if-then-else` statements with tests performed on Boolean parameters. The `dynamic_lift` operator converts a bit to a Boolean parameter. Hence, the result of a measurement over a qubit can be stored in a bit, and then converted to a Boolean and used as guard in a test. Moreover, Boolean parameters can be used to initialize qubits through the `qinit` operator. In Example 5.3.2 we show how these can be combined inside a simple recursive Quipper circuit.

Example 5.3.2. In the following example we show a recursive instance of the quantum coin flipping: a qubit is initialized to $|0\rangle$, then the Hadamard gate is applied to it and it is measured. If the outcome is 0, i.e. the value associated to the state $|0\rangle$, then the circuit is re-initialized, otherwise it terminates. This is repeated until the result of the measurement is 1. Hence, the circuit halts after an unpredictable number of iterations, returning the qubit $|1\rangle$.

```
coinFlipCirc :: (Qubit) -> Circ (Qubit)
coinFlipCirc (q) = do
  q <- qinit[False]
  hadamard_at q
  m <- measure q
  bool <- dynamic_lift m
  if bool
  then
    return (q)
  else
    coinFlipCirc (q)
```

In the above example the circuit is tail-recursive. In classical computation tail recursion corresponds to **while-loops** which together with concatenation of instructions, assignments, increments, and comparisons give rise to a Turing-complete formalism. In the case of quantum circuits tail-recursion is the most natural form of recursion that we can imagine. A sequence of unitary gates is applied, the result is measured over some qubits and such result is used to decide whether to stop or repeat the circuit.

Example 5.3.3. Let us consider a circuit `Black_Box` that on input $|0\rangle$ returns the desired output bit with probability $p > 0.5$. We could construct the following tail-recursive circuit.

```
double_Black_Box :: (Qubit) -> Circ Bit
double_Black_Box(q) = do
  q <- qinit[False]
  b1 <- black_Box(q)
  q<- qinit[False]
  b2 <- black_Box(q)
  bool1 <- dynamic_lift(b1)
  bool2 <- dynamic_lift(b2)
  if (bool1=bool2)
  then
    return(b1)
  else
    double_Black_Box(q)
```

The probability that the new circuit terminates with a wrong answer is $(1 - p)^2$ which is smaller than the probability that the original circuit terminates with a wrong answer.

The fragment of Quipper we are interested in is basically the `Circ` monad where we allow tail-recursion. In particular, we allow the use of the initialization operator `reset`, of unitary operators, and of measurements, and we call this sublanguage *Quip-E*. The results of measurements can be lifted to Boolean values and used inside a guard condition to decide whether to terminate the circuit or to restart it. *Quip-E* uses the following syntax to specify the tail-recursive programs. The *body* `Body_C` of a tail-recursive program `trc_C` is defined by the following grammar:

```

Body_C ::= reset_at q | U_at [qi1, ..., qij] | m <- measure q |
         bool <- dynamic_lift m | if (bool) Body_C1 else Body_C2 |
         Body_C1 Body_C2

```

where q , q_{i_1}, \dots, q_{i_j} are qubits that occur as formal parameters of the program, U_{at} is a unitary operator of dimension j , m is a bit variable name, b is a Boolean parameter.

Remark. By nesting `if-then-else` constructors it is possible to mimic conditions that depend on any possible Boolean combination of sets of Boolean parameters. Hence, in the formal definition of the language we omit Boolean combinations without losing expressive power, while our compiler of *Quip-E* allows their use to ease the programming task.

Hence, the *Quip-E* syntax can be informally summarised as a sequence consisting of a finite list of instructions taken from the following set:

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

A general *tail-recursive program* `trc_C` has the form

```

trc_C :: (Qubit, Qubit, ...) -> Circ RecAction
trc_C (q1, q2, ...) = do
  -- beginning of body
  Body_C
  -- end of body
  exitOn bool

```

where q_1, q_2, \dots are the qubits occurring in `Body_C` and `bool` is a Boolean parameter occurring in `Body_C`. In this case we say that `trc_C` is the *Quip-E* program defined by the body `Body_C` and the exit condition `exitOn bool`.

We impose that whenever a Boolean parameter `bool` is used as a guard of `if-then-else` and `exitOn` constructors, its value has been previously defined in the body (e.g., through a `measure` instruction followed by a `dynamic_lift`).

Intuitively, the execution of `Body_C` is repeated until `bool` becomes true. It is possible to define programs which are not recursive by simply defining `bool` as `True`.

Remark. Non recursive programs can be defined as recursive ones using exit conditions that are always true. Hence, we omit them in the formal definition of *Quip-E* even if our compiler allows their explicit use.

Example 5.3.4. The following is a small example of a *Quip-E* program. In the program two qubits are initialized to $|0\rangle$ and $|1\rangle$, respectively, then Hadamard is applied to the second one, after the second qubit is measured and the result of the measurement is used both to decide which gate has to be applied to the first qubit and whether the program has to loop or terminate.

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

The same program can be written in Quipper native formalism as follows:

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

Notice that the `reset` function is a way to provide a unitary operator for the `qinit` instruction. In particular, instruction `reset_at q` in *Quip-E* is equivalent to the Quipper instruction `q <- qinit False`, which initializes the qubit to $|0\rangle$. If `reset_at q` is followed by the application of a `not` gate on `q` (e.g., `gate_X_at q`), then the sequence of two instructions of *Quip-E* is equivalent in Quipper to `q <- qinit True`, which initializes the qubit to $|1\rangle$.

5.4 Structural Operational Semantics for *Quip-E*

In this section we show how the translation of each component of `Body_C` is formalized. In particular, for each instruction we provide a structural operational semantics (S.O.S) of *Quip-E* programs in terms of QMCs. Intuitively, a transition system through S.O.S defines the operational rules for all the programs in a given language. The nodes of such transition system represent the states during the computation and the transitions mimic the state changes. In the general case, the transition system associated to a program could have an infinite number of nodes. Even when it is finite, its size could

depend on the input of the program, i.e., the transition system cannot be constructed on a generic input. In this Section we will see that the restrictions imposed on *Quip-E* ensure that we can associate to any *Quip-E* program a finite transition system. Such transition system turns out to be a QMC, which would not be the case for a general Quipper program. In detail, in Quipper the use of lists of qubits together with recursion allows to represent an infinite family of circuits using a single program. The semantics we define in this section cannot be easily generalized to such Quipper programs.

As a matter of fact, *Quip-E* denotes a fragment of Quipper programs which generate only finite state, possibly circular, graphs of computations. Moreover, the dimension of such state spaces can be determined at compiling time. This is not the case if we consider generic Quipper programs having, for example, lists of qubits as formal parameters. In such cases, even if the state spaces are finite, their sizes depend on the length of the input qubits lists.

It is not restrictive to fix an a-priori finite set \mathcal{Q} of qubits together with a finite set \mathcal{B} of bits and Booleans. In this section we consider *Quip-E* programs whose variables and parameters are included in such sets. We could avoid this assumption, but this would make the description of the semantics more complex without increasing its expressibility.

Let `trc.C` be a *Quip-E* program having body `Body.C`. Let \mathcal{L} be the set of functions from \mathcal{B} to $\{0, 1\}$. Intuitively, a function $L \in \mathcal{L}$ is an assignment of values for the bits and Booleans occurring in the program. The rules in Table 5.1 define by induction on the structural complexity of `Body.C` its operational semantics in terms of QMCs. The states of such QMCs are pairs, whose first element is either the body of a program or the *empty body*, denoted by `—`. The second element of a pair is a function belonging to \mathcal{L} , which stores the current values of the bits and Booleans. All the operators that label the edges of the chain have dimension $2^{|\mathcal{Q}|}$. Intuitively, if `Body.C` is `reset_at qk`, then the qubit `qk` is measured along the standard basis, applying the operators \mathcal{M}_0^k and \mathcal{M}_1^k . When \mathcal{M}_0^k is applied the empty body is reached, when \mathcal{M}_1^k is applied the body `X_at qk` is reached, and `X_at` is the not operator. In both cases there are no effects on the function L . In the case of `U_at [qi1], ..., qij]` the superoperator $\mathcal{U}_{i_1, \dots, i_j}$ corresponding to `U` is applied and the empty body is reached, without affecting the function L . Such superoperator is computed by applying the identity operator to the qubits in $\mathcal{Q} \setminus \{q_{i_1}, \dots, q_{i_j}\}$ and by swapping the qubits to preserve their order (see also Section 5.1.1). In the case of `m ← measure qk` the measure operators \mathcal{M}_0^k and \mathcal{M}_1^k are applied and the result of the measurement is stored by modifying $L(m)$ accordingly. In particular, $L[L(m) = i]$ denotes the function L' which behaves as L on $\mathcal{B} \setminus \{m\}$, while $L'(m)$ has value i . In the case of `bool ← dynamic_lift m` the identity superoperator \mathcal{I} is applied, i.e., the qubits are unchanged, and the value stored in $L(m)$ is copied in $L(\text{bool})$. In the case of an `if-then-else` instruction on the guard `bool` either the first or the second branch is chosen depending on the value of $L(\text{bool})$, without modifying the values of the qubits. In the case of a sequence `Body.C1 Body.C2` the first instruction of `Body.C1`, is executed applying the corresponding rule and the computation proceeds. Finally, the last rule is added only to ensure that also the empty body satisfies the second condition in the definition of QMC.

Remark. Notice that the values of the qubits are not stored in the state of the Markov chain. Their final values can be computed considering the composition of the operators which label the edges of the chain and by applying the resulting superoperator to their

$$\begin{array}{c}
\frac{}{(\text{reset_at } q_k, L) \xrightarrow{\mathcal{M}_0^k} (-, L)} \qquad \frac{}{(\text{reset_at } q, L) \xrightarrow{\mathcal{M}_1^k} ((X_at \ q_k, L))} \\
\\
\frac{}{(\text{U_at } [q_{i_1}, \dots, q_{i_j}], L) \xrightarrow{\mathcal{U}_{i_1, \dots, i_j}} (-, L)} \\
\\
\frac{}{(\text{m} \leftarrow \text{measure } q_k, L) \xrightarrow{\mathcal{M}_i^k} (-, L[L(\text{m}) = i])} \qquad \text{for } i \in \{0, 1\} \\
\\
\frac{}{(\text{bool} \leftarrow \text{dynamic_lift } m, L) \xrightarrow{\mathcal{I}} (-, L[L(\text{bool}) = L(\text{m})])} \\
\\
\frac{L(\text{bool}) = i}{(\text{if } (\text{bool}) \text{ Body_C}_1 \text{ else Body_C}_0, L) \xrightarrow{\mathcal{I}} (\text{Body_C}_i, L)} \qquad \text{for } i \in \{0, 1\} \\
\\
\frac{(\text{Body_C}_1, L) \xrightarrow{\mathcal{S}} (\text{Body_C}_1', L')}{(\text{Body_C}_1 \ \text{Body_C}_2, L) \xrightarrow{\mathcal{S}} (\text{Body_C}_1' \ \text{Body_C}_2, L')} \\
\\
\frac{(\text{Body_C}_1, L) \xrightarrow{\mathcal{S}} (-, L')}{(\text{Body_C}_1 \ \text{Body_C}_2, L) \xrightarrow{\mathcal{S}} (\text{Body_C}_2, L')} \\
\\
\frac{}{(-, L) \xrightarrow{\mathcal{I}} (-, L)}
\end{array}$$

Table 5.1: Operational Semantics of *Quip-E*

initial values (see [35]). In fact, all the operators that label the edges of the chain have dimension $2^{|\mathcal{Q}|}$ and it is fundamental that the order of the qubits is the same along all the chain. As a matter of facts, Quipper and *Quip-E* aim to provide a flexible programming framework and allow us to specify at each step which are the qubits of interest and the order in which they enter a quantum gate. On the other hand, QMCs are a low level description language for quantum processes and as such they prefer minimality rather

than flexibility. Hence, in a QMC all the gates are applied to all the qubits and these are always considered in the same order. This does not restrict the expressibility of QMC, since by exploiting swapping and identity operators it is always possible to extend a gate to all the qubits in the desired order.

Having now defined the S.O.S. of a *Quip-E* program, we are now almost ready to define the QMC associated to the body of a *Quip-E* program.

Definition 5.4.1 ($QC(s)$). Let $s = (\text{Body_C}, L)$ we define the structure

$$QC(s) = (S(s), Q(s), AP(s), Lab(s))$$

as follows:

- $S(s)$ is the set of pairs reachable from s by applying the rules of Table 5.1;
- $Q(s) : S(s) \times S(s) \rightarrow \mathcal{S}^{\mathcal{I}}(\mathbb{C}^{2^{|\mathcal{Q}|}})$ is defined by the rules of Table 5.1;
- $AP(s) = \mathcal{B}$;
- $Lab(s)((B', L')) = \{b \in \mathcal{B} \mid L'(b) = 1\}$.

$QC(s)$ is said to be the *quantum chain* of s .

Lemma 5.4.1. Given a state $s = (\text{Body_C}, L)$, the quantum chain $QC(s)$ is a QMC.

Proof. In order to prove that the quantum chain $QC(s)$ is a QMC we have to verify that the sum of the superoperators labelling the edges outgoing from each state is a trace preserving superoperator. We proceed by cases as follows:

- (1) $s = (\text{U_at } [q_{i1}, \dots, q_{ij}], L) \Rightarrow (s, L) \xrightarrow{\mathcal{U}_{i_1, \dots, i_j}} (_, L)$ is the only outgoing edge from s and, since $\mathcal{U}_{i_1, \dots, i_j}$ is the superoperator associated to the unitary operator U , it is trivially trace preserving.
- (2) $s = (\text{m} \leftarrow \text{measure } [q_k], L) \Rightarrow Q(s) = \mathcal{M}_i^k$, $i \in \{0, 1\}$. In this case there are two outgoing edges from s , labelled \mathcal{M}_0^k and \mathcal{M}_1^k , i.e., the superoperators associated to the projection operators M_0^k and M_1^k respectively. The property $\sum_i M_i^k = \mathbb{I}$ which follows from the definition of projection operator, can be lifted to the case of superoperators, hence $\sum_i \mathcal{M}_i^k = \mathcal{I}$ which verifies the requirement of the sum being trace preserving.
- (3) $s = (\text{reset_at } [q_k], L)$. This is as case (2).
- (4) $s = (\text{bool} \leftarrow \text{dynamic_lift } [m], L)$. In this case there is only one outgoing edge with label \mathcal{I} .
- (5) $s = (\text{if } (\text{bool}) \text{ Body_C}_1 \text{ else Body_C}_2, L)$. As in case (4), since L satisfies either $L(\text{bool}) = \text{true}$ or $L(\text{bool}) = \text{false}$ but not both.
- (6) $s = (\text{Body_C}_1 \text{ Body_C}_2, L)$. This follows by induction on Body_C_2

Since Body_C_1 and Body_C_2 are compositions of states as in (1)–(5), the superoperator \mathcal{S} is, with certainty, among the kinds already presented, hence it is trace preserving.

(7) $s = (_, L) \Rightarrow QC(s) = \mathcal{I}$. As in case (4).

□

Definition 5.4.2 (QMC associated to a body). Let $Body_C$ be a *Quip-E* body. The QMC associated to $Body_C$, denoted by $QC(Body_C)$, is

$$QC((Body_C, O))$$

where O is the function that assigns value 0 to all the variables in \mathcal{B} .

Notice that $QC(s)$, apart from the self-loops on the “empty body states”, is acyclic. In order to define the semantics of *Quip-E* programs, it is convenient to define an acyclic version of $QC(s)$ in which self-loops are removed.

Definition 5.4.3 (Quasi QMC associated to a body). Let $Body_C$ be a *Quip-E* body. We define the *Quasi QMC associated to $Body_C$* , denoted by $QC^-(Body_C)$, as the structure obtained by removing the self-loops in $QC(Body_C)$.

The structure $QC^-(Body_C)$ is not a QMC, since for the terminal states, i.e., the pairs whose first element is the empty body, the second condition of the definition of QMC is not satisfied. In the following definition we associate a QMC to a *Quip-E* program by introducing two rules that fix the violation.

Definition 5.4.4 (QMC associated to a program). Let trc_C be a tail recursive *Quip-E* program defined by a body $Body_C$ and an exit condition *exitOn bool*. The QMC associated to trc_C , denoted by $QC(trc_C)$, is the QMC obtained from $QC^-(Body_C)$ by adding the edges defined by the following rules:

$$\frac{L(\mathit{bool}) = 1}{(_, L) \xrightarrow{\mathcal{I}} (_, L)} \qquad \frac{L(\mathit{bool}) = 0}{(_, L) \xrightarrow{\mathcal{I}} (Body_C, O)}$$

The following theorem states that our definition is correct, i.e., that the structure we associate to a program is a QMC.

Theorem 5.4.2. Let trc_C be a tail recursive *Quip-E* program. $QC(trc_C)$ is a QMC.

Proof. In order to prove that $QC(s)$ is a QMC we have to verify that the sum of the superoperators labelling the edges outgoing from each state is a trace preserving superoperator. The first part follows from cases (1)–(6) of the proof of Lemma 5.4.1, while case (7) is replaced by two possibilities. Since either $L(\mathit{bool})=1$ or $L(\mathit{bool})=0$ are satisfied and they do not hold at the same time, there is always one outgoing edge with label \mathcal{I} , which is trace preserving. □

5.5 Translation of `trc_C` Programs

In the following we will show the translation of (the most significative) instructions of a `trc_C` programs into QMCs, according to their underlying operational semantics. For each of them we will provide a graphical representation of the resulting QMC as a directed graph, in which the nodes are the states of the chain, and the edges are labeled

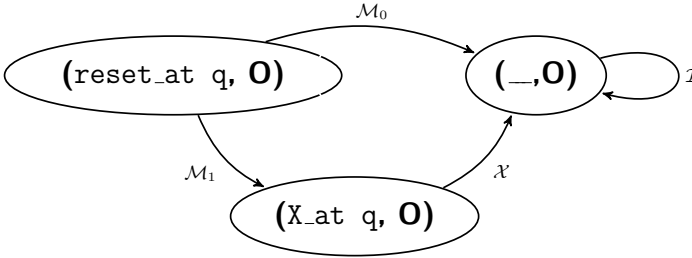
by the unitary or measurement superoperators, according to the order of the *Quip-E* instructions. The states are labelled according to the S.O.S defined.

The first four examples show the QMCs for single instructions, such as reset, unitary transformations, measurements and if-then-else respectively.

Example 5.5.1. In this example we show a single-qubit `trc_C` program in which a reset gate is applied. The *Quip-E* reset instruction together with its corresponding QMC, can be represented as follows:

```
1 reset_at q
```

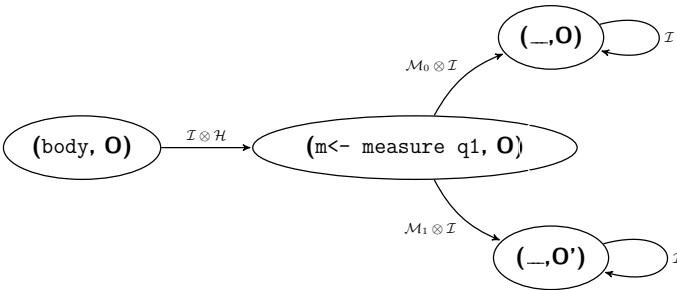
with $\mathcal{B} = \emptyset$.



Example 5.5.2. In this example we show a two-qubit program, in which an Hadamard gate is applied on the second qubit, then a measurement instruction is performed on the first one. The *Quip-E* instructions, together with the corresponding QMC, can be represented as follows:

```
1 hadamard_at q2
2 m <- measure q1
```

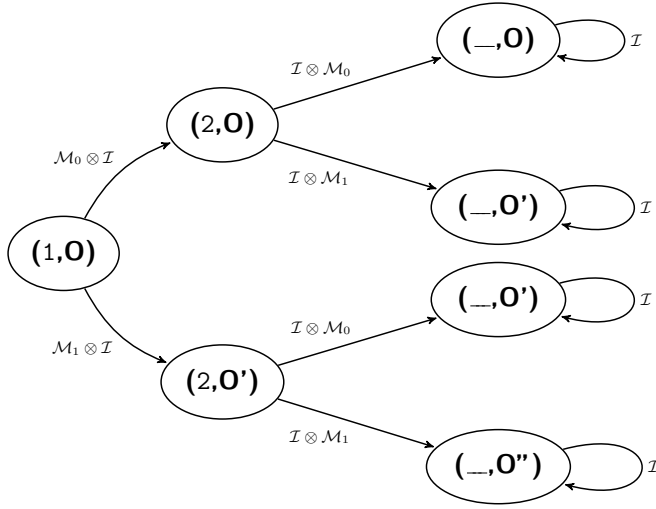
with $\mathcal{B} = \{m\}$. For space reason, we will group the two instructions under the name `body`, and we recall that $\mathbf{0}' = \mathbf{0}[0(m)=1]$.



Example 5.5.3. In this example we show a two-qubit program, in which two measurement are applied on the first and second qubit, respectively. The *Quip-E* instructions, together with the corresponding QMC, can be represented as follows:

```
1 m1 <- measure q1
2 m2 <- measure q2
```

with $\mathcal{B} = \{m1, m2\}$. For space reason, we will refer to the two instructions by the line-number, i.e., `m1 <- measure q1 = 1` and `m2 <- measure q2 = 2`.

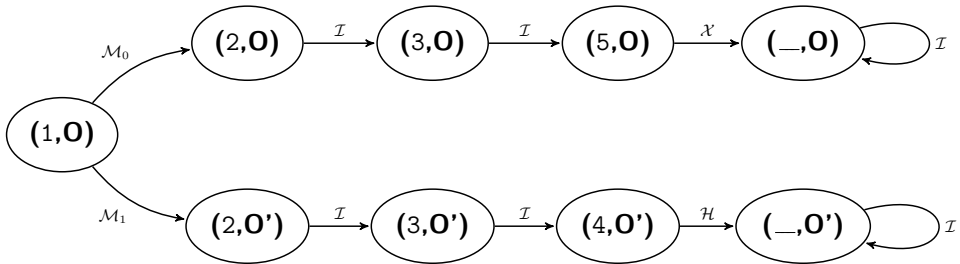


Example 5.5.4. In this example we show a single-qubit program, in which a measurement is performed and it is followed by a dynamic lifting, which transforms the resulting bit into a Boolean value, and by a conditional branch in which, according to the result an Hadamard or a Pauli X gate are applied. The *Quip-E* instructions, together with the corresponding QMC, can be represented as follows:

```

1 m <- measure q
2 b <- dynamic_lift m
3 if b
4     then hadamard_at q
5     else X_at q
    
```

with $\mathcal{B} = \{m, b\}$. For space reason, we will refer to the two instructions by the line-number, i.e., `m <- measure q` = 1, `b <- dynamic_lift m` = 2, `if bool` = 3, `hadamard_at q` = 4 and `X_at q` = 5.



In the following we present two examples of `trc_C` programs in order to show the behaviour of the *QUIP-E* tail-recursive instruction `exitOn`. As in the previous examples, for space reasons we will refer to the single instruction by its line number.

Example 5.5.5. In this example we show a single-qubit program, in which a measurement is performed and it is followed by a dynamic lifting, and by a conditional branch in which, according to the result an Hadamard or a Pauli X gate are applied. As

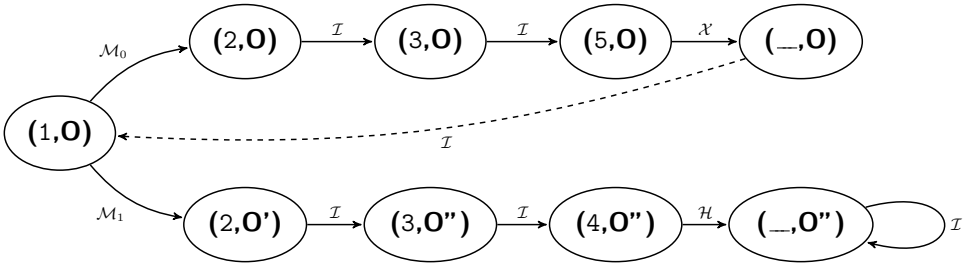
last instruction, we have a recursive instruction which allows the program to terminate only when the Boolean value $b = true$. The *Quip-E* instructions, together with the corresponding QMC, can be represented as follows:

```

1 m <- measure q
2 b <- dynamic_lift m
3 if b
4     then hadamard_at q
5     else X_at q
6 exit0n b

```

with $\mathcal{B} = \{m, b\}$. It is guaranteed to always terminate.



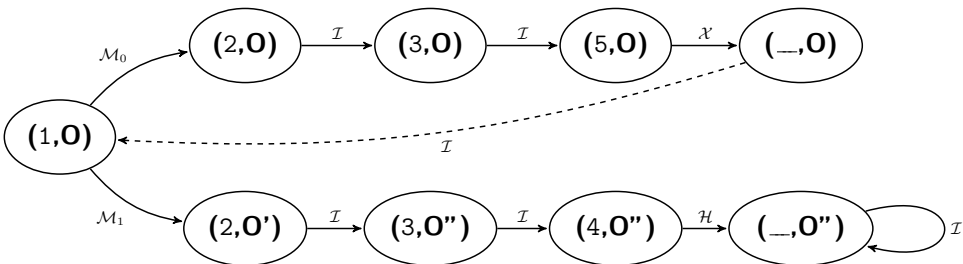
Example 5.5.6. In this example we show a two-qubit program, in which a measurement is performed and it is followed by a dynamic lifting, and by a conditional branch in which, according to the result an Hadamard or a Pauli X gate are applied. As last instruction, we have a recursive instruction which allows the program to terminate only when the Boolean value $b = true$. The *Quip-E* instructions, together with the corresponding QMC, can be represented as follows:

```

1 m1 <- measure q1
2 b1 <- dynamic_lift m1
3 if b1
4     then hadamard_at q2
5     else X_at q2
6 exit0n b1

```

with $\mathcal{B} = \{m1, b1\}$.



5.6 The tool Entangle

We developed an extension of our framework, called *Entangle*, which has been implemented in Haskell, allowing to import and re-use the libraries already developed for both its previous version and Quipper. We implemented several functions to map a *Quip-E* program into the correspondig QMC. In order to provide a more intuitive layout, *Entangle* has been provided with a web based graphical interface written in Elm ¹, together with a preferred style for writing the quantum programs. A snapshot of *Entangle*'s interface is shown in Fig. 5.10. It is divided into three main blocks: *Quipper*, *Tree* and *QPMC*, which will be analysed in the following.

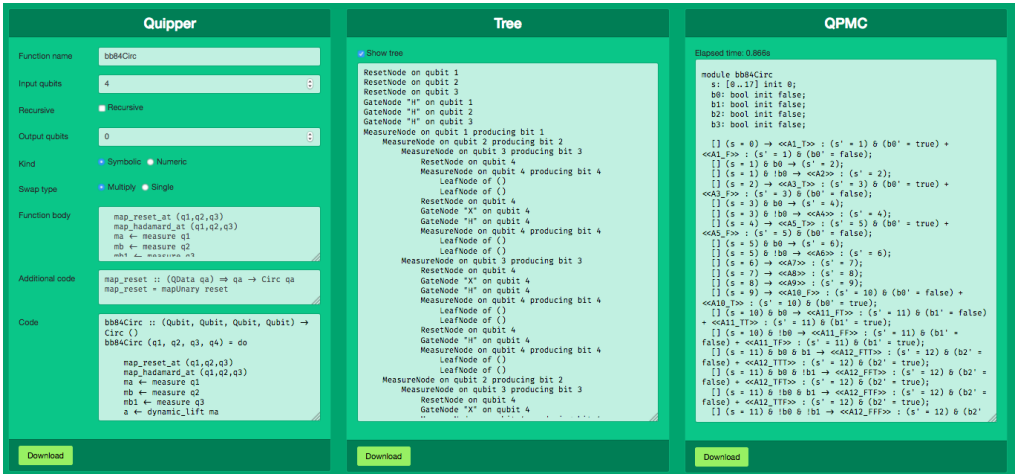


Figure 5.10: Entangle GUI.

5.6.1 Quipper

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

- Function name:** the name of the Quipper program that we want to write;
- Input qubits:** the number of qubits in input to the Quipper program;
- Recursive:** if checked it automatically changes the type signature of the program;
- Output qubits:** the number of output qubits, provided that the program is non-recursive;

¹<http://elm-lang.org>.

- Type:** switches the matrix representation from symbolic, which uses embedded QPMC instructions, to numeric, which provides a numeric, MATLAB-style, representation of matrices, and vice versa;
- Swap:** allows the selection of a different algorithm to build the swap matrices: `multiply` generates the swaps using an algorithm based on permutation by transposition, while `single` uses an algorithm which outputs the matrix representation of the swaps;
- Function body:** the body (i.e., the set of instructions in the right order) of the quantum program to be translated. In this box it is not required to provide the method signature since the complete *Quip-E* program will be generated automatically by `Entangle` in the Code section;
- Additional code:** auxiliary Quipper/Haskell code that cannot be translated from the function body;
- Code:** the final Quipper program, generated by `Entangle` by using all the information provided above. A `.hs` file containing the source code can be downloaded by clicking the Download button.

In the **Function body** we constrained the programmer to a preferred coding style, i.e., we restrict the *Quip-E* syntax to certain choices: the name of the variables of type `Qubit`, if more than one, are denoted by q_i , with $i = 1, \dots, n$, with n number of input qubits, otherwise, if there is only one input qubit it should be labeled `q`. Moreover, measured qubits cannot be re-initialized in the last part of the body. The instruction order to be preserved is: unitary and reset instructions, measurements, dynamic lifting, and control flow. `Entangle` translates tail recursive programs, thus the recursive call must be (when present) the last instruction. A zoomed-in snapshot of the Quipper block can be seen in Figure 5.11.

5.6.2 Tree Block

The Tree block displays an abstract representation of the instructions in the program. It represents the intermediate translation steps. In this block, each quantum gate is associated to the qubits on which it is acting. Measurements produce a binary branch terminating with two leaves which can be of different type, according to the type of program that we want to translate. If the user does not want to see this intermediate representation, (s)he can choose to hide this block.

5.6.3 QPMC Block

The QPMC block displays the translation of the Quipper program into the corresponding QPMC model. It provides both matrices and module to be checked using QCTL. This final code can be downloaded as a PRISM file and used as an input for the QPMC model checker. A snapshot of the QPMC block can be seen in Figure 5.12. For further references about the intermediate translation steps see [6], [5].

Quipper

Function name

Input qubits

Recursive Recursive

Kind Symbolic Numeric

Function body

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

Additional code

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

Code

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

Figure 5.11: Quipper Block

QPMC

Elapsed time: 0.374s

Show tree

qmc

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

Figure 5.12: QPMC Block

5.7 Experimental Results

We have tested `Entangλe` with our implementation of different algorithms and protocols, i.e., Deutsch–Jozsa, Grover’s Search, our (novel) Grover–based implementation of a *quantum switch* function, Teleportation and the BB84 quantum key distribution protocol. As a last example, we explored the possibilities of our tool in the evaluation of quantum properties, i.e., *entanglement*. For the algorithms in which noise can occur or for the probabilistic ones, we provided both the recursive and non–recursive version. In the following, for each algorithm we briefly recall their behaviour, we provide the *Quip-E* implementation and, for the ones having a size which allows it, we provide their representation in terms of QMCs. The proper translation into QPMC code, due to space reason, can be found in Appendix A. We must address a difference in our representation of `reset_` operator: in detail, while the semantics is given as in Table 5.1, the representation that we generate uses an intermediate node, which doesn’t change the semantics of the QMC, and it has been used in order to simplify the design of the tool only.

5.7.1 Deutsch–Jozsa

Let’s consider a function f from n bits to 1 bit, $f : \{0,1\}^n \rightarrow \{0,1\}$. Deutsch–Jozsa algorithm allows to distinguish between two different classes of functions, i.e., the constant and balanced ones. The function is constant if it evaluates the same on all inputs, i.e., the function is either $f(x) = 0$ or $f(x) = 1$ for every x , while it is balanced if the function is 0 on one half of the possible inputs and 1 on the other half. By using a quantum device, one single oracle query is needed to deterministically know whether the function is constant (the circuit output is a register $|0\rangle^{\otimes n}$) or not (the circuit output contains at least one state set to $|1\rangle$). Classically, the problem is solved by using $\frac{1}{2^n} + 1$ queries, in the worst case since we have to apply the function on half the inputs instead of using a linear superposition of them.

In the following we show an implementation of the algorithm using 3–qubits oracles, plus an ancilla to implement it in a reversible way; the first oracle is constant and returns 0 on all the inputs, while the second one is balanced.

Implementation and Translation In the following we present the *Quip-E* implementation for an instance of constant oracle, on the left, and a balanced one, on the right. The translation into QPMC code is available in Appendix A.

Constant

```
dJozsaConst :: (Qubit, Qubit, Qubit,
  Qubit) -> Circ ()
dJozsaConst (q1, q2, q3, q4) = do
  map_reset_at (q1,q2,q3,q4)
  gate_X_at q4

  map_hadamard_at (q1,q2,q3, q4)

  map_hadamard_at (q1,q2,q3)
  measure (q1,q2,q3)
  return ()
```

Balanced

```
dJozsaBal :: (Qubit, Qubit, Qubit,
  Qubit) -> Circ ()
dJozsaBal (q1, q2, q3, q4) = do
  map_reset_at (q1,q2,q3,q4)
  gate_X_at q4

  map_hadamard_at (q1,q2,q3, q4)
  qnot_at q4 'controlled' [q1]
  qnot_at q4 'controlled' [q2]
  qnot_at q4 'controlled' [q3]
  map_hadamard_at (q1,q2,q3)
  measure (q1,q2,q3)
  return ()
```

Test: Some examples of QCTL formulae that we tested are presented in the following. In the case in which the output is a matrix, instead of displaying it, we put its representation in the Appendix, for space reasons. In order to show the probability associated, we provide the traces of the matrices.

QCTL Fomula	Output	Trace
qeval(Q=? [F (s = 19 & !b0 & !b1 & !b2)], r);	(A)	1
qeval(Q=? [F (s = 19 & b0 & !b1 & !b2)], r);	(A)	0
Q=1[F(s=19 & !b0 & !b1 & !b2)];	true	
Q=1[F(s=19 & b0 & !b1 & !b2)];	false	

Table 5.2: Deutsch–Jozsa Constant Verification.

The first formula computes the probability that, given an initial state $\mathbf{r} = |0001\rangle\langle 0001|$ we reach the final state $|0\rangle$. Such probability is equal to 1, while the probability of reaching a final state in which at least a state $|1\rangle$ occurs is equal to 0, as expected. In the following we consider the same queries in the case of the balanced oracle; the results change accordingly, since at least one state $|1\rangle$ should occur for the algorithm to success.

The last formula, in both cases, investigates whether the probability of reaching the attended final state is equal to 1, which is true since the two instances are deterministic.

QCTL Fomula	Output	Trace
qeval(Q=? [F (s = 22 & !b0 & !b1 & !b2)], r);	(A)	0
qeval(Q=? [F (s = 22 & b0 & b1 & !b2)], r);	(A)	1
Q>0.5[F(s=21 & b0 & b1 & !b2)];	true	
Q<0.5[F(s=21 & !b0 & !b1 & !b2)];	true	

Table 5.3: Deutsch–Jozsa Balanced Verification.

5.7.2 Grover’s Quantum Search

The aim of Grover’s algorithm is searching for the index x of an element in a N -dimensional space with no structure. We assume $N = 2^n$, so that the indexes are represented by n -bit strings. The algorithm solves the problem by considering a function $f : \{0,1\}^n \rightarrow \{0,1\}$ such that $f(x) = 1$ if and only if the string x is a solution. Classically, this problem can be solved in $O(N)$ steps, while using a quantum oracle

it can be probabilistically solved in $O(\sqrt{N})$ steps. Then the oracle marks the strings corresponding to possible solutions. At this point, the algorithm performs some steps of amplitude amplification in order to maximize the probability of getting the desired result after the measurement. The result is the index of the searched element. The algorithm is probabilistic, because of the amplitude amplification step. Anyway, for $N = 4$, after one iteration it behaves in a deterministic way, giving the right result with probability equal to 1. Instances of the Grover's algorithm can be used as a starting point to solve different problems, such as graph-coloring. Grover's algorithm exploits quantum parallelism by giving to the quantum oracle all the possible input strings at the same time.

For the experiment we decided to use a search space of size $N = 4$. The oracle returns the string $x = 3$, so the state after the measurement will collapse to $|11\rangle|1\rangle$. The algorithm needs an ancilla qubit that can be easily discarded at the end of the computation.

Implementation and Translation In Figure 5.13 is depicted the Quipper circuit for the three qubit Grover's algorithm. The *Quip-E* implementation can be seen in the

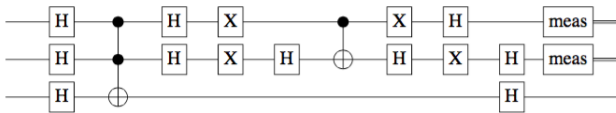


Figure 5.13: Grover's algorithm in Quipper

following:

Grover Deterministic

```
grover :: (Qubit, Qubit, Qubit) -> Circ ()
grover (q1,q2,q3) = do
  map_reset_at (q1,q2,q3)
  gate_X_at q3
  map_hadamard_at (q1,q2,q3)
  qnot_at q3 'controlled' [q1, q2]
  map_hadamard_at (q1,q2)
  gate_X_at q1
  gate_X_at q2
  hadamard_at q2
  qnot_at q2 'controlled' q1
  hadamard_at q2
  gate_X_at q1
  gate_X_at q2
  map_hadamard_at (q1,q2,q3)
  measure (q1,q2)
  return ()
```

The first three applications of the Hadamard gate are needed in order to obtain a linear superposition of the input qubits. The `qnot_at q3 'controlled' [q1,q2]` is the oracle and corresponds to a controlled-not gate. The last gates implement both the amplitude amplification and the interference steps. Finally, the last two qubits are measured. The oracle and the amplitude amplification are repeated a single time, according to the size of the search space.

For space reasons we omit the automatically generated QPMC code, which can be found in the Appendix A. A representation of the QMC for the deterministic version can be seen in Figure 5.14.

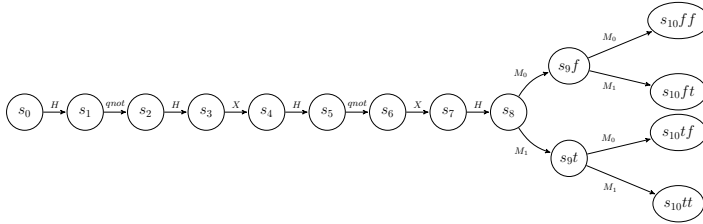


Figure 5.14: QMC associated to Grover's Algorithm

Since, in the general case, the Grover's search algorithm is probabilistic, we also provided a recursive version with 4 qubits (plus the an ancilla). The larger instance of the algorithm halts if the terminal state is the expected one otherwise it reinitialises the circuit.

Grover Recursive

```

groverRec :: (Qubit, Qubit, Qubit, Qubit) -> Circ RecAction
groverRec (q1,q2,q3, q4) = do
  map_reset_at (q1,q2,q3,q4)
  gate_X_at q4

  map_hadamard_at (q1, q2, q3)
  qnot_at q4 'controlled' [q1, q2, q3]
  map_hadamard_at (q1, q2, q3)
  map_X_at (q1, q2, q3)
  hadamard_at q3
  qnot_at q4 'controlled' [q1, q2, q3]
  hadamard_at q3
  map_X_at (q1, q2, q3)
  map_hadamard_at (q1, q2, q3)
  hadamard_at q4

  m1 <- measure q1
  m2 <- measure q2
  m3 <- measure q3
  b1 <- dynamic_lift m1
  b2 <- dynamic_lift m2
  b3 <- dynamic_lift m3

  exit0n $ b1 && b2 && b3

```

Test: In the deterministic version, according to the calculations, we should reach the terminal state s_{14} with probability equal to 1, while the other terminal states must have an associated probability equal to 0. We tested QCTL the formulae to evaluate the density matrix associated to each terminal state with input state $r = |001\rangle\langle 001|$ and the results are presented in Table 5.4.

QCTL Formula	Output	Trace
<code>qeval(Q=? [F (s = 10 & !b0 & !b1)], r);</code>	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$	0
<code>qeval(Q=? [F (s = 10 & b0 & !b1)], r);</code>	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$	0
<code>qeval(Q=? [F (s = 10 & !b0 & b1)], r);</code>	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$	0
<code>qeval(Q=? [F (s = 10 & b0 & b1)], r);</code>	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	1

Table 5.4: Grover’s Algorithm Verification.

The trace of the first three matrices is equal to 0, thus the probability of reaching those states is null. The last matrix has trace equal to 1, thus the computation will reach that state. We also tested formulas to calculate the accumulated superoperators for each state, but since the resulting matrices have size $2^6 \times 2^6$ for space reasons we do not report them here.

In the probabilistic tail-recursive version we we should reach the terminal states s_{48} in which at least one Boolean is equal to *true* with probability higher than other states, since they encode the desired result. We tested some QCTL the formulae to evaluate the density matrix associated to each terminal state with input state $r = |1\rangle\langle 1|$ of appropriate dimension; the results are reported in Table 5.5.

QCTL Formula	Output	Trace
<code>qeval(Q=? [F (s = 48 & !b0 & !b1 & !b2)], r);</code>	(A)	0.25
<code>qeval(Q=? [F (s = 48 & !b0 & b1 & !b2)], r);</code>	(A)	0.25
<code>qeval(Q=? [F (s = 48 & b0 & b1 & b2)], r);</code>	(A)	0.75

Table 5.5: Grover’s Algorithm Verification.

5.7.3 Quantum Switch

Classically, a Boolean switch function (or switch statement) checks for equality a discrete variable (or a Boolean expression) against a list of values, called cases. The variable to be switched is checked for each case. Just as the classical version, a quantum switch returns, according to the value of the input qubits, the index of the correct gate (function) to be applied on them. In this way, we are sure that a given set of functions works properly on each possible combination of variables in input.

The idea behind the quantum switch is to use Grover’s algorithm on a superposition of Boolean functions, represented by quantum *oracles*, rather than on a superposition of basis states. A linear superposition of oracles is an operator which has the following matrix representation:

$$\hat{O} = \begin{pmatrix} U_0 & & & \\ & U_1 & & \\ & & \ddots & \\ & & & U_n \end{pmatrix} \equiv U_0 \oplus \dots \oplus U_n \tag{5.1}$$

While the aim of Grover’s algorithm is searching for the index i (represented by an n -bit string) of an element in an unstructured N -dimensional space, the aim of our Quantum Switch is to search for the index of the i -th Boolean function according to the index of the input qubits. In particular, the quantum switch relies on the diffusion operator of Grover’s algorithm in order to amplify the probability that the right answer occurs. In order to do so, we have to extend the search space in order to provide in input both the variables to be switched and the linear superposition of oracles. Thus the search space will have size $N = 2^{2n} + 1$, with 2^n control qubits, i.e. the variables, 2^n qubits on which Grover’s diffusion operator is applied plus one ancillary qubit. Let us consider a non-trivial circuit with $N = 16$, in Figure 5.15 where, according to the

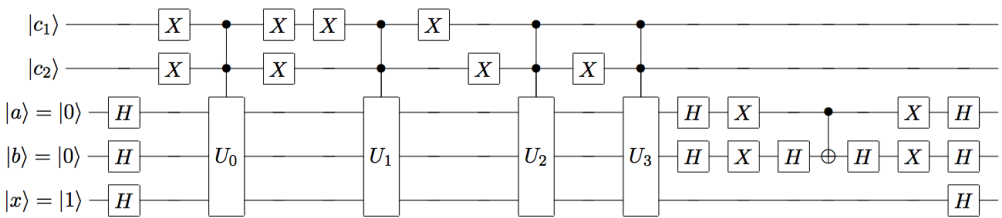


Figure 5.15: Quantum Switch without superposition of variables.

values of $|c_1\rangle$ and $|c_2\rangle$, the circuit applies the correspondent oracle function and return in output the state $|c_1, c_2\rangle U_i |a, b\rangle$. In this particular case, the algorithm is deterministic and requires only one iteration of the diffusion operator. This is due to the fact that the Grover’s diffusion operator is applied to a subspace of size $N_{sub} = 4$.

A more interesting example can be found in Figure 5.16 where as input we provide a linear superposition of variables, thus we are considering all the possible inputs at the same time, and we want to check whether the output matches our expectations

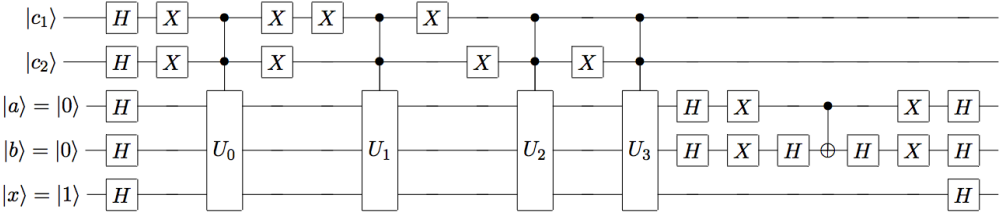


Figure 5.16: Quantum Switch with Superposition of Variables.

or not. The quantum switch exploits quantum parallelism to give to the superposition of quantum oracles all the possible input strings at the same time. In the end we obtain in output a quantum state of the form $\frac{1}{2} \sum_{i,j} |c_i, c_j\rangle U_{i,j} |a, b\rangle$ which is a uniform distribution of oracles. In general, our quantum switch takes a linear superposition $\frac{1}{\sqrt{N}} |c_1, \dots, c_n, a_1, \dots, a_n\rangle$, applies a linear superposition of oracles $U_0 \oplus \dots \oplus U_{2^n}$ and returns a uniform distribution of states, in the form $\frac{1}{\sqrt{N}} \sum_{i_1, \dots, i_n} |c_{i_1}, \dots, c_{i_n}\rangle U_{i_1, \dots, i_n} |a_{i_1}, \dots, a_{i_n}\rangle$.

Translation and Validation

We translated an instance of the quantum switch algorithm, i.e., the deterministic one, with two variables and a search space of $N = 32$, due to the ancillary qubit. Instances with more variables are still to be verified due to the high complexity of generating and performing verification of larger operators. In the following we show the *Qqip-E* program, while the QPMC automatically generated by *Entangl e* can be found in Appendix A.

Switch (1)

```

qswitchCirc :: (Qubit, Qubit, Qubit,
  Qubit, Qubit) -> Circ ()
qswitchCirc (q1, q2, q3, q4, q5) = do
  map_reset_at (q1,q2,q3,q4,q5)
  gate_X_at q5
  map_hadamard_at (q1,q2,q3,q4,q5)
  map_X_at (q1,q2,q3,q4)
  qnot_at q5 'controlled' [q3,q4, q1,
    q2]
  map_X_at (q1,q2,q3,q4)
  map_X_at (q1,q3)
  qnot_at q5 'controlled' [q3,q4, q1,
    q2]
  map_X_at (q1,q3)

```

Switch (2)

```

map_X_at (q2,q4)
qnot_at q5 'controlled' [q3,q4,
  q1, q2]
map_X_at (q2,q4)
qnot_at q5 'controlled' [q3,q4,
  q1, q2]
map_hadamard_at (q3,q4)
map_X_at (q3,q4)
hadamard_at q4
qnot_at q4 'controlled' q3
hadamard_at q4
map_X_at (q3,q4)

```

Test: We simulated the Quantum Switch code and provided a Matlab plot of the probability of success, which can be seen in Figure 5.17, noting that the eight peaks growing up to 0.25, represent the desired output states, while the ones with a lower probability are those which are not solutions. From this simulation it is also possible to see the cyclic behaviour of the Grover operators.

We tested some QCTL formulae:

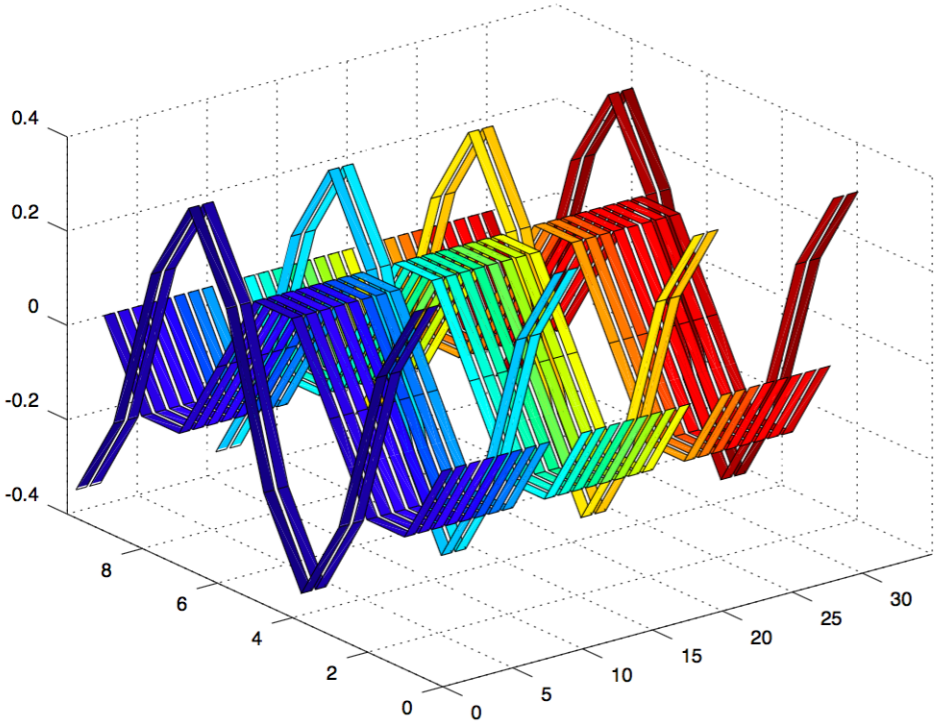


Figure 5.17: Quantum Switch Simulation.

QCTL Formula	Output
$Q \geq 0.25 [F(s=39 \ \& \ !b0 \ \& \ !b1 \ \& \ !b2 \ \& \ !b3)]$;	true
$Q \geq 0.25 [F(s=39 \ \& \ !b0 \ \& \ !b1 \ \& \ !b2 \ \& \ b3)]$;	false
$Q \geq 0.25 [F(s=39 \ \& \ b0 \ \& \ b1 \ \& \ b2 \ \& \ b3)]$;	true
$Q \geq 0.25 [F(s=39 \ \& \ b0 \ \& \ b1 \ \& \ b2 \ \& \ !b3)]$;	false

Table 5.6: Quantum Switch Verification.

the formulae verify that, in the future, a desired state (we restricted the example to two solutions for space reasons) is reached with probability bounded by 0.25, while the probability to reach other, undesired states, is less than 0.25, validating our expectations.

A larger version of the circuit, with a search space of $N = 512$ can be found at <https://github.com/miniBill/entangle> (path res/Entangle Tests).

5.7.4 Teleportation

The aim of quantum teleportation is to move a qubit from one location to another, without physically transporting or copying it, with the aid of a classical channel and a shared quantum entanglement pair between the sender and the receiver. The steps of the teleportation protocol have been explained in Section 2 and, given an unknown qubit that is to be teleported between two parties, namely Alice and Bob, it can be summarised as follows: first, an entangled pair is generated between Alice and Bob, then Alice performs a Bell measurement (a specific sequence of unitary operators followed by a measurement) of her part of the entangled pair qubit and the qubit to be teleported. The measurement yields one of four measurement outcomes, which are then encoded using two classical bits. By using the classical communication channel, Alice sends the two bits to Bob. As a last step, according to the two received bits, Bob applies a pre-determined sequence of unitary gates on its part of the entangled pair, obtaining always the qubit that was chosen for teleportation. In the following, since the protocol always succeeds in absence of noise, we provide both the non-recursive and the recursive versions.

Implementation and Translation The *Quip-E* implementation of the teleportation algorithm presented in the previous sections can be seen in the following:

Teleportation

```
teleport :: (Qubit, Qubit, Qubit) -> Circ ()
teleport (q1, q2, q3) :: do
  reset_at q2
  reset_at q3
  hadamard_at q2
  qnot_at q3 'controlled' q2
  qnot_at q2 'controlled' q1
  hadamard_at q1
  c1 <- measure q1
  c2 <- measure q2
  if c1==1 && c2==1
    then do
      gate_Y_at q3
    else if c1==1 && c2==0
      then do
        gate_Z_at q3
      else if c1==0 && c2==1
        then do
          gate_X_at q3
        else return ()
```

The first hadamard gate, followed by a controlled-not is used to create a maximally entangled state between the second and the third qubit. The automatically generated QPMC code, which can be found in the Appendix A. A representation of the QMC for the teleportation version can be seen in Figure 5.18.

Since noise and decoherence may occur, making the correlations vanish and the protocol not reliable as required, we also provided a recursive version.

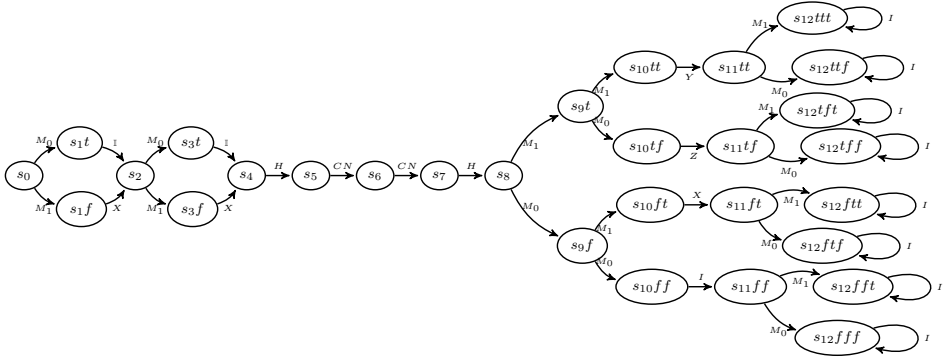


Figure 5.18: QMC for the Teleportation Protocol.

Teleportation Recursive

```

teleportRec :: (Qubit, Qubit, Qubit) -> Circ RecAction
teleportRec (q1, q2, q3) = do
  reset_at q2
  reset_at q3
  hadamard_at q2
  qnot_at q3 'controlled' q2
  qnot_at q2 'controlled' q1
  hadamard_at q1
  c1 <- measure q1
  c2 <- measure q2
  b1 <- dynamic_lift c1
  b2 <- dynamic_lift c2
  if b1 && b2
  then do
    gate_Y_at q3
  else if b1 && (not b2)
  then do
    gate_Z_at q3
  else if (not b1) && b2
  then do
    gate_X_at q3
  else do
    gate_X_at q3
    gate_X_at q3
  c3 <- measure q3
  b3 <- dynamic_lift c3
  exit0n $ b1==b3
    
```

QCTL Formula	Output
$Q > 0.25 [F(s=11 \ \& \ !b0 \ \& \ !b1)];$	true
$Q = 0 [F(s=12 \ \& \ !b0 \ \& \ !b1 \ \& \ !b2)];$	false

Table 5.7: Teleportation Protocol Verification.

The formulae bounds the probability to reach a desired state (after the first conditional branch) to a value greater or equal to 0.25, and ascertain that the probability to reach a final, desired state (after the second conditional branch), never goes to 0.

5.7.5 BB84 Protocol

In this example we focus on protocols that use quantum effects such as entanglement to guarantee secure communication between two parties (Alice and Bob) communicating on classical channels. Errors of different type can occur during the communication, in particular due to noise and decoherence effects. Moreover, in QKD protocols such as BB84, we want not only to guarantee that the channel is free of noise effects that could change the output, but also that no eavesdropper can obtain the key the two parties are exchanging. In particular, the BB84 protocol enables two parties to share a random and secure key, which could be used for classical encryption schemes such as the one-time pad. The protocol has been analysed step-by-step in Section 2.2.1. In the following experiments, we have implemented two versions of the BB84 protocol where only one-bit strings are transmitted. The first assumes that the channel is free of noise, while the second is recursive and halts only if certain conditions are verified. Using a single bit string is the simplest setting possible, but it can be generalized to strings of arbitrary length. Since quantum measurement outcomes are random, we decided to generate the classical strings using this kind of technique, i.e., we measure two qubits previously put in a uniform superposition by the application of an Hadamard gate and then we get the classical outcome.

Non-recursive BB84 Implementation and Translation: The *Quip-E* implementation for the non-recursive version can be seen in the following:

BB84 (1)

```
bb84Circ :: (Qubit, Qubit, Qubit,
            Qubit) -> Circ ()
bb84Circ (q1, q2, q3, q4) = do
  bb84defCirc (q1,q2,q3,q4)
  ma1<- measure q4
  a1<-dynamic_lift ma1
  return ()
--AUX--
bb84defCirc :: (Qubit, Qubit, Qubit,
               Qubit) -> Circ Qubit
bb84defCirc (q1, q2, q3, q4) = do
  map_reset_at (q1,q2,q3)
  map_hadamard_at (q1,q2,q3)
  ma <- measure q1
  mb <- measure q2
  mb1 <- measure q3
  a <- dynamic_lift ma
  b <- dynamic_lift mb
  b1 <- dynamic_lift mb1
  if (not a) && (not b) && (b==b1)
    then do
      reset_at q4
```

BB84 (2)

```
reset_at q4
return q4
else if a && (not b) &&
      (b==b1)
  then do
    reset_at q4
    gate_X_at q4
    return q4
  else if (not a) && b
    && (b==b1)
  then do
    reset_at q4
    hadamard_at q4
    return q4
  else if (b==b1)
  then do
    reset_at q4
    gate_X_at q4
    hadamard_at q4
    return q4
  else do
    return q4
```

The corresponding QPMC translation, i.e., module `bb84Circ`, without the definition of constants, can be found in Appendix A. The graph representation of the QMC, can be seen in Figure 5.19 and in the details below:

In the first three blocks the random strings a , b and b' are generated by means of quantum measurement on an equal superposition of basis states, holding $1/2$ of probability to get $|0\rangle$ and $1/2$ to get $|1\rangle$ for each qubit. Then, the BB84 protocol is performed according to the generated strings. In the end, checks are performed: some states are

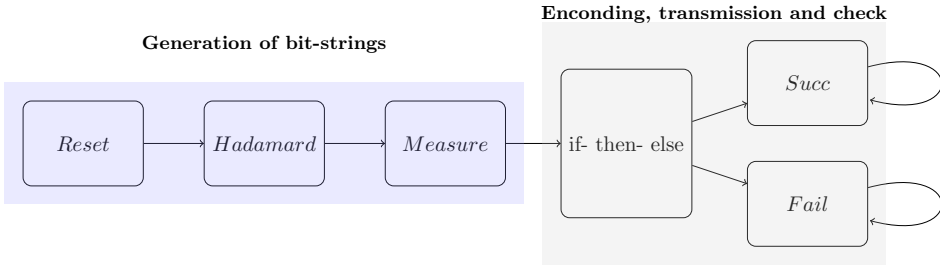


Figure 5.19: High level representation of BB84 steps.

accepted as solutions, thus the protocol terminates successfully, while other states are denied, meaning that something occurred during the transmission of the key. In this case we have to manually discard the results and run again the whole protocol. The portion of QMC associated to the generation of the random bit strings a , b and b' is represented in Figure 5.20.

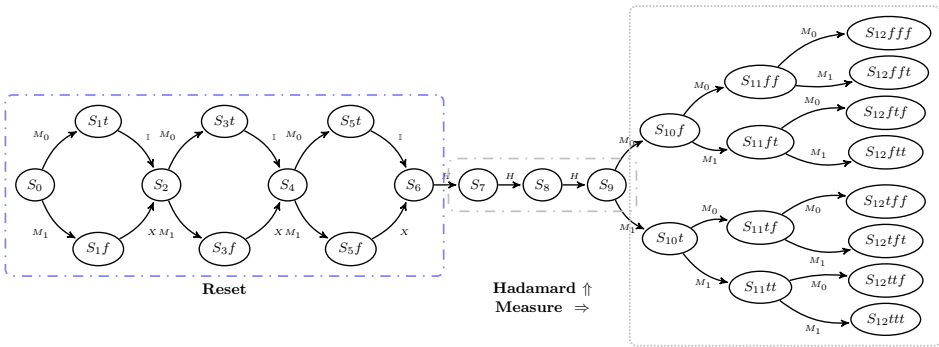


Figure 5.20: Random string generation.

The portion related to the encoding and check part of the protocol is represented in Figure 5.21. The blue states are those accepted at the end of the protocol, while the red ones are the denied ones.

Recursive BB84 translation:

In the following we provide, as an example, our *Quip-E* implementation of a tail recursive BB84 protocol.

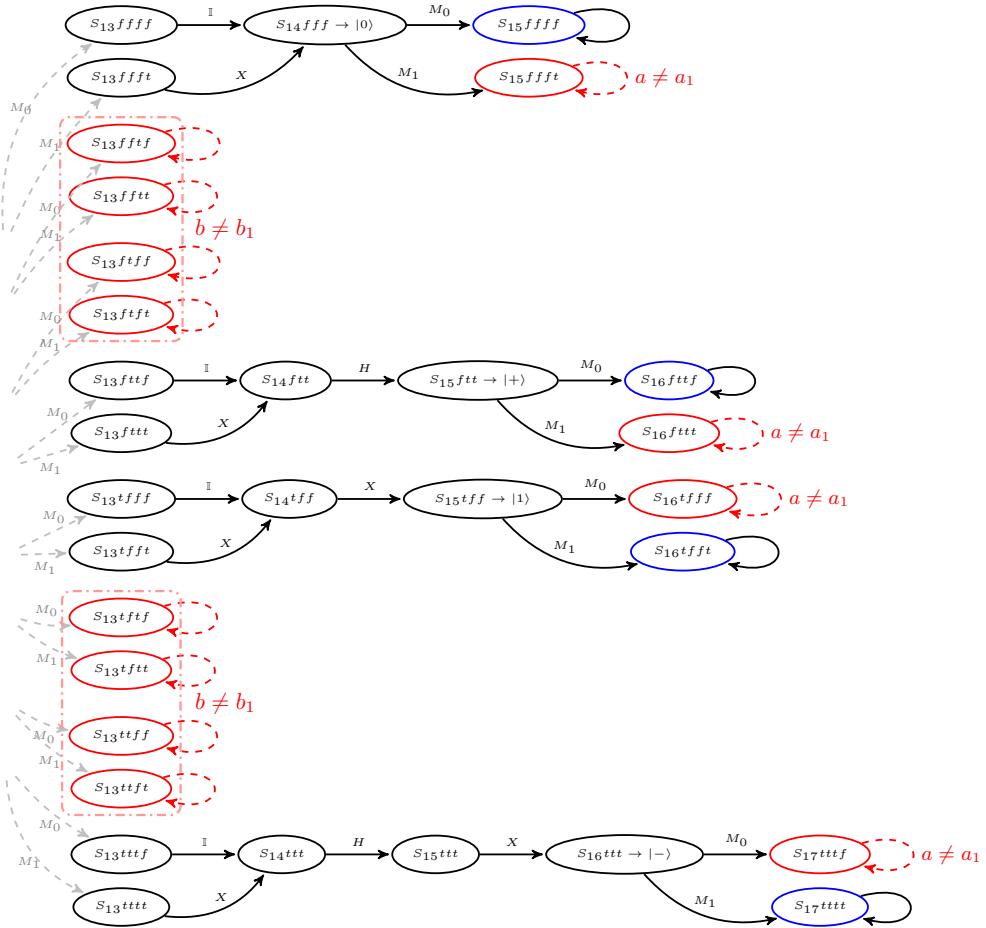


Figure 5.21: BB84 Body.

BB84 rec (1)

```

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

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

BB84 rec (2)

```

then do
  reset_at q4
  gate_X_at q4

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

else do
  reset_at q4
  gate_X_at q4
  hadamard_at q4

ma1<- measure q4
a1 <- dynamic_lift ma1
exitOn $ b==b1 && a==a1
  
```

The idea is the same as the non-recursive BB84 protocol, but in this version, instead of manually performing the checks to assess whether the state is accepted or denied and then restart the protocol, this is done automatically by the algorithm. In this case, the QMC in Figure 5.22 shows the portion related to the encoding and check part of the recursive protocol, since the initialization part in which the random strings are generated is the same as the non-recursive case.

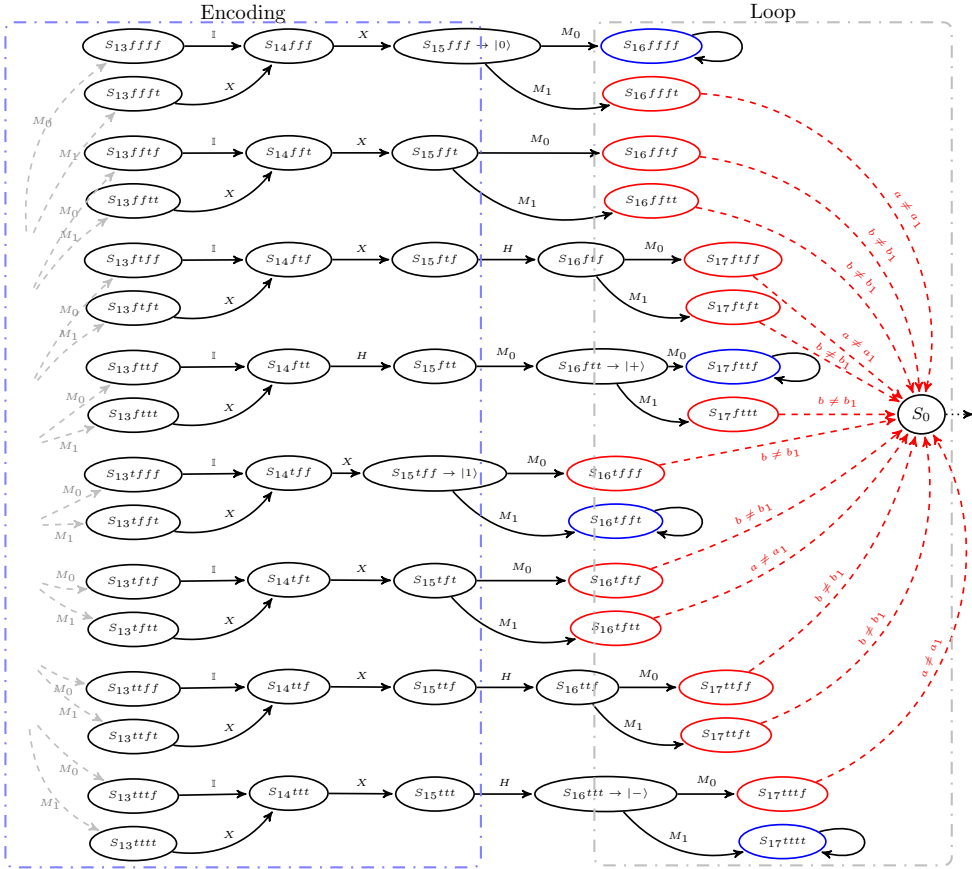


Figure 5.22: Recursive BB84 Body.

The corresponding QPMC translation, can be found in Appendix A and the abstract model of the quantum Markov chain can be seen in Figure 5.23.

5.7.6 BB84 Tests:

We show in Table 5.8 some tests performed by using QCTL formulae [35] on both the implementations of the BB84 protocol. The first two formulae refer to the probability that a *success* state never occurs before a *failure* state. The third and fourth formulae consider the probability that a failure state cannot occur before a success state. The

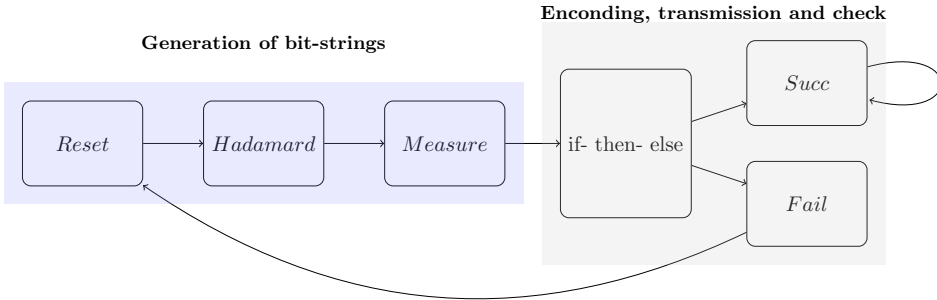


Figure 5.23: High level representation of recursive BB84 steps.

QCTL Formula	Output
$Q=0[!F(\text{succ}) (!\text{fail}) U(\text{succ})];$	true
$Q>0[!F(\text{succ}) (!\text{fail}) U(\text{succ})];$	false
$Q=0[!F(\text{fail}) (!\text{succ}) U(\text{fail})];$	true
$Q>0[!F(\text{fail}) (!\text{succ}) U(\text{fail})];$	false
$Q>0.3[\text{true} U(\text{succ})];$	true
$Q>0.3[\text{true} U(\text{fail})];$	true
$Q>0.5[\text{true} U(\text{succ})];$	false
$Q>0.5[\text{true} U(\text{fail})];$	true

Table 5.8: QCTL example tests for recursive and non-recursive BB84 protocol.

fifth and sixth formulae show that both *success* and *failure* states can be reached with probability > 0.3 . The last set of formulae bounds the probability that a state *success* is reached with a value < 0.5 . By generating the density matrices, and by computing the trace, these results are consistent, since $\text{trace}(\text{SUCC}) = 0.37500$ and $\text{trace}(\text{FAIL}) = 0.62500$. Note that the two traces sum to 1.

5.7.7 Entanglement Detection:

In this section we show how to use our framework for detecting entanglement in a Quipper program. Before we proceed, we recall that by computing the von Neumann entropy it is possible to assess if a pure state is entangled or not. This can be useful not only to evaluate the loss of coherence throughout the execution of a (more complex) quantum protocol due to the presence of noise or of an eavesdropper, but also to check if new correlations are created due to the presence of the environment, with consequences on the information carried by the protocol.

In the following we briefly recall the notions of entanglement and the von Neumann entropy seen in Section 2. Recall that von Neumann entropy is a measure of information, and it is useful for deciding whether a state is mixed or pure, and it is also used as a measure of entanglement for pure states. In this work we will only consider the entanglement of a bipartite system, since a more in depth analysis of the subject is out of the scope of this paper.

Let $\mathcal{H} = \mathcal{H}^{(1)} \otimes \mathcal{H}^{(2)}$ be a composite (bipartite) quantum system. A state $|\psi(1,2)\rangle$ on \mathcal{H} is *entangled* if and only if it does not exist any state vector $|\phi(1)\rangle \in \mathcal{H}^{(1)}$ and $|\omega(2)\rangle \in \mathcal{H}^{(2)}$ such that:

$$|\psi(1,2)\rangle = |\phi(1)\rangle \otimes |\omega(2)\rangle$$

The von Neumann entropy $S(\rho)$ of a given density matrix ρ is defined as follows:

$$S(\rho) = -\text{tr}(\rho \log \rho) = -\sum_i \lambda_i \log \lambda_i$$

where λ_i are the (strictly) positive eigenvalues of ρ .

It is possible to determine whether a pure state, i.e., a state such that $\rho^2 = \rho$, is entangled or not by resorting to the von Neumann entropy. Let $\rho = |\psi(1,2)\rangle\langle\psi(1,2)|$ be the state under consideration, the following statement holds:

$$S(\rho) = 0 \wedge S(\rho^{(1)}) > 0, S(\rho^{(2)}) > 0 \Leftrightarrow \rho \text{ entangled}$$

where $\rho^{(1)} = \text{tr}^{(2)}[\rho]$ and $\rho^{(2)} = \text{tr}^{(1)}[\rho]$ are the reduced density matrices of the composite pure state ρ . Given a pure state $\rho = |\psi\rangle\langle\psi|$ it is possible to determine whether it is entangled or not by simply calculating first its reduced density matrices, and then by calculating the entropy associated to its (non-zero) eigenvalues. The state is entangled if and only if such entropy turns out to be greater than zero, i.e., $S(\rho^{(1)}) > 0$ and $S(\rho^{(2)}) > 0$.

We translated a simple circuit that entangles two qubits. This is represented by the following *Quip-E* code:

```
entCirc :: (Qubit, Qubit) -> Circ (Qubit, Qubit)
entCirc (q1, q2) = do
  reset_at q1
  reset_at q2
  hadamard_at q1
  qnot_at q2 'controlled' q1
  return (q1,q2)
```

which can be automatically translated into the following QMC:

```

module entCirc
  s: [0..6] init 0;
  b0: bool init false;

  [] (s = 0) -> <<A1_T>> : (s' = 1) & (b0' = true) + <<A1_F>> : (s' = 1) & (b0' =
    false);
  [] (s = 1) & b0 -> (s' = 2);
  [] (s = 1) & !b0 -> <<A2>> : (s' = 2);
  [] (s = 2) -> <<A3_T>> : (s' = 3) & (b0' = true) + <<A3_F>> : (s' = 3) & (b0' =
    false);
  [] (s = 3) & b0 -> (s' = 4);
  [] (s = 3) & !b0 -> <<A4>> : (s' = 4);
  [] (s = 4) -> <<A5>> : (s' = 5);
  [] (s = 5) -> <<A6>> : (s' = 6);
  [] (s = 6) -> true;
endmodule

```

<pre> qeval(Q=?[F(s=1)], 0>_4 <0 _4); </pre>	<pre> 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </pre>
<pre> qeval(Q=?[F(s=6)], 0>_4 <0 _4); </pre>	<pre> 0.5 0 0 0.5 0 0 0 0 0 0 0 0 0.5 0 0 0.5 </pre>

Figure 5.24: QCTL formulas and results.

In our example, the first `qeval(Q=? [F(s=1)], |0>⟨0|) = ρI` returns the density operator for the initial state, while `qeval(Q=? [F(s=6)], |0>⟨0|) = ρF` computes the density operator for the final state of the algorithm.

We used the functions `PartialTrace` and `Entropy` of the Matlab library `QETLAB` to compute the partial trace and the entropy of the density operator and its reduced. Since $S(\rho_I) = S(\rho_I^{(1)}) = S(\rho_I^{(2)}) = 0$ the initial state ρ_I is not entangled. On the contrary, the final state ρ_F is fully entangled, since it has entropy equal to zero, and both its reduced operators have maximal entropy (i.e., $S(\rho_F) = 0$, $S(\rho_F^{(1)}) = S(\rho_F^{(2)}) = 1/2$).

5.7.8 Multiple SWAP Optimisation

We implemented an optimisation for our previous swap algorithm (herein `multiply`), in which we directly generate the swap matrices without the composition of binary swaps. We will refer to the optimised method as `single`. Remember that, as we have introduced in the previous section, in the Quipper block it is possible to change the two methods in order to perform scalability tests.

Given the permutation of qubits in Fig. 5.25, by using the `multiply` method, the permutation is obtained by transposition, i.e., by the composition of two binary-SWAP operators S_1 and S_2 ; the first performs a swap between the second and the third qubits,

while the second acts on the first and second qubits, respectively. The composition of the two operators is a swap operator S :

$$S = S_2(S_1)$$

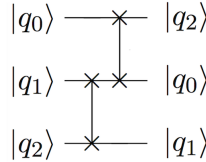


Figure 5.25: Composition of SWAP.

It can be proved that the `multiply` method works on any number of swaps.

When S acts on a linear superposition of basis states we obtain the following state in output:

$$S|q_0, q_1, q_2\rangle = |q_2, q_0, q_1\rangle$$

Which, once applied to the vectors of the standard computational basis behaves as follows:

$$\begin{aligned} S|0, 0, 0\rangle &= |0, 0, 0\rangle ; & S|0, 0, 1\rangle &= |1, 0, 0\rangle ; \\ S|0, 1, 0\rangle &= |0, 0, 1\rangle ; & S|0, 1, 1\rangle &= |1, 0, 1\rangle \\ S|1, 0, 0\rangle &= |0, 1, 0\rangle ; & S|1, 0, 1\rangle &= |1, 1, 0\rangle ; \\ S|1, 1, 0\rangle &= |0, 1, 1\rangle ; & S|1, 1, 1\rangle &= |1, 1, 1\rangle \end{aligned} \quad (5.2)$$

The `single` method allows to build the operator S without composing n binary swaps; this is obtained by taking into consideration the index of the elements inside the state $|i\rangle$. In this way, we are able to rewrite the permutation of Eq. 5.2 as follows:

$$(i, j, k) \rightarrow (i, k, j) \rightarrow (k, i, j)$$

We build a swap matrix S by computing its matrix elements, i.e., given an initial configuration $|i\rangle$ and a final configuration $|j\rangle$ we compute $S_{ij} = \langle j|S|i\rangle$. In order to perform this efficiently, we have to compute the positions which the elements equal to 1 will occupy in the permutation matrix, by indexing the qubits.

The `single` method can be implemented with the following MATLAB function:

```
function S = swap(T)      %T target positions in base 0

S = zeros(2^length(T))
for i = 1:(2^length(T))
    origin = dec2bin(i-1,length(T)); % original position of the 1
    target = dec2bin(0,length(T)); % target position of the 1
    % fill the target array doing the actual swap
    for j = 1:length(T)
        target(j) = origin(T(j)+1);
```

```

    end
% put the 1 in the swap matrix in the target-th row
    S(bin2dec(target)+1,i) = 1;
end
end

```

Our swap algorithm –the `multiply` method– has computational cost $O(2^n)$, where n is the number of qubits. If we let $m = 2^n$ be the size of the $m \times m$ square swap matrix, our algorithm has cost $O(m)$, i.e. is linear with the size of the swap matrix. In fact, the outer for loop has complexity $\Theta(m)$, and each operation inside the loops costs at most n .

On the other hand, the naïve swap algorithm for n qubits used before costs $O(m^3)$. In details, we can build a swap matrix for a binary swap operation in time $O(m^2)$, by performing $n - 1$ tensor product operations. For example, let I_2 the 2×2 identity matrix and S the 4×4 binary swap matrix. If we need to swap the second and third qubit of a four qubit system, we need to perform $I_2 \otimes SWAP$, which results in a 8×8 matrix. Then, we have to perform the tensor product of such matrix with I_2 , resulting in a 16×16 matrix. If we consider the computational cost of the last tensor product (i.e. the most expensive one), it requires exactly $2 \times 2 \times 8 \times 8 = 2^8$ steps, i.e. its computational cost is $\Theta(2^{n^2}) = \Theta(m^2)$.

However, we have to build a swap matrix for each qubit couple we want to swap, and then multiply them, i.e., if we want to swap n qubit (the worst case), we need to perform n matrix multiplication operations, each of cost $O(m^3)$.² This means that, overall, the naïve swap algorithm requires at least to build a $m \times m$ binary swap matrix, i.e. it has cost $\Omega(m^2)$, and at worst it has to multiply n $m \times m$ binary swap matrices, i.e. it has cost $O(m^3)$.

We tested the circuits, built as in the previous case, using from 3 to 7 qubits, where the number of swap has been maximised. Times are recorded using an Elm function. In this case as well, for each size of the input the program has been executed five times and the mean time has been computed. The results are shown in Figure 5.26.

Now, it is possible to estimate the translation time by looking at the parameter *elapsed time* in the QPMC block. We made a comparison between the `multiply` and the `single` methods. Please note that the results we present here are slower than the one presented in [5] due to a different implementation (i.e. using the web interface to run the code and assess the performance). However, the reader should not focus on their absolute value, but just on the higher speed of our new swap algorithm when compared to the classical one.

5.8 Summary

In this chapter we presented `Entangle`, a framework providing an integrated environment to translate quantum programs written using a recursive fragment of Quipper, into QMCs, which are structures that can be given in input to the QPMC model checker. Other model checking techniques for quantum protocols have been considered, e.g., in [39][41][9] and [68], but, even if the techniques proposed are interesting, they were either

²We are aware of the existence of algorithms for matrix multiplication that slightly lower the exponent; however, for the sake of simplicity, and since we don't have discovered a quadratic square multiplication algorithm yet, we will treat this operation as computationally cubic.

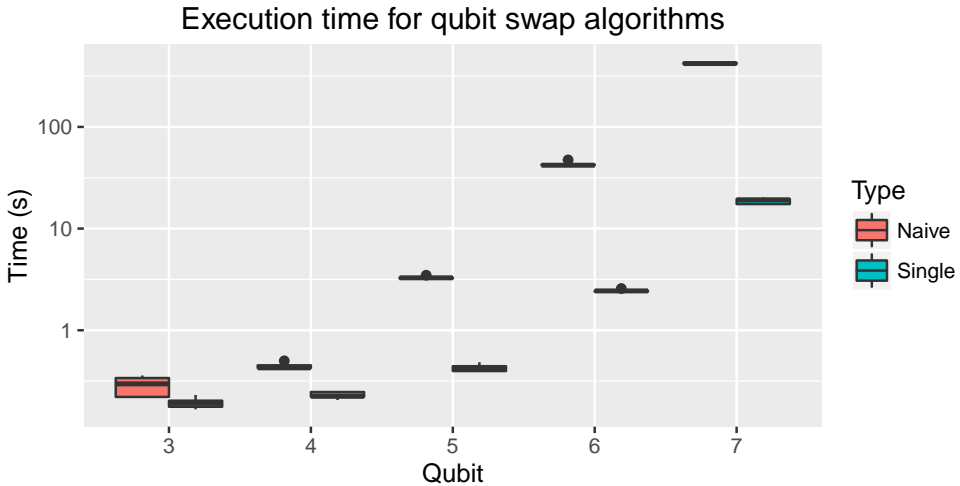


Figure 5.26: Comparison between `multiply` (naive method) and `single`.

not publicly available or mainly restricted to a particular class of quantum circuit, while Quipper and QPMC provided a more flexible paradigm.

We first developed a simplified version of the tool for the translation of Quipper circuits, written inside the `Circ` monad, into QMCs. This allowed us to simulate and make formal analysis on quantum protocols using a high-level language, but still was restricted to circuits, not allowing to translate proper programs, in which conditional branches and loops might occur.

In order to provide the verification of a Turing-complete fragment, we then extended the previous tool by considering also *tail-recursive* quantum programs. The extended version of our tool, called `Entangle`, plus a unification framework from a fragment of Quipper, called *Quip-E*, has been presented. We described both the syntax and the operational semantics of the language, and we showed how a *Quip-E* program is translated into a QMCs, ready to be verified by QPMC.

`Entangle` allows to both write and verify quantum protocols, using an high-level programming language. It is able to translate also `trc_C` programs, i.e., programs in which measurement results may control termination of quantum protocols and algorithms.

`Entangle` is endowed with a graphic interface, which has been described and where the programmer can write *Quip-E* programs which are automatically translated into QPMC code. We showed many tests of `Entangle`, which has been used to translate and verify several quantum algorithms, from the simpler Deutsch-Jozsa algorithm to the BB84 quantum key distribution protocol in both recursive and non-recursive versions, among others. The final results validated our expectations. Moreover, we found a way to include the validation of quantum properties in the verification process as well –by now as a post-processing operation, using the results of the tested formulae.

We are still working on the optimization of our framework in order to match the requirement of validating complex algorithms and protocols, e.g., the ones involving a

wider number of qubits which are actually difficult to translate due to the huge cost to generate the matrices. Anyways, some steps in this direction has been done, i.e., by implementing a multiple swap optimisation algorithm to build generalised permutation matrix by the computation of its matrix elements instead than using the composition of binary swaps. Moreover, we intend to investigate the specification of properties involving typical quantum effects, in particular automatic entanglement detection. This would be useful in particular for the quantum key distribution protocols.

Optimisations should be done from the model-checker point of view, involving the automatic verification of more complex properties, i.e., entanglement and other quantum effects. A possible solution to this problem has been investigated and brought to the development of an early version of a custom model-checker. which will be briefly presented in Sectionsec:conclusion. The tool is currently under specification and development; it will allow us to explicitly generate the graph structure of a discrete time Markov chain representing the quantum program, whose edges are labelled with the matrix representation of the unitary and measurement operators. We aim at substituting QPMC, by relying more on the standard version of *PRISM*. Nevertheless, since the tool is still in its infancies, by now we regard it as an interesting research direction for the future.

6

Entangled Evolving Hypergraphs

In this part of the thesis we show an extension of the idea presented in [42] which was in turn based on the concept of entangled graphs, which can be found in [73].

Quantum systems show behaviors that are different from the classical ones, i.e., superposition, interference, and *entanglement* [51]. Some of the aforementioned behaviors can be considered as one-particle effects, since there is no need for the system to be composed by more than one particle to show them. Other effects, such as entanglement, can be observed only in composite quantum systems, i.e., systems composed by two (or more) subsystems. These subsystems can exhibit both classical and non-classical correlations. *Entanglement* is a kind of non-classical correlation, displayed by certain classes of quantum states called *entangled states*, and it is an important notion in QIC. It constitutes a fundamental resource for many quantum protocols –ranging, e.g., from the simplest case of quantum teleportation[14] to more complex scenarios such as quantum key distribution (herein QKD) and quantum cryptography[13]– which rely on entanglement to be properly performed. In order to perform formal reasoning on quantum protocols, entanglement *as a resource* needs to be classified, since each *entanglement class* is associated with a different set of tasks in quantum information processing. However, this classification is not trivial, since the notion of entanglement differs according to the number of entangled particles of the quantum system under consideration.

In *bipartite entanglement*, we deal with quantum systems which are composed by two subsystems. Thus, a bipartite quantum system is associated to the Hilbert space $\mathcal{H} = \mathcal{H}^{(1)} \otimes \mathcal{H}^{(2)}$ ¹ and, given two states $|\psi_1\rangle \in \mathcal{H}^{(1)}$ and $|\psi_2\rangle \in \mathcal{H}^{(2)}$, we can always build a state $|\Psi_{sep}\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$, $|\Psi_{sep}\rangle \in \mathcal{H}$. Quantum states that can be written in this form are called *separable* (or product states). It is important to recall that not every state in \mathcal{H} is separable; there are, indeed, bipartite quantum states that cannot be decomposed in the latter form. In this case the state is called *entangled*.

Quantum entanglement in bipartite systems of pure states –i.e., states corresponding to vectors in a Hilbert space– is (almost) completely understood, since it can be characterized using the *Schmidt decomposition* [81]. This decomposition allows to write, by means of local unitary transformation only, any pure state $|\Psi\rangle$ of a bipartite system in

¹ $\mathcal{H}^{(i)}$ are the Hilbert spaces associated to a single particle.

the canonical form displayed in Eq.(6.1):

$$|\Psi\rangle = \sqrt{\lambda_i} \sum_i |\psi_i\rangle \otimes |\varphi_i\rangle \quad (6.1)$$

where λ_i are called *Schmidt coefficients* and the local bases $\{|\psi_i\rangle\}$ and $\{|\varphi_i\rangle\}$ are guaranteed to always exist. The sum is limited by the dimension of the smaller Hilbert space, i.e., $\mathcal{H}^{(1)}$ or $\mathcal{H}^{(2)}$. The non-local properties of the state are encoded in the positive Schmidt coefficients, which tell us whether the state is separable or not: in fact, if at least two Schmidt coefficients $\sqrt{\lambda_i}, \sqrt{\lambda_j}$ are different from zero, it is not possible to express the state in a separable form, meaning that it is entangled.

Quantum entanglement in multipartite systems of pure states is not easy to classify as in the bipartite case. Indeed, when addressing *multipartite entanglement*, which refers to correlations between more than two subsystems, it is not enough to know whether the subsystems are entangled or not, but also *how* they are entangled. There are different ways in which a pure state $|\Psi\rangle \in \mathcal{H}^{(1)} \otimes \dots \otimes \mathcal{H}^{(N)}$ of an N -partite system can be entangled. In particular, in tripartite systems of qubits –i.e., systems represented by the Hilbert space $\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2$ – we can have separable states, biseparable states and two kinds of *tripartite* entangled states, i.e., Greenberger-Horn-Zeilinger (herein GHZ) states [45] which have the form $|GHZ\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$ and W -states, which are entangled states of three qubits having the form $|W\rangle = \frac{1}{\sqrt{3}}(|001\rangle + |010\rangle + |100\rangle)$. GHZ and W states are locally inequivalent, i.e., they cannot be transformed in each other by means of stochastic local operations assisted by classical communication (herein SLOCC) [31].

In the case of multipartite entanglement, as in the bipartite one, there exists a *generalized Schmidt decomposition* (herein GSD) [2, 23] which will be used by the classification approach considered in this work. GSD allows us to put any state in a canonical form, which is then used as a starting point for other algorithmic steps, involving entanglement measures. We use *concurrence* [88] as the measure to quantify bipartite entanglement in multipartite systems, as defined in Eq.(2.85), $\mathcal{C}(\rho) = \max(0, \lambda_1 - \lambda_2 - \lambda_3 - \lambda_4)$ where the λ_i are the non-negative eigenvalues, in decreasing order, of the non-hermitian matrix $\rho\tilde{\rho}$. Here $\tilde{\rho}$ is the matrix given by $\tilde{\rho} = (\sigma_y \otimes \sigma_y) \rho^* (\sigma_y \otimes \sigma_y)$ where ρ^* is the complex conjugate of ρ ² when it is expressed in a standard basis such as $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ and σ_y represents the Pauli Y operator. In this paper, since we focus on the case of three-qubit systems, we use *tangle* [26] to assess tripartite entanglement. In fact, in tripartite systems, tangle allows the quantification of entanglement between states that are not pairwise entangled. However, tangle cannot properly quantify the tripartite entanglement of W state since $\tau(|W\rangle) = 0$) but it still remains a well-known quantity allowing to distinguish GHZ-type (which have non-vanishing tangle) and W -type states. Given $\mathcal{H} = \mathbb{C}_A^2 \otimes \mathbb{C}_B^2 \otimes \mathbb{C}_C^2$, tangle is defined as in Eq.(6.2):

$$\tau = \mathcal{C}_{A(BC)}^2 - \mathcal{C}_{AB}^2 - \mathcal{C}_{AC}^2, \quad (6.2)$$

where $\mathcal{C}_{A(BC)} ::= \sqrt{2(1 - \text{Tr}(\rho_A^2))}$ is called *purity* (and can be regarded intuitively as a bipartite concurrence), and Tr represents the *trace* function in its usual definition.

²i.e., the density operator associated to the quantum state under consideration, according to its usual definition

Understanding *how* multipartite systems are entangled is a formal problem that, if solved efficiently, has positive effects in many applications. In order to do that, we aim at building a finite classification of entangled systems of three qubits, by now restricted to the tripartite case, using methods that can be further investigated from a logical and computational point of view as well (e.g., graphs and hypergraphs). Since entanglement is an important resource in QIC, its classification can be useful to verify properties of quantum protocols, in particular in the context of quantum cryptography. Formal techniques, such as *model checking*, can be used to test the reliability of a quantum protocol in a realistic scenario, thus an approach allowing us to have a finite classification of quantum entanglement in the multipartite case can be a further step in the direction of building, and then testing, realistic protocols.

The chapter is structured in the following way: Section 6.1 explores a method to classify three-qubit entanglement by means of the concept of entangled hypergraphs, with details to algorithmically build the classification; we aim at extending this approach to the multipartite case, but this problem is currently still under investigation. Section 6.2 uses the proposed classification to build a new data structure, called *evolving entangled hypergraph*, suitable to represent evolving quantum systems in which entanglement is an emergent behaviour. Section 6.3 shows an example of the proposed method, applied to a QKD protocol which uses entanglement. In the end we explore future ideas and implementation proposals.

6.1 Classification of three-qubit Entanglement

In this section we provide a classification of three-qubit pure states inspired by the approach presented in [42]. The classification proposed in this work uses the notion of GSD of three-qubit pure states, together with the concept of *entangled hypergraphs* which can be considered as a generalization of entangled graphs. In fact, we can define the latter as follows:

Definition 6.1.1 (Entangled Graph). An entangled graph is a graph $G = (V, E)$ in which each vertex $v_i \in V$ represents a qubit in multi-qubit system, and an edge $e(v_i, v_j) \in E$ between two different vertices denotes bipartite entanglement between the corresponding qubits [73].

However, by using the concept of entangled graph we can only assess bipartite entanglement in multi-qubit systems [74]. To overcome this problem, in [42], the authors used a circle including the graph indicating global entanglement in order to avoid ambiguity between corresponding GHZ and separable states in the same entangled graph. It is important to note that, when we consider systems composed by four (or more) qubits, it is not enough to investigate just bipartite and global entanglement, thus it would be more beneficial to use another data structure; for this reason we exploited the concept of entangled hypergraph. According to the latter consideration, we decided to associate an entangled hypergraph to each possible entangled state (see Fig. 6.1).

Definition 6.1.2 (Entangled Hypergraph). An entangled hypergraph $G = (V, H_G)$ is a graph in which each vertex $v_i \in V$ represents a qubit of a multi-qubit system and each hyperedge $h_k(v_i, v_j, \dots, v_m) \in H_G$ (also called k -edge) links $k \geq 2$ vertices, indicating k -partite entanglement between the corresponding qubits.

Table 6.1: Three-qubit GSD's coefficients that make nonzero concurrences and tangle

$\mathcal{C}_{1,2}$	$\mathcal{C}_{1,3}$	$\mathcal{C}_{2,3}$	τ
(λ_0, λ_3)	(λ_0, λ_2)	(λ_1, λ_4)	(λ_0, λ_4)
		(λ_2, λ_3)	

In [2], it has been shown a GSD decomposition such that, for any three-qubit pure state, there exist a local base allowing to rewrite the state in a unique canonical form, by using a set of five orthogonal tensor product states. This can be expressed as in Eq. (6.3):

$$\begin{aligned}
 |\Psi\rangle_3 &= \lambda_0|000\rangle + \lambda_1 e^{i\theta}|100\rangle + \lambda_2|101\rangle + \lambda_3|110\rangle + \lambda_4|111\rangle, \\
 \lambda_i &\geq 0, \quad 0 \leq \theta \leq \pi, \quad \sum_i \lambda_i^2 = 1.
 \end{aligned} \tag{6.3}$$

The first step of our approach is thus to compute the GSD of the state under consideration. Now, let us consider all the possible bipartite factorizations with nonzero concurrence of this state, i.e., $\mathcal{C}_{i,j} \neq 0 \quad \forall i, j$ with i, j referring to the i -th and j -th qubits. This allows us to find coefficients of the GSD which make nonzero concurrences and which guarantee a weighted edge (i.e., 2-edge) between two vertices. Then, in addition, we also need to consider tripartite entanglement, which corresponds to global entanglement in the case of tripartite systems. Hence, we use the tangle and find coefficients of the GSD which make nonzero tangle, i.e., $\tau \neq 0$, providing a weighted 3-edge. These steps are summarized in Table 6.1.

Permutations of entangled hypergraphs can be considered by labeling the vertices. For instance, in Figure 6.2 we labeled the vertices of entangled hypergraphs in the biseparable case. In this way, we can have a relation between this classification and the SLOCC classification, since we know there are three different SLOCC classes of biseparable states, but all of them belong to the same family. Since the entangled hypergraphs corresponding to separable and W states are symmetric, i.e., permutationally invariant, and the ones corresponding to GHZ-type states contain an hyperedge, we have not labeled their vertices, while we labelled the biseparable ones.

It is important to notice that, if we draw all the possible hypergraphs, there are some of them not corresponding to any entangled pure state (i.e., starting from the hypergraphs, it is not possible to retrieve the associated state). We call this kind of structures *forbidden entangled hypergraphs*. In the three-qubit case, the forbidden entangled hypergraph is the one with two edges but no hyperedge, i.e., with no global entanglement. Indeed, for every possible choice of the coefficients to have two edges, we always end up having either the third edge or the hyperedge (see Fig. 6.2).

6.2 Evolving Entangled Hypergraphs

The core part of this work is devoted to use the classification introduced in Section 6.1 as a starting point to model quantum protocols which use entanglement as a resource.

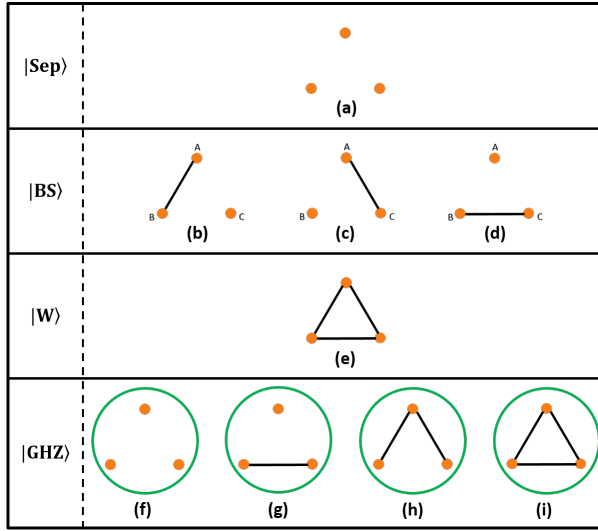


Figure 6.1: Classification of three-qubit entanglement in terms of entangled hypergraphs

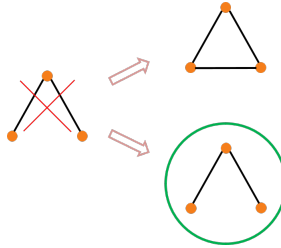


Figure 6.2: Forbidden entangled hypergraph of 3-qubit entanglement

We propose the concept of *evolving entangled hypergraph* (herein EEH), a data structure inspired by both concepts of entangled hypergraphs and hypergraph states[79], taking into account not only correlations (i.e., entanglement) but also interactions and the time evolution of the system under consideration. Interactions in this case are considered in the context of a dynamical process, i.e., the evolution of the quantum system in successive, discrete time steps. Since we wanted to keep track of patterns in the underlying hypergraphs, we decided to create a structure that merged the notion of entangled hypergraphs together with the one of *multilayer graphs*. This approach turns out to be useful, in particular, when we try to model quantum protocols in which entanglement is an important resource to be preserved, e.g., QKD and teleportation.

Multilayer graphs belong to the family of evolving graphs, also known as temporal graphs [50]. Evolving graphs highlight the change in time of a graph. According to [67], if in an evolving graph $G = (V, E)$ the time is taken discrete and only the relationships (i.e., edges) between entities (i.e., nodes) may change leaving the graph topologically unchanged, then G is a sequence G_1, \dots, G_n of static graphs over the same set of nodes. Evolving graphs [37] have also been proposed as a theoretic model in order to capture

the changes in time of a dynamic network topology. According to [22], they are suitable to analyze the quality of a communication protocol, since its *history* is made explicit as a sequence of graph topologies. A *multilayer graph* [15] is an evolving graph made of layers which are distinct copies of the main *spatial* graph, i.e., a graph in which nodes or edges are spatially located according to a certain metric. Each layer is then connected to its neighbour according to some measure of time (e.g., causality, probability, etc) which encodes the evolution of the network.

The multilayer graphs approach is thus suitable to describe the behavior of a dynamic system taking into account both spatial and temporal dimension. For this reason, we decided to use EEHs as a method to represent quantum dynamics in which entanglement is an emergent behavior.

Definition 6.2.1 (Evolving Entangled Hypergraph). Let $G = (V, H_G)$ be an entangled hypergraph. An evolving entangled hypergraph $EEH = (L, H)$ is a evolving multilayer graph in which:

1. $L = \{L_0, \dots, L_{t-1}\}$, with the variable t representing the timesteps, is a set of *layers*. Each layer $L_i \in L$ represents an instance of G at time t_i ;
2. $H = \{H_1, \dots, H_{t-1}\}$ is the set of hyperedges from a layer to the following one. Each hyperedge $h_i \in H$ is labeled with a CPTP (i.e., completely positive and trace preserving) linear map acting on the states of G .

In other words, H contains edges from a layer to the following one, which are called *inter-layer* hyperedges, while edges within a single layer L_i are called *intra-layer* hyperedges.

Each L_i corresponds to one of the allowed entangled hypergraphs, belonging to the classification presented in Section 6.1 (i.e., one of the allowed equivalence classes). Figures (6.3, 6.4) show two examples of hypothetical EEHs in a tripartite system. It is important to note that the representation of states is given in the density matrix formalism, since it allows a broader set of unitary and non-unitary evolution operators. In Fig.6.3, an initial state ρ (i.e., a given state of three qubits) at time t_0 is biseparable and, after the evolution through two given CPTP maps \mathcal{E}_1 and \mathcal{E}_2 ends in a completely separable state at time t_2 . The information extracted by this (hypothetical) model allows us to know that, during the execution of the quantum protocol under consideration, in the quantum channel \mathcal{E}_2 something occurred, causing decoherence.

In the example in Fig. 6.4, we show an EEH in which the state ρ is in a completely separable configuration at time t_0 . Then, after the evolution through the maps $\mathcal{E}_1, \dots, \mathcal{E}_n$, at time t_n it becomes totally entangled, allowing us to witness a process of entanglement creation along the channel.

Intra-layer edges $e_i \in E$ are not labelled; a weight can eventually be added, referring to the amount of entanglement between two qubits, quantified by concurrence, or by von-Neumann entropy. Nodes $v_i \in V$ in the EH (i.e., within a layer) are labeled and should be interpreted as spatial locations of qubits (i.e., qubit positions) instead of qubit “names”. We suggest that this subtle difference, even if not used in this early stage of the work, might become more important when dealing with identical particles, thus the notion of *indistinguishability* will be enforced by using locations of qubits instead of labeling them.

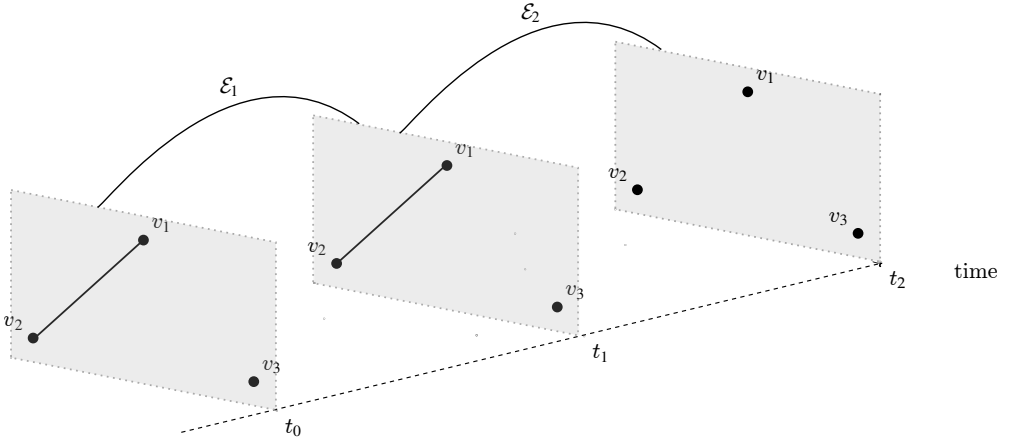


Figure 6.3: Example of an EEH in which decoherence occurs.

The EEH model proposed in this work is an *abstract* structure taking in input a class L_i from a finite set L of equivalence classes, evolving according some CPTP map \mathcal{E} and producing as output another class from the same set; i.e., $\{L_i\} \xrightarrow{\mathcal{E}} \{L_j\}$. The main purpose of EEHs is to create a structure that models properties emerging from quantum dynamics like, in this case, entanglement. Such properties must be suitable to be verified using formal methods such as model checking and logic, thus it is important to keep EEHs as abstract as possible. This is useful since by abstracting the domain, we “forget” some details, making the model computer representable.

We do not aim at computing the numerical representation of the operators which are labelling the inter-layer hyperedges. In this way, the formal verification tool should rely not on an explicit –and computationally expensive– representation of the CPTP maps, but insted it focuses on the abstract representation of their action on the graph. Using this approach, which can intuitively be regarded as an *abstract interpretation* one, a formal verification tool gains information about the semantics and the properties of the quantum protocol under consideration, without performing calculations.

In the EEHs, each layer representation (i.e., sub-hypergraph G) encodes an action induced by the CPTP map entering it. In this way a quantum channel (i.e., CPTP map) has a representation at the hypergraph level and its behavior in time removes (or adds) *intra-layer* edges. The rules determining how many edges are allowed to be removed (or added) refer to the classification presented in the first part of this work.

6.3 Example: Modeling Quantum Protocols

In this section we show an application for EEHs by using them to model two quantum protocols, i.e., teleportation and QKD with W states. We suggest that EEHs are suitable to model quantum dynamics in which entanglement is a property to be preserved or detected, because their structure allows to track the structural and morphological evolution encoding both spatial and temporal behaviors of an evolving quantum system.

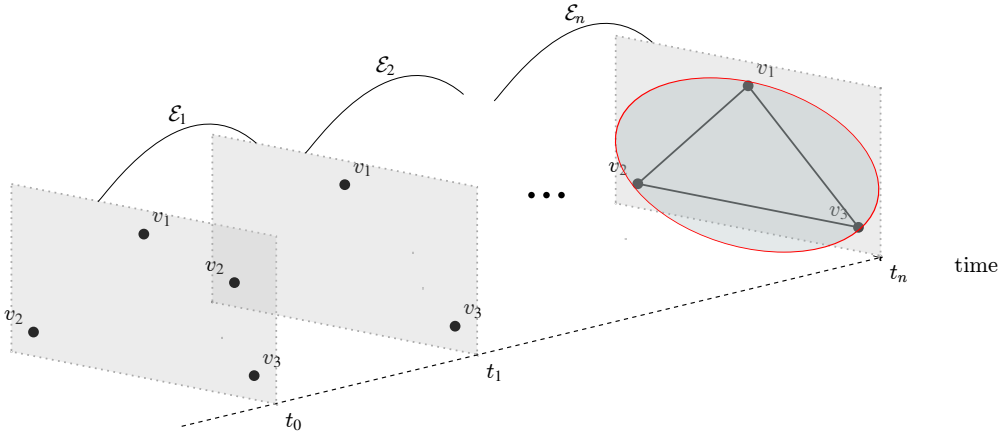


Figure 6.4: Example of an EEH in which entanglement creation occurs.

EEHs are abstract structures, thus they allow us to remove unwanted, and computationally expensive, information by focusing on the structure of the hypergraph. Moreover, they can be constructed by using an algorithmic approach.

In the following we will provide two instances of “real world” quantum teleportation protocols in which tri-partite entanglement is used as a resource and for which EEHs provide a good model. We represent the protocols by using both a Quipper-like [44] pseudocode and a graphical representation of the associated EEH.

It is also important to stress that, since this is just a preliminary work, we do not consider measurements in detail, which will be further investigated in a future extension. Instead, in this work we will focus our attention just on the channel.

6.3.1 Teleportation Protocol

Quantum teleportation is a protocol allowing to transmit quantum information (e.g. a quantum state) from one location to another, using both classical communication and quantum entanglement between the sender and the receiver. In this context we abstract from the underlying physical and mathematical details, since they are out of the scope of this preliminary work; further references can be found in [14].

In Fig. 6.5 we provided a pseudocode implementation for the teleportation protocol together with its circuit representation, where the *EPR* gate creates an entangled (e.g., Bell) state of the form $\frac{1}{\sqrt{2}}|00\rangle + |11\rangle$ and *CN* represent the standard *controlled-not* gate.

In the case in which no noise along the channel occurs, this protocol –without considering measurement and post measure corrections– can be summarized as follows (note that normalization factors are omitted for simplicity):

```

teleport :: (Qubit, Qubit, Qubit) ->
  Qubit
teleport (q1, q2, q3) :: do
  reset_at (q2,q3)
  EPR_at (q2,q3)
  CN_at q2 controlled q1
  hadamard_at q1
  c1 <- measure q1
  c2 <- measure q2
  if c1==1 && c2==1
    then do
      gate_Y_at q3
    else if c1==1 && c2==0
      then do
        gate_Z_at q3
      else if c1==0 && c2==1
        then do
          gate_X_at q3
        else do
          identity_at q3
  return q3

```

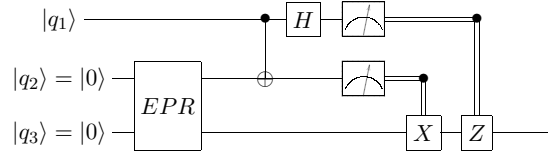


Figure 6.5: Teleportation Pseudocode and Circuit

$$\begin{aligned}
& |q_1 \ 0 \ 0\rangle && \text{separable} \\
\frac{EPR}{\longrightarrow} & |q_1\rangle(|00\rangle + |11\rangle) && \text{biseparable} \\
\frac{CN}{\longrightarrow} & |0\rangle(|00\rangle + |11\rangle) + |1\rangle(|10\rangle + |01\rangle) && \text{fully entangled} \\
\frac{H}{\longrightarrow} & |+\rangle(|00\rangle + |11\rangle) + |-\rangle(|10\rangle + |01\rangle) && \text{fully entangled}
\end{aligned} \tag{6.4}$$

where $|\pm\rangle$ represent the state $|0\rangle \pm |1\rangle$. It is important to note that the application of a controlled-not gate allows to “extend” the entanglement to the first subsystem creating a fully entangled state which can be then reduced to a *GHZ*-like state with both bipartite and tripartite entanglement, according to the underlying classification.³ This process results in the EEH in Fig. 6.6, in which the quantum gates are represented by $\mathcal{E}_{\text{gate-name}}$, i.e., the quantum operations associated to the circuit gate. This choice allows to deal with more realistic scenarios, in which noise and decoherence (i.e., processes that are not unitary) may occur.

Let us now consider the same teleportation protocol in which we add a *phase flip* channel instead of an ideal, not noisy one. This simulates a situation in which a loss of coherence may occur and can be realized by the pseudocode in Fig. 6.7, in which `PhaseFlip_at` represents a phase flip channel.

Given a generic state ρ , the phase flip channel is represented by the quantum operation $\rho' = \sum_{i=1}^2 E_i \rho E_i^\dagger$, with $E_1 = \sqrt{p}\sigma_z$, $E_2 = \sqrt{1-p}\mathbb{I}$ and $0 \leq p \leq 1$.⁴ After the application of the channel, the sum of the entries of the off-diagonal of the density matrix ρ' is lower than in the initial state; by iterating the process k times, the state

³The calculations have been performed by using the QETLAB[53] toolkit for MATLAB, computing both von-Neumann entropy –function `Entropy`– and concurrence –function `Concurrence` of the subsystems.

⁴ σ_z is the Pauli Z operator, applied to the whole system and \mathbb{I} the identity matrix.

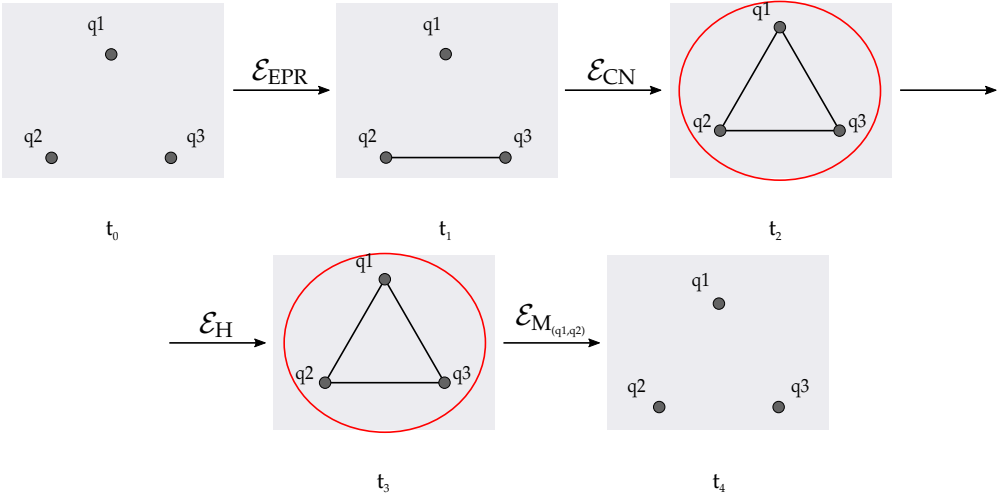


Figure 6.6: EEH for an ideal teleportation protocol.

loses all the information about coherences (i.e., decoherence), becoming less entangled with the rest of the system at each iteration. In this case we suppose that the probability p is high enough to guarantee a complete decoherence with just one application of the channel. Thus, the EEH associated is the one in Fig.6.8. The hypergraph is different from the previous one in two layers, namely t_2 and t_3 , and since the entanglement is not modelled in the same way as before, it is not possible to guarantee the effectiveness of the protocol, which could fail to teleport the correct state.

Different teleportation methods, i.e., through quantum channels containing more than two entangled qubits, have been investigated. In [55, 70] tripartite GHZ and W states can also be used as quantum channels for teleportation, which can be an interesting case study, since they deal with more than three qubits.

6.3.2 QKD using W states

In this example we consider a quantum key distribution protocol using W states, as presented in [54, 49]. W states, due to their pairwise entanglement, have been considered suitable and robust configurations for QKD protocols. Indeed, after tracing out one subsystem, there is still the possibility of bipartite entanglement, while the GHZ state, once that a subsystem have been traced away, becomes completely separable. The QKD protocol via W states between three parties can be summarized as follows:

1. The three parties share respectively one qubit each, which belongs from a previously entangled tripartite W state;
2. They randomly choose a basis to locally measure their qubit (e.g., x or z -basis);
3. Each part announces a bit of information on the basis of the local measurement (not the outcome);


```

teleport :: (Qubit, Qubit, Qubit) ->
  Qubit
teleport (q1, q2, q3) :: do
  reset_at (q2, q3)
  EPR_at (q2,q3)
  PhaseFlip_at (q1,q2,q3)
  qnot_at q2 controlled q1
  hadamard_at q1
  c1 <- measure q1
  c2 <- measure q2
  if c1==1 && c2==1
    then do
      gate_Y_at q3
    else if c1==1 && c2==0
      then do
        gate_Z_at q3
      else if c1==0 && c2==1
        then do
          gate_X_at q3
        else do

```

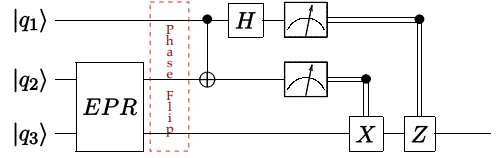


Figure 6.7: Teleportation Pseudocode and Circuit with Noise

4. For security reasons there might be a requests to announce the outcomes at random, to discover if previously eavesdropping has occurred;
5. If the three measurement basis are not (z, x, x) , (x, z, x) or (x, x, z) , then the protocol is restarted;
6. If the outcome of the part who measured in the z -basis is $|0\rangle$ then the protocol ends and the other two parties know for sure that they have the same outcome, otherwise the protocol is restarted.

This protocol heavily relies on the preservation of the entanglement of the W -states. If the entanglement is preserved, then the protocol ends in a success, otherwise, if noise along the channel or eavesdropping occur before the measurement, the protocols ends in a failure and should be restarted. For this reason we will focus only on the part of the protocol in which the entangled state is generated and transmitted along the channel, i.e., the first point of the above list. In Fig. 6.9 we provide the pseudocode implementation for the QKD protocol via W states, together with the portion of EEH relative to the pre-measurement part, where the identity operator \mathcal{E}_I refers to a channel without noise.

In Fig. 6.10 we can see how the EEH associated might change if eavesdropping, denoted by EVE , or any other source of noise occurs. The two EEHs are different, and just by comparing their structure we can note that something happened and the second protocol will result in a possible failure. Moreover, if an hypergraph is represented by one of the forbidden configurations, we might also infer that something within our specification, or the underlying language/hardware, is faulty.

6.3.3 Technical improvements and implementations

In this Section we presented two examples of EEHs relative to the teleportation protocol and a simplified version of a QKD protocol with W states. We are currently developing

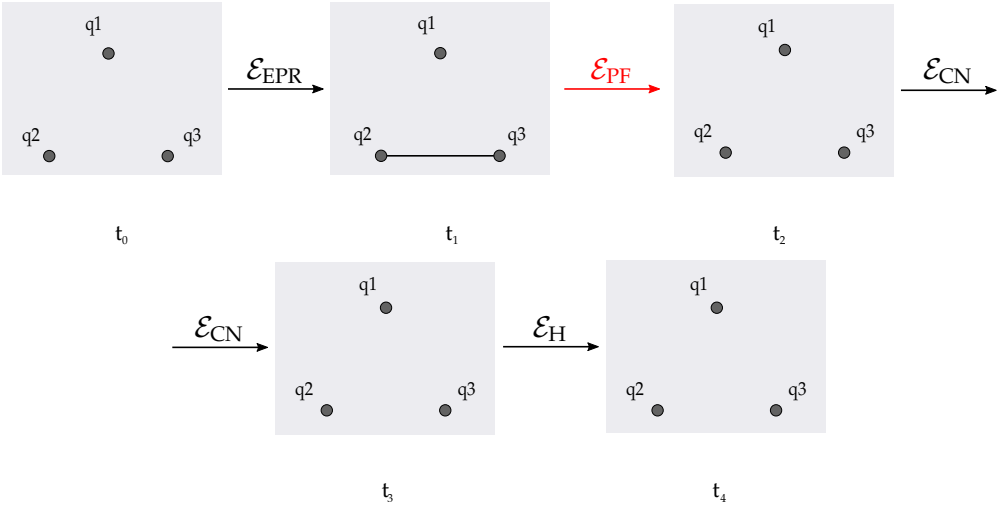


Figure 6.8: EEHs for a teleportation protocol with noise.

```

qkdW :: (Qubit, Qubit, Qubit) -> (Qubit, Qubit, Qubit)
qkdW (q1, q2, q3) = do
  W_at (q1, q2, q3)
  A <- measure_x-z q1
  B <- measure_x-z q2
  C <- measure_x-z q3
  if (axis(A,B,C)==(z,x,x) || axis(A,B,C)==(x,z,x) ||
      axis(A,B,C)==(x,x,z)) && C == |z+>
  then do
    return (m1,m2,m3)
  else qkdW(q1, q2, q3)

```

an automatic tool that allows to build the EEH from its specification in a quantum programming language, e.g., Quipper. The tool, which is still in its infancies and by now requires a numerical representation of the states, is structured as follows: first, given a pure state in the standard computational basis, it returns its entangled graph by means of GSD and pairwise concurrence quantification/tangle. Since a quantum protocol can be thought as a sequence $state \rightarrow state$, the complete EEH is built after the computation of all the intermediate states. Since its behavior in time is represented as a causal process, we are investigating whether to use a guarded command language such as PRISM to model the abstract EEH (before the generation of its explicit visual representation) and its transition from a state to the following one.

We are working on a visualization of the graph, which at each step will be represented as a lattice, or a grid of pixels. At present, we use as *incidence matrix* of the hypergraph an instance of Table 6.1, where, in order to render graphically the most explicit visual representation of the EEH topology, the columns represent concurrence and tangle and the rows represent the qubits. The changing in time of the shapes within the grid might be useful to investigate possible patterns associated to certain protocols and entangle-

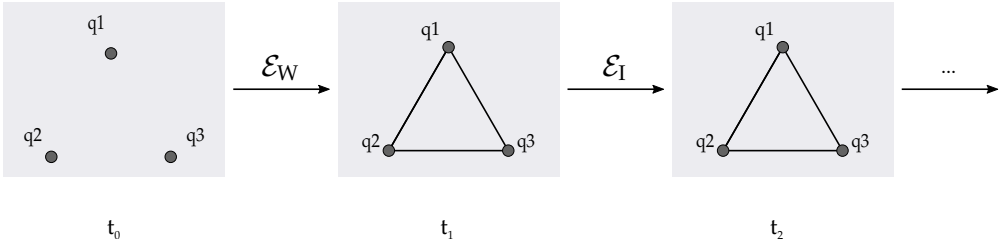


Figure 6.9: QKD with W states pseudocode and EEH.

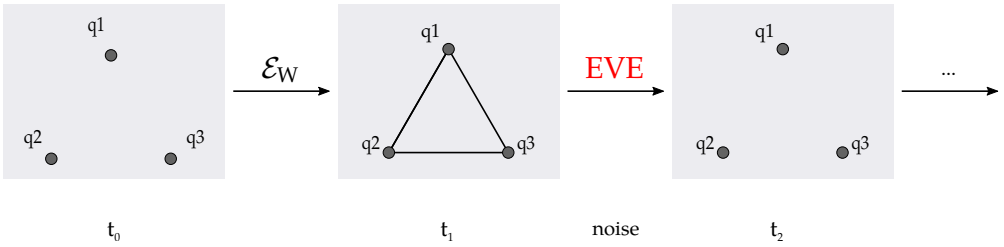


Figure 6.10: QKD with W states EEH noisy channel.

ment classes and the color can be related to the “amount” of entanglement measured by concurrence and tangle. As an example, let us consider a W state, whose entangled graph is taken unweighted for simplicity; by computing both pairwise concurrences and tangle we obtain its incidence matrix:

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

where $M_{i,j} = 1$ means that the qubit i has the property j . In our example, the qubit couples (v_1, v_2) , (v_1, v_3) , and (v_2, v_3) are pairwise entangled, but there is no tangle between them. This can be translated in the visual representation of Fig. 6.11.

More optimized structures, e.g., triangular graphs, could be proposed in an extended work.

We are also planning to provide the possibility to perform formal verification on the model obtained. Such verification can be done both temporally, i.e., by using model checking and a suitable temporal logic, e.g., QCTL, and spatially, by using a quantum extension of the SSTL spatial-temporal logic, presented in [19]. The last one in particular is currently used to analyze emergent behaviors in dynamic systems (e.g., morphogenesis and pattern formation). A possible example of property to be verified on the grid can be the following (proper syntax and semantics are still to be investigated): $Q=? [F(S[w(v_i, v_j)>0])]$.

where:

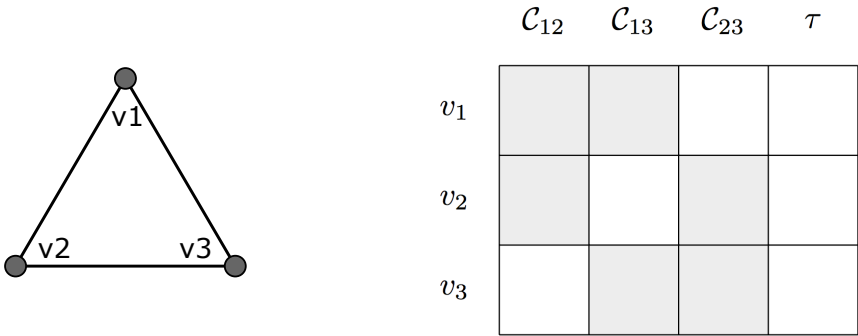


Figure 6.11: Graphical representation of a W state EH via incidence matrix

- $w(v_1, \dots, v_n)$ represents the weight (i.e., in this case the entanglement property) on a list of vertices;
- $F\phi$ is the temporal *in the future* operator (as in the usual model checking theory) stating that, at a certain point during the execution of the protocol, the formula ϕ must be true;
- $S\phi$ is the spatial *somewhere* operator, which requires the formula ϕ to hold in a location reachable from the current one;
- $Q \sim_{\mathcal{E}} [\phi]$, with $\sim \in \{\geq, =, \leq\}$ is the QCTL quantum probability operator, which states that the probability that the formula ϕ is verified is bounded by a probability expressed by the quantum operation \mathcal{E} .

Roughly speaking, the formula investigates which is the probability that, starting at time t_0 , in the future there will be an entangled couple in the graph. Again, we stress that this can be a direction in which this work can be extended.

The hypergraph structure is also suitable be used in machine learning tasks, both from the classification and the verification point of views, thus further investigations on a mixed approach (ML and MC techniques) should be carried on. Moreover, proofs about the mapping from the concrete domain (the quantum system) and the abstract one (the EEH) will be provided in the extended version of this work.

6.4 Summary and Future Work

In this chapter we proposed a new data structure, allowing to represent protocols in which entanglement is an important property, called Evolving Entangled Hypergraph (EEH). The data structure is based on a classification of tripartite entanglement which, in turn, uses entangled hypergraphs (EHs).

The EEH model is an abstract structure taking in input a class L_i from a finite set L of equivalence classes, evolving according some CPTP \mathcal{E} and giving in output another class from the same set. This model allows us to track interactions between qubits of a multipartite system and the evolution of the quantum system under consideration. We

suggest that it will allow to verify whether the entanglement properties are preserved by the time evolution or not. Since we want to model real world quantum systems, we assume that the evolution may not be unitary, and that some coupling with the environment may cause noise and decoherence, thus destroying the entanglement which is supposed to be preserved for the success of the quantum protocol under consideration.

The advantages in using the entanglement classification with EHs is that it is a finitary method and that it is not required to compute the mathematical representation of operators. The hypergraph is computed by an algorithmic procedure, for which we are planning to build a tool to automate the process. In order to extract information about the entanglement from the system, modeled as an EEH, we can work on an high level structure. Then, by having an explicit classification declaring which are the allowed and forbidden configurations, we can verify whether the structure matches the requirements.

This method, which is still in its infancy, needs further investigation in different directions in order to provide both a reliable classification of multipartite entanglement for systems with more than three particles and a formal model, suitable to represent and formally verify quantum protocols. Moreover, it is required to understand whether the evolution of an EH does map it into a single class or into a set of classes. We also may need to deal with protocols which do not use tripartite entanglement only but also the multipartite one, which now has been proposed as a resource for QKD protocols, for example in [33].

Finally, we suggest that EEHs can be used to perform formal verification of quantum protocols. In particular, in the context of model-checking EEHs provide a good model of the protocol execution, on which automatic and logical verification can be performed. Since they encode also a spatial dimension, we will be able to both define spatial locations of qubits (i.e., the vertices of the EH can be interpreted as qubit positions) and the temporal behavior of the system. We suggest that a spatial-temporal logic such as SSTL [19] can be used to verify a protocol modelled by an EEH. In this way we will be able to query the model whether a property of the system (e.g., entanglement) is preserved at a certain time in a specific location.

An important improvement that should also be investigated is whether EEHs can be used to model quantum systems of identical particles, by exploiting the spatial locations of qubits instead of labeling them.

We are planning to extend the work proposed here with further formal details and proofs from both the classification (i.e., whether it is possible to use the presented method to classify also ≥ 4 -partite entanglement or not) and the abstract interpretation point of view. A tool which, given a state, provides the corresponding entangled hypergraph is currently under development.

Conclusion and Future Work

7.1 Future Work

7.1.1 *Quipks*: improving `Entangle`

`Entangle` heavily relies on the QPMC IscasMC-based model checker. QPMC is designed to be used as a web-based tool, and it displays some drawbacks which are difficult to fix. For instance, there is not direct access to the source code, hence it is not possible to extend the set of formulae with those allowing to test directly some classical and quantum properties, e.g., the *von-Neuman entropy* or *concurrency* among others.

For this reason, together with a group of supervised M.Sc. students (thank you Francesco Saverio Comisso and Luca Foschiani for your valuable work) we have started developing a prototype model checker for quantum circuits. The tool, called *Quipks*, has been built upon the existing code of `Entangle` and it is a framework for debugging *Quip-E* code, with the main aim to replace the current model checker, i.e., the aforementioned QPMC, with a custom version, always based on PRISM.

The general idea of the model checker is to build a discrete time Markov chain representing all the possible behaviours of a given quantum circuit and then passing the model to PRISM, in order to check LTL properties. `Entangle` already had an internal representation of the *Quip-E* circuit (i.e., `CircTree`). Our code generates a labelled graph from the tree, where each node is a state (possibly in a superposition of classical states) and each edge represents the application of a unitary gate over a given set of qubits. *Quipks* performs a visit of the generated graph and assesses the probability of a state to evolve into another one. We keep track of the entangled states by using the classical state vector representation. For every visited edge we generate the unitary and measurement gates by computing the matrix representation of unitary and measurement operators (which label the edges) and an explicit vector representation for the states; then we perform a matrix-vector multiplication in order to update the state. We do not store any information about the quantum state into the Markov chain, keeping track only of the measured bits, one for each qubit. If a qubit has not been measured yet, then its associated bit is set to zero. Even if this approach is very straightforward, it

revealed to be effective due to the laziness of the product operator and the sparsity of the matrices. The Markov chain is then stored in a format suitable for PRISM, and we also generate a `.dot` file readable by Graphviz (a tool that builds a visual representation of a graph) containing the visited labelled graph.

Since, by now, the size of the representation of the DTMC is exponential in the number of qubits, we aim to optimise the model by investigating a formalism allowing to use polynomial space. In this way it would not be necessary to store in memory all the possible paths, which will be “clustered”, using an approach similar to the one using the OBDD.

The tool, even if still requires a thorough investigation and testing, is available as a work-in-progress preview at <https://github.com/SaverioFrancesco/quipks>.

7.1.2 Applying the Evolving Entangled Hypergraphs

A second perspective work is to use the notion of evolving entangled hypergraphs as models for the formal verification of quantum protocols and aims at the development of a quantum version of the Spatial-Temporal logics in [19], since in the EEHs we will be able to both define spatial locations of qubits (i.e., the vertices of each EH can be interpreted as qubit positions) and the temporal behavior of the system. In this way, we suppose that we could perform spatio-temporal model checking using EEHs as models, in order to assess the emergent behaviour of quantum properties (e.g., entanglement) and pattern formation related to the evolution of a quantum system. An important improvement that should also be investigated is whether EEHs can be used to model quantum systems of identical particles, by exploiting the spatial locations of qubits instead of simply labeling them.

Further investigations should be done in different directions, in order to provide a reliable classification of multipartite entanglement for systems with more than three particles, to understand whether the evolution of a EH does map it into a single class or into a set of classes and to assess if our proposal can be modelled as an abstract interpretation.

7.2 Conclusion

In this thesis we aimed at the presentation of a high-level unification framework for quantum programming and quantum model checking. The main motivation leading towards this goal lies in the substantial lack of a formalism allowing this kind of integration when dealing with quantum computation.

We have developed a framework allowing on the one hand the description of quantum algorithms in an high-level programming language, and on the other hand their formal verification through model checking. In order to achieve this goal we had to perform a translation from the semantics of the programming language, to the semantics of the model checking guarded command language.

The aforementioned framework, that we called **Entangle**, allows to define –by using a sublanguage of the quantum programming language Quipper, that we defined and called *Quip-E*– and automatically verify –by using the quantum model checker QPMC–

formal properties of quantum algorithms and protocols by abstracting away from low-level features.

Quip-E's syntax and operational semantics have been formalised and defined, allowing to provide a notion of **Body** of a tail-recursive quantum program, i.e., programs in which the results of a measurement are used as guard conditions for recursive calls.

Entangle has been implemented using Haskell, which allowed to re-use libraries already developed for Quipper, and has been provided with a web based GUI, allowing a more intuitive approach.

Different tests have been performed by using **Entangle**. In particular, in this thesis we presented two different versions of the Deutsch-Jozsa algorithm, and instances of the Grover's search, which allowed also to define a quantum switch program, recursive and non-recursive versions of the teleportation and the BB84 quantum key distribution protocols. In the end, since one of the claim behind QPMC was that the tool allowed to verify only classical properties, we provided an example in which also a quantum property was assessed, i.e., entanglement, with some custom post-processing. The translations and the tests behaved as expected, validating in this way our expectations.

We worked on the enrichment and the optimisation of **Entangle**, in order to match the requirements of validating more complex algorithms and protocols. Some steps have been moved in this direction, by implementing a more efficient version of the algorithms computing the swap matrices. Indeed, the version that we proposed computes the matrix elements of a given swap operator, allowing to automatically build the swap matrix without composing binary swaps (i.e., an instance of permutation by transposition), which is a method computationally less efficient than ours. Nevertheless, more tests should be performed in order assess its correctness and to generalise it also to other operators.

A preliminary version of **Entangle**, which was limited to the translation of quantum circuits has been presented in [5]; this one was later enriched in order to deal with tail-recursive quantum programs. The updated version is freely available at <https://github.com/miniBill.entangle>.

The main perspective work on which we are already working aims at extending **Entangle** by providing a framework for debugging *Quip-E* programs, with the main aim to replace the current model checker, i.e., QPMC, with a custom version, based on PRISM. This framework, called *Quipks*, builds a DTMC representing all the possible behaviours of an input quantum program. *Quipks* generates a labelled graph from the tree-representation of the programs, where each node is a state (possibly in a superposition of classical states) and each edge represents the application of a unitary gate over a given set of qubits. The tool is already available to be tested, but is still under development, hence errors may occur.

Another contribution that we presented in this thesis is relative to the notion of *entangled hypergraphs* (EHs), a generalization of entangled graphs suitable to be applied in the context of multipartite entanglement; in an EH each vertex v_i represents a qubit, and each hyperedge e_i connecting a set of vertices represents multipartite entanglement. We proposed an algorithmic framework, based on the generalised Schmidt decomposition and entanglement measures such as concurrence and tangle, to build a classification of tripartite entanglement of pure states using EHs. Moreover, by starting from the equivalence classes modeled with EH, we can use an algebraic framework to write the

pure state associated to each class. We have found that some EHs configurations are forbidden, hence we cannot write a pure state describing them, e.g., the star shape hypergraph without a 3-edge is the example we considered in this work.

The method proposed works as expected in the tripartite case; nevertheless it is not general since some problems arise in this classification when deal with ≥ 4 -partite cases. Anyways, we argue that this method can be a good starting point for the investigation of other multipartite classification techniques, since it has a finitary approach, which is suitable for further modeling and verification tasks.

A proposal has been made in order to use the classification introduced as a starting point to *model* quantum protocols as *Evolving Entangled Hypergraph* (EEH), a structure inspired by both EH and hypergraph states, taking into account not only correlations (entanglement) but also the evolutions of the qubits under consideration. We assume that this approach can be used to model quantum protocols in which entanglement is an important resource to be preserved, e.g., QKD protocols and teleportation.

We give the definition of a EEH as a *causal multilayer hypergraph* in which each layer L_i represents the EH at time t_i and the hyperedges from a layer to the following one are labeled with the CPTP (completely positive and trace preserving) linear map acting on the states of EH. The EEH model allows us to track the interactions between qubits of a multipartite system, and thus to verify if the entanglement properties are preserved by the time evolution. Since we want to model real quantum systems, we assume that the evolution may not be unitary, and that some coupling with the environment may cause noise and decoherence, while entanglement is supposed to be preserved for the success of the quantum protocol under consideration. Moreover, we can also use forbidden EHs to identify whether the protocol is valid or not from a merely computational point of view. In the last part we also provided two examples of possible uses of EEHs.

Finally, we concluded this Thesis by proposing two different directions for our research, i.e. *Quipks*, a tool that extends `Entangle` by offering a graphical, more efficient representation of QMCs, with the final goal of building a new PRISM-based quantum model checker, and to better formalize Evolving Entangled Hypergraphs in order to apply them on quantum model checking as well.

A

QPMC code

In the following we show the examples of translated quantum algorithms, together with some results that have been omitted for space reason from the previous Chapters.

Exploiting our implementation of `Entangle` the following code has been automatically generated.

Deutsch–Jozsa Constant Oracle QMC

```
qmc

const matrix A1_T = kron(M0, ID(8));
const matrix A1_F = kron(M1, ID(8));
const matrix A2 = kron(PauliX, ID(8));
const matrix A3_T = kron(kron(ID(2), M0), ID(4));
const matrix A3_F = kron(kron(ID(2), M1), ID(4));
const matrix A4 = kron(kron(ID(2), PauliX), ID(4));
const matrix A5_T = kron(kron(ID(4), M0), ID(2));
const matrix A5_F = kron(kron(ID(4), M1), ID(2));
const matrix A6 = kron(kron(ID(4), PauliX), ID(2));
const matrix A7_T = kron(ID(8), M0);
const matrix A7_F = kron(ID(8), M1);
const matrix A8 = kron(ID(8), PauliX);
const matrix A9 = kron(ID(8), PauliX);
const matrix A10 = kron(Hadamard, ID(8));
const matrix A11 = kron(kron(ID(2), Hadamard), ID(4));
const matrix A12 = kron(kron(ID(4), Hadamard), ID(2));
const matrix A13 = kron(ID(8), Hadamard);
const matrix A14 = kron(Hadamard, ID(8));
const matrix A15 = kron(kron(ID(2), Hadamard), ID(4));
const matrix A16 = kron(kron(ID(4), Hadamard), ID(2));
const matrix A17_F = kron(M0, ID(8));
const matrix A17_T = kron(M1, ID(8));
const matrix A18_FF = kron(kron(ID(2), M0), ID(4));
const matrix A18_TF = kron(kron(ID(2), M1), ID(4));
const matrix A19_FFF = kron(kron(ID(4), M0), ID(2));
const matrix A19_TFF = kron(kron(ID(4), M1), ID(2));
const matrix A19_FTF = kron(kron(ID(4), M0), ID(2));
const matrix A19_TTF = kron(kron(ID(4), M1), ID(2));
const matrix A18_FT = kron(kron(ID(2), M0), ID(4));
const matrix A18_TT = kron(kron(ID(2), M1), ID(4));
const matrix A19_FFT = kron(kron(ID(4), M0), ID(2));
const matrix A19_TFT = kron(kron(ID(4), M1), ID(2));
const matrix A19_FTT = kron(kron(ID(4), M0), ID(2));
const matrix A19_TTT = kron(kron(ID(4), M1), ID(2));

module dJozsaConst
s: [0..19] init 0;
b0: bool init false;
b1: bool init false;
b2: bool init false;

[] (s = 0) -> <<A1_T>> : (s' = 1) & (b0' = true) + <<A1_F>> : (s' = 1) & (b0' = false);
[] (s = 1) & b0 -> (s' = 2);
```



```

const matrix A22_TTT = kron(M1, ID(8));

module dJozsaBal
s: [0..22] init 0;
b0: bool init false;
b1: bool init false;
b2: bool init false;

[] (s = 0) -> <<A1_T>> : (s' = 1) & (b0' = true) + <<A1_F>> : (s' = 1) & (b0' = false);
[] (s = 1) & b0 -> (s' = 2);
[] (s = 1) & !b0 -> <<A2>> : (s' = 2);
[] (s = 2) -> <<A3_T>> : (s' = 3) & (b0' = true) + <<A3_F>> : (s' = 3) & (b0' = false);
[] (s = 3) & b0 -> (s' = 4);
[] (s = 3) & !b0 -> <<A4>> : (s' = 4);
[] (s = 4) -> <<A5_T>> : (s' = 5) & (b0' = true) + <<A5_F>> : (s' = 5) & (b0' = false);
[] (s = 5) & b0 -> (s' = 6);
[] (s = 5) & !b0 -> <<A6>> : (s' = 6);
[] (s = 6) -> <<A7_T>> : (s' = 7) & (b0' = true) + <<A7_F>> : (s' = 7) & (b0' = false);
[] (s = 7) & b0 -> (s' = 8);
[] (s = 7) & !b0 -> <<A8>> : (s' = 8);
[] (s = 8) -> <<A9>> : (s' = 9);
[] (s = 9) -> <<A10>> : (s' = 10);
[] (s = 10) -> <<A11>> : (s' = 11);
[] (s = 11) -> <<A12>> : (s' = 12);
[] (s = 12) -> <<A13>> : (s' = 13);
[] (s = 13) -> <<A14>> : (s' = 14);
[] (s = 14) -> <<A15>> : (s' = 15);
[] (s = 15) -> <<A16>> : (s' = 16);
[] (s = 16) -> <<A17>> : (s' = 17);
[] (s = 17) -> <<A18>> : (s' = 18);
[] (s = 18) -> <<A19>> : (s' = 19);
[] (s = 19) -> <<A20_F>> : (s' = 20) & (b0' = false) + <<A20_T>> : (s' = 20) & (b0' = true);
[] (s = 20) & b0 -> <<A21_FF>> : (s' = 21) & (b1' = false) + <<A21_TT>> : (s' = 21) & (b1' = true);
[] (s = 20) & !b0 -> <<A21_FF>> : (s' = 21) & (b1' = false) + <<A21_TF>> : (s' = 21) & (b1' = true);
[] (s = 21) & b0 & b1 -> <<A22_TTT>> : (s' = 22) & (b2' = false) + <<A22_TTT>> : (s' = 22) & (b2' = true);
[] (s = 21) & b0 & !b1 -> <<A22_FTT>> : (s' = 22) & (b2' = false) + <<A22_TFT>> : (s' = 22) & (b2' = true);
[] (s = 21) & !b0 & b1 -> <<A22_FFF>> : (s' = 22) & (b2' = false) + <<A22_TFF>> : (s' = 22) & (b2' = true);
[] (s = 21) & !b0 & !b1 -> <<A22_FFF>> : (s' = 22) & (b2' = false) + <<A22_TFF>> : (s' = 22) & (b2' = true);
[] (s = 22) & !b0 & !b1 & !b2 -> true;
[] (s = 22) & !b0 & !b1 & b2 -> true;
[] (s = 22) & !b0 & b1 & !b2 -> true;
[] (s = 22) & !b0 & b1 & b2 -> true;
[] (s = 22) & b0 & !b1 & !b2 -> true;
[] (s = 22) & b0 & !b1 & b2 -> true;
[] (s = 22) & b0 & b1 & !b2 -> true;
[] (s = 22) & b0 & b1 & b2 -> true;
endmodule

```

Recursive BB84 QMC

```

qmc

const matrix A1_T = kron(M0, ID(8));
const matrix A1_F = kron(M1, ID(8));
const matrix A2 = kron(PauliX, ID(8));
const matrix A3_T = kron(kron(ID(2), M0), ID(4));
const matrix A3_F = kron(kron(ID(2), M1), ID(4));
const matrix A4 = kron(kron(ID(2), PauliX), ID(4));
const matrix A5_T = kron(kron(ID(4), M0), ID(2));
const matrix A5_F = kron(kron(ID(4), M1), ID(2));
const matrix A6 = kron(kron(ID(4), PauliX), ID(2));
const matrix A7 = kron(Hadamard, ID(8));
const matrix A8 = kron(kron(ID(2), Hadamard), ID(4));
const matrix A9 = kron(kron(ID(4), Hadamard), ID(2));
const matrix A10_F = kron(M0, ID(8));
const matrix A10_T = kron(M1, ID(8));
const matrix A11_FF = kron(kron(ID(2), M0), ID(4));
const matrix A11_TF = kron(kron(ID(2), M1), ID(4));
const matrix A12_FFF = kron(kron(ID(4), M0), ID(2));
const matrix A12_TFF = kron(kron(ID(4), M1), ID(2));
const matrix A13_TFFF = kron(ID(8), M0);
const matrix A13_FFFF = kron(ID(8), M1);
const matrix A14_FFFF = kron(ID(8), PauliX);
const matrix A15_FFFF = kron(ID(8), PauliX);
const matrix A16_FFFF = kron(ID(8), M0);
const matrix A16_TFFF = kron(ID(8), M1);
const matrix A13_TTFF = kron(ID(8), M0);
const matrix A13_FTFF = kron(ID(8), M1);
const matrix A14_TFF = kron(ID(8), PauliX);
const matrix A15_TFF = kron(ID(8), PauliX);
const matrix A16_TFF = kron(ID(8), M0);
const matrix A16_TTFF = kron(ID(8), M1);
const matrix A12_FTF = kron(kron(ID(4), M0), ID(2));
const matrix A12_TTF = kron(kron(ID(4), M1), ID(2));
const matrix A13_TFTF = kron(ID(8), M0);
const matrix A13_FFTF = kron(ID(8), M1);
const matrix A14_FTF = kron(ID(8), PauliX);

```

```

const matrix A15_FTF = kron(ID(8), PauliX);
const matrix A16_FTF = kron(ID(8), Hadamard);
const matrix A17_FFFF = kron(ID(8), M0);
const matrix A17_TFFF = kron(ID(8), M1);
const matrix A13_TTTF = kron(ID(8), M0);
const matrix A13_FFFF = kron(ID(8), M1);
const matrix A14_TTF = kron(ID(8), PauliX);
const matrix A15_TTF = kron(ID(8), PauliX);
const matrix A16_TTF = kron(ID(8), Hadamard);
const matrix A17_FFFF = kron(ID(8), M0);
const matrix A17_TFFF = kron(ID(8), M1);
const matrix A11_FT = kron(kron(ID(2), M0), ID(4));
const matrix A11_TT = kron(kron(ID(2), M1), ID(4));
const matrix A12_FFT = kron(kron(ID(4), M0), ID(2));
const matrix A12_TFT = kron(kron(ID(4), M1), ID(2));
const matrix A13_TFFT = kron(ID(8), M0);
const matrix A13_FFFT = kron(ID(8), M1);
const matrix A14_FFT = kron(ID(8), PauliX);
const matrix A15_FFT = kron(ID(8), PauliX);
const matrix A16_FFFT = kron(ID(8), M0);
const matrix A16_TFFT = kron(ID(8), M1);
const matrix A13_TFFT = kron(ID(8), M0);
const matrix A13_FFFT = kron(ID(8), M1);
const matrix A14_TFT = kron(ID(8), PauliX);
const matrix A15_TFT = kron(ID(8), PauliX);
const matrix A16_FFFT = kron(ID(8), M0);
const matrix A16_TFFT = kron(ID(8), M1);
const matrix A12_FTT = kron(kron(ID(4), M0), ID(2));
const matrix A12_TTT = kron(kron(ID(4), M1), ID(2));
const matrix A13_TFTT = kron(ID(8), M0);
const matrix A13_FFTT = kron(ID(8), M1);
const matrix A14_FTT = kron(ID(8), PauliX);
const matrix A15_FTT = kron(ID(8), PauliX);
const matrix A16_FTT = kron(ID(8), Hadamard);
const matrix A17_FFTT = kron(ID(8), M0);
const matrix A17_TFTT = kron(ID(8), M1);
const matrix A13_TTTT = kron(ID(8), M0);
const matrix A13_FTTT = kron(ID(8), M1);
const matrix A14_TTT = kron(ID(8), PauliX);
const matrix A15_TTT = kron(ID(8), PauliX);
const matrix A16_TTT = kron(ID(8), Hadamard);
const matrix A17_FTTT = kron(ID(8), M0);
const matrix A17_TTTT = kron(ID(8), M1);

module bb84RecCirc
s: [0..17] init 0;
b0: bool init false;
b1: bool init false;
b2: bool init false;
b3: bool init false;

[] (s = 0) -> <<A1_T>> : (s' = 1) & (b0' = true) + <<A1_F>> : (s' = 1) & (b0' = false);
[] (s = 1) & b0 -> (s' = 2);
[] (s = 1) & !b0 -> <<A2>> : (s' = 2);
[] (s = 2) -> <<A3_T>> : (s' = 3) & (b0' = true) + <<A3_F>> : (s' = 3) & (b0' = false);
[] (s = 3) & b0 -> (s' = 4);
[] (s = 3) & !b0 -> <<A4>> : (s' = 4);
[] (s = 4) -> <<A5_T>> : (s' = 5) & (b0' = true) + <<A5_F>> : (s' = 5) & (b0' = false);
[] (s = 5) & b0 -> (s' = 6);
[] (s = 5) & !b0 -> <<A6>> : (s' = 6);
[] (s = 6) -> <<A7>> : (s' = 7);
[] (s = 7) -> <<A8>> : (s' = 8);
[] (s = 8) -> <<A9>> : (s' = 9);
[] (s = 9) -> <<A10_F>> : (s' = 10) & (b0' = false) + <<A10_T>> : (s' = 10) & (b0' = true);
[] (s = 10) & b0 -> <<A11_FT>> : (s' = 11) & (b1' = false) + <<A11_TT>> : (s' = 11) & (b1' = true);
[] (s = 10) & !b0 -> <<A11_FF>> : (s' = 11) & (b1' = false) + <<A11_TF>> : (s' = 11) & (b1' = true);
[] (s = 11) & b0 & b1 -> <<A12_FTT>> : (s' = 12) & (b2' = false) + <<A12_TTT>> : (s' = 12) & (b2' = true);
[] (s = 11) & b0 & !b1 -> <<A12_FFT>> : (s' = 12) & (b2' = false) + <<A12_TFT>> : (s' = 12) & (b2' = true);
[] (s = 11) & !b0 & b1 -> <<A12_TTT>> : (s' = 12) & (b2' = false) + <<A12_TTF>> : (s' = 12) & (b2' = true);
[] (s = 11) & !b0 & !b1 -> <<A12_FFF>> : (s' = 12) & (b2' = false) + <<A12_TFF>> : (s' = 12) & (b2' = true);
[] (s = 12) & b0 & b1 & b2 -> <<A13_TTTT>> : (s' = 13) & (b3' = true) + <<A13_FTTT>> : (s' = 13) & (b3' = false);
[] (s = 12) & b0 & b1 & !b2 -> <<A13_TFTT>> : (s' = 13) & (b3' = true) + <<A13_FFTT>> : (s' = 13) & (b3' = false);
[] (s = 12) & b0 & !b1 & b2 -> <<A13_TFFF>> : (s' = 13) & (b3' = true) + <<A13_TFTF>> : (s' = 13) & (b3' = false);
[] (s = 12) & b0 & !b1 & !b2 -> <<A13_TFTF>> : (s' = 13) & (b3' = true) + <<A13_FFTF>> : (s' = 13) & (b3' = false);
[] (s = 12) & !b0 & b1 & b2 -> <<A13_TTTT>> : (s' = 13) & (b3' = true) + <<A13_TFTT>> : (s' = 13) & (b3' = false);
[] (s = 12) & !b0 & b1 & !b2 -> <<A13_TFTT>> : (s' = 13) & (b3' = true) + <<A13_TFTF>> : (s' = 13) & (b3' = false);
[] (s = 12) & !b0 & !b1 & b2 -> <<A13_TFFF>> : (s' = 13) & (b3' = true) + <<A13_TFTF>> : (s' = 13) & (b3' = false);
[] (s = 12) & !b0 & !b1 & !b2 -> <<A13_TFFF>> : (s' = 13) & (b3' = true) + <<A13_FFFF>> : (s' = 13) & (b3' = false);
[] (s = 13) & b0 & b1 & b2 & b3 -> (s' = 14);
[] (s = 13) & b0 & b1 & b2 & !b3 -> <<A14_TTT>> : (s' = 14);
[] (s = 13) & b0 & b1 & !b2 & b3 -> (s' = 14);
[] (s = 13) & b0 & b1 & !b2 & !b3 -> <<A14_FTT>> : (s' = 14);
[] (s = 13) & b0 & !b1 & b2 & b3 -> (s' = 14);
[] (s = 13) & b0 & !b1 & b2 & !b3 -> <<A14_TFT>> : (s' = 14);
[] (s = 13) & b0 & !b1 & !b2 & b3 -> (s' = 14);
[] (s = 13) & b0 & !b1 & !b2 & !b3 -> <<A14_FFT>> : (s' = 14);
[] (s = 13) & !b0 & b1 & b2 & b3 -> (s' = 14);
[] (s = 13) & !b0 & b1 & b2 & !b3 -> <<A14_TTT>> : (s' = 14);
[] (s = 13) & !b0 & b1 & !b2 & b3 -> (s' = 14);
[] (s = 13) & !b0 & b1 & !b2 & !b3 -> <<A14_TFT>> : (s' = 14);
[] (s = 13) & !b0 & !b1 & b2 & b3 -> (s' = 14);
[] (s = 13) & !b0 & !b1 & b2 & !b3 -> <<A14_FTT>> : (s' = 14);
[] (s = 13) & !b0 & !b1 & !b2 & b3 -> (s' = 14);
[] (s = 13) & !b0 & !b1 & !b2 & !b3 -> <<A14_FFT>> : (s' = 14);

```


Bibliography

- [1] S. Abramsky and B. Coecke. A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 415–425, 2004.
- [2] A. Acín, A. Andrianov, L. Costa, E. Jané, J. I. Latorre, and R. Tarrach. Generalized schmidt decomposition and classification of three-quantum-bit states. *Phys. Rev. Lett.*, 85:1560–1563, Aug 2000.
- [3] T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 249–258, 2005.
- [4] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, Jul 1999.
- [5] L. Anticoli, C. Piazza, L. Taglialegne, and P. Zuliani. Towards quantum programs verification: From quipper circuits to QPMC. In *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*, pages 213–219, 2016.
- [6] L. Anticoli, C. Piazza, L. Taglialegne, and P. Zuliani. Verifying quantum programs: From quipper to qpmc. Available from arXiv:1708.06312, 2016.
- [7] C. Baier and J. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [8] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, Aug 1998.
- [9] P. Baltazar, R. Chadha, and P. Mateus. Quantum computation tree logic – model checking and complete calculus. *International Journal of Quantum Information*, 2008.
- [10] F. Benatti, M. Fannes, R. Floreanini, and D. Petritis, editors. *Quantum Information, Computation and Cryptography*, volume 808 of *Springer Lecture Notes in Physics*. Springer-Verlag Berlin Heidelberg, 2010.
- [11] P. Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, 22(5):563–591, 1980.

- [12] C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing*, page 175, 1984.
- [13] C. H. Bennett and G. Brassard. Quantum Cryptography: Public Key Distribution and Coin Tossing. In *Proceedings of the IEEE International Conference on Computers, Systems and Signal Processing*, pages 175–179, New York, 1984. IEEE Press.
- [14] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Phys. Rev. Lett.*, 70:1895–1899, Mar 1993.
- [15] K. Benzi, B. Ricaud, and P. Vandergheynst. Principal patterns on graphs: Discovering coherent structures in datasets. *IEEE Transactions on Signal and Information Processing over Networks*, 2(2):160–173, June 2016.
- [16] E. Bernstein and U. Vazirani. Quantum complexity theory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, New York, NY, USA, 1993. ACM.
- [17] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *The European Physical Journal D - Atomic, Molecular, Optical and Plasma Physics*, 25(2), Aug 2003.
- [18] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 499–513, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [19] L. Bortolussi and L. Nenzi. Specifying and monitoring properties of stochastic spatio-temporal. systems in signal temporal logic. In *Proceedings of the 8th International Conference on Performance. Evaluation Methodologies and Tools, VAL-UETOOLS 2014*, pages 66–73, 2014.
- [20] H. J. Briegel, D. E. Browne, W. Dür, R. Raussendorf, and M. Van den Nest. Measurement-based quantum computation. *Nature Physics*, 2009.
- [21] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3), 1992.
- [22] B. Bui-Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(2):267–285, 2003.
- [23] H. A. Carteret, A. Higuchi, and A. Sudbery. Multipartite generalization of the schmidt decomposition. *Journal of Mathematical Physics*, 41(12):7932–7939, dec 2000.
- [24] R. Chadha, P. Mateus, and A. Sernadas. Reasoning about imperative quantum programs. *Electronic Notes in Theoretical Computer Science*, 158:19–39, 2006.

- [25] E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, 1982.
- [26] V. Coffman, J. Kundu, and W. K. Wootters. Distributed entanglement. *Phys. Rev. A*, 61:052306, Apr 2000.
- [27] V. Danos, E. Kashefi, and P. Panangaden. The measurement calculus. *J. ACM*, 54(2), April 2007.
- [28] T. S. Davidson, S. J. Gay, H. Mlnarik, R. Nagarajan, and N. Papanikolaou. Model Checking for Communicating Quantum Processes. *IJUC*, 2012.
- [29] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
- [30] D. Deutsch. Quantum theory, the church—turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 400(1818):97–117, 1985.
- [31] W. Dür, G. Vidal, and J. I. Cirac. Three qubits can be entangled in two inequivalent ways. *Phys. Rev. A*, 62:062314, Nov 2000.
- [32] O. Grumberg E.M. Clarke and D.A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
- [33] M. Epping, H. Kampermann, C. Macchiavello, and D. Bruß. Multi-partite entanglement can speed up quantum key distribution in networks. *New Journal of Physics*, 19(9), 2017.
- [34] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser. Quantum computation by adiabatic evolution. 2000.
- [35] Y. Feng, E. M. Hahn, A. Turrini, and L. Zhang. QPMC: A Model Checker for Quantum Programs and Protocols. In Nikolaj Bjørner and Frank D. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, June 24-26, 2015, Proceedings*, Lecture Notes in Computer Science. Springer, 2015.
- [36] Y. Feng, N. Yu, and M. Ying. Model checking quantum Markov chains. *Journal of Computer and System Sciences*, 2013.
- [37] A. Ferreira. On models and algorithms for dynamic communication networks: The case for evolving graphs. In *4^e rencontres francophones sur les Aspects Algorithmiques des Telecommunications (ALGOTEL'2002)*, Mèze, France, May 2002.
- [38] Richard P. Feynman. Simulating physics with computers. *Int. J. Theor. Phys.*, 21:467–488, 1982.
- [39] S. Gay, R. Nagarajan, and N. Papanikolaou. Probabilistic model-checking of quantum protocols. In *Proceedings of the 2nd International Workshop on Developments in Computational Models*, 2006.

- [40] S. J. Gay. Quantum programming languages: Survey and bibliography. In *Bulletin of the EATCS*, 2005.
- [41] S. J. Gay, N. Papanikolaou, and R. Nagarajan. QMC: A model checker for quantum systems. In *In Proceeding of the 20th International Conference on Computer Aided Verification*, 2008.
- [42] M. Gharahi Ghahi and S. J. Akhtarshenas. Entangled graphs: a classification of four-qubit entanglement. *Eur. Phys. J. D*, 2016.
- [43] A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. An Introduction to Quantum Programming in Quipper. In *Proceedings of the 5th International Conference on Reversible Computation*. Springer-Verlag, 2013.
- [44] A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. Quipper: A Scalable Quantum Programming Language. *SIGPLAN Not.*, 48(6), 2013.
- [45] D. M. Greenberger, M. Horne, and A. Zeilinger. A bell theorem without inequalities. *Am. J. Phys.* 58 1131, 1990.
- [46] S. Gudder. Quantum Markov Chains. *Journal of Mathematical Physics*, 49, 2008.
- [47] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, Sep 1994.
- [48] C. Heunen and J. Bart. Quantum logic in dagger kernel categories. *Order*, 27(2), Jul 2010.
- [49] A. Hillebrand. Superdense coding and quantum direct communication with ghz in the zx-calculus. In *Proceedings of the 8th Workshop on Quantum Physics and Logic, QPL 2011*.
- [50] P. Holme and J. Saramäki. *Temporal Networks*. Understanding Complex Systems. Springer-Verlag Berlin Heidelberg, 2013.
- [51] R. Horodecki, P. Horodecki, M. Horodecki, and K. Horodecki. Quantum entanglement. *Rev. Mod. Phys.*, 81:865–942, Jun 2009.
- [52] C.J. Isham. *Lectures on Quantum Theory: Mathematical and Structural Foundations*. Imperial College Press, 1995.
- [53] N. Johnston. QETLAB: A MATLAB toolbox for quantum entanglement, version 0.9. <http://qetlab.com>, January 2016.
- [54] J. Joo, J. Lee, J. Jang, and Y.J. Park. Quantum secure communication with w states. Available from arXiv:quant-ph/0204003, 2002.
- [55] A. Karlsson and M. Bourennane. Quantum teleportation using three-particle entanglement. *Phys. Rev. A*, 58:4394–4400, Dec 1998.
- [56] A. Kitaev and C. Laumann. Topological phases and quantum computations. *Exact Methods in Low-Dimensional Statistical Physics and Quantum Computing*, 2010.

- [57] E. Knill. Conventions for Quantum Pseudocode. Technical report, Los Alamos National Laboratory, 1996.
- [58] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *LNCS*, volume 6806, 2011.
- [59] P. Maymin. Extending the lambda calculus to express randomized and quantumized algorithms. 1996.
- [60] A. Messiah. *Quantum Mechanics*. Number v. 2 in Quantum Mechanics. North-Holland Publishing Company, 1962.
- [61] J.A. Miszczak. Models of quantum computation and quantum programming languages. *Bulletin of the Polish Academy of Sciences: Tech. Sci.*, 59, 2011.
- [62] R. Nagarajan, N. Papanikolaou, and D. Williams. Simulating and compiling code for the sequential quantum random access machine. *Electr. Notes Theor. Comput. Sci.*, 170:101–124, 2007.
- [63] M. A. Nielsen. Quantum computation by measurement and quantum memory. *Physics Letters A*, 2003.
- [64] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2011.
- [65] B. Ömer. Procedural quantum programming. In *Proceedings of the AIP conference on Computing Anticipatory Systems*, 2001.
- [66] B. Ömer. Classical Concepts in Quantum Programming. *International Journal of Theoretical Physics*, 2005.
- [67] M. Othon. An introduction to temporal graphs: An algorithmic perspective. In *Algorithms, Probability, Networks, and Games - Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, pages 308–343. Springer, 2015.
- [68] P. Mateus P. Baltazar, R. Chadha and A. Sernadas. Towards model-checking quantum security protocols. In *Proceedings of the first workshop on Quantum Security: QSEC'07*. iee Press, 2007.
- [69] Scott Pakin. Performing fully parallel constraint logic programming on a quantum annealer. *CoRR*, <http://arxiv.org/abs/1804.00036>, 2018.
- [70] A. Pankaj and P. Arun. Perfect teleportation and superdense coding with w states. *Phys. Rev. A*, 74:062320, Dec 2006.
- [71] J. Philippe and M. Lalire. Toward a quantum process algebra. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 111–119, New York, NY, USA, 2004. ACM.

- [72] J. Philippe and S. Perdrix. Innovation and intellectual property rights. In I. Mackie S. Gay, editor, *Semantic Techniques in Quantum Computation*, chapter 6. Cambridge University Press, 2010.
- [73] M. Plesch and V. Bužek. Entangled graphs: Bipartite entanglement in multiqubit systems. *Phys. Rev. A*, 67:012322, Jan 2003.
- [74] M. Plesch, J. Novotný, Z. Dzuráková, and V. Bužek. Controlling bi-partite entanglement in multi-qubit systems. *Journal of Physics A: Mathematical and General*, 37(5):1843, 2004.
- [75] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, Washington, DC, USA, 1977. IEEE Computer Society.
- [76] J. Preskill. *Lecture Notes for Physics 229: Quantum Information and Computation*. CreateSpace Independent Publishing Platform, 1998.
- [77] A.N. Prior. *Time and Modality*. Clarendon Press, 1957.
- [78] M. A. Reniers and T. A. C. Willemse. Folk theorems on the correspondence between state-based and event-based systems. In *SOFSEM 2011: Theory and Practice of Computer Science*. Springer Berlin Heidelberg, 2011.
- [79] M. Rossi, M. Huber, D. Bruss, and C. Macchiavello. Quantum hypergraph states. *New Journal of Physics*, 15(11), 2013.
- [80] J. W. Sanders and P. Zuliani. Quantum Programming. *Mathematics of Program Construction*, 2000.
- [81] E. Schmidt. *Math. Annalen*. 63, 1906.
- [82] P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4), 2004.
- [83] S. Siddiqui, M. Jahirul Islam, and O. Shehab. Five Quantum Algorithms Using Quipper.
- [84] J.M. Smith, N.J. Ross, P. Selinger, and B. Valiron. Quipper: concrete resource estimation in quantum algorithms. Extended abstract for a talk given at the 12th International Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2014, Grenoble. Available from arxiv1412.0625, 2014.
- [85] K.M. Svore, A.V. Aho, A.W. Cross, I Chuang, and I.L. Markov. A layered software architecture for quantum computing design tools. *Computer*, 39(1), 2006.
- [86] A. van Tonder. A lambda calculus for quantum computation. 2003.
- [87] D. Wecker and K. M. Svore. Liquid: A software design architecture and domain-specific language for quantum computing.
- [88] W. K. Wootters. Entanglement of formation of an arbitrary state of two qubits. *Phys. Rev. Lett.*, 80:2245–2248, Mar 1998.

-
- [89] M. Ying. Floyd–hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.*, 33(6), January 2012.
- [90] M. Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.
- [91] M. Ying and Y. Feng. An algebraic language for distributed quantum computing. *IEEE Transactions on Computers*, 58(6), June 2009.
- [92] M. Ying and Y. Feng. A flowchart language for quantum programming. *IEEE Transactions on Software Engineering*, 37(4), 2011.
- [93] P. Zuliani. *Quantum programming*. PhD dissertation, University of Oxford, 2001.
- [94] P. Zuliani. Compiling quantum programs. *Acta Informatica*, 41, 2005.