Exploiting the Power of Quantum Computers: Programming, Access Control and Concurrency

Zhicheng Zhang

November 2025

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

in the

Centre for Quantum Software and Information
Faculty of Engineering and Information Technology
University of Technology Sydney

Supervisor:	Prof. Mingsheng Ying
Co-Supervisors:	Prof. Zhengfeng Ji
	Prof. Sanjiang Li

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Zhicheng Zhang, declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research was supported by an Australian Government Research Training Program (RTP) Scholarship doi.org/10.82133/C42F-K220.

Signed: Production Note: Signature removed prior to publication.

Date: 18/11/2025

Dedicated to my parents, teachers and friends.

Abstract

Exploiting the power of quantum computing relies on foundational software that ensures its convenient, efficient, and safe utilisation. The distinct nature of quantum mechanics introduces new challenges for quantum software design. This thesis contributes to identifying and addressing such challenges from the following three perspectives:

- **Programming**: The first part explores quantum recursive programming, an emerging paradigm that enables compact and elegant programming of complex quantum algorithms. We focus on efficiently implementing such programs, which involve an intricate interplay between two features: quantum control flow and recursive procedure calls. To handle this interplay, we propose the quantum register machine, a new architecture that provides simultaneous instruction-level support for both features. Based on this, we describe a comprehensive implementation process, including compilation, partial evaluation of quantum control flow, and execution on the quantum register machine. Significantly, our efficient implementation of quantum recursive programs also offers automatic parallelisation of quantum algorithms.

- **Access control**: To ensure the security of multi-programming quantum computers, the second part investigates access control in quantum operating systems. Access control is a cornerstone of computer security that prevents unauthorised access to resources. We identify a security threat arising from quantum entanglement as existing operating systems integrate quantum computing. In particular, we present an explicit scenario in which a security breach occurs when a classically secure access control system is straightforwardly adapted to the quantum setting. To protect against such threats, we propose several new models of quantum access control and rigorously analyse their security, flexibility, and efficiency.

- **Concurrency**: The third part examines the atomicity assumption in distributed quantum computing, a fundamental concept in concurrency control, which is crucial for scaling up quantum computational power through distributed systems. While atomic actions have well-established guarantees in classical computing, their rigorous basis in quantum computing remains largely unexplored. We identify key challenges in guaranteeing the atomicity assumption that arise from quantum entanglement and the quantum measurement problem. To address these challenges, we establish a formal model of non-atomic distributed quantum systems and use it to provide a rigorous guarantee for the atomicity of local actions in the quantum setting.

Acknowledgements

The completion of this thesis, and most of my work, would not have been possible without the support of many people.

First and foremost, I would like to thank my supervisor Mingsheng Ying for guiding and supporting me throughout my research journey. Mingsheng has always been a great teacher, consistently encouraging, and generous in sharing his amazing vision of quantum computing and the broad field of computer science. As a researcher, his infinite passion and good taste have always inspired me. I learned a lot from our discussions about various research topics, the history of science, and the philosophy of research. Without his continuous encouragement and extensive support, I would not have had the opportunity to explore many interesting problems.

I also wish to thank my two co-supervisors. I am grateful to Zhengfeng Ji for his encouragement, support, and many inspiring discussions. I especially enjoyed the reading group led by Zhengfeng, where a group of his students and I had a great time sharing our own work and discussing the recent results in the field. His many sharp insights have enlightened me. I am indebted to Sanjiang Li for his continued help in both my scientific and non-scientific life. Sanjiang has always been careful and patient in guiding me through my studies at UTS and generous in his support whenever I faced obstacles.

I sincerely thank Mario Szegedy and Lirong Xia for hosting me at Rutgers University. Mario shared many interesting insights with me in our joyful discussions, some of which profoundly reshaped my understanding of the field. I thank him for his guidance and for the ping-pong game we played. Lirong was generous in providing invaluable advice on my career development. I am especially grateful for his hospitality and for offering me the opportunity to deliver a mini-course at Rutgers. I also wish to thank Ryan O'Donnell for his guidance during my undergraduate visit to CMU. Ryan had a crucial impact on my research interests from the early days, and I learned many fascinating theoretical tools from him.

I am deeply grateful to Yufei Ding, Yuan Feng, Min-Hsiu Hsieh, Liang Jiang, Sumeet Khatri, Mária Kieferová, Robert König, Nana Liu, Kae Nemoto, Youming Qiao, Robert Rand, Runzhou Tao, Nathan Wiebe, Mark Wilde, and Xiaodi Wu for their invaluable guidance on my career development. I appreciate Daniel Burgarth, Yuan Feng, Troy Lee, Mária Kieferová, and Thomas Volz for their excellent quantum courses I took during my PhD. I extend my thanks to Penghui Yao for inviting me to give a talk at NJU on my first research project.

I have been fortunate to collaborate with many great researchers, to whom I want to express my gratitude. Most of my papers on quantum algorithms were written with Qisheng Wang, who always has a passion for solving new problems and often encouraged me. I especially enjoyed our wide-ranging discussions about research and life, some of which eventually turned into papers. I appreciate Kean Chen, who knows a great deal of cool math and is generous with his insights. He often came up with the key techniques in our collaborations. I am very grateful to Junyi Liu for mentoring me at the very beginning of my quantum research, when we spent a wonderful time reading a quantum computing textbook together. I owe my gratitude to Ji Guan and Wang Fang, who have guided and encouraged me on my research journey. It was my great pleasure to meet Nana Liu and Mark Wilde in Sydney, and I learned much about quantum information from them in our joyful collaboration. I sincerely thank Nengkun Yu and Zhan Yu, who are not only fantastic collaborators but also great friends.

For many other stimulating and inspiring discussions, I would like to thank the following people not yet mentioned: Samson Abramsky, Jinge Bao, Dominic Berry, Dolev Bluvstein, Chenfeng Cao, Boyang Chen, Yu-Fang Chen, Zihan Chen, Bin Cheng, Thiparat Chotibut, Simon Devitt, Yangjing Dong, Jens Eisert, Di Fang, Chris Ferrie, Honghao Fu, Minbo Gao, Alexei Gilchrist, András Gilyén, Jingzhe Guo, Naixu Guo, Alexander Hahn, Qishen Han, Zhengyi Han, Zhiyang He, Zhuangfei Hu, Hsin-Yuan Huang, Qifan Huang, Yipeng Huang, Zixin Huang, Mingrui Jing, Robin Kothari, Greg Kuperberg, Ondřej Lengál, Gushu Li, Riling Li, Xingjian Li, Zirui Li, Zhiding Liang, Yupan Liu, Alessandro Luongo, Urmila Mahadev, Angsar Manatuly, Yusuke Matsushita, Mauro Morales, Timothy Ralph, Patrick Rebentrost, Barry Sanders, Yuval Sanders, Ritvik Sharma, Rolando Somma, Mark Squillante, Yuan Su, Mingyu Sun, Kristan Temme, Nathaniel Tornow, Francisca Vasconcelos, Kangning Wang, Howard Wiseman, Zhean Xu, Shenggang Ying, Chuanqi Zhang, Kezhen Zhang, Shengyu Zhang, Benchi Zhao, and Li Zhou.

I am also grateful to my thesis examiners Canh Minh Do and Chen Qian for reading this thesis and providing invaluable feedback.

My four years at the Center for Quantum Software and Information at UTS have been wonderful, and I am grateful to all the members and friends here for creating such a great environment. Special thanks to Zhili Chen, Bin Cheng, Yunqi Huang, Jingtong Ge, Gang Tang, Peng Yan, and Chuanqi Zhang for many joyful hikes and fun events in Sydney. I thank the Sydney Quantum Academy for its PhD scholarship and for the events it organised, where I enjoyed many engaging conversations with Arinta Auza, Adrien Di Lonardo, Giorge Gemisis, Karl Lin, Angsar Manatuly, Mauro Morales, Tuyen Nguyen, Treerat Srivipat, and Gozde Ustun. I appreciate my friend Yangfan Wu, who has always been available to help.

Finally, I want to thank my parents for their unconditional love and support throughout my life. Many thanks to all the teachers and friends who have guided me.

Contents

Chapter 1

Introduction

Significant advancements in quantum hardware over the last three decades have made quantum computing increasingly tangible. To fully exploit the power of quantum computers, foundational software is indispensable. Ideally, future quantum computers — much like their classical counterparts — should be convenient, efficient, and safe to use. However, the unique nature of quantum mechanics introduces challenges that demand novel quantum software design beyond classical paradigms. This thesis is therefore motivated by the pursuit of harmonious solutions to foundational questions in quantum software design, covering the aspects of programming, access control, and concurrency.

1.1 From Sequential to Distributed Computing

Since the design of quantum software is greatly inspired by its well-established classical counterpart, this section provides the necessary background. We briefly review the evolution of classical computers through three stages: sequential, multiprogramming, and distributed computers; and we pay particular attention to the foundational software relevant to this thesis.

Sequential Computers

The modern computing era began with machines operating sequentially on a single central processing unit (CPU). A foundational challenge of software design at this time was bridging the gap between humans and computers. To this end, early high-level programming languages such as Fortran, COBOL, Lisp, and ALGOL were developed. These languages introduced many concepts profoundly impacting today's programming, chief among them structured programming [1, 2]. This paradigm promotes program clarity and modularity and emphasises a disciplined use of control flow constructs (such as if-statements, loop statements, and procedure calls), enabling programmers to design complicated programs in a modular way.

For a computer to execute these high-level programs, they must be translated into low-level instructions. This translation is performed by a compiler [3], which typically involves multiple structured intermediate layers. A successful compiler should both preserve the source program's semantics and produce a compiled program that executes efficiently.

Multiprogramming Computers

Multiprogramming [4] was introduced to better utilise fast CPUs that would otherwise idle while waiting for slower input/output (I/O) devices. Multiple programs can share the CPU time: when one program is being executed, others can wait for the I/O devices. This fundamental idea of resource sharing has become a cornerstone of modern operating systems.

Two challenges were introduced by multiprogramming. The first is concurrency: operations from different programs can have a non-deterministic order [5]. System software needs to be carefully designed to guarantee correct computation regardless of this non-determinism. The second is access control, the central element of computer security: resources should be shared safely. Foundational system software must guarantee that a user (or more generally, a subject) can only access a resource (an object) by performing authorised actions (rights). While modern access control schemes are diverse, their foundation lies in the seminal access matrix model [6], which explicitly manages each right granted to a subject to access an object.

Distributed Computers

Distributed computing represents the next stage, where multiple computers are connected to form a distributed system. While multiprogramming involves sharing a single CPU, a distributed system allows all individual computers to run concurrently. This poses a significant challenge to foundational system software: concurrency must be carefully controlled to ensure correct and efficient computation. For example, to prevent corruption, a file should be guaranteed to be written by at most one process at any time. This generalises to the celebrated mutual exclusion problem [7].

A fundamental notion in solving the mutual exclusion problem is atomic actions [7, 8]. Two atomic actions a and b are guaranteed to be sequential: either a precedes b, or b precedes a. Assuming all actions in a distributed system are atomic can significantly simplify the system model and analysis. However, actual actions from individual computers can be non-atomic and have no temporal order at all. Therefore, a number of hardware and software guarantees [9–11] have been established for atomic actions.

1.2 From Classical to Quantum Computing

The last three decades have witnessed remarkable advancements in quantum computing. Numerous quantum algorithms promising computational advantages have been proposed, including notable examples like Shor's factoring algorithm [12], Grover's search algorithm [13], and quantum simulation algorithm [14]. In parallel, quantum hardware has progressed remarkably, with quantum processors now reaching hundreds to thousands of qubits [15, 16] and early experiments demonstrating the potential for quantum error correction [17, 18].

To connect theoretical quantum advantages to practical applications, quantum software serves as the fundamental bridge. In this section, we review prior research in quantum software, following an analogous evolutionary path from sequential to distributed computers. Along this path, we identify the key questions that are addressed by this thesis.

Sequential Quantum Computers

The most near-term model for quantum computers is sequential. To translate abstract quantum algorithms into practical implementations, quantum programming languages are indispensable. Numerous languages and associated compilers have been developed, ranging from industry-backed platforms like IBM's Qiskit [19], Google's Cirq [20], and Microsoft's Q# [21], to research-oriented languages with various features (e.g., [22–29]). A substantial body of prior work in this area is devoted to quantum program verification. Rich theories have been established using quantum Hoare-like logics [25, 30–38] and other formal methods [39–48], leading to various automated verification tools [49–60]. Equivalence checking, which is crucial for verifying quantum compilers and circuit optimisers [61–63], has also been extensively studied [64–75].

Recently, high-level modular design of quantum programs has gained significant attention, with a focus on unique quantum programming features like quantum control flow [22, 25, 26, 28, 76–78]. An emerging paradigm is quantum recursive programming [37], which enables one to write complex quantum algorithms as elegant and compact programs. However, a fundamental challenge is how to implement these programs efficiently. Addressing this, Part I of this thesis is dedicated to developing a systematic framework for the efficient implementation of quantum recursive programs.

Multiprogramming Quantum Computers

At a lower level, a quantum computer behaves as a natural multiprogramming system, requiring complex coordination between multiple quantum and classical resources. This

coordination is managed by quantum operating systems, whose importance is underscored by the recent active development of quantum-centric supercomputing [79–83]. Research in this area includes holistic design of quantum operating systems [84–88], as well as work on specific tasks like quantum circuit cutting (e.g., [89–99]), job scheduling (e.g., [100–103]), multiprogramming (e.g., [101, 104–110]), and memory management (e.g., [111–115]).

On the other hand, the unique properties of quantum operating systems raise new security concerns [116]. A significant question is whether the security guarantees of access control still hold when existing classical systems integrate quantum computing services, like in the case of quantum-centric supercomputing [79–83]. Answering this, Part II of this thesis identifies threats to access control stemming from quantum entanglement and develops new models to protect against such threats.

Distributed Quantum Computers

Connecting distributed quantum processors is a promising approach to scale beyond the limits of a single device, with early experimental demonstrations already achieved [117, 118]. The theory of this field has been extensively studied, including distributed quantum algorithms with provable quantum advantages [119–126] and various formal models like quantum process algebra [127–129], quantum versions of the LOCAL [130–135] and CONGEST models [136–141], and distributed quantum programming [36, 142–144]. Closely related is the area of quantum networks, which focuses on communication between distributed quantum processors, covering topics like quantum repeaters (e.g., [145–150]), network simulation (e.g., [151–158]), and quantum routing (e.g., [159–171]).

Like in the classical case, nearly all distributed quantum computing models either explicitly or implicitly assume that actions are atomic. However, this foundational assumption has never been seriously studied in the quantum context. Therefore, Part III of this thesis confronts this issue by examining the rigorous guarantee for this atomicity assumption and developing a non-atomic model for distributed quantum computing.

1.3 Challenges from Unique Quantum Properties

This section briefly reviews several unique properties of quantum mechanics (see Chapter 2 for a brief introduction to quantum computing), highlighting the foundational software challenges they pose that will be addressed in this thesis.

The first key property is *quantum superposition*. In the classical world, a basic unit of information is a bit with state space $\{0, 1\}$. The quantum counterpart of a bit is called a qubit, whose state space is a Hilbert space $\mathbb{C}^2$. The state of a qubit can be $\alpha |0\rangle + \beta |1\rangle$,

a superposition of the basis states $|0\rangle$ and $|1\rangle$. More generally, the state of n qubits composed together can be $\sum_{x\in\{0,1\}^n} \alpha_x |x\rangle$, a superposition of $|0\rangle, |1\rangle, \ldots, |2^n-1\rangle$. Quantum superposition enables powerful quantum programming features like quantum control flow. In Part I, we will see how the intricate interplay between quantum control flow and recursive procedure calls poses a fundamental challenge to the efficient implementation of quantum recursive programs.

The second property is *quantum entanglement*, a consequence of the first property. A quantum state $|\psi\rangle$ is said to be entangled iff it cannot be represented as a tensor product state $|\psi_1\rangle \otimes |\psi_2\rangle$. For example, an EPR state $|+\rangle_{AB} = \frac{1}{\sqrt{2}}(|0\rangle_A |0\rangle_B + |1\rangle_A |1\rangle_B)$ is entangled, where the subscripts A and B label the two qubits. Entanglement is a unique form of quantum correlation that goes beyond classical mechanics. In Part II, we will identify entanglement as a source of threats to access control when existing computer systems integrate quantum computing. Further, in Part III, entanglement will pose an obstacle to guaranteeing the atomicity assumption in distributed quantum computing, as prior classical results implicitly rely on the state being a product.

The third property is that quantum operations are fundamentally different from classical ones. The first type of quantum operations is unitary gates, which are deterministic and reversible, originating from the evolution of closed quantum systems. The second type is quantum measurements, which are probabilistic and irreversible, serving as a probe to extract classical information from quantum states. The nature of quantum measurement, in particular, how the state evolves during a measurement, has not been fully understood. This is known as the notorious measurement problem, and in Part III of this thesis, it will impose a challenge in modeling the real-time behaviours of non-atomic distributed quantum systems.

1.4 Overview of the Thesis

This thesis is divided into three parts, focusing on the foundational software for quantum computers from the following three perspectives: programming, access control, and concurrency. We have seen why these topics are foundational in the previous sections. Figure 1.1 visualises the structure of this thesis and the relationship between chapters.

1.4.1 Programming

Part I of this thesis is dedicated to the efficient implementation of quantum recursive programs. Classical recursive programming is a powerful paradigm for modular design, enabling programmers to elegantly describe complex algorithms like Hoare's quick sort [172], recursively defined data structures [173], and divide-and-conquer algorithms. In the same vein, quantum recursive programming [37] is an emerging paradigm that

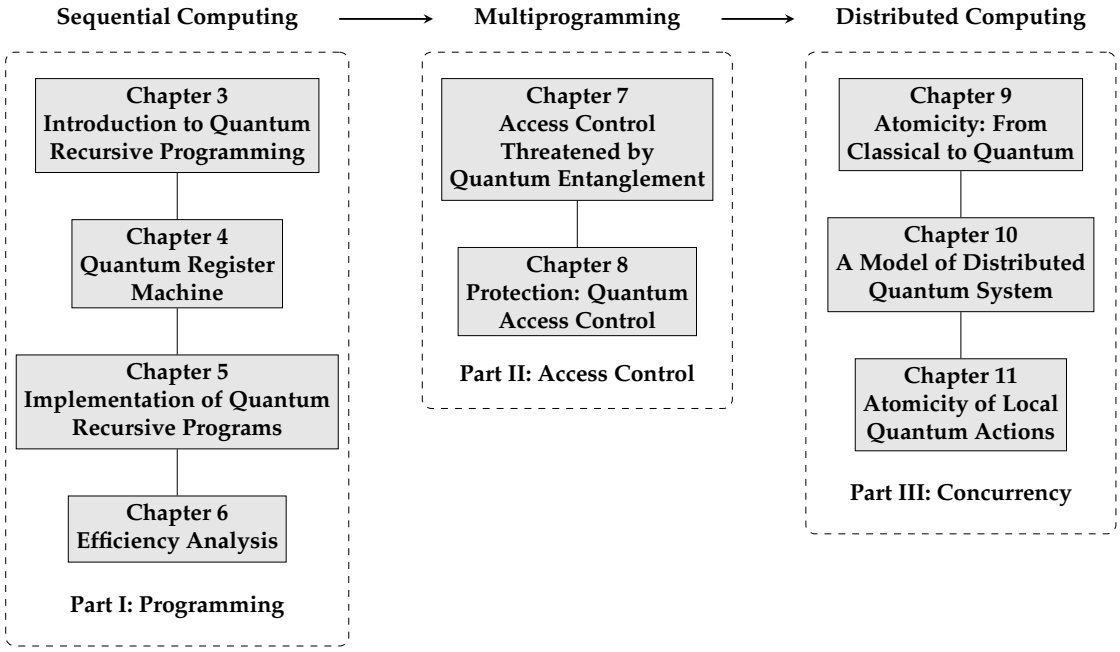


FIGURE 1.1: Structure of the thesis.

combines the classical recursive procedure calls with a uniquely quantum feature: quantum control flow [22, 25, 26, 28, 76–78], which allows executing multiple quantum programs in superposition. The intricate interplay between quantum control flow and recursive procedure calls grants the paradigm a distinct expressive power, demonstrated through examples like quantum Fourier transform, preparation of recursively defined quantum states, and quantum random access memory [37]. However, it also poses a significant challenge for the efficient implementation of quantum recursive programs. To resolve this challenge, a systematic framework is established in Part I.

Chapter 3 provides a detailed introduction to quantum recursive programming. The chapter motivates this paradigm from the history of classical recursive programming and quantum control flow. It then introduces **RQC**$^{++}$, a programming language for describing quantum recursive programs [37], detailing its formal syntax and semantics, as well as its core features. We conclude with a series of illustrative examples, demonstrating how complicated quantum algorithms can be written as elegant and compact quantum recursive programs.

Chapter 4 presents the quantum register machine, a novel quantum architecture offering instruction-level support for quantum recursion. The chapter provides an overview of the systematic framework to resolve the implementation challenges of quantum recursion. It then details the storage components of the quantum register machine, including quantum registers and a quantum random access memory (QRAM) for storing both compiled programs and data. We explain how the quantum register machine operates like

a classical CPU by repeatedly fetching, decoding and executing instructions from the QRAM. The associated instruction set **QINS** is carefully designed to support quantum control flow and recursive procedure calls. This architecture provides the foundation for the comprehensive implementation process in Chapter 5 and the efficiency analysis in Chapter 6.

Chapter 5 details our comprehensive process of implementing quantum recursive programs written in $\mathbf{RQC}^{++}$, which consists of three steps. (a) Compilation: the source program written in the high-level language $\mathbf{RQC}^{++}$ is translated into low-level instructions in **QINS**. (b) Partial evaluation: Using only the classical inputs (without knowing the quantum inputs), the quantum control flow information in the compiled program is partially evaluated and stored in a data structure. (c) Execution: the compiled program and the partial evaluation results are loaded into the QRAM, where the quantum register machine executes by repeatedly applying a fixed unitary cycle by cycle, like a classical CPU.

Chapter 6 provides a rigorous analysis of our implementation process in Chapter 5. The chapter analyses the theoretical complexity of the partial evaluation and execution steps, first in terms of elementary operations and then refined into parallel time complexity measured by the standard circuit depth. Such efficient implementation of quantum recursive programs offers automatic parallelisation as a bonus. For implementing certain quantum algorithmic subroutine, like the quantum multiplexor [174], it even yields exponential parallel speed-up (over the straightforward implementation).

1.4.2 Access Control

Part II of this thesis investigates access control for quantum operating systems. In classical operating systems, access control is a framework for specifying how rights are granted to users (subjects) to access resources (objects) [175]. In quantum operating systems with quantum resources, unique quantum properties have raised new security concerns [116], but no explicit threats were previously known. This leaves a crucial question, especially with the rise of quantum-centric supercomputing [79–83]: do the security guarantees of access control still hold when existing operating systems integrate quantum computing? To answer this question, Part II presents an explicit scenario revealing threats to access control from quantum entanglement and proposes new quantum access control models to protect against such threats.

Chapter 7 presents an explicit scenario of a security breach when a classically secure access control system is straightforwardly adapted to the quantum setting. The chapter starts with the formal definition of an access control system. It then presents an explicit classical system and proves its security, showing that a user's confidential information cannot be significantly leaked to others. However, when some registers in this system

become quantum (e.g., when quantum computing is integrated) this security fails. The threat we reveal essentially stems from quantum entanglement. Our construction and proofs precisely exploit the fact that Mermin inequality [176] holds in the classical world but can be violated by quantum entanglement.

Chapter 8 presents a series of quantum access control models designed to protect against the threats revealed in Chapter 7. Specifically, we introduce subsystem control, group control and entanglement control models. As quantum entanglement is the source of threats, these models work by enabling explicit control over multi-object quantum operations or the entanglement resource itself. We then rigorously evaluate and compare these models against three metrics: (a) security against threats from quantum entanglement; (b) flexibility regarding the granularity of specifying the access control; and (c) efficiency regarding the space and time complexity for implementing the model.

1.4.3 Concurrency

Part III of this thesis examines the atomicity assumption in distributed quantum computing. Assuming actions are atomic, as is done in nearly all models of distributed computing, significantly simplifies the system modeling and analysis. In the classical setting, this atomicity assumption has well-established hardware and software guarantees [4, 9, 177–179]. However, while still widely adopted in distributed quantum computing, the assumption has not been seriously studied. This raises a fundamental question: can the atomicity assumption be rigorously guaranteed in distributed quantum systems? In response, Part III identifies the key challenges for guaranteeing atomicity in the quantum setting and addresses them by establishing a model of non-atomic distributed quantum systems.

Chapter 9 presents the background on atomicity and concurrency. It then uses a series of motivating examples to demonstrate the non-triviality of establishing guarantees for the atomicity assumption in distributed quantum computing, even when considering only the atomicity of local actions. From these examples, we identify two key challenges from quantum entanglement and the measurement problem. Specifically, entanglement forbids decomposing the real-time system state into a product of object states, and the measurement problem obstructs modeling the real-time evolution of a distributed quantum system.

Chapter 10 presents a formal model of non-atomic distributed quantum systems. The chapter first defines the notions of actions, quantum processes, and distributed quantum systems. Then it establishes two semantics: (a) real-time semantics characterising the real-time evolution of a distributed quantum system; and (b) observable semantics, induced from the real-time one, characterising the probabilities of all classical observable events in the system. Based on the observable semantics, we define the equivalence

between systems. These definitions are carefully designed to circumvent the challenges identified in Chapter 9 and are used later in Chapter 11 to rigorously guarantee the atomicity of local actions.

Chapter 11 establishes a rigorous basis for the atomicity of local actions in a distributed quantum system. Specifically, based on the model developed in Chapter 10, we prove that any non-atomic distributed quantum system is equivalent to another system where local actions are atomic.

Summary

Through my PhD, I have been fortunate to collaborate with many great researchers. This thesis is based on the following co-authored papers, for which I was a major contributor:

- **Zhicheng Zhang** and Mingsheng Ying, "Quantum Register Machine: Efficient Implementation of Quantum Recursive Programs". In: *Proceedings of the ACM on Programming Languages* 9.PLDI (2025), pp. 822–847. [180]

  (Part I)

- **Zhicheng Zhang** and Mingsheng Ying, "Access Control Threatened by Quantum Entanglement". Work in submission, available on arXiv:2507.02622. [181]

  (Part II)

- **Zhicheng Zhang** and Mingsheng Ying, "Atomicity in Distributed Quantum Computing". Work in submission, available on arXiv:2404.18592. [182]

  (Part III)

My other publications and preprints that are not specifically explored in this thesis are listed below:

- Qisheng Wang, **Zhicheng Zhang**, Kean Chen, Ji Guan, Wang Fang, Junyi Liu, and Mingsheng Ying, "Quantum Algorithm for Fidelity Estimation". In: *IEEE Transactions on Information Theory* 69.1 (2023), pp. 273—282. [183]

- **Zhicheng Zhang**, Qisheng Wang, and Mingsheng Ying, "Parallel Quantum Algorithm for Hamiltonian Simulation". In: *Quantum* 8 (2024), p. 1228. [184]

- Qisheng Wang and **Zhicheng Zhang**, "Quantum Lower Bounds by Sample-to-Query Lifting". In: *SIAM Journal on Computing* 54.5 (2025), pp. 1294–1334. [185]

- Qisheng Wang and **Zhicheng Zhang**, "Fast Quantum Algorithms for Trace Distance Estimation". In: *IEEE Transactions on Information Theory* 70.4 (2024), pp. 2720–2733. [186]

- Qisheng Wang, Ji Guan, Junyi Liu, **Zhicheng Zhang**, and Mingsheng Ying, "New Quantum Algorithms for Computing Quantum Entropies and Distances". In: *IEEE Transactions on Information Theory* 70.8 (2024), pp. 5653–5680. [187]
- Qisheng Wang and **Zhicheng Zhang**, "Time-Efficient Quantum Entropy Estimator via Samplizer". In: *Proceedings of the 32nd Annual European Symposium on Algorithms* 308.ESA (2024), pp. 101:1–101:15. [188]
- Nana Liu, Qisheng Wang, Mark M. Wilde, and **Zhicheng Zhang**, "Quantum Algorithms for Matrix Geometric Means". In: *npj Quantum Information* 11.101 (2025). [189]
- Qisheng Wang and **Zhicheng Zhang**, "Tight Quantum Depth Lower Bound for Solving Systems of Linear Equations". In: *Physical Review A* 110 (2024), p. 012422. [190]
- Qisheng Wang and **Zhicheng Zhang**, "Sample-Optimal Quantum Estimators for Pure-State Trace Distance and Fidelity via Samplizer". Work in submission, available on arXiv:2410.21201. [191]
- Mingsheng Ying and **Zhicheng Zhang**, "Verification of Recursively Defined Quantum Circuits". Work in submission, available on arXiv:2404.05934. [37]
- Kean Chen, Nengkun Yu, and **Zhicheng Zhang** "Tight Bound for Quantum Unitary Time-Reversal". Work in submission, available on arXiv:2507.05736. [192]
- Kean Chen, Qisheng Wang, Zhan Yu, and **Zhicheng Zhang**, "Simultaneous Estimation of Nonlinear Functionals of a Quantum State". Work in submission, available on arXiv:2505.16715. [193]
- Kean Chen, Qisheng Wang, and **Zhicheng Zhang**, "Local Test for Unitarily Invariant Properties of Bipartite Quantum States". Work in submission, available on arXiv:2404.04599. [194]

Chapter 2

Preliminaries

This chapter provides a brief introduction to the basic concepts and notations in quantum computing. The focus is on building intuition rather than providing a fully formal treatment; for a more thorough introduction, the reader is referred to the textbook by Nielsen and Chuang [195]. A basic familiarity with linear algebra is assumed. The formalism of quantum mechanics adopted here is the standard axiomatic framework pioneered by von Neumann [196] and Dirac [197].

2.1 State Space of a Quantum System

Single Systems

The state space of a closed quantum system is described by a *Hilbert space* $\mathcal{H}$. For the purposes of this introduction, let us restrict our focus to finite-dimensional systems where $\mathcal{H} = \mathbb{C}^N$, an N-dimensional complex vector space. These are sufficient for most of this thesis, with the exception of Chapter 10, which involves continuous-dimensional Hilbert spaces.

A vector in $\mathcal{H}$ is denoted by $|\psi\rangle$. Here, $|\cdot\rangle$ is the Dirac notation for a column vector, pronounced as "ket"; and ψ is how we name this vector. Correspondingly, the Dirac notation for a row vector is $\langle\cdot|$, pronounced as "bra", and $\langle\psi|$ represents the conjugate transpose of $|\psi\rangle$. For example, if

$$|\psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{N-1} \end{bmatrix}, \quad \text{then} \quad \langle\psi| = \begin{bmatrix} \alpha_0^* & \alpha_1^* & \dots & \alpha_{N-1}^* \end{bmatrix},$$

where c^* denotes the complex conjugate of $c \in \mathbb{C}$.

The Hilbert space $\mathcal{H} = \mathbb{C}^N$ is spanned by the *computational basis*, a set composed of the following vectors:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \ldots, \quad |N-1\rangle = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

Any vector $|\psi\rangle \in \mathcal{H}$ can be expressed as a linear combination of these basis vectors:

$$|\psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{N-1} \end{bmatrix} = \sum_{j=0}^{N-1} \alpha_j |j\rangle .$$

The Hilbert space $\mathcal{H} = \mathbb{C}^N$ is equipped with the Hermitian inner product. For any two vectors

$$|\psi\rangle = \sum_{j=0}^{N-1} \alpha_j |j\rangle \quad \text{and} \quad |\phi\rangle = \sum_{k=0}^{N-1} \beta_k |k\rangle ,$$

their inner product $\langle\psi|\phi\rangle$ is defined as:

$$\langle\psi|\phi\rangle = \sum_{j=0}^{N-1} \alpha_j^* \beta_j.$$

Note that $\langle\psi|\phi\rangle$ is equivalent to the matrix multiplication of the row vector $\langle\psi|$ and the column vector $|\phi\rangle$:

$$\begin{bmatrix} \alpha_0^* & \alpha_1^* & \ldots & \alpha_{N-1}^* \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{N-1} \end{bmatrix}.$$

The inner product induces the norm of a vector $|\psi\rangle = \sum_{j=0}^{N-1} \alpha_j |j\rangle$:

$$\| |\psi\rangle \| = \sqrt{\langle\psi|\psi\rangle} = \sqrt{\sum_{j=0}^{N-1} |\alpha_j|^2},$$

where $|c| = \sqrt{c^* c}$ denotes the magnitude of $c \in \mathbb{C}$.

Consequently, the computational basis $\{|0\rangle, |1\rangle, \ldots, |N-1\rangle\}$ is an orthonormal basis for $\mathcal{H}$, that is

$$\langle j|k\rangle = \begin{cases} 0, & j \neq k, \\ 1, & j = k. \end{cases}$$

Composite Systems

The composite of two or more quantum systems is also a quantum system with a state space. If systems A and B have Hilbert spaces $\mathcal{H}_A$ and $\mathcal{H}_B$, respectively, their composite system AB has Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$, a *tensor product* of $\mathcal{H}_A$ and $\mathcal{H}_B$.

For illustration, let $\mathcal{H}_A = \mathbb{C}^N$ and $\mathcal{H}_B = \mathbb{C}^M$, with computational bases $\{|j\rangle\}_{j=0}^{N-1}$ and $\{|k\rangle\}_{k=0}^{M-1}$. For any two vectors

$$|\psi\rangle = \sum_{j=0}^{N-1} |j\rangle \in \mathcal{H}_A, \quad \text{and} \quad |\phi\rangle = \sum_{k=0}^{M-1} |k\rangle \in \mathcal{H}_B,$$

their tensor product $|\psi\rangle \otimes |\phi\rangle$ is a vector in $\mathcal{H}_A \otimes \mathcal{H}_B$, defined as:

$$|\psi\rangle \otimes |\phi\rangle = \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} \alpha_j \beta_k \, |j\rangle \otimes |k\rangle ,$$

where $|j\rangle \otimes |k\rangle$ are vectors forming the computational basis for $\mathcal{H}_A \otimes \mathcal{H}_B$. Note that $|\psi\rangle \otimes |\phi\rangle$ is equivalent to the Kronecker product of the vectors' matrix representations:

$$\begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_{N-1} \end{bmatrix} \otimes \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_{N-1} \end{bmatrix} = \begin{bmatrix} \alpha_0\beta_0 \\ \cdots \\ \alpha_0\beta_{N-1} \\ \vdots \\ \alpha_{N-1}\beta_0 \\ \vdots \\ \alpha_{N-1}\beta_{N-1} \end{bmatrix}.$$

For convenience, $|\psi\rangle \otimes |\phi\rangle$ is often abbreviated as $|\psi\rangle\,|\phi\rangle$ or $|\psi, \phi\rangle$.

The Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$ is equipped with an inner product satisfying:

$$\langle j, k|l, r\rangle = \langle j|l\rangle \cdot \langle k|r\rangle ,$$

for any $j, l = 0$ to $N-1$ and $k, r = 0$ to $M-1$. It follows that $\{|j, k\rangle\}_{j,k}$ is an orthonormal basis for $\mathcal{H}_A \otimes \mathcal{H}_B$.

2.2 Quantum States

A *quantum state* of a system is described by a unit vector in the system's Hilbert space $\mathcal{H}$. More precisely:

- A vector $|\psi\rangle \in \mathcal{H}$ with $\||\psi\rangle\| = 1$ represents a quantum state; and
- Two vectors $|\psi\rangle$ and $|\phi\rangle$ that differ by a global phase, i.e., $|\psi\rangle = e^{i\theta}|\phi\rangle$ for some $\theta \in [0, 2\pi)$, represent the identical quantum state.

A *qubit* is a quantum system with a Hilbert space $\mathcal{H} = \mathbb{C}^2$. The state of a qubit can be written as $\alpha|0\rangle + \beta|1\rangle$ with $|\alpha|^2 + |\beta|^2 = 1$, for example:

$$|0\rangle, |1\rangle, \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad \text{and} \quad \frac{i}{\sqrt{5}}|0\rangle + \frac{-2i}{\sqrt{5}}|1\rangle.$$

The computational basis states $|0\rangle$ and $|1\rangle$ can be thought of as quantum counterparts of the classical bit states 0 and 1. Physically, $\alpha|0\rangle + \beta|1\rangle$ is a *superposition* of states $|0\rangle$ and $|1\rangle$. This ability of a quantum system to exist in a superposition state represents a fundamental difference between quantum and classical mechanics.

Now consider a composite system of two qubits A and B, with the Hilbert space $\mathcal{H}_{AB} = \mathbb{C}^2 \otimes \mathbb{C}^2$. A state in $\mathcal{H}_{AB}$ can be written as

$$|\psi\rangle_{AB} = \alpha_0|0\rangle_A|0\rangle_B + \alpha_1|0\rangle_A|1\rangle_B + \alpha_2|1\rangle_A|0\rangle_B + \alpha_3|1\rangle_A|1\rangle_B,$$

where $\sum_{j=0}^{3}|\alpha_j|^2 = 1$, and the subscripts A and B explicitly indicate the qubits holding the states. There exists a state $|\psi\rangle_{AB}$ that cannot be written in the form of a tensor product $|\phi\rangle_A \otimes |\eta\rangle_B$. An example of such $|\psi\rangle_{AB}$ is one of the Bell states (a.k.a., EPR pairs):

$$\frac{1}{\sqrt{2}}(|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B).$$

Another example is:

$$\frac{i}{\sqrt{3}}|0\rangle_A|1\rangle_B + \frac{\sqrt{2}}{\sqrt{3}}|1\rangle_A|0\rangle_B.$$

More generally, a quantum state $|\psi\rangle$ is said to be a *product state* if $|\psi\rangle = |\phi\rangle \otimes |\eta\rangle$ for some $|\phi\rangle$ and $|\eta\rangle$. Otherwise, it is said to be *entangled*. The phenomenon of *quantum entanglement* is a fundamental feature of quantum mechanics with no classical counterpart.

2.3 Quantum Operations

In quantum computing, we can perform two types of quantum operations: unitary transformations and quantum measurements.

Unitary Transformations

The evolution of a closed quantum physical system is described by a *unitary transformation* U, which originates from the Schrödinger equation. More precisely, for a system with Hilbert space $\mathcal{H}$, a unitary transformation is a linear operator $U : \mathcal{H} \to \mathcal{H}$ that satisfies

$$UU^\dagger = U^\dagger U = \mathbb{1},$$

where $U^\dagger$ denotes the *conjugate transpose* of U and $\mathbb{1}$ denotes the identity operator. For convenience, we often abbreviate a unitary transformation as a unitary.

If the system's current state is $|\psi\rangle$, applying the unitary transformation U transforms the state to $U|\psi\rangle$. The unitary transformation is essentially a reversible process, as applying $U^\dagger$ to $U|\psi\rangle$ can reverse the effect of U.

Note that for a composite system AB with Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$, performing a unitary U on A and a unitary V on B corresponds to performing the larger unitary $U \otimes V$ on A, B. When $V = \mathbb{1}$, performing U only on the subsystem A is equivalent to performing $U \otimes \mathbb{1}$ on the whole system.

In quantum computing, complex unitary transformations are implemented by *quantum circuits* constructed from elementary *quantum gates*. Each quantum gate is typically a small unitary acting on one or two qubits. At a low level, the design of quantum algorithms is therefore the design of quantum circuits.

For illustration, let us consider some examples of quantum gates. Common single-qubit gates include

- Pauli gates:

$$\mathbb{1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

- Hadamard gate:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

- Phase gates:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \quad \text{and} \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

Next we introduce some common two-qubit gates, using the Dirac notation.

- Controlled-NOT (CNOT) gate: This gate flips the second (target) qubit, conditioned on the first (control) qubit being in state $|1\rangle$.

$$CNOT = |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes X.$$

- Controlled-Z (CZ) gate: This gate applies a Z gate to the second qubit, conditioned on the first being in state $|1\rangle$.

$$CZ = |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes Z.$$

- SWAP gate: This gate exchanges the states of the two qubits.

$$SWAP = \sum_{x,y\in\{0,1\}} |x,y\rangle\langle y,x| .$$

Quantum Measurements

To extract classical information from a quantum system, we need to perform quantum measurements. A measurement is described by a set of Kraus operators $\{M_m\}_m$. More precisely, for a system with Hilbert space $\mathcal{H}$, a measurement $\{M_m\}_m$ is a set of linear operators $M_m : \mathcal{H} \to \mathcal{H}$ satisfying

$$\sum_m M_m^\dagger M_m = \mathbb{1}. \tag{2.1}$$

Suppose the system's state before the measurement is $|\psi\rangle$. After applying the measurement $\{M_m\}_m$, with probability $\|M_m|\psi\rangle\|^2$, the current state becomes

$$\frac{M_m|\psi\rangle}{\|M_m|\psi\rangle\|} \tag{2.2}$$

and an associated classical outcome m is produced in the measurement device. A measurement is said to be *complete* if the range of m is equal to the dimension $\dim\mathcal{H}$ of the system being measured.

The fact that the post-measurement states differ from the initial quantum state is usually referred to as the *collapse* of quantum states. Quantum measurements are generally irreversible, transforming quantum information into classical information. They also naturally introduce probabilities into quantum computation. The outcomes from quantum measurements can serve as the classical output of a quantum algorithm.

The simplest is a computational basis measurement, where $M_m = |m\rangle\langle m|$. Similar to the unitary case, a measurement $\{M_m\}_m$ can be applied only to a subsystem A of a composite system AB, equivalent to applying a larger measurement with Kraus operators $\{M_m \otimes \mathbb{1}_B\}_m$. For example, consider a two-qubit system in the entangled state:

$$\frac{1}{\sqrt{2}}(|0\rangle|0\rangle + i|1\rangle|1\rangle). \tag{2.3}$$

If we measure the first qubit in the computational basis $\{|0\rangle\langle 0|, |1\rangle\langle 1|\}$, we can obtain:

- outcome 0 and post-measurement state $|0\rangle\,|0\rangle$, with probability $\frac{1}{2}$; and
- outcome 1 and post-measurement state $|1\rangle\,|1\rangle$, with probability $\frac{1}{2}$.

Note that the measurement *disentangles* the state.

Now consider performing another measurement $\{|+\rangle\langle+|, |-\rangle\langle-|\}$ on the first qubit of Equation (2.3), where

$$|+\rangle = H\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \text{ and } |-\rangle = H\,|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

After applying this measurement, we can obtain:

- outcome $+$ and post-measurement state $\frac{1}{\sqrt{2}}\,|+\rangle\,(|0\rangle + i\,|1\rangle)$, with probability $\frac{1}{2}$; and
- outcome $-$ and state $\frac{1}{\sqrt{2}}\,|-\rangle\,(|0\rangle - i\,|1\rangle)$, with probability $\frac{1}{2}$.

In quantum computing, the quantum hardware usually provides the computational basis measurements for each qubit. This is sufficient, as any general measurement can be reduced to a computational basis measurement by adding ancilla qubits and an additional unitary. In general, a quantum circuit consists of quantum gates and measurements.

It is crucial to note that the above formalism only describes the state *before and after* a measurement. The continuous evolution of the state *during* the measurement process is not described, a gap in our understanding known as *the measurement problem*. In a broader sense, we do not know where the boundary is between the classical and quantum worlds. From an implementation perspective, a measurement involves a complex interaction between the system being measured and the macroscopic measurement device, which is itself a quantum system. The dynamics of such interaction is often too complicated to be precisely modeled.

2.4 Density Operator Formalism

While the state-vector formalism in the previous sections is sufficient for much of quantum computing, a more general and often useful formalism is based on density operators, pioneered by von Neumann [198] and Landau [199].

Mixed States

The state vectors discussed in Section 2.2 are known as *pure states*. When classical probability is introduced, such as through quantum measurements, we need the more general notion of *mixed states*.

Consider a system with Hilbert space $\mathcal{H}$. A mixed state of this system is described by a density operator $\rho : \mathcal{H} \to \mathcal{H}$, which is *positive semi-definite* with trace in $[0, 1]$. Formally:

- Positive semi-definite: $\langle\psi| \rho |\psi\rangle \geq 0$ for any $|\psi\rangle \in \mathcal{H}$; and
- Trace condition: $\mathrm{tr}(\rho) \in [0, 1]$.

The mixed state ρ is normalised if $\mathrm{tr}(\rho) = 1$.

Any pure state $|\psi\rangle$ has a corresponding density operator $\rho = |\psi\rangle\langle\psi|$. A general mixed state ρ can also be viewed as an ensemble of pure states via its spectral decomposition:

$$\rho = \sum_j p_j |\psi_j\rangle\langle\psi_j| , \tag{2.4}$$

where the system is in the pure state $|\psi_j\rangle$ with probability $p_j > 0$. Note that when $\mathrm{tr}(\rho) = 1$, the probabilities sum to one: $\sum_j p_j = 1$.

As an illustration, consider the following example of a single-qubit density operator:

$$\rho = \frac{1}{4}|0\rangle\langle 0| + \frac{1}{2}|+\rangle\langle +| = \begin{bmatrix} 0.5 & 0.25 \\ 0.25 & 0.25 \end{bmatrix},$$

where $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. In this example, $\mathrm{tr}(\rho) = 0.75 \leq 1$.

We use $\mathcal{D}(\mathcal{H})$ to denote the set of all density operators on Hilbert space $\mathcal{H}$.

General Quantum Operations

Unitary transformations and quantum measurements from Section 2.3 can be described within the unified framework of quantum operations (a.k.a., quantum channels) on density operators.

A general quantum operation $\mathcal{E}$ on Hilbert space $\mathcal{H}$ is a *completely positive, trace-non-increasing* map on the set of density operators $\mathcal{D}(\mathcal{H})$. Equivalently, by the Kraus representation theorem, $\mathcal{E}$ can be represented by a set of Kraus operators $\{E_k\}_k$ satisfying

$$0 \sqsubseteq \sum_k E_k^\dagger E_k \sqsubseteq \mathbb{1}, \tag{2.5}$$

where $\sqsubseteq$ is the Loewner order ($A \sqsubseteq B$ iff $B - A$ is positive semi-definite). Note that Equation (2.5) is a generalised condition of Equation (2.1).

If the system's state is a mixed state ρ, applying $\mathcal{E}$ to ρ transforms the current state to

$$\mathcal{E}(\rho) = \sum_k E_k \rho E_k^\dagger.$$

Unitary transformations and quantum measurements are special cases of quantum operations: a unitary U has a single Kraus operator $\{U\}$, and a measurement $\{M_m\}_m$

has Kraus operators $\{M_m\}_m$. They are *trace-preserving* quantum operations, meaning $\mathrm{tr}(\mathcal{E}(\rho)) = \mathrm{tr}(\rho)$.

In general, quantum operations need not be trace-preserving. For example, from a measurement $\{M_m\}_m$, we can select a partial measurement $\{M_{m_0}, M_{m_1}\}$, corresponding to two specific classical outcomes m_0 and m_1. Then, $\{M_{m_0}, M_{m_1}\}$ is also a quantum operation.

We use $\mathcal{QO}(\mathcal{H})$ to denote the set of all quantum operations on Hilbert space $\mathcal{H}$. For convenience, we adopt the following convention. When the context is clear, a quantum operation $\mathcal{E} \in \mathcal{QO}(\mathcal{H})$ is also used to represent its extension to a larger space $\mathcal{E} \otimes \mathbb{1} \in \mathcal{QO}(\mathcal{H} \otimes \mathcal{H}')$. Here, $\mathbb{1} \in \mathcal{QO}(\mathcal{H}')$ is the identity quantum operation with Kraus operators $\{\mathbb{1}\}$. In this case, $\mathcal{E} \in \mathcal{QO}(\mathcal{H})$ only indicates that $\mathcal{E}$ acts non-trivially on $\mathcal{H}$. As an example of this convention, consider two quantum operations $\mathcal{E} \in \mathcal{QO}(\mathcal{H}_1 \otimes \mathcal{H}_2)$ and $\mathcal{F} \in \mathcal{QO}(\mathcal{H}_1 \otimes \mathcal{H}_3)$. Then,

$$\mathcal{E} + \mathcal{F} = \mathcal{E} \otimes \mathbb{1}_3 + \mathcal{F} \otimes \mathbb{1}_2 \in \mathcal{QO}(\mathcal{H}_1 \otimes \mathcal{H}_2 \otimes \mathcal{H}_3),$$

where $\mathbb{1}_2 \in \mathcal{QO}(\mathcal{H}_2)$ and $\mathbb{1}_3 \in \mathcal{QO}(\mathcal{H}_3)$ are identity quantum operations on $\mathcal{H}_2$ and $\mathcal{H}_3$, respectively.

Part I

Programming

Chapter 3

Introduction to Quantum Recursive Programming

In this chapter, we provide a thorough introduction to the emerging paradigm of quantum recursive programming. We begin with the motivation for quantum recursion, which originates from the well-established notion of classical recursion but possesses a quantum nature due to the unique feature of quantum control flow. Then, we provide the background on the quantum programming language $\mathbf{RQC}^{++}$ [25, 37] for describing such quantum recursion, detailing its syntax and semantics. Finally, we present a series of examples to showcase the expressive power of quantum recursive programs.

3.1 Motivation

3.1.1 Recursion

Recursion is a fundamental concept in computer science. In the context of programming languages, recursion enables programmers to conveniently describe complicated computations as compact programs. By allowing a procedure to call itself recursively, a short static program text can generate a long, dynamic (and potentially unbounded) program execution [1], as illustrated in Figure 3.1. Well-known examples of classical recursion include Hoare's quicksort algorithm [172], Cooley-Tukey's fast Fourier transform [200], various recursive data structures [173], and divide-and-conquer algorithms. The implementation of classical recursion is well-studied and was a key feature of the influential programming language ALGOL 60 [201–204]. In modern imperative programming languages, the recursive procedure call is a standard programming primitive.

3.1.2 Quantum Control Flow

To achieve the most general form of recursion in quantum programming, we need another programming feature called quantum control flow, which has been repeatedly introduced in the literature [22, 25, 26, 28, 76–78]. Recall that in classical programming, the

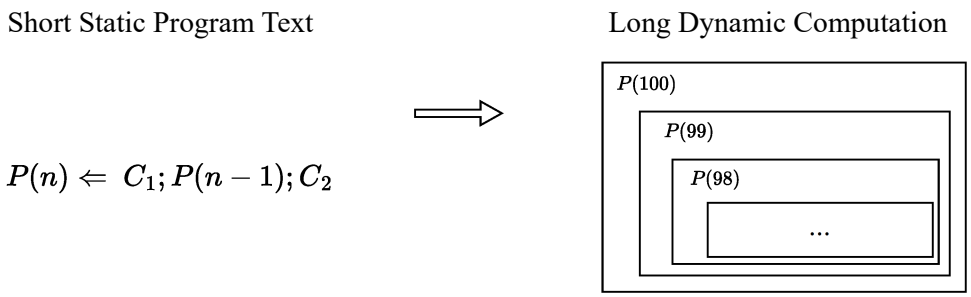


FIGURE 3.1: Recursive procedure calls generate long dynamic computation from short static program text.

control flow is only classical. For example, when executing an if-statement of the form **if** b **then** C_1 **else** C_2 **fi**, the control flow only enters either branch C_1 or C_2, depending on the value of the boolean expression b.

In contrast, quantum programming supports two types of control flow. The first, classical control flow, is similar to the standard case: e.g., a classical if-statement in quantum programming still conditions on a classical boolean expression. The second, quantum control flow, is unique and usually managed by the *quantum if-statement* of the form **qif**$[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)$ **fiq**. Here, q is a qubit variable that can be in a superposition state $\alpha|0\rangle + \beta|1\rangle$. When this **qif** statement is executed, the control flow enters both quantum branches (executing C_0 and C_1) in superposition, conditioned on the quantum state of q. Figure 3.2 illustrates the distinction between classical and quantum control flow.

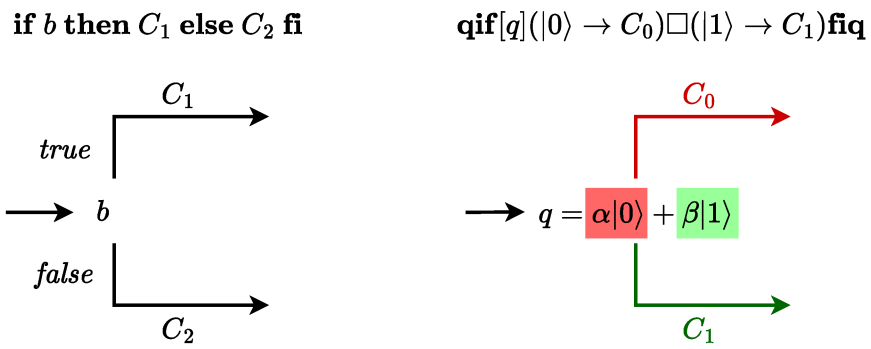


FIGURE 3.2: Classical control flow vs. quantum control flow.

3.1.3 Quantum Recursion

The most general form of recursion in quantum programming, termed quantum recursion, is realised by combining recursive procedure calls with quantum control flow. It should be contrasted with the following two weaker forms of recursion:

- Classical recursion in quantum programming (see e.g., [28, 205–207]), which involves recursive procedure calls but only has classical control flow.
- Classically bounded recursion in superposition (see e.g., [27, 208]), which only allows a single layer of interleaving between recursive procedure calls and quantum control flow.

The quantum recursive programming language **RQC**$^{++}$, introduced by [25, 37], provides an elegant framework for describing this general form of quantum recursion. Its expressive power has been demonstrated by various examples [37], such as quantum Fourier transform, generation of recursively defined quantum states, quantum random access memory and quantum state preparation.

As an illustrative example, we consider the quantum Fourier transform (QFT), a key component of Shor's integer factoring algorithm [12]. The QFT on n qubits is a unitary transformation $QFT(n)$, defined as:

$$QFT(n)\,|j\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi ijk/2^n}\,|k\rangle$$

for $j = 0, 1, \ldots, 2^n - 1$.

The textbook implementation [195] of $QFT(n)$ is based on the following observation:

$$QFT(n)\,|j_1 \ldots j_n\rangle = \frac{1}{\sqrt{2^n}} \bigotimes_{l=1}^{n} \left(|0\rangle + e^{2\pi i 0.j_{n-l+1}\ldots j_n}\,|1\rangle\right),$$

where we use the bit string representations $k_1 \ldots k_m = \sum_{l=1}^{m} k_l 2^{m-l}$ and $0.k_1 \ldots k_m = \sum_{l=1}^{m} k_l 2^{-l}$. This observation leads to the standard quantum circuit for $QFT(n)$, shown in Figure 3.3 (in program form), where $CR(l)$ is the controlled version of the single-qubit unitary gate $R(l)$ defined by

$$R(l) = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^l} \end{bmatrix},$$

and $Reverse[q_1, \ldots, q_n]$ is a quantum circuit reversing the order of the qubits $q_1, \ldots, q_n$.

Note that the standard program in Figure 3.3 requires a program text that grows with the number of qubits n, making it inconvenient for quantum programmers. In contrast, as observed in [37], quantum recursion enables an elegant and compact QFT program (in the language **RQC**$^{++}$) whose size is independent of n, as shown in Figure 3.4. Here,

$$
\begin{aligned}
QFT(n)[q_1,\ldots,q_n] \equiv\ & H[q_1]; CR(2)[q_2,q_1]; CR(3)[q_3,q_1]; \ldots; CR(n-1)[q_{n-1},q_1]; CR(n)[q_n,q_1]; \\
& H[q_2]; CR(2)[q_3,q_2]; CR(3)[q_4,q_2]; \ldots; CR(n-1)[q_n,q_2]; \\
& \ldots \\
& H[q_{n-1}]; CR(2)[q_n,q_{n-1}]; \\
& H[q_n]; \\
& Reverse[q_1,\ldots,q_n]
\end{aligned}
$$

FIGURE 3.3: Textbook quantum circuit (in the program form) for the QFT [195].

$$
\begin{aligned}
QFT(m,n) \Leftarrow\ & \textbf{if } m = n \textbf{ then } S(0)[q[m]] \\
& \qquad \textbf{else } Rot(m,n,0); QFT(m+1,n); Shift(m,n) \\
& \textbf{fi} \\
Rot(m,n,\theta) \Leftarrow\ & \textbf{if } m = n \textbf{ then } S(\theta)[q[m]] \\
& \qquad \textbf{else qif}[q[n]]\left(\square_{x=0}^{1} |x\rangle \rightarrow Rot(m,n-1,(\theta+x)/2)\right) \textbf{fiq} \\
& \textbf{fi} \\
Shift(m,n) \Leftarrow\ & \textbf{if } m < n \textbf{ then} \\
& \qquad Swap[q[m],q[n]]; Shift(m+1,n) \\
& \textbf{fi}.
\end{aligned}
$$

FIGURE 3.4: A quantum recursive program for the QFT.

$QFT(m,n)$ is the main procedure, describing a unitary acting on an array $q[m:n]$ of qubits indexed by integers $[m:n] = \{m, m+1, \ldots, n\}$, and $S(\theta)$ is a single-qubit unitary gate defined by

$$S(\theta) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ e^{\pi i \theta} & -e^{\pi i \theta} \end{bmatrix}.$$

We note that this program features quantum recursion, which creates an intricate interplay between recursive procedure calls and quantum control flow. This is exemplified by the procedure $Rot(m,n,\theta)$, calling itself $Rot(m,n-1,(\theta+x)/2)$ in two quantum branches created by the **qif** statement, with different classical parameters. Indeed, the construction of this program is inspired [37] by the classical Cooley-Tukey algorithm for fast Fourier transform [200].

The next section provides the background on the programming language $\mathbf{RQC}^{++}$, followed by a section showing more examples of quantum algorithms written as quantum recursive programs.

3.2 Quantum Recursive Programming Language RQC^{++}

In this section, we introduce **RQC**$^{++}$ [37], a high-level programming language for describing quantum recursive programs. We adopt the original definitions [37] with slight modifications to facilitate the implementations discussed later in this thesis.

We first list several features of the language **RQC**$^{++}$:

- Two key features of **RQC**$^{++}$, compared to other existing quantum programming languages, are *quantum control flow* and *recursive procedure calls*, which together enable quantum recursion.
- For simplicity, **RQC**$^{++}$ is not explicitly typed.
- A program in **RQC**$^{++}$ describes a *unitary* quantum circuit without measurements, whose size is parameterised. (A discussion about the unitary restriction is provided in Section 3.4.1.)
- The alphabet of **RQC**$^{++}$ contains classical and quantum variables, while classical variables solely serve for *specifying the control* of the programs.
- Quantum control flow in **RQC**$^{++}$ is *fully managed* by the **qif** statements: quantum branches are only created by **qif**, and only merged by **fiq**.

3.2.1 Program Variables and Procedure Identifiers

Classical Variables

Classical variables in **RQC**$^{++}$ are solely for specifying the control of programs. They can be used to define formal parameters (of procedure declarations), store intermediate results, and express conditions (in **if** and **while** statements) and actual parameters (of procedure calls). For example, in Figure 3.4, m and n define the formal parameters of $QFT(m,n)$, and are used in the if-statement and the actual parameters for procedure calls. The dimension of the quantum unitary transformation described by a program in **RQC**$^{++}$ can also depend on classical parameters.

Classical variables are classical only in the eyes of the programmer, or more specifically, in the eyes of the enclosing procedure. Meanwhile, the procedure calls can be used in quantum superposition, e.g., within quantum branches created by the **qif** statements. In the implementation, classical variables will be realised by the quantum hardware instead.

A classical variable x has a type $T(x)$, which can be thought of as a set; i.e., $x \in T(x)$. In this thesis, we will only consider three types of classical variables, **Uint**, **Int**, and **Bit**, standing for unsigned integer type, integer type, and bit type, respectively. We use $\overline{x} = x_1 x_2 \dots x_n$ to denote a list of classical variables.

Quantum Variables

Quantum variables are very different from classical variables. The state of quantum variables can be in superposition, and different quantum variables can be entangled. An elementary quantum variable q has a type $T(q) = \mathcal{H}$, which represents the corresponding Hilbert space of q. The type (Hilbert space) of a list $\overline{q}$ of distinct quantum variables $q_1, \ldots, q_n$ is then the tensor product $T(\overline{q}) = \bigotimes_{i=1}^{n} \mathcal{H}_i$, where each $\mathcal{H}_i = T(q_i)$ is the type of q_i. In this thesis, we will only consider two types of quantum variables, **Qint** and **Qbit**, standing for quantum integer type and qubit type, respectively.

Procedure Identifiers

Procedure identifiers are the names of procedures and are of a designated type **Pid**. For each procedure identifier P, we can associate with it a classical variable $P.ent$ of type **Uint**, storing the entry address of the procedure declaration of P, in our later implementation of quantum recursive programs. The value of $P.ent$ is determined and static after the program is compiled and loaded into the memory. In the compiled program, $P.ent$ will be used for handling procedure calls (see also Section 5.1.4).

Arrays

Classical and quantum variables can all be generalised to array variables. Procedure identifiers can also be generalised to procedure arrays. An array can be subscripted by classical values. The notion of array is standard, e.g., if x is a 1-indexed one-dimensional classical array, then $x[10]$ represents the 10^{th} element in x. For simplicity, in this thesis, we only consider one-dimensional arrays. High-dimensional arrays can easily be simulated by one-dimensional arrays.

The type of an array depends on the type of its elements:

1. The type of a classical array x is $(T \to X) \equiv X^T$, where X is the type of the elements in x, and T is the type of the subscript.

2. The type of a quantum array q is $(T \to \mathcal{H}) \equiv \bigotimes_{t \in T} \mathcal{H}_t$,[1] where $\mathcal{H}_t = \mathcal{H}$ is the type of the element, and T is the type of the subscript.

3. The type of a procedure array is $(T \to \textbf{Pid}) \equiv \textbf{Pid}^T$, where T is the type of the subscript.

Elements in a classical or quantum array are assigned contiguous addresses in the memory (see also Section 5.1.3 in later implementation), and hence can be efficiently addressed.

[1] Here, we assume the elements in T are ordered.

Given an array variable, we can also write corresponding subscripted variables. For classical array x of type $T \to X$ and quantum array q of type $T \to \mathcal{H}$, we can write subscripted variables $x[t]$, $q[t]$ for classical expression t of type T, respectively. They are of types X and $\mathcal{H}$, respectively. For example, one can write subscripted quantum variable $q[5x + 2y]$, where x, y are classical variables. Similarly, given a procedure array P, we can also write corresponding subscripted procedure identifiers $P[t]$.

For simplicity, we restrict the use of nested subscriptions. In particular, for classical subscripted variable $x[t]$, we require that t contains no more subscripted variables. For example, we do not allow subscripted variable $x[y[10]]$ for classical x, y, but allow $q[x[7z]]$ for quantum q and classical x, z. The implementation of nested subscriptions can actually be treated in similar but more complicated ways.

Suppose x is a classical array of type $T \to X$ with $T =$ **Uint** or **Int**. We use the notation $x[k : l]$ to denote the restriction of x to the interval $[k : l] = \{k, k+1, \ldots, l-1, l\}$. Then, $x[k : l]$ is a variable of type $[k : l] \to X$. The same convention applies to quantum and procedure arrays.

Global Variables vs. Local Variables

In **RQC**$^{++}$, a variable is not declared before its use. All variables are treated as global variables and initialised to 0. Local (classical) variables will be realised by the block statement

$$\textbf{begin local } \overline{x} := \overline{t}; \ldots \textbf{end}.$$

Within the scope of the block, the list of classical variables $\overline{x}$ are regarded as local variables, initialised to new values specified by the list of expressions $\overline{t}$ at the begining of the block and restore their old values at the end of the block. A consequence of this treatment is that in a procedure call, the callee can use the variables set up by the caller.

3.2.2 Syntax

The syntax of **RQC**$^{++}$ is summarised in Figure 3.5. Here, a program is specified by $\mathcal{P}$, a set of procedure declarations, with a main procedure P_{main}. Each procedure declaration has the form $P(\overline{u}) \Leftarrow C$, where P is the procedure identifier, $\overline{u}$ is a list of formal parameters (which can be empty), and C is the procedure body. The recursion is supported by that C can contains P itself. A statement C is inductively defined, where U represents an elementary unitary gate and b represents a classical binary expression. We further explain as follows.

$\mathcal{P} = \{P_1(\overline{u}_1) \Leftarrow C_1, ..., P_n(\overline{u}_n) \Leftarrow C_n\}$	Procedure declaration
$C ::= \textbf{skip} \mid C_1; C_2$	Sequential composition
$\mid \overline{x} := \overline{t}$	Classical assignment
$\mid U[\overline{q}]$	Quantum unitary gate
$\mid P(\overline{t})$	Procedure call
$\mid \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}$	Classical if-statement
$\mid \textbf{while } b \textbf{ do } C \textbf{ od}$	Classical loop
$\mid \textbf{begin local } \overline{x} := \overline{t};\ C \textbf{ end}$	Local classical variable block
$\mid \textbf{qif}[q](\lvert 0\rangle \rightarrow C_0)\square(\lvert 1\rangle \rightarrow C_1)\textbf{fiq}$	Quantum if-statement

FIGURE 3.5: The syntax of quantum recursive programming language **RQC**$^{++}$.

Classical Assignment, If-Statement and Loop

In **RQC**$^{++}$, the classical assignment, if-statement, and loop are similar to their counterparts in classical programming languages.

- The assignment $\overline{x} := \overline{t}$ simultaneously assigns the values of the list of expressions $\overline{t}$ to the list of classical variables $\overline{x}$. Note that in $\overline{t} = t_1 t_2 \dots t_n$, t_i might contains variables in $\overline{x} = x_1 x_2 \dots x_n$.
- The **if** b **then** C_1 **else** C_2 **fi** statement chooses one of statements C_1 and C_2 to execute, depending on the value of the boolean expression b.
- The **while** b **do** C **od** statement repeatedly execute statement C, conditioned on that the value of the boolean expression b is 1.

Block Statement

The block statement is used to declare classical variables as local variables. In particular, the statement

$$\textbf{begin local } \overline{x} := \overline{t};\ C \textbf{ end}$$

declares the list of variables $\overline{x}$ as local variables within the scope of **begin** ... **end**, initialised to values of the list of expressions $\overline{t}$. At the end of the block, $\overline{x}$ will restore the old values. The block statement is also useful in defining the semantics of procedure calls (to be introduced in Section 3.2.3).

Procedure Call

The procedure call and procedure declaration are also similar to their classical counterparts.

- The procedure declaration $P(\overline{u}) \Leftarrow C$ declares a procedure identifier P with the list of formal parameters $\overline{u}$ and procedure body C.
- The procedure call $P(\overline{t})$ calls the procedure with identifier P with the list of actual parameters $\overline{t}$.

Quantum Unitary Gate

The statement $U[\overline{q}]$ applies an elementary quantum unitary gate U on the list $\overline{q}$ of quantum variables. For simplicity, we restrict that $\overline{q}$ contains at most two quantum variables, which are of type **Qbit**. The type of the unitary gate U needs to be matched with $\overline{q}$. If $\overline{q} = q_1$, then U is a single-qubit unitary gate; if $\overline{q} = q_1 q_2$, then U is a two-qubit unitary gate. Here, for the two-qubit case, note that we need to promise q_1 and q_2 are distinct quantum variables, which will be formally stated in Section 3.2.3. Additionally, we restrict that U is chosen from a fixed set of elementary unitary gate set $\mathcal{G}$ of size $|\mathcal{G}| = O(1)$. For example, $\mathcal{G} = \{H, T, CNOT\}$. More complicated unitaries, like parameterised rotation $R_X(\theta)$, can be implemented by procedure call with classical parameters.

Quantum Control Flow

The quantum if-statement $\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}$ in $\mathbf{RQC}^{++}$ explicitly manages quantum control flow. It executes C_i, conditioned on the qubit variable q (a.k.a., quantum coin): when q is in state $|0\rangle$, C_0 is executed; when q is in state $|1\rangle$, C_1 is executed. When q is in a quantum superposition state, the subsystems that C_0 and C_1 act on will be entangled with q. Unlike the classical if-statement where the control flow only runs through one of the two branches, the quantum control flow in the **qif** statement is essentially quantum — it runs through both quantum branches created by the **qif** statement, in superposition. Note that the superposition state is held in the composite system including q and the quantum variables in C_0, C_1. The condition of the external quantum coin is to promise the physicality of the **qif** statement, which will be formally stated in Condition 1, and will be further explained later when we come to the semantics of $\mathbf{RQC}^{++}$.

It is also worth pointing out that in $\mathbf{RQC}^{++}$, the quantum control flow is fully guarded by the **qif** statements. In particular, quantum branches are only created by **qif**, and only merged by **fiq**.

3.2.3 Semantics

Now we introduce the operational semantics of $\mathbf{RQC}^{++}$. Let us fix a finite set of classical and quantum variables. A configuration is represented by $(C, \sigma, |\psi\rangle)$, where C is the remaining statement to be executed or $C = \downarrow$ (standing for termination; and we denote

(SK) $(\mathbf{skip}, \sigma, |\psi\rangle) \to (\downarrow, \sigma, |\psi\rangle)$

(AS) $(\overline{x} := \overline{t}, \sigma, |\psi\rangle) \to (\downarrow, \sigma[\overline{x} := \sigma(\overline{t})], |\psi\rangle)$

(GA) $$\frac{\sigma \models Dist(\overline{q})}{(U[\overline{q}], \sigma, |\psi\rangle) \to \left(\downarrow, \sigma, \left(U_{\sigma(\overline{q})} \otimes \mathbb{1}\right)|\psi\rangle\right)}$$

(SC) $$\frac{(C_1, \sigma, |\psi\rangle) \to (C_1', \sigma', |\psi'\rangle)}{(C_1; C_2, \sigma, |\psi\rangle) \to (C_1'; C_2, \sigma', |\psi'\rangle)}$$

(IF) $$\frac{\sigma \models b}{(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, \sigma, |\psi\rangle) \to (C_1, \sigma, |\psi\rangle)}, \quad \frac{\sigma \models \neg b}{(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, \sigma, |\psi\rangle) \to (C_2, \sigma, |\psi\rangle)}$$

(LP) $$\frac{\sigma \models b}{(\mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, \sigma, |\psi\rangle) \to (C; \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, \sigma, |\psi\rangle)}, \quad \frac{\sigma \models \neg b}{(\mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}), \sigma, |\psi\rangle \to (\downarrow, \sigma, |\psi\rangle)}$$

(BS) $(\mathbf{begin\ local}\ \overline{x} := \overline{t}; C\ \mathbf{end}, \sigma, |\psi\rangle) \to (\overline{x} := \overline{t}; C; \overline{x} := \sigma(\overline{x}), \sigma, |\psi\rangle)$

(RC) $$\frac{P(\overline{u}) \Leftarrow C \in \mathcal{P}}{(P(\overline{t}), \sigma, |\psi\rangle) \to (\mathbf{begin\ local}\ \overline{u} := \overline{t}; C\ \mathbf{end}, \sigma, |\psi\rangle)}$$

(QIF) $$\frac{|\psi\rangle = \alpha_0 |0\rangle_{\sigma(q)} |\theta_0\rangle + \alpha_1 |1\rangle_{\sigma(q)} |\theta_1\rangle, \qquad (C_i, \sigma, |\theta_i\rangle) \to^* (\downarrow, \sigma, |\theta_i'\rangle)\ (i = 0, 1)}{(\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}, \sigma, |\psi\rangle) \to \left(\downarrow, \sigma, \alpha_0 |0\rangle_{\sigma(q)} |\theta_0'\rangle + \alpha_1 |1\rangle_{\sigma(q)} |\theta_1'\rangle\right)}$$

FIGURE 3.6: Transition rules for defining the operational semantics of $\mathbf{RQC}^{++}$.

$\downarrow; C' \equiv C'$), σ is the current classical state of all classical variables, and $|\psi\rangle$ is the current quantum state of all quantum variables. Let $\mathcal{C}$ be the set of configurations. Then, the operational semantics is defined as the transition relation $\to \subseteq \mathcal{C} \times \mathcal{C}$, of the form $(C, \sigma, |\psi\rangle) \to (C', \sigma', |\psi'\rangle)$, by a series of transition rules. The transition rules for defining the operational semantics of $\mathbf{RQC}^{++}$ are presented in Figure 3.6. Let us also further explain some of them as follows.

- In the (GA) rule, $\sigma(\overline{q})$ denotes the subsystem specified by q with respect to the classical state σ. In particular, if $\overline{q} = q_1$, where q_1 is not subscripted, then $\sigma(\overline{q}) = q_1$; if $\overline{q} = q_1$ with $q_1 = r[t_1]$ for quantum array r, then $\sigma(\overline{q}) = r[\sigma(t_1)]$; if $\overline{q} = q_1 q_2$ with $q_1 = r[t_1]$ and $q_2 = r'[t_2]$ for quantum arrays r and r', then $\sigma(\overline{q}) = r[\sigma(t_1)]r'[\sigma(t_2)]$. The condition $\sigma \models Dist(\overline{q})$ means in the classical state σ, $\overline{q}$ is a list of distinct quantum variables. In particular, if $\overline{q} = q_1$, then $Dist(\overline{q})$ is always true; if $\overline{q} = q_1 q_2$ with $q_1 = r[t_1]$ and $q_2 = r'[t_2]$ for quantum arrays r and r', then $Dist(\overline{q})$ is the logical formula $r = r' \to t_1 \neq t_2$. For simplicity, we assume all programs considered always satisfy this condition.

- In the (QIF) rule, $i = 0, 1$ correspond to the two quantum branches, controlled by the external quantum coin q. Here, $\sigma(q)$ denotes the subsystem specified by q with respect to classical state σ. As usual, $\to^k$ denotes the composition of k copies of $\to$, and $\to^* = \bigcup_{k=0}^{\infty} \to^k$. The semantics of the **qif** statement is exactly a quantum multiplexor [174] with one control qubit q: if each C_i describes a unitary U_i, then $\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}$ describes the unitary $U_0 \oplus U_1 = |0\rangle\langle 0|_{\sigma(q)} \otimes U_0 + |1\rangle\langle 1|_{\sigma(q)} \otimes U_1$. We can also see how the condition of external quantum coin (see

the formal Condition 1 later) is used: it promises that q is separated from the subsystems that C_0 and C_1 act on, and therefore the current state $|\psi\rangle$ can be written as $\alpha_0 |0\rangle_{\sigma(q)} |\theta_0\rangle + \alpha_1 |1\rangle_{\sigma(q)} |\theta_1\rangle$, given the classical state σ.

Moreover, note that $(C_i, \sigma, |\theta_i\rangle)$ are required to terminate in the same classical state σ for both branches ($i = 0, 1$) to *prevent classical variables from being in superposition*. In other word, classical variables are kept *disentangled* from quantum variables after the **qif** statement. In [37], originally, both quantum branches are only required to terminate in the same classical state σ' that may differ from the initial σ. For simplicity of later implementation, here in Figure 3.6, the requirement has been made slightly stricter, while remaining easy to meet in practice. The requirement seems inevitable to separate classical and quantum variables in the presence of quantum control flow. As a result, only local classical variables can be arbitrarily modified in the **qif** statement. If one wishes to return different data from two quantum branches, then the data becomes intrinsically quantum and should therefore be stored in quantum variables.

It is worth stressing again that classical variables in **RQC**$^{++}$ are classical only in the eyes of the programmer (or the enclosing procedure); they become quantum when the program is implemented on quantum hardware.

- In the (BS) rule, the sequential composition $\overline{x} := \overline{t}; C; \overline{x} := \sigma(\overline{x})$ formalises the meaning of initialising $\overline{x}$ at the beginning of the block and restoring their old values (i.e., $\sigma(\overline{x})$) at the end of the block. Note that all variables are initialised to 0 (see Section 3.2.1) at the start of the program.

- In the (RC) rule, the formal parameters $\overline{u}$ to be used by the callee will be locally replaced by the actual parameters $\overline{t}$ set by the caller.

Conditions for Well-defined Semantics

A program written in the syntax in Figure 3.5 is not yet promised to have well-defined semantics. We present three conditions for a program in **RQC**$^{++}$ to have well-defined semantics, in particular, for the (QIF) rule to be properly and easily applied.

The first condition guarantees that in every **qif** statement, q is external to C_0 and C_1. This is introduced for the **qif** statement to be physically meaningful. We use $qv(C, \sigma)$ to denote the quantum variables in statement C with respect to a given classical state σ. For simplicity, its precise definition is given later in this section.

Condition 1 (External quantum coin)**.** For any procedure declaration $P(\overline{u}) \Leftarrow C \in \mathcal{P}$, and any **qif**$[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)$**fiq** appearing in C, and any classical state σ (of concern), $q \notin qv(C_0, \sigma) \cup qv(C_1, \sigma)$.

The second condition says that in every **qif** statement, both C_0 and C_1 contain no free changed (classical) variables. A classical variable is *free* if it is not declared as local variable. It is *changed* if it appears on the LHS of an assignment. We use $fcv(C, \sigma)$ to denote the free changed variables in C with respect to σ. Again, its precise definition is given later. This condition is introduced as the (QIF) rule requires $(C_i, \sigma, |\psi\rangle)$ to terminate in the same classical state σ for both branches $i = 0, 1$.

Condition 2 (No free changed variables in **qif** statements)**.** For any $P(\overline{u}) \Leftarrow C \in \mathcal{P}$, any $\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}$ appearing in C, and any classical state σ (of concern), $fcv(C_0, \sigma) = fcv(C_1, \sigma) = \emptyset$.

The third condition says that every procedure body contains no free changed variables. This condition is introduced to simplify the process of compilation, as it allows the procedure calls to be arbitrarily used together with the **qif** statements without violating Condition 2.

Condition 3 (No free changed variables in procedure bodies)**.** For any $P(\overline{u}) \Leftarrow C \in \mathcal{P}$ and any classical state σ (of concern), $fcv(C, \sigma) = \emptyset$.

Two notions in the above three conditions remain to be formalised. The first notion is the quantum variables $qv(C, \sigma)$ in a statement C with respect to a given classical state σ. It is used in Condition 1. Before formally defining qv, we need some additional notions. We use $\mathcal{QV}$ to denote the set of all quantum variables. Let $\mathcal{C}'$ be the set of classical configurations of the form (C, σ). Let $\mathcal{F} \equiv \mathcal{C}' \to 2^{\mathcal{QV}}$ be the set of functions f that maps a configuration $(C, \sigma) \in \mathcal{C}'$ to a set of quantum variables $f(C, \sigma) \in 2^{\mathcal{QV}}$. Then, we can define a partial order $\sqsubseteq$ on $\mathcal{F}$, such that for $f, g \in \mathcal{F}$, $f \sqsubseteq g$ iff $\forall (C, \sigma) \in \mathcal{C}'$, $f(C, \sigma) \subseteq g(C, \sigma)$.

Definition 1 (Quantum variables)**.** For a statement $C \in \mathbf{RQC}^{++}$ and a classical state σ, the quantum variables of C with respect to σ is denoted by $qv(C, \sigma)$, and $qv(\cdot, \cdot) : \mathcal{C}' \to 2^{\mathcal{QV}}$ is defined as the least element $f \in \mathcal{F}$ (with respect to the order $\sqsubseteq$) that satisfies the following conditions:

1. $f(C, \sigma) \supseteq \emptyset$ if $C \equiv \mathbf{skip} \mid \overline{x} := \overline{t}$.
2. $f(U[\overline{q}], \sigma) \supseteq \overline{q}$.
3. $f(C_1; C_2, \sigma) \supseteq f(C_1, \sigma) \cup f(C_2, \sigma')$, where σ' satisfies $(C_1, \sigma, |\psi\rangle) \to^* (\downarrow, \sigma', |\psi'\rangle)$ for any quantum state $|\psi\rangle$.
4. $f(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, \sigma) \supseteq \begin{cases} f(C_1, \sigma), & \sigma \models b, \\ f(C_2, \sigma), & \sigma \models \neg b. \end{cases}$
5. $f(\mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, \sigma) \supseteq \begin{cases} f(C; \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, \sigma), & \sigma \models b, \\ \emptyset, & \sigma \models \neg b. \end{cases}$

6. $f(\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}, \sigma) \supseteq \{q\} \cup f(C_0, \sigma) \cup f(C_1, \sigma)$.
7. $f\big(\mathbf{begin\ local}\ \overline{x} := \overline{t}; C\ \mathbf{end}, \sigma\big) \supseteq f\big(\overline{x} := \overline{t}; C; \overline{x} := \sigma(\overline{x}), \sigma\big)$.
8. $f\big(P(\overline{t}), \sigma\big) \supseteq f\big(\mathbf{begin\ local}\ \overline{u} := \overline{t}; C\ \mathbf{end}, \sigma\big)$, if $P(\overline{u}) \Leftarrow C \in \mathcal{P}$.

The following lemma shows that $qv(\cdot, \cdot)$ in Definition 1 is well-defined.

Lemma 1. *Let S be the set of all functions $f \in \mathcal{F}$ that satisfy the conditions in Definition 1. Then S has the least element.*

Proof. It is easy to see $S \neq \emptyset$, because the constant function h with $\forall (C, \sigma) \in \mathcal{C}', h(C, \sigma) = \mathcal{QV}$ is in S. Let $g \in \mathcal{F}$ be defined by $g(C, \sigma) = \bigcap_{f \in S} f(C, \sigma)$ for all $(C, \sigma) \in \mathcal{C}'$. Then it suffices to show that $g \in S$; that is, g satisfies the conditions in Definition 1. Here, we only prove that g satisfies Condition 6 in Definition 1; other conditions can be verified similarly.

$$\begin{aligned} g(\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}, \sigma) &= \bigcap_{f \in S} f(\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}, \sigma) \\ &\supseteq \bigcap_{f \in S} (\{q\} \cup f(C_0, \sigma) \cup f(C_1, \sigma)) \\ &\supseteq \{q\} \cup \Big(\bigcap_{f \in S} f(C_0, \sigma)\Big) \cup \Big(\bigcap_{f \in S} f(C_1, \sigma)\Big) \\ &= \{q\} \cup g(C_0, \sigma) \cup g(C_1, \sigma) \end{aligned}$$

□

The second notion is free changed (classical) variables $fcv(C, \sigma)$ in a statement C with respect to a given classical state σ. It is used in Conditions 2 and 3. Similar to the case of defining qv, we also need some additional notions. We use $\mathcal{CV}$ to denote the set of all classical variables. Let $\mathcal{G} \equiv \mathcal{C}' \to 2^{\mathcal{CV}}$, which also has a partial order $\sqsubseteq$ induced by the subset order $\subseteq$.

Definition 2 (Free changed (classical) variables)**.** For a statement $C \in \mathbf{RQC}^{++}$ and a classical state σ, the free changed (classical) variables in C with respect to σ is denoted by $fcv(C, \sigma)$, and $fcv(\cdot, \cdot) : \mathcal{C}' \to 2^{\mathcal{CV}}$ is defined as the least element $f \in \mathcal{G}$ (with respect to the order $\sqsubseteq$) that satisfies the following conditions:

1. $f(C, \sigma) \supseteq \emptyset$ if $C \equiv \mathbf{skip} \mid U[\overline{q}] \mid P(\overline{t})$.
2. $f(C, \sigma) \supseteq \overline{x'}$ if $C \equiv \overline{x} := \overline{t}$, where $x'_i = y_i$ if $x_i = y_i$ for some basic classical variable y_i, and $x'_i = y_i[\sigma(e_i)]$ if $x_i = y_i[e_i]$ for some classical array variable y_i and expression e_i.

3. $f(C_1;C_2,\sigma) \supseteq f(C_1,\sigma) \cup f(C_2,\sigma')$, where σ' satisfies $(C_1,\sigma,|\psi\rangle) \to^* (\downarrow,\sigma',|\psi'\rangle)$ for any quantum state $|\psi\rangle$.

4. $f(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi},\sigma) \supseteq \begin{cases} f(C_1,\sigma), & \sigma \models b, \\ f(C_2,\sigma), & \sigma \models \neg b. \end{cases}$

5. $f(\textbf{while } b \textbf{ do } C \textbf{ od},\sigma) \supseteq \begin{cases} f(C;\textbf{while } b \textbf{ do } C \textbf{ od},\sigma), & \sigma \models b, \\ \emptyset, & \sigma \models \neg b. \end{cases}$

6. $f(\textbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\textbf{fiq},\sigma) \supseteq f(C_0,\sigma) \cup f(C_1,\sigma)$.

7. $f\big(\textbf{begin local } \overline{x} := \overline{t};\ C \textbf{ end}\big) \supseteq f(C) - \overline{x}$.

We can also prove *fcv* in Definition 2 is well-defined, as shown in the following lemma.

Lemma 2. *Let R be the set of all functions in $\mathcal{G}$ that satisfy the conditions in Definition 2. Then R has the least element.*

Proof. The proof is very similar to that of Lemma 1 and thus omitted. □

3.3 More Examples

In the last section, we have introduced the syntax and semantics of the programming language **RQC**$^{++}$. In this section, for further illustrating the power of quantum recursion, we provide several more examples of quantum algorithms written as programs in **RQC**$^{++}$. All of these examples feature compactness and elegance, demonstrating that quantum recursive programming provides a uniform way to capture and exploit the recursive structures in these algorithms.

Multi-Controlled Gate

We start with a simple example of multi-controlled unitary gate. It illustrates both **qif** statements and recursive procedure calls.

Example 1 (Multi-controlled Gate)**.** Let U be an elementary unitary gate. Then the multi-controlled U gate with $n-1$ control qubits is defined by:

$$C^{(*)}(U)\,|i_1,\ldots,i_{n-1}\rangle\,|\psi\rangle = \begin{cases} |i_1,\ldots,i_{n-1}\rangle\,U\,|\psi\rangle\,, & i_m = \ldots = i_{n-1} = 1, \\ |i_1,\ldots,i_{n-1}\rangle\,|\psi\rangle\,, & o.w. \end{cases}$$

for all $i_1, \ldots, i_{n-1} \in \{0,1\}$ and quantum state $|\psi\rangle$. We can describe $C^{(*)}(U)$ by the following program in $\mathbf{RQC}^{++}$ [37]:

$$\begin{aligned} P_{\text{main}}(n) &\Leftarrow P(1,n) \\ P(m,n) &\Leftarrow \textbf{if } m = n \\ &\qquad \textbf{then } U[q[n]] \\ &\qquad \textbf{else qif}[q[m]]\,|0\rangle \to \textbf{skip} \\ &\qquad\qquad\quad \square \quad |1\rangle \to P(m+1,n) \\ &\qquad \textbf{fiq} \\ &\quad \textbf{fi}. \end{aligned} \tag{3.1}$$

Generation of Recursively Defined Quantum States

Many physically relevant quantum states can be recursively defined. It is natural to generate these states using quantum recursive programs. One simple example is the generation of the Greenberger-Horne-Zeilinger (GHZ) state [209], although it only involves recursive procedure calls but no **qif** statements.

Example 2 (Generation of GHZ states)**.** The (generalised) Greenberger-Horne-Zeilinger (GHZ) state [209] of n qubits is defined as:

$$|\text{GHZ}(n)\rangle = \frac{1}{\sqrt{2}}\Big(|0\rangle^{\otimes n} + |1\rangle^{\otimes n}\Big).$$

We can use the following program in $\mathbf{RQC}^{++}$ to generate the GHZ state [37]:

$$\begin{aligned} P_{\text{main}}(n) &\Leftarrow \textbf{if } n = 1 \\ &\qquad \textbf{then } H[q[n]] \\ &\qquad \textbf{else } P_{\text{main}}(n-1); CNOT[q[n-1], q[n]] \\ &\quad \textbf{fi}. \end{aligned} \tag{3.2}$$

Another example is the generation of the Dicke state [210], which perfectly demonstrates the elegance of quantum recursion.

Example 3 (Generation of Dicke states)**.** The Dicke state [210] on n qubits with Hamming weight $0 \leq k \leq n$ is recursively defined as:

$$|D(n,k)\rangle = \begin{cases} \sqrt{\frac{k}{n}}\,|D(n-1,k-1)\rangle\,|1\rangle + \sqrt{\frac{n-k}{n}}\,|D(n-1,k)\rangle\,|0\rangle\,, & k \geq 1, \\ |0\rangle^{\otimes n}\,, & k = 0. \end{cases}$$

We can use the following program in $\mathbf{RQC}^{++}$ to generate the Dicke state:

$$\begin{aligned} P_{\text{main}}(n,k) \Leftarrow \textbf{if } & n = 0 \vee k = 0 \\ & \textbf{then skip} \\ & \textbf{else } R_Y\left(2\arcsin(\sqrt{k/n})\right)[q[n]]; \\ & \quad \textbf{qif}[q[n]]\,|0\rangle \to P_{\text{main}}(n-1,k); \\ & \quad\quad \square \quad |1\rangle \to P_{\text{main}}(n-1,k-1) \\ & \quad \textbf{fiq} \\ & \textbf{fi}. \end{aligned} \tag{3.3}$$

Here, $R_Y(\theta)$ is a single-qubit rotation gate defined by

$$R_Y(\theta) = \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}.$$

Quantum State Preparation

Quantum state preparation is a common subroutine in many quantum algorithms, e.g., Hamiltonian simulation [211–213], quantum machine learning [214, 215], and solving quantum linear system of equations [216, 217]. This example involves both **qif** statements and recursive procedure calls.

Example 4 (Quantum State Preparation)**.** The task of quantum state preparation is to generate the n-qubit quantum state

$$|\psi\rangle = \sum_{j\in[N]} \alpha_j |j\rangle, \tag{3.4}$$

where $N = 2^n$ and $\alpha_j \in \mathbb{C}$ satisfy $\sum_{j\in[N]} |\alpha_j|^2 = 1$.

For $j \in [N]$, let us define $\theta_j \in [0, 2\pi)$ such that $\alpha_j = e^{i\theta_j}|\alpha_j|$. For $l, r \in [N]$, let $S_{l,r} = \sum_{j=l}^{r} |\alpha_j|^2$. We define a single qubit unitary $U_{k,x}$ such that:

$$U_{k,x}|0\rangle = \sqrt{\gamma_x}\,|0\rangle + e^{i\beta_x}\sqrt{1-\gamma_x}\,|1\rangle, \tag{3.5}$$

where $\gamma_x = \frac{S_{u,w-1}}{S_{u,v-1}}$, $\beta_x = \theta_w - \theta_u$, $u = 2^{n-k}x$, $v = 2^{n-k}(x+1)$ and $w = (u+v)/2$.

Now we can use the following program in $\mathbf{RQC}^{++}$ to generate the state $|\psi\rangle$ [37]:

$$\begin{aligned} P_{\text{main}}(n) &\Leftarrow P(0,n,0) \\ P(k,n,x) &\Leftarrow \textbf{if } k \neq n \textbf{ then} \\ &\qquad Q(k,x); \\ &\qquad \textbf{qif}[q[k]]\ |0\rangle \rightarrow P(k+1,n,2x) \\ &\qquad\quad \square\ \ |1\rangle \rightarrow P(k+1,n,2x+1) \\ &\qquad \textbf{fiq} \\ &\quad \textbf{fi} \\ Q(k,x) &\Leftarrow C, \end{aligned} \tag{3.6}$$

where C is a quantum circuit that performs $U_{k,x}[q[k+1]]$. (In practice, when the elementary gate set is simple, e.g., $\{H,S,T,CNOT\}$, what C performs is only an approximation of $U_{k,x}$, and C depends on the explicit α_j (for $j \in [N]$).)

Quantum Random Access Memory

The last example is quantum random access memory (QRAM), the quantum counterpart of classical RAM and a useful subroutine in many quantum algorithms. More precisely, here, we focus on a circuit implementation of quantum random access quantum memory [218].

Example 5 (Circuit QRAM)**.** The swap access to a circuit QRAM storing $N = 2^n$ qubits is defined by

$$|x\rangle_r |i\rangle_a |M\rangle_{mem} \mapsto |M_i\rangle_r |i\rangle_a |M_0,\ldots,M_{i-1},x,M_{i+1},\ldots,M_{N-1}\rangle_{mem},$$

for all x, i and $M = (M_0,\ldots,M_{N-1})$. Here, r is the target register, a is the address register, and *mem* is the QRAM.

Let us regard a and mem as qubit arrays. Then, we can use the following program in $\mathbf{RQC}^{++}$ to perform a swap access [37]:

$$
\begin{aligned}
P_{\text{main}} &\Leftarrow U(0, N-1, 1) \\
U(l, r, k) &\Leftarrow \textbf{if } k \leq n \textbf{ then} \\
&\quad \textbf{begin local } m := \lfloor (l+r)/2 \rfloor; \\
&\quad\ \textbf{qif}[a[k]] \ \ |0\rangle \rightarrow U(l, m, k+1) \\
&\quad\quad\quad \square \quad |1\rangle \rightarrow U(m+1, r, k+1); \\
&\quad\quad\quad\quad\quad\ SWAP[mem[l], mem[m+1]]; \\
&\quad\quad\quad\quad\quad\ U(m+1, r, k+1) \\
&\quad\ \textbf{fiq} \\
&\quad \textbf{end} \\
&\textbf{fi}.
\end{aligned}
\tag{3.7}
$$

It is worth mentioning, in the above example, we regard an access to the QRAM as a unitary at the algorithmic level and describe it as a program. QRAM will appear later in Chapter 4 again, where it will be a low-level storage component of the quantum register machine.

3.4 Discussion

3.4.1 Related Work

Quantum control flow and data structures

Many works [22, 26, 28, 76–78, 208, 219, 220] on quantum programming languages incorporate quantum control flow as a feature. Some [78, 219] also discuss its limitations. For example, it is shown in [219] that the semantics of quantum recursion cannot be defined using Tarski's fixpoint theorem, when quantum measurements are involved. However, $\mathbf{RQC}^{++}$ only describes unitary operators and therefore circumvents this issue. A similar unitary restriction is also used in [78] to support instruction-level quantum control flow. We make two further remarks about this restriction. First, the restriction appears inevitable for quantum programs with quantum control flow, and thus it is not required for those with only classical recursion (e.g., [207]). Second, to handle measurements under the unitary restriction, one can use the standard *deferred measurement* technique to split the computation into layers of unitary operators and measurements.

Another related topic is data structures in superposition [27, 208]. The language Tower [27], for example, can describe recursive programs in superposition, which allows

a *single* layer of interleaving between quantum control flow and recursion. However, its syntax does not contain unitary gates and quantum if-statement, preventing it from expressing the most general form of quantum recursion. In contrast, **RQC**$^{++}$ allows *arbitrary*, nested interleaving between quantum if-statements and recursive procedure calls. As we will see in Chapters 4 and 5, this expressive power of **RQC**$^{++}$ also makes its implementation non-trivial.

3.4.2 Summary

In summary, this chapter has introduced the emerging paradigm of quantum recursive programming. The most general form of quantum recursion is realised by combining two programming features: recursive procedure calls and quantum control flow. It is the quantum nature of the control flow that fundamentally distinguishes quantum recursion from its classical counterpart.

We provided a detailed introduction to the quantum recursive programming language **RQC**$^{++}$ [37] and illustrated its expressive power through a series of concrete examples. These examples highlighted the intricate interplay between recursive procedure calls and quantum control flow in quantum recursive programs.

Chapter 4

Quantum Register Machine

> ... increase of efficiency always comes down to exploitation of structure ...
>
> EDSGER W. DIJKSTRA [221]

The power of quantum recursive programming relies on the intricate interplay between *quantum control flow* and *recursive procedure calls*. However, the arbitrary interleaving between these two programming features also makes the implementation of quantum recursive programs challenging. An ideal implementation must not only support these two features harmoniously but also be efficient.

To resolve this fundamental challenge, the following three chapters — forming the major contribution of Part I of this thesis — present a systematic framework for the efficient implementation of quantum recursive programs. This framework is based on a new quantum architecture called quantum register machine. This chapter begins with the motivation and a high-level overview of this framework. Then, we focus on the architecture part of the quantum register machine, covering its storage components and an instruction set **QINS** for specifying how the machine operates.

4.1 Introduction

4.1.1 Motivating Example: Quantum Multiplexor

To illustrate the insight behind our efficient implementation of quantum recursive programs, let us start with an algorithmic subroutine called quantum multiplexor [174]. We will see how quantum recursion can benefit its description and implementation. Quantum multiplexor is used in a wide range of quantum algorithms, for example, linear combination of unitaries (LCU) [211–213, 222], Hamiltonian simulation [223–226], quantum state preparation [227–230], and solving quantum linear system of equations [217]. Let $N = 2^n$ and $[N] = \{0, 1, \ldots, N-1\}$. A quantum multiplexor can be described by the unitary

$$U = \sum_{x \in [N]} |x\rangle\langle x| \otimes U_x. \tag{4.1}$$

$$
\begin{aligned}
P_{\text{main}}(n) &\Leftarrow P(n,0)\\
P(k,x) &\Leftarrow \textbf{if } k=0 \textbf{ then } Q[x]\\
&\qquad \textbf{else}\\
&\qquad\quad \textbf{qif}[q[k]]\ \ |0\rangle \to P(k-1,2x)\\
&\qquad\qquad\quad \square\ \ |1\rangle \to P(k-1,2x+1)\\
&\qquad\quad \textbf{fiq}\\
&\qquad \textbf{fi}\\
Q[0] &\Leftarrow C_0\\
&\cdots\\
Q[N-1] &\Leftarrow C_{N-1}.
\end{aligned}
$$

FIGURE 4.1: Quantum multiplexor as a quantum recursive program.

Here, every unitary U_x is described by a quantum circuit, or more generally, a quantum program, say C_x. The quantum multiplexor U applies U_x, conditioned on the state $|x\rangle$ of the first n qubits.

A straightforward implementation of U is by applying a sequential product of N controlled-U_x:

$$\prod_{x\in[N]} \Big(|x\rangle\langle x| \otimes U_x + \sum_{y\neq x} |y\rangle\langle y| \otimes \mathbb{1} \Big). \tag{4.2}$$

This implementation has time complexity $O\Big(\sum_{x\in[N]} T_x\Big)$, where T_x is the time for executing C_x (i.e., implementing U_x). On the other hand, there exists a more efficient parallel implementation [228, 229] of U, with parallel time complexity $O\Big(n + \max_{x\in[N]} T_x\Big)$ (measured by the quantum circuit depth), using rather involved constructions similar to the bucket-brigade quantum random access memories [218, 231–233]. The implementation in [228, 229] achieves exponential parallel speed-up (with respect to n) over the straightforward one. The price for obtaining such efficiency is the manual design of rather low-level quantum circuits.

It is natural to ask *if we can design at high-level and still obtain an efficient implementation*. For this example of quantum multiplexor, the intuition is as follows. First, U can be described by a high-level quantum recursive program $\mathcal{P}$, which encapsulates both the control structure in Equation (4.1) and all programs C_x for describing unitaries U_x. Then, by storing the program $\mathcal{P}$ in a quantum memory, we can design a *quantum register machine* (to be formally defined in Sections 4.2 and 4.3) that automatically exploits the structure of $\mathcal{P}$ and executes all C_x's (i.e., implements all U_x's) in quantum superposition, thereby outperforming the straightforward implementation that only sequentially executes C_x's.

Let us make the above intuition more concrete, by describing U in a quantum recursive program in the language $\mathbf{RQC}^{++}$ [37] (see Chapter 3 for an introduction) as in

Figure 4.1. Here, the main procedure $P_{\text{main}}(n)$ describes U, and every $Q[x]$ (or their procedure body C_x) describes U_x. Procedure $P(k, x)$ recursively collects the control information x using the quantum if-statement (**qif** statement) and calls $Q[x]$ when $k = 0$. At this point, we only need to note that the program in Figure 4.1 involves the interplay between the quantum control flow (managed by the **qif** statement) and recursive procedure calls. The **qif** statement in $P(k, x)$ creates two *quantum branches* (in superposition): when $q[k]$ is in state $|0\rangle$, $P(k-1, 2x)$ is called; when $q[k]$ is in state $|1\rangle$, $P(k-1, 2x+1)$ is called.

If the program in Figure 4.1 is compiled and stored into a quantum memory, then a quantum register machine that supports quantum control flow and recursive procedure calls can *run through the two quantum branches in superposition*. The cost for executing the **qif** statement only depends on the quantum branch that takes longer running time. This will incur a final time complexity proportional to the maximum $\max_{x\in[N]} T_x$ (compared to the sum $\sum_{x\in[N]} T_x$ in the straightforward implementation) and lead to an exponential parallel speed-up, similar to [228, 229].

4.1.2 Overview of Framework

Now we present an overview of our systematic framework for efficiently implementing quantum recursive programs. The high-level source language is chosen to be **RQC**$^{++}$ [37], but we expect that the techniques developed here can work for other quantum programming languages that support recursion. We propose a new architecture called quantum register machine, which is the focus of this chapter. Based on it, we describe a comprehensive implementation process of quantum recursive programs, the focus of Chapter 5. Finally, we analyse the efficiency of such implementation, which is the focus of Chapter 6. The framework is visualised in Figure 4.2.

Architecture: Quantum Register Machine

We propose a notion of quantum register machine, a quantum architecture that provides instruction-level support for quantum control flow and procedure calls at the same time. Its storage components include a constant number of quantum registers (simply called registers in the sequel) and a quantum random access memory (QRAM). The QRAM stores both compiled quantum programs and quantum data. The machine operates on registers like a classical CPU, executing the compiled program by fetching instructions from the QRAM. The machine is also accompanied by a set of low-level instructions, each specifying operations to be carried out by the machine. We briefly explain how the quantum register machine handles quantum control flow and recursive procedure calls as follows.

Handle quantum control flow: Inspired by the previous work [78] (which borrows ideas from classical reversible architectures [234–237]), we put the program counter into

a quantum register, which can be in quantum superposition. However, existing techniques are insufficient to *automatically* handle a challenge introduced by the quantum control flow, known as the *synchronisation problem* [78, 238–246]. Specifically, previous work [78] circumvents this problem by manually inserting nop (no operation) into the low-level programs. This approach changes the static program text and is not extendable to handle quantum recursive programs, because the length of dynamic computation generated by quantum recursion cannot be pre-determined from the static program text (see Section 5.2.1 and Section 5.5.1 for further discussion).

In contrast, to automatically handle the synchronisation problem (without changing the static program text), our solution is to use a partial evaluation of quantum control flow (to be explained soon) before execution, and design a few corresponding quantum registers and mechanisms to exploit the partial evaluation result at runtime.

Handle recursive procedure calls: We allocate a call stack in the QRAM. Stack operations are made reversible by borrowing techniques from the classical reversible computing (e.g., [247]). Note that at runtime, all quantum registers and the QRAM (where the dynamic call stack is stored) can be in an entangled quantum state.

It is worth pointing out that the quantum register machine does not aim to model any existing quantum hardware (typically controlled by classical pulses to implement standard quantum circuits). Indeed, quantum register machine should better be thought of as an abstract machine (that does *not* require hardware-level quantum control flow; see also [78]). Its execution is by repeatedly applying some fixed unitary operator per instruction cycle. Such unitary operator will be efficiently implemented by standard quantum circuits composed of one- and two-qubit gates.

Implementation: Compilation, Partial Evaluation and Execution

We propose a comprehensive process of implementing high-level quantum recursive programs (described in the language $\mathbf{RQC}^{++}$) on the quantum register machine. This includes the following three steps: the first two are purely classical and the last is quantum.

Step 1. Compilation: The high-level program in $\mathbf{RQC}^{++}$ is compiled into a low-level one described by instructions, together with a series of transformations. The low-level instruction set is designed such that the high-level program structure can be exploited for later execution. This step only depends on the static program text and is independent of inputs.

Step 2. Partial evaluation: Given the classical inputs (typically specifying the size of quantum inputs), the quantum control flow information of the compiled program is evaluated and stored into a data structure. In later execution, it will be loaded into the QRAM

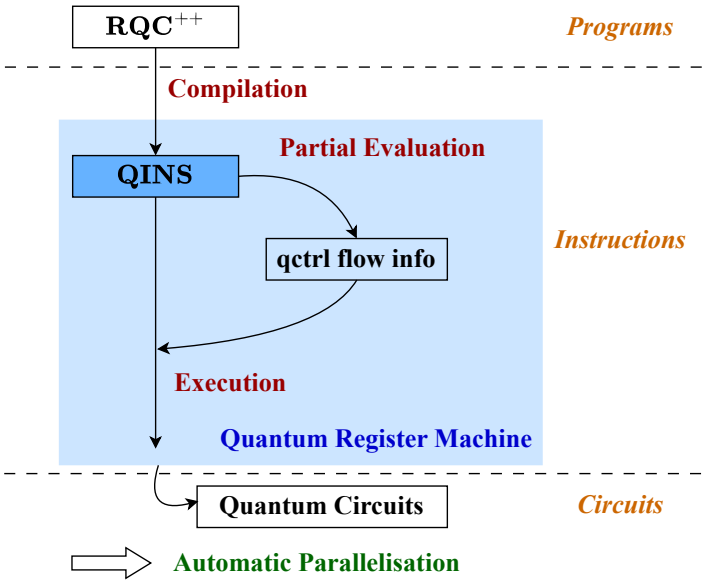


FIGURE 4.2: Overview of the framework for implementing quantum recursive programs.

to help address the aforementioned synchronisation problem. This step is independent of quantum inputs.

Step 3. Execution: With the compiled program and partial evaluation results loaded into the QRAM, the quantum inputs are finally considered, and the compiled program is executed with the aid of the partial evaluation results. The execution is done by repeatedly applying a fixed unitary (independent of the program) per cycle, which will be eventually implemented by standard quantum circuits with rigorously analysed complexity.

We will describe the above three steps in detail in Chapter 5. Then in Chapter 6, we will rigorously analyse the theoretical complexity of **Step 2** and **3**. The final parallel time complexity, measured by the standard asymptotic (classical and quantum) circuit depth, is $O\big(T_{\text{exe}}(\mathcal{P}) \cdot \big(T_{\text{reg}} + T_{\text{QRAM}}\big)\big)$. Intuitively, $T_{\text{exe}}(\mathcal{P})$ is the time for executing the longest quantum branch in program $\mathcal{P}$; and T_{reg} and T_{QRAM} are complexities for elementary operations on registers and the QRAM, independent of the program.

Bonus: Automatic Parallelisation

We show that quantum recursive programming can be *win-win* for both modularity of programs (demonstrated in [37] via various examples) and efficiency of their implementation (realised here). In particular, as a bonus, our efficient implementation also offers *automatic parallelisation*. For implementing certain quantum algorithmic subroutine, like

the quantum multiplexor in Section 4.1.1, an exponential speed-up (over the straightforward implementation) can be obtained from this automatic parallelisation, in terms of (classical and quantum) parallel time complexity. Here, the classical parallel time complexity is relevant because the partial evaluation will be performed by a classical parallel algorithm.

For implementing the quantum multiplexor, we obtain the following theorem from the automatic parallelisation, whose proof is to be shown in Chapter 6.

Theorem 1 (Automatic parallelisation of quantum multiplexor)**.** *Via the quantum register machine, the quantum multiplexor in Equation (4.1) with each U_x consisting of T_x elementary unitary gates can be implemented in (classical and quantum) parallel time complexity (i.e., circuit depth) $\widetilde{O}\left(n \cdot \max_{x\in[N]} T_x + n^2\right)$, where $\widetilde{O}(\cdot)$ hides logarithmic factors.*

Although the complexity in Theorem 1 is slightly worse than that in [228, 229] by a factor of $\widetilde{O}(n)$, it is worth stressing that the parallelisation in Theorem 1 is obtained automatically. Our framework steps towards a *top-down* design of (parallel) efficient quantum algorithms: the programmer only needs to design the high-level quantum programs (like in Figure 4.1), and the parallelisation is automatically realised by our implementation based on the quantum register machine. Further comparison of Theorem 1 and [228, 229] can be found in Section 6.7.1.

4.2 Components of Quantum Register Machine

Now we start to introduce the notion of quantum register machine, an architecture that provides instruction-level support for quantum control flow and recursive procedure calls at the same time. Unlike most existing quantum architectures that use classical controllers to implement quantum circuits, the quantum register machine stores quantum programs and data in a quantum random access memory (QRAM) and executes on quantum registers. As aforementioned in Section 4.1.2, since existing quantum hardware is typically controlled by classical pulses, it would be better to think quantum register machine as an abstract machine (that does not require hardware-level quantum control flow). Like a classical CPU, the machine works by repeatedly applying a fixed unitary U_{cyc} (independent of the program) per instruction cycle, which consists of several stages, including fetching an instruction from the QRAM, decoding it and executing it by performing corresponding operations. To support quantum control flow, additional stages related to the partial evaluation are also needed. The unitary U_{cyc} will be eventually implemented by standard quantum circuits, as described and visualised later in Section 5.3.

In this section, we first explain the storage components of the quantum register machine: quantum registers and the QRAM. We will cover the elementary operations on

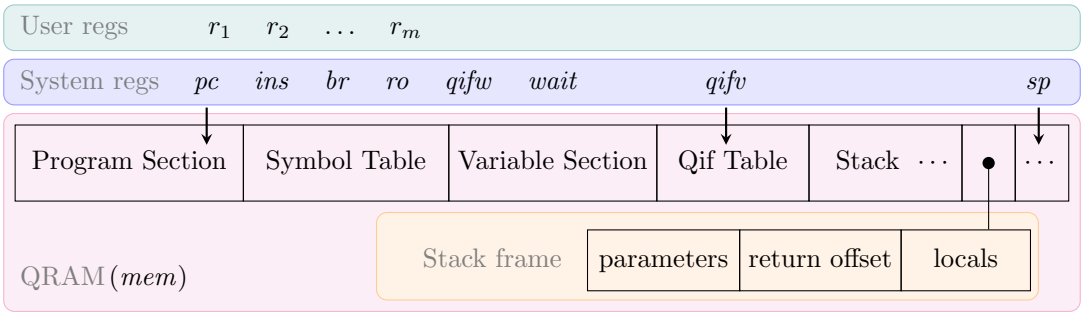


FIGURE 4.3: Storage components of the quantum register machine and the layout of the QRAM. All components can be together in a quantum superposition state.

registers and accesses to the QRAM, and how they are arranged to form an architecture, as visualised in Figure 4.3. In the next section, a low-level instruction set **QINS** (quantum instructions) for specifying how the quantum register machine operates will be presented.

4.2.1 Quantum Registers

The quantum register machine has a constant number of quantum registers (or simply, registers), each storing a quantum word composed of L_{word} (called word length) qubits. Registers are directly accessible. The machine can perform a series of elementary operations on registers, including word-level arithmetic operations (see also Section 4.3), each assumed to take time T_{reg}. The precise definition of elementary operations will be presented in Section 4.2.2 later.

Registers are grouped into two types: system and user registers. There are eight system registers. The first five are rather standard and borrowed from the classical reversible architectures [234–237], as quantum unitaries are intrinsically reversible. We describe their classical effects as follows.

- Program counter *pc* records the address of the current instruction.
- Instruction *ins* records the current instruction.
- Branching offset *br* records the offset of the address of the next instruction to go from *pc*. More specifically, if $br = 0$, then the address of the next instruction will be $pc + 1$. Otherwise, the address of the next instruction will be $pc + br$.
- Return offset *ro* records the offset for *br* in the return of a procedure call.
- Stack pointer *sp* records the current topmost location of the call stack.

In contrast, the last three system registers are novelly introduced to support an efficient implementation of the **qif** statements. They are related to the *qif table*, a data structure

generated by the partial evaluation of quantum control flow and used during execution to address the aforementioned synchronisation problem. We briefly describe their classical effects as follows, and will explain further details in Sections 5.2 and 5.3.

- Qif table pointer *qifv* records the current node in the qif table.
- Qif wait counter *qifw* records the number of instruction cycles to wait at the current node in the qif table.
- Qif wait flag *wait* records whether the current instruction cycle needs to be skipped.

We also set the initial values of these registers: *pc*, *sp* and *qifv* are initialised to $|j\rangle$, where j are the starting addresses of the main program, the call stack and the qif table, respectively. Other system and user registers are initialised to $|0\rangle$.

4.2.2 Elementary Operations on Registers

In Section 4.2.1, we have briefly mentioned that the quantum register machine can perform a series of elementary operations on registers, each taking time T_{reg}. Now let us make these elementary operations more specific as follows.

Definition 3 (Elementary operations on registers)**.** The quantum register machine can perform the following elementary operations on registers.

- (Reversible elementary arithmetic): For binary operator $* \in \{\oplus, +, -\}$, let

$$U_*(r_1, r_2) := \sum_{x,y} |x * y\rangle\langle x|_{r_1} \otimes |y\rangle\langle y|_{r_2}$$

  for distinct registers $r_1 \neq r_2$. Here, $\oplus$ denotes the XOR operator. Also, let $U_{\text{neg}}(r) := \sum_x |-x\rangle\langle x|_r$ for register r.

- (Reversible versions of possibly irreversible arithmetic): Let $\mathcal{OP}$ be a fixed set of concerned unary and binary arithmetic operators with $|\mathcal{OP}| = O(1)$. For unary operator $op \in \mathcal{OP}$, let $U_{op}(r_1, r_2) := \sum_{x,y} |x \oplus (op\ y)\rangle\langle x|_{r_1} \otimes |y\rangle\langle y|_{r_2}$ for distinct registers r_1, r_2.

  For binary operator $op \in \mathcal{OP}$, let $U_{op}(r_1, r_2, r_3) := \sum_{x,y,z} |x \oplus (y\ op\ z)\rangle\langle x|_{r_1} \otimes |y\rangle\langle y|_{r_2} \otimes |z\rangle\langle z|_{r_3}$ for distinct registers r_1, r_2, r_3.

- (Swap): Let the swap unitary $U_{\text{swap}}(r_1, r_2) := \sum_{x,y} |y\rangle\langle x|_{r_1} \otimes |x\rangle\langle y|_{r_2}$ for distinct registers r_1, r_2.

- (Unitary gate): For unitary gate specified by $G \in \mathcal{G}$ (see also Section 3.2.2), let $U_G(r)$ be its corresponding unitary applied on register r.

- (Controlled operations): For a register r and an elementary operation (unitary) U not acting on r, let the controlled versions of U be

$$\bullet(r)\text{-}U := \sum_{x\neq 0} |x\rangle\langle x|_r \otimes U + |0\rangle\langle 0| \otimes \mathbb{1},$$
$$\circ(r)\text{-}U := \sum_{x\neq 0} |x\rangle\langle x|_r \otimes \mathbb{1} + |0\rangle\langle 0| \otimes U.$$

Suppose that performing any of the above elementary operations takes time T_{reg}.

Elementary quantum operations in Definition 3 will eventually be implemented at a lower level by standard quantum circuits composed of one- and two-qubit gates. Further analysis will be shown in Section 6.4.

4.2.3 Quantum Random Access Memory

The quantum register machine has a quantum random access memory (QRAM)[1] composed of N_{QRAM} memory locations, each storing a quantum word. The QRAM is not directly accessible. Like a classical memory, access to QRAM is by providing an address register specifying the address, and a target register to hold the information retrieved from the specified location. Unlike the classical case, the address register can be in quantum superposition, and registers can be entangled with the QRAM. The machine can perform two types of elementary QRAM accesses, each assumed to take time T_{QRAM}. The precise definition of elementary QRAM accesses will be presented in Section 4.2.4 later.

Layout of the QRAM

The QRAM in the quantum register machine stores both programs and data. In particular, it contains the following sections.

1. Program section stores the compiled program in a low-level language **QINS** (to be defined in Section 4.3).

2. Symbol table section stores the name of every variable and its corresponding address. Here, unlike in the classical case, the symbol table is used at runtime instead of compile time (see also Section 5.1.3), because arrays in **RQC**$^{++}$ are not declared with fixed size.

3. Variable section stores the classical and quantum variables.

[1] In particular, the QRAM considered here is quantum random access quantum memory (QRAQM). The reader is referred to [248] for a review of QRAM.

4. Qif table section stores the qif table (to be defined in Section 5.2.2).

5. Stack section stores the call stack to handle the procedure calls. The stack is composed of multiple stack frames, each storing the actual parameters and return offset (from the caller to the callee), and the local data used by the callee, in a procedure call.

The layout of the QRAM was previously visualised in Figure 4.3.

4.2.4 Elementary QRAM Accesses

In Section 4.2.3, we have briefly mentioned the quantum register machine can perform two types of elementary QRAM accesses, each taking time T_{QRAM}. Now let us make them more specific.

Definition 4 (Elementary QRAM accesses)**.** The quantum register machine can perform the following two types of elementary QRAM accesses, commonly adopted in the literature [248].

- QRAM (swap) load. This access performs the unitary $U_{\text{ld}}(r, a, mem)$ defined by the mapping:

$$|x\rangle_r |i\rangle_a |M\rangle_{mem} \mapsto |M_i\rangle_r |i\rangle_a |M_0, \ldots, M_{i-1}, x, M_{i+1}, \ldots, M_{N_{\text{QRAM}}-1}\rangle_{mem}, \tag{4.3}$$

  for all x, i and $M = (M_0, \ldots, M_{N_{\text{QRAM}}-1})$. Here, r is the target register, a is the address register, and mem is the QRAM.

- QRAM (xor) fetch. This access performs the unitary $U_{\text{fet}}(r, a, mem)$ defined by the mapping:

$$|x\rangle_r |i\rangle_a |M\rangle_{mem} \mapsto |x \oplus M_i\rangle_r |i\rangle_a |M\rangle_{mem}, \tag{4.4}$$

  for all x, i, M.

Moreover, the controlled versions (controlled by a register) of elementary QRAM accesses are also considered elementary, since the number of registers is constant and the control only incurs a constant overhead. Suppose every elementary QRAM access takes time T_{QRAM}.

Remarks on the QRAM

Some readers might notice that the physical realisation of QRAM is not yet near-term, a challenge shared by most works leveraging QRAM (e.g., [27, 249–251]). Nevertheless, there are ongoing efforts towards feasible QRAM implementations (e.g., [232, 233, 252]). Importantly, the final complexity of our implementation of quantum recursive programs

	Instruction	Classical Effect
Load	`ld(r,i)`	$r \leftrightarrow M_i$
	`ldr(r1,r2)`	$r_1 \leftrightarrow M_{r_2}$
	`fetr(r1,r2)`	$r_1 \leftarrow r_1 \oplus M_{r_2}$
Gate	`uni(G,r1)`	Apply gate G on r_1
	`unib(G,r1,r2)`	Apply gate G on $r_1 r_2$
Arith	`xori(r,i)`	$r \leftarrow r \oplus i$
	`addi(r,i)`	$r \leftarrow r + i$
	`subi(r,i)`	$r \leftarrow r - i$
	`swap(r1,r2)`	$r_1 \leftrightarrow r_2$
	`add(r1,r2)`	$r_1 \leftarrow r_1 + r_2$
	`sub(r2,r2)`	$r_1 \leftarrow r_1 - r_2$
	`neg(r1)`	$r_1 \leftarrow -r_1$
	`ari(op,r1,r2)`	$r_1 \leftarrow r_1 \oplus (op\ r_2)$
	`arib(op,r1,r2,r3)`	$r_1 \leftarrow r_1 \oplus (r_2\ op\ r_3)$
Branch	`bra(i)`	$br \leftarrow br \oplus i$
	`bez(r,i)`	$br \leftarrow br \oplus (i \cdot [r = 0])$
	`bnz(r,i)`	$br \leftarrow br \oplus (i \cdot [r \neq 0])$
	`swbr(r)`	$br \leftrightarrow r$
Qif	`qif(r)`	update *qifv* and *qifw*
	`fiq(r)`	update *qifv*
Special	`start`	none
	`finish`	none

(A) Instructions and corresponding classical effects. Here, $\oplus$ denotes the XOR operator; $[b] = 1$ if b is true and $[b] = 0$ otherwise.

Type	Format (ins)				
I	*opcode*	*reg*	*imm*		
	c	r	i		
R	*opcode*	reg_1	reg_2		
	c	r_1	r_2	0	0
O	*opcode*	*para*	reg_1	reg_2	reg_3
	c	G/op	r_1	r_2	r_3

$$U_{\text{dec}} = \sum_c |c\rangle\langle c| \otimes \underbrace{\sum_d |d\rangle\langle d| \otimes U_{c,d}}_{U_c}$$

Type	c	$U_{c,d}$
I	`ld`	$U_{\text{ld}}(r, imm, mem)$
	`xori`	$U_{\oplus}(r, imm)$
	`bnz`	$\circ(r)$-$U_{\oplus}(br, imm)$
R	`ldr`	$U_{\text{ld}}(r_1, r_2, mem)$
	`swap`	$U_{\text{swap}}(r_1, r_2)$
	`qif`	$U_{\text{qif}}(r_1)$
O	`uni`	$U_G(r_1)$
	`ari`	$U_{op}(r_1, r_2)$

(B) Instruction formats, the decoding unitary U_{dec}, and selected examples of instruction implementations.

FIGURE 4.4: The low-level language **QINS** and selected examples.

is measured by the standard circuit depth and unaffected by the near-term feasibility of QRAM.

It is also worth pointing out that managing entanglement between registers and the QRAM is crucial, as improper handling can result in incorrect output states [114]. To this end, the instruction set **QINS** (Section 4.3) and the compilation process (Section 5.1) are carefully designed to ensure that, after the execution, quantum variables are *disentangled* from other registers and the remaining part of the QRAM. A key of the design is *proper uncomputation* of intermediate results. The idea traces back to the classical techniques of reversible computing [253, 254], and has been applied in [78, 234–237]. Moreover, in our design, the creation and removal of entanglement during the execution align with the program structure (in **RQC**$^{++}$).

4.3 Instruction Set QINS

We have specified the components of the quantum register machine. Now let us proceed to specify how it operates. In this section, we present **QINS**, an instruction set for describing the compiled programs. Each instruction specifies a series of elementary operations to be carried out by the quantum register machine. There are 22 instructions in **QINS**,

which are listed with their classical effects in Figure 4.4a. Here, we leave the explanation of instructions `qif` and `fiq` to Section 5.3. The classical effects of other instructions are lifted to quantum in the standard way when being executed by the quantum register machine.

The design of **QINS** is inspired by the existing classical reversible instruction sets [234–237]. Nevertheless, several instructions in **QINS** are essentially new. The most important are instructions `qif` and `fiq`, which are designed for a *structured management* of quantum control flow (generated by the **qif** statements in **RQC**$^{++}$), in particular, aiding later partial evaluation and execution. Instructions `uni` and `unib` are designed for quantum unitary gates.

We group the instructions into three types: I (immediate-type), R (register-type) and O (other-type), according to their formats, as shown in Figure 4.4b. During the execution (to be described in Section 5.3), we decode the instruction in register *ins* by performing a unitary U_{dec} (see Figure 4.4b). U_{dec} is a quantum multiplexor, with section *opcode* as its first part of control, and other sections (depending on the type I/R/O) in *ins* as its second part of control. Let c be a computational basis in the first part and d in the second part, then the unitary being controlled is denoted by $U_{c,d}$.

For illustration, selected instructions and corresponding $U_{c,d}$ are presented in Figure 4.4b. Here, U_{ld} is the QRAM load access in Definition 4. U_{qif} (and similarly U_{fiq} for `fiq`) will be defined in Section 5.3. Other unitaries are elementary operations on registers (see Definition 3): (a) $U_{\oplus}$ performs the mapping $|x\rangle\,|y\rangle \mapsto |x \oplus y\rangle\,|y\rangle$; (b) $\circ(r)$-U stands for the controlled version $|0\rangle\langle 0|_r \otimes U + \sum_{x\neq 0} |x\rangle\langle x|_r \otimes \mathbb{1}$ of unitary U; (c) U_{swap} performs the mapping $|x\rangle\,|y\rangle \mapsto |y\rangle\,|x\rangle$; (d) U_G applies the elementary gate G (chosen from a fixed set $\mathcal{G}$ of size $O(1)$); (e) U_{op} performs the mapping $|x\rangle\,|y\rangle \mapsto |x \oplus (op\ y)\rangle\,|y\rangle$ for unary operator *op* (chosen from a fixed set $\mathcal{OP}$ of size $O(1)$).

Let us further explain the classical effects of instructions in Figure 4.4a.

- Load instructions are used to retrieve information (including instructions and data) from the QRAM. In particular, `ld(r,i)` swaps the value M_i at address i specified by the immediate number and the value in register r. `ldr(`$\mathtt{r}_1$`,`$\mathtt{r}_2$`)` works similarly with the address specified by the value in register r_2. `fetr(`$\mathtt{r}_1$`,`$\mathtt{r}_2$`)` instead copies (via the XOR operator $\oplus$) the value M_i at address i specified by the immediate number into register r.

- Unitary gate instructions are used to apply elementary quantum gates (chosen from the fixed set $\mathcal{G}$, specified by **RQC**$^{++}$ in Section 3.2.2) on registers. In particular, `uni(G,`$\mathtt{r}_1$`)` applies the single-qubit unitary gate specified by G on register r_1. `unib(G,`$\mathtt{r}_1$`,`$\mathtt{r}_2$`)` applies the two-qubit unitary gate specified by G on registers r_1 and r_2.

- Arithmetic instructions are used to perform word-level elementary arithmetic operations, including reversible and (possibly) irreversible ones, as previously introduced in Definition 3.

  Reversible arithmetic includes XOR (`xori` and `xor`), addition (`addi` and `add`), subtraction (`subi` and `sub`), negation (`neg`), and swap (`swap`). These arithmetic operations can also take the immediate number i as one of the inputs.

  A constant number of flexible choices of irreversible arithmetic operations are also allowed. Specifically, `ari(op,r1,r2)` copies (by the XOR operator $\oplus$) the result of $(op\ r_2)$ into register r_1, where op specifies an unary arithmetic operation. Instruction `arib(op,r1,r2,r3)` works similarly with the op specifying a binary arithmetic operation. In both cases, we assume $op \in \mathcal{OP}$ for a fixed set $\mathcal{OP}$ of $O(1)$ size (see also Definition 3). Here, the irreversible arithmetic operations are actually implemented reversibly by introducing garbage data.

- Branch instructions are used to realise transfer of control flow. In particular, `bra(i)` copies (by the XOR operator $\oplus$) the branch offset i specified by the immediate number i into register br. `bez(r,i)` and `bnz(r,i)` are conditional versions of `bra`: the former performs `bra` when the value in register r is zero; while the latter works in the non-zero case. `swbr(r)` swaps the values in register r and register br.

- Qif instructions are used to update the registers $qifv$ and $qifw$. Their effects will be described in Section 5.3.

- Special instructions are used to specify the start and finishing points of the main program.

Now we provide more details of implementing instructions in **QINS**. To execute an instruction stored in register ins, the quantum register machine performs a unitary

$$U_{\text{dec}} = \sum_c |c\rangle\langle c| \otimes \sum_d |d\rangle\langle d| \otimes U_{c,d},$$

which is previously defined in Figure 4.4b. Here, c ranges over possible values of the section *opcode* (see Figure 4.4b) of ins, which corresponds to the name of the instruction in ins. Depending on the type (I/R/O) of the instruction (indicated by c; see Figure 4.4b), d ranges over possible values of other sections in ins:

- If the instruction is of type I, then d ranges over possible values of the section reg in ins.

- If the instruction is of type R, then d ranges over possible values of the sections reg_1 and reg_2 in ins.

Type	c	$U_{c,d}$
I	`ld`	$U_{\text{ld}}(r, imm, mem)$
	`xori`	$U_{\oplus}(r, imm)$
	`addi`	$U_{+}(r, imm)$
	`subi`	$U_{-}(r, imm)$
	`bra`	$U_{\oplus}(br, imm)$
	`bez`	$\bullet(r)$-$U_{\oplus}(br, imm)$
	`bnz`	$\circ(r)$-$U_{\oplus}(br, imm)$
R	`ldr`	$U_{\text{ld}}(r_1, r_2, mem)$
	`fetr`	$U_{\text{fet}}(r_1, r_2, mem)$
	`swap`	$U_{\text{swap}}(r_1, r_2)$
	`add`	$U_{+}(r_1, r_2)$
	`sub`	$U_{-}(r_1, r_2)$
	`neg`	$U_{\text{neg}}(r_1)$
	`swbr`	$U_{\text{swap}}(br, r_1)$
	`qif`	$U_{\text{qif}}(r_1)$
	`fiq`	$U_{\text{fiq}}(r_1)$
	`start`	$\mathbb{1}$
	`finish`	$\mathbb{1}$
O	`uni`	$U_G(r_1)$
	`unib`	$U_G(r_1, r_2)$
	`ari`	$U_{op}(r_1, r_2)$
	`arib`	$U_{op}(r_1, r_2, r_3)$

FIGURE 4.5: Full list of unitaries $U_{c,d}$ for implementing instructions in **QINS**.

- If the instruction is of type O, then d ranges over possible values of the sections *para*, reg_1, reg_2 and reg_3 in *ins*.

In Figure 4.5, we present the full list of unitaries $U_{c,d}$ for implementing instructions in **QINS**. Most unitaries on the third column were defined in Definitions 3 and 4. The unitaries U_{qif} and U_{fiq} for implementing instructions `qif` and `fiq` will be defined in Section 5.3.

4.4 Discussion

4.4.1 Related Work

Low-level quantum instructions

Several quantum instruction set architectures have been proposed in the literature, e.g., OpenQASM [255], Quil [256], eQASM [257]. Among these, only the quantum control machine [37] supports a program counter in superposition and, consequently, supports

quantum control flow at the instruction level. However, while the quantum control machine also supports conditional jumps, it does not support arbitrary procedure calls that are native to our quantum register machine.

Classical reversible languages

There is also extensive research in classical reversible programming languages, including the high-level language Janus [258–260], low-level instruction set architectures PISA [234–236] and BobISA [237]. Some of these reversible languages support local variables, specified by a pair of local-delocal statements, which have explicitly reversible semantics. In contrast, while our source language **RQC**$^{++}$ allows for irreversible classical computation on local variables, our framework ensures the compiled programs expressed as instructions in **QINS** become reversible.

4.4.2 Summary

In this chapter, we provided an overview of our systematic framework for implementing quantum recursive programs written in **RQC**$^{++}$. The foundation of this framework is a new quantum architecture, the quantum register machine, which simultaneously supports quantum control flow and recursive procedure calls at the instruction level.

The quantum register machine features quantum registers and a QRAM that stores both compiled programs and data. It supports quantum control flow by putting the program counter into a quantum register and designing additional mechanisms to automatically handle the synchronisation problem (to be explained in Chapter 5). It supports recursive procedure calls by allocating a call stack in the QRAM. To specify how the quantum register machine operates, we designed an associated instruction set **QINS**. The machine operates like a classical CPU by repeatedly applying a fixed unitary per instruction cycle, which performs fetching, decoding and executing of an instruction from the QRAM. The detailed implementation will be presented in the next chapter.

Chapter 5

Implementation of Quantum Recursive Programs

In this chapter, we describe a comprehensive process for implementing quantum recursive programs, based on the architecture *quantum register machine* established in the last chapter. The implementation includes three steps: *compilation*, *partial evaluation*, and *execution*. We start with the compilation of high-level quantum recursive programs in **RQC**$^{++}$ [37] to low-level instructions in **QINS**. Next, to address the synchronisation problem mentioned in the last chapter, we proceed to the partial evaluation of quantum control flow on the compiled program. Finally, we describe how the quantum register machine executes the compiled program, with the aid of the partial evaluation results.

5.1 Compilation

As usual, the first step in the implementation of quantum recursive programs is their compilation. The compilation of a program $\mathcal{P}$ in **RQC**$^{++}$ is independent of the classical and quantum inputs. It consists of the following passes.

1. First, a series of high-level transformations are performed on the original $\mathcal{P}$ to obtain $\mathcal{P}_h$, which simplify the program structure and make further compilation easier.

2. Then, the transformed program $\mathcal{P}_h$ is translated into an intermediate program $\mathcal{P}_m$ in the mid-level language composed of instructions similar to those in **QINS** but more flexible.

3. Finally, the mid-level $\mathcal{P}_m$ is translated into a program $\mathcal{P}_l$ in the low-level language **QINS**.

The compilation process is visualised in Figure 5.1. The remainder of this section is devoted to describing these passes carefully. In the sequel, we always assume the source program $\mathcal{P}$ to be compiled satisfies Conditions 1 to 3 in Section 3.2.3 and *has well-defined semantics*. We will not bother checking the syntax and semantics of $\mathcal{P}$.

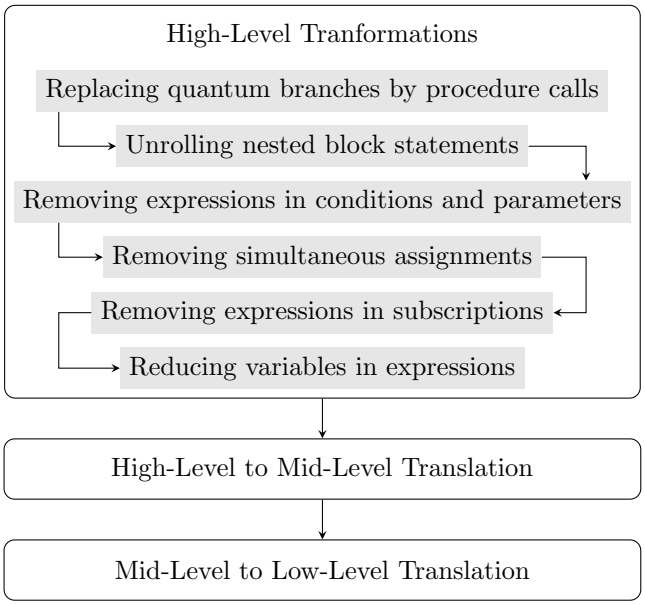


FIGURE 5.1: The compilation process from **RQC**$^{++}$ to **QINS**.

5.1.1 High-Level Transformations

In this section, we describe the first pass of high-level transformation from $\mathcal{P}$ to $\mathcal{P}_h$. The major target of this pass is to simplify the automatic uncomputation of classical variables in later passes.

A program $\mathcal{P}$ in **RQC**$^{++}$ may contain irreversible classical statements, e.g., assignment $x := 1$. Reversibly implementing these statements introduces garbage data, e.g., through the standard Landauer [253] and Bennett [254] methods. For the overall correctness of the quantum computation, these garbage data should be properly uncomputed. Moreover, the block statement in **RQC**$^{++}$ explicitly requires uncomputation of local variables at the end of the block.

In the execution of a program, when should we perform uncomputation? First, we observe a difficulty from the uncomputation of local variables in nested block statements. Consider the example in Figure 5.2. The inner block modifies x, which is used by the outer block. If one tries to uncompute the local variable y at the end of the inner block, the change on x (by the inner block) is also uncomputed, which is an undesirable side effect.

To overcome this difficulty, we will perform a series of transformations on the source program $\mathcal{P}$, such that the transformed $\mathcal{P}_h$ no longer contains nested block statements. Along the way, we also simplify the structure of the program. Consequently, for $\mathcal{P}_h$, we only need to perform uncomputation at the end of every procedure body (of procedure declarations), which will be automatically done in the high-level to mid-level translation (in Section 5.1.2).

begin local $x := 1$;
 begin local $y := 2x$;
  $x := y + 1$;
  $y := 2x$
 end;
 $U[q[x + 2]]$
end

FIGURE 5.2: An example of nested block statement.

An overview of high-level transformations was previously shown in Figure 5.1. Now we delineate every step in the high-level transformations, as well as the simplified program syntax after each step. Note that the last four steps are rather simple and standard (see e.g., the textbook [3]), but we describe them in our context (adapted to support some features of the language **RQC**$^{++}$) for completeness.

Step 1: Replacing Quantum Branches by Procedure Calls

The first step is to replace the programs used in every quantum branch of the **qif** statements by procedure calls, which can simplify later compilation of **qif** statements. More specifically, for every procedure declaration $Q(\overline{u}) \Leftarrow C \in \mathcal{P}$ and every **qif** statement appearing within, if its two branches C_0, C_1 are not procedure identifiers or **skip** statements, then we perform the replacement:

$$\textbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\textbf{fiq} \quad \Rightarrow \quad \textbf{qif}[q](|0\rangle \to P_0(\overline{u}))\square(|1\rangle \to P_1(\overline{u}))\textbf{fiq},$$

where P_0, P_1 are fresh procedure identifiers, and we add new procedure declarations $P_i(\overline{u}) \Leftarrow C_i$ (for $i \in \{0, 1\}$) to $\mathcal{P}$. If only one of C_0, C_1 is procedure identifier or **skip**, then we only perform the replacement for the other branch. Note that Conditions 1 to 3 are kept after this step. In particular, Condition 2 promises that $fcv(C_0, \sigma) = fcv(C_1, \sigma) = \emptyset$ for any classical state σ, which therefore implies the newly introduced procedure declarations $P_i \Leftarrow C_i$ satisfy Condition 3.

After this step, the program $\mathcal{P} = \{P(\overline{u}) \Leftarrow C\}_P$ has the following simplified syntax:

$$\begin{aligned} C ::= {} & \textbf{skip} \mid \overline{x} := \overline{t} \mid U[\overline{q}] \mid C_0; C_1 \mid P(\overline{t}) \mid \textbf{if } b \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi} \mid \textbf{while } b \textbf{ do } C \textbf{ od} \\ & \mid \textbf{begin local } \overline{x} := \overline{t}; C \textbf{ end} \mid \textbf{qif}[q](|0\rangle \to P_0)\square(|1\rangle \to P_1)\textbf{fiq}. \end{aligned} \tag{5.1}$$

Step 2: Unrolling Nested Block Statements

The second step is to unroll all nested block statements, which helps circumvent the obstacle for uncomputation of local variables (as illustrated in Figure 5.2). After this step,

we are promised that the program no longer contains block statements, but when the program is implemented, we need to perform uncomputation of classical variables at the end of every procedure body for the program to preserve its original semantics. To this end, for every procedure declaration $P(\overline{u}) \Leftarrow C' \in \mathcal{P}$ and every block statement B appearing in C', we perform the replacement:

$$B \equiv \textbf{begin local } \overline{x} := \overline{t};\ C \textbf{ end} \qquad \Rightarrow \qquad \overline{x'} := \overline{t}; C\left[x'/x\right],$$

where $\overline{x'}$ is a list of fresh variables, and $[x'/x]$ stands for replacing variable x by x'. Moreover, we need to append $\overline{x'} := \overline{0}$ at the beginning of C'.

After this step, the program $\mathcal{P} = \{P(\overline{u}) \Leftarrow C\}_P$ has the following simplified syntax:

$$\begin{aligned} C ::= \textbf{skip} \mid \overline{x} := \overline{t} \mid U[\overline{q}] \mid C_0; C_1 \mid P(\overline{t}) \mid \textbf{if } b \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi} \mid \textbf{while } b \textbf{ do } C \textbf{ od} \\ \mid \textbf{qif}[q](|0\rangle \to P_0)\square(|1\rangle \to P_1)\textbf{fiq}. \end{aligned} \tag{5.2}$$

It is worth mentioning that the program is technically in some new language with the same syntax as $\mathbf{RQC}^{++}$, but whose semantics requires the uncomputation of classical variables at the end of every procedure body. The uncomputation is to be automatically done in the high-to-mid-level translation (see Section 5.1.2).

Step 3: Removing Expressions in Conditions and Parameters

The third step is to remove the expression b in every **if** statement and every **while** loop, and the list of expressions $\overline{t}$ in every procedure call $P(\overline{t})$. After this step, classical expressions only appear in the assignment statements $\overline{x} := \overline{t}$ and the subscriptions of variables and procedure identifiers.

- For every **if** statement, we perform the replacement:

  $$\textbf{if } b \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi} \qquad \Rightarrow \qquad x := b; \textbf{if } x \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi}$$

  where x is a fresh boolean variable.

- For every **while** statement, we perform the replacement:

  $$\textbf{while } b \textbf{ do } C \textbf{ od} \qquad \Rightarrow \qquad x := b; \textbf{while } x \textbf{ do } C; x := b \textbf{ od}$$

  where x is a fresh boolean variable.

- For every procedure call $P(\overline{t})$, we perform the replacement:

  $$P(\overline{t}) \qquad \Rightarrow \qquad \overline{x} := \overline{t}; P(\overline{x}),$$

where $\overline{x}$ is a list of fresh variables.

After this step, the program $\mathcal{P} = \{P(\overline{u}) \Leftarrow C\}_P$ has the following simplified syntax:

$$C ::= \textbf{skip} \mid \overline{x} := \overline{t} \mid U[\overline{q}] \mid C_0; C_1 \mid P(\overline{x}) \mid \textbf{if } x \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi} \mid \textbf{while } x \textbf{ do } C \textbf{ od} \\ \mid \textbf{qif}[q](|0\rangle \to P_0)\square(|1\rangle \to P_1)\textbf{fiq}.$$

Step 4: Removing Simultaneous Assignments

The fourth step is to remove every simultaneous assignment $\overline{x} := \overline{t}$ by converting it to a series of unary assignments $x := t$. To do this, for every $\overline{x} := \overline{t}$ with $\overline{x} \equiv x_1, x_2, \ldots, x_n$ and $\overline{t} \equiv t_1, t_2, \ldots, t_n$, we perform the replacement

$$\overline{x} := \overline{t} \qquad \Rightarrow \qquad \begin{array}{l} x_1' := t_1; \ldots; x_n' := t_n; \\ x_1 := x_1'; \ldots; x_n := x_n'; \end{array}$$

where $x_1', \ldots, x_n'$ are fresh variables.

After this step, the program $\mathcal{P} = \{P(\overline{u}) \Leftarrow C\}_P$ has the following simplified syntax:

$$C ::= \textbf{skip} \mid x := t \mid U[\overline{q}] \mid C_0; C_1 \mid P(\overline{x}) \mid \textbf{if } x \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi} \mid \textbf{while } x \textbf{ do } C \textbf{ od} \\ \mid \textbf{qif}[q](|0\rangle \to P_0)\square(|1\rangle \to P_1)\textbf{fiq}.$$

Step 5: Removing Expressions in Subscriptions

The fifth step is to remove expressions in the subscriptions of subscripted variables. Note that subscripted quantum variables only appear in the $U[\overline{q}]$ statements and the $\textbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\textbf{fiq}$ statements, and subscripted procedure identifiers only appear in the procedure calls $P(\overline{x})$. By the previous steps of high-level transformations, subscripted classical variables only appear in the assignments $x := t$ and the subscriptions of quantum variables and procedure identifiers.

1. For every unitary gate statement $U[\overline{q}]$, suppose that subscripted quantum variables appearing in it are $q_1[e_1], \ldots, q_n[e_n]$ (by our assumption in Section 3.2.2, $n \leq 2$), where $q_1, \ldots, q_n$ are quantum arrays and $e_1, \ldots, e_n$ are expressions. We perform the replacement:

$$U[\overline{q}] \qquad \Rightarrow \qquad \begin{array}{l} x_1 := e_1; \ldots; x_n := x_n; \\ (U[q])[q_1[x_1]/q_1[e_1], \ldots, q_n[x_n]/q_n[e_n]], \end{array}$$

where $x_1, \ldots, x_n$ are fresh variables, and $(U[\overline{q}])[q_1[x_1]/q_1[e_1], \ldots, q_n[x_n]/q_n[e_n]]$ represents replacing every expression e_i in the subscription of q_i by variable x_i in the statement $U[\overline{q}]$.

2. For every $\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq}$ statement with $q = q'[e]$ for some quantum array q' and expression e, we perform the replacement:

$$\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq} \quad \Rightarrow \quad x := e; \mathbf{qif}[q'[x]](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq},$$

   where x is a fresh variable.

3. For every procedure call $P(\overline{x})$ (note that at this point, actual parameters in procedure calls are variables) with $P = Q[t]$ for some procedure array Q and expression t, we perform the replacement:

$$P(\overline{x}) \quad \Rightarrow \quad y := t; Q[y](\overline{x}),$$

   where y is a fresh variable.

4. For every classical assignment $x := t$, suppose that subscripted classical variables appearing in it are $y_1[e_1], y_2[e_2], \ldots, y_n[e_n]$, where $y_1, \ldots, y_n$ are classical arrays and $e_1, \ldots, e_n$ are expressions. We perform the replacement:

$$x := t \quad \Rightarrow \quad \begin{array}{l} z_1 := e_1; \ldots; z_n := e_n; \\ (x := t)[y_1[x_1]/y_1[e_1], \ldots, y_n[x_n]/y_n[e_n]], \end{array}$$

   where $z_1, \ldots, z_n$ are fresh variables, and $(x := t)[y_1[z_1]/y_1[e_1], \ldots, y_n[z_n]/y_n[e_n]]$ represents replacing every expression e_i in the subscription of y_i by variable z_i in the assignment $x := t$. Note that here $e_1, \ldots, e_n$ no more contain subscripted classical variables because of our assumption in Section 3.2.1.

After this step, every subscription in the program will be a classical variable.

Step 6: Reducing Variables in Expressions

The sixth step is to replace every assignments $x := t$ with t involving multiple variables by a series of equivalent assignments

$$x_1 := t_1; \ \ldots \ x_n := t_n; \ x := t_{n+1},$$

such that every expression t_i contains at most two variables; that is, either $t_i \equiv a \ op \ b$, where a and b are variables or constants, and $op \in \mathcal{OP}$ (see also Definition 3) is an elementary binary operator; or $t_i \equiv op \ a$, where a is a variable or constant, and $op \in \mathcal{OP}$ is an elementary unary operator. We will not bother describing the details of this standard conversion.

This step is the last step of the high-level transformations. The transformed program is denoted by $\mathcal{P}_h = \{P(\overline{u}) \Leftarrow C\}_P$, and has the following simplified syntax:

$$C ::= \textbf{skip} \mid x := t \mid U[\overline{q}] \mid C_0; C_1 \mid P(\overline{x}) \mid \textbf{if } x \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi} \mid \textbf{while } x \textbf{ do } C \textbf{ od}$$
$$\mid \textbf{qif}[q](|0\rangle \rightarrow P_0)\square(|1\rangle \rightarrow P_1)\textbf{fiq}.$$

Moreover, every subscripted variable and procedure identifier has a basic classical variable as its subscription (e.g., $z[x], q[y], P[w]$), and every expression has the form $t \equiv op\ x$ or $t \equiv x\ op\ y$.

Summary: After the High-Level Transformations

We observe that the program $\mathcal{P}_h = \{P(\overline{u}) \Leftarrow C\}_P$ after all the high-level transformations in Figure 5.1 has the following simplified syntax:

$$C ::= \textbf{skip} \mid x := t \mid U[\overline{q}] \mid C_0; C_1 \mid P(\overline{x}) \mid \textbf{if } x \textbf{ then } C_0 \textbf{ else } C_1 \textbf{ fi} \mid \textbf{while } x \textbf{ do } C \textbf{ od}$$
$$\mid \textbf{qif}[q](|0\rangle \rightarrow P_0)\square(|1\rangle \rightarrow P_1)\textbf{fiq},$$

where every subscripted variable and procedure identifier has a basic classical variable as its subscription (e.g., $x[y]$), and every expression has the form $t \equiv op\ x$ or $t \equiv x\ op\ y$. As aforementioned, the semantics is slightly changed: uncomputation of classical variables are needed at the end of every procedure body in the implementation, (which will be automatically done in the high-level to mid-level translation in Section 5.1.2).

For illustration, on the LHS of Figure 5.3, we show an after-the-high-level-transformations version of the quantum multiplexor program in Figure 4.1.

5.1.2 High-Level to Mid-Level Translation

Now we translate the transformed high-level program $\mathcal{P}_h$ obtained in the previous subsection into $\mathcal{P}_m$ in a mid-level language, which is different from the low-level language **QINS** (defined in Section 4.3) in the following aspects:

- We do not consider the memory allocation. Thus, instructions `ld`, `ldr`, and `fetr` are not needed at this stage.
- Beyond registers and numbers, instructions can also take variables and labels as input. Here, like in the classical assembly language, a label is an identifier for the address of an instruction. (When the program is further translated into **QINS**, in the next section, every label l will be replaced by the offset of the address of where l is defined from the address of where l is used.)

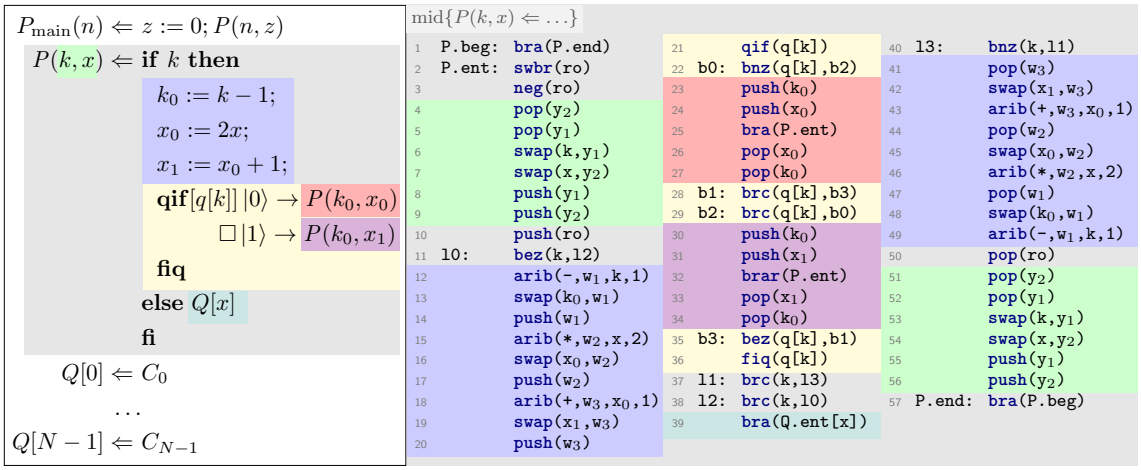


FIGURE 5.3: Example of high-level transformations and high-to-mid-level translation. The original program is the quantum multiplexor program in Figure 4.1. Here, on the LHS is the program after the high-level transformations. On the RHS is the high-to-mid-level translation of the procedure $P(k, x)$. Their connections are highlighted in colors. Note that new variables x_0, x_1, k_0 are introduced by the third step of high-level transformations, and some transformations have no effect on this example.

- We have additional instructions `push` and `pop` for stack operations. Also, an additional branching instruction `brc` will be used in pair with `bez` (or `bnz`). In particular, `brc(x,1)`, compared to `bra(1)`, has the additional information of some variable x.

The high-to-mid-level translation also automatically handles the initialisation of formal parameters and the uncomputation of classical variables at the end of procedure bodies (see Section 5.1.1).

We use mid$\{D\}$ to denote the high-to-mid-level translation of a statement (or declaration) D in the high-level language **RQC**$^{++}$. The definition of mid$\{\cdot\}$ is recursive, which further involves init$\{\cdot\}$ and uncp$\{\cdot\}$: the former stands for the initialisation of formal parameters in procedure declarations, and the latter stands for the uncomputation of classical variables (as required by the slightly changed program semantics after high-level transformations in Section 5.1.1).

In Figure 5.4, we present the full list of the high-to-mid-level translation mid$\{\cdot\}$, as well as the initialisation init$\{\cdot\}$ and the uncompuation uncp$\{\cdot\}$. We further explain as follows.

- For $D \equiv$ **skip** $\mid C_1; C_2 \mid U[q] \mid U[q_1 q_2]$, the translation mid$\{D\}$ and uncomputation uncp$\{D\}$ of D are self-explanatory. Note that our uncomputation is only for classical variables.

- For $D \equiv x := op\ y \mid x := y\ op\ z$, in the translation mid$\{D\}$, the standard technique of introducing garbage data in reversible computing [253, 254] is used. We first

compute the new value of x into a fresh variable w, then swap the values of x and w. Finally, the old value of x will be pushed into the stack. The uncomputation $\mathrm{uncp}\{D\}$ is simply the inverse of $\mathrm{mid}\{D\}$.

- For $D \equiv \textbf{if } x \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}$, the construction of $\mathrm{mid}\{D\}$ is inspired by [247]. Similar to the classical reversible architectures [234–237], we use a pair of branching instructions (e.g., `bez` and `brc`) to realise reversible (conditional) branching. Note that by the high-level transformation in Section 5.1.1, variable x is not changed by C_1 and C_2. So, we can use the value of x to determine which branch ($x = 0, 1$) the control flow has run through. Here, the mid-level instruction `brc` (a variant of `bra`) is used in pair with conditional branch instructions `bez` and `bnz` to indicate the variable x for the condition, which will be helpful later in the mid-to-low-level translation. The uncomputation $\mathrm{uncp}\{D\}$ is similarly constructed.

- For $D \equiv \textbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\textbf{fiq}$, the translation $\mathrm{mid}\{D\}$ is similar to that of the **if** statement.

  What is *new* is our introduction of a pair of instructions `qif(q)` and `fiq(q)`, which indicate the creation and join of quantum branching controlled by the quantum coin q. They are relevant to operations on the qif table in the partial evaluation (described in Section 5.2) and the execution (described in Section 5.3). Note that the uncomputation $\mathrm{uncp}\{D\}$ of the **qif** statement is $\emptyset$ due to Condition 2 in Section 3.2.3: there are no free changed variables and therefore the uncomputation of classical variables is not required.

- For $D \equiv \textbf{while } x \textbf{ do } C \textbf{ od}$, in the translation $\mathrm{mid}\{D\}$, a fresh variable y is introduced to records the number of steps taken in the loop. Note that by the high-level transformation in Section 5.1.1, x is not changed by C. So, we can use the value of x and y to determine where the control flow comes from. The construction of the uncomputation $\mathrm{uncp}\{D\}$ exploits the same fresh variable y as the one in $\mathrm{mid}\{D\}$.

- For $D \equiv P(\overline{x})$, in the translation $\mathrm{mid}\{D\}$, we first push n actual parameters $x_1, \ldots, x_n$ into the stack. Then, we branch to the declaration of the procedure P, whose address is specified by the variable $P.ent$. Finally, when the procedure call is returned, we pop the actual parameters from the stack. Note that the procedure P here can be subscripted, e.g., $P = Q[y]$ for some procedure array Q and variable y. In this case, $P.ent = Q.ent[y]$ is also subscripted. The uncomputation $\mathrm{uncp}\{D\} = \emptyset$, due to Condition 3 in Section 3.2.3.

- For $D \equiv P(\overline{u}) \Leftarrow C$, we need to handle the initialisation of formal parameters $\overline{u}$ and the automatic uncomputation of classical variables at the end of the procedure body C (see Section 5.1.1). The design of control flow in the translation $\mathrm{mid}\{D\}$ is

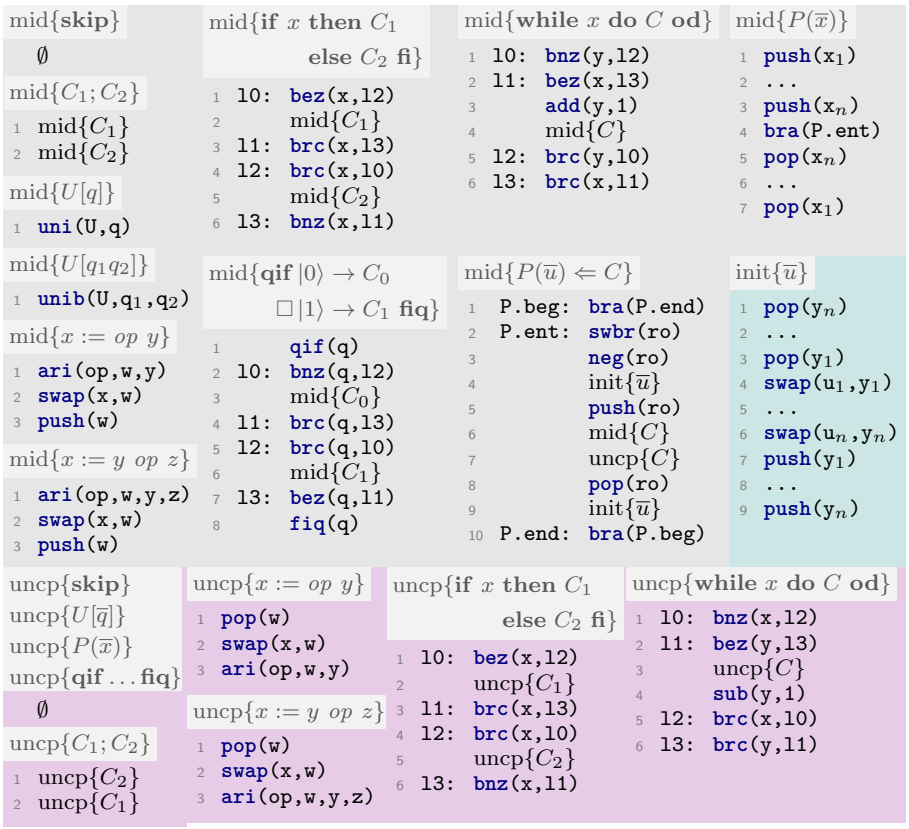


FIGURE 5.4: Full list of the high-to-mid-level translation mid{·}. Here, init{·} and uncp{·} denote the initialisation of formal parameters and uncomputation of classical variables, respectively. Variables that have not appeared in the original high-level programs are all fresh variables. Also, mid{**while**} and uncp{**while**} use the same fresh variable y.

inspired by that in the translation of classical reversible languages [247]. Specifically, the entry point and exit point to the declaration of procedure P are the same, specified by *P.ent*. Instruction **swbr** is used to update the return offset register *ro* (at both entry and exit), whose value will be temporarily pushed into the stack (see also Section 4.2.3) for supporting recursive procedure calls.

What is *new* here is our design of the *automatic uncomputation* of classical variables, performed by the program uncp{C} at the end of procedure body C (see also Section 5.1.1), which reverses the changes on classical variables in C. The uncomputation uncp{·} is also recursively defined, where uncp{$P(\overline{x})$} and uncp{**qif** ... **fiq**} are set to empty, due to Conditions 2 and 3.

- The initialisation init{$\overline{u}$} of formal parameters $\overline{u}$ is rather standard: we simply pop the actual parameters from the stack, then update the formal parameters, followed by pushing their old values into the stack.

```
1                start              ; start
2                push(n)            ; call P_main(n)
3                bra(P_main.ent)    ; call P_main(n)
4                pop(n)             ; call P_main(n)
5                finish             ; finish
6   P_main.beg:  bra(P_main.end)
7   P_main.ent:  swbr(ro)
8                neg(ro)
9                pop(y3)            ; init n
10               swap(n,y3)         ; init n
11               push(y3)           ; init n
12               push(ro)
13               push(n)            ; call P(n,z)
14               push(z)            ; call P(n,z)
15               bra(P.ent)         ; call P(n,z)
16               pop(z)             ; call P(n,z)
17               pop(n)             ; call P(n,z)
18               pop(ro)
19               pop(y3)            ; init n
20               swap(n,y3)         ; init n
21               push(y3)           ; init n
22  P_main.end:  bra(P_main.beg)
23  P.beg:       bra(P.end)
24  P.ent:       swbr(ro)
25               neg(ro)
26               pop(y2)            ; init k,x
27               pop(y1)            ; init k,x
28               swap(k,y1)         ; init k,x
29               swap(x,y2)         ; init k,x
30               push(y1)           ; init k,x
31               push(y2)           ; init k,x
32               push(ro)
33  l0:          bez(k,l2)          ; if k then ...
34               arib(-,w1,k,1)     ; k0 := k-1
35               swap(k0,w1)        ; k0 := k-1
36               push(w1)           ; k0 := k-1
37               arib(*,w2,x,2)     ; x0 := 2x
38               swap(x0,w2)        ; x0 := 2x
39               push(w2)           ; x0 := 2x
40               arib(+,w3,x0,1)    ; x1 := x0+1
41               swap(x1,w3)        ; x1 := x0+1
42               push(w3)           ; x1 := x0+1
43               qif(q[k])          ; qif(q[k]) ...
44  b0:          bnz(q[k],b2)       ; |0⟩ → ...
45               push(k0)           ; call P(k0,x0)
46               push(x0)           ; call P(k0,x0)
47               bra(P.ent)         ; call P(k0,x0)
48               pop(x0)            ; call P(k0,x0)
49               pop(k0)            ; call P(k0,x0)
50  b1:          brc(q[k],b3)

51  b2:          brc(q[k],b0)       ; |1⟩ → ...
52               push(k0)           ; call P(k0,x1)
53               push(x1)           ; call P(k0,x1)
54               bra(P.ent)         ; call P(k0,x1)
55               pop(x1)            ; call P(k0,x1)
56               pop(k0)            ; call P(k0,x1)
57  b3:          bez(q[k],b1)
58               fiq(q[k])          ; fiq(q[k]) ...
59  l1:          brc(k,l3)
60  l2:          brc(k,l0)          ; else ...
61               bra(Q.ent[x])      ; call Q[x]
62  l3:          bnz(k,l1)
63               pop(w3)            ; uncp k0,x0,x1
64               swap(x1,w3)        ; uncp k0,x0,x1
65               arib(+,w3,x0,1)    ; uncp k0,x0,x1
66               pop(w2)            ; uncp k0,x0,x1
67               swap(x0,w2)        ; uncp k0,x0,x1
68               arib(*,w2,x,2)     ; uncp k0,x0,x1
69               pop(w1)            ; uncp k0,x0,x1
70               swap(k0,w1)        ; uncp k0,x0,x1
71               arib(-,w1,k,1)     ; uncp k0,x0,x1
72               pop(ro)
73               pop(y2)            ; init k,x
74               pop(y1)            ; init k,x
75               swap(k,y1)         ; init k,x
76               swap(x,y2)         ; init k,x
77               push(y1)           ; init k,x
78               push(y2)           ; init k,x
79  P.end:       bra(P.beg)
80  Q0.beg:      bra(Q0.end)
81  Q.ent[0]:    swbr(ro)
82               neg(ro)
83               push(ro)
84               mid{C0}
85               uncp{C0}
86               pop(ro)
87  Q0.end:      bra(Q0.beg)
88               ...
89  QN-1.beg:    bra(QN-1.end)
90  Q.ent[N-1]:  swbr(ro)
91               neg(ro)
92               push(ro)
93               mid{CN-1}
94               uncp{CN-1}
95               pop(ro)
96  QN-1.end:    bra(QN-1.beg)
```

FIGURE 5.5: Full high-to-mid-level translation of the quantum multiplexor program. The original program is in Figure 4.1, with part of the high-to-mid-level translation previously presented in Figure 5.3.

For illustration, let us apply the high-to-mid-level translation to the transformed quantum multiplexor program on the LHS of Figure 5.3. On the RHS of Figure 5.3, we present part of the high-to-mid-level translation of the quantum multiplexor program, where we only show the translation of the recursive procedure $P(k, x)$. The full translated program is presented in Figure 5.5, which is annotated to show the correspondence with the program on the LHS of Figure 5.3. For simplicity of presentation, we have also done some manual optimisation on the translated program.

5.1.3 Symbol Table and Memory Allocation of Variables

Before we move to the mid-to-low-level translation, let us consider the memory allocation of variables and how the symbol table will be used at runtime.

After the high-level to mid-level translation, we can determine the names of all variables and procedure identifiers by scanning the whole program in the mid-level language, because the mid-to-low-level translation will no more introduce new variables or procedure identifiers. The addresses of all variables (corresponding to their names) will be stored in the symbol table and loaded into the QRAM for the addressing of variables at runtime.

For classical (either basic or array) variable x, we use $@x$ to denote the address of the variable name x, which falls in the symbol table section of the QRAM (see Section 4.2.3). The memory location at address $@x$ stores $\&x$, which stands for the address of the variable x. If x is an array variable, then $\&x$ will be the base address of x; i.e., $\&x = \&(x[0])$. As an address, $\&x$ falls in the classical variable section of the QRAM, and stores the value of x. The same convention applies to any quantum variable.

An example of the symbol table is visualised in Figure 5.6. Here, the symbol table section is composed of classical and quantum symbols. The memory location in the QRAM with address $@x$ stores $\&x$, which is the address of the memory location that stores $x[0]$, for classical array variable x. Similarly, for quantum variable q, the memory location with address $@q$ stores $\&q$, which is the address of the memory location that stores q.

Note that the sizes of classical and quantum arrays (except those classical variables $P.ent$ that correspond to procedure arrays P) are not determined at compile time as the classical inputs are not yet given. So, the values in the symbol table (i.e., addresses of all variables) need to be filled in after the classical inputs are considered (e.g., during the partial evaluation described in Section 5.2). Still, for every classical variable x, the address $@x$ of the name x is determined. The same holds for every quantum variable. Therefore, the compiled program will be completely determined and independent of classical and quantum inputs.

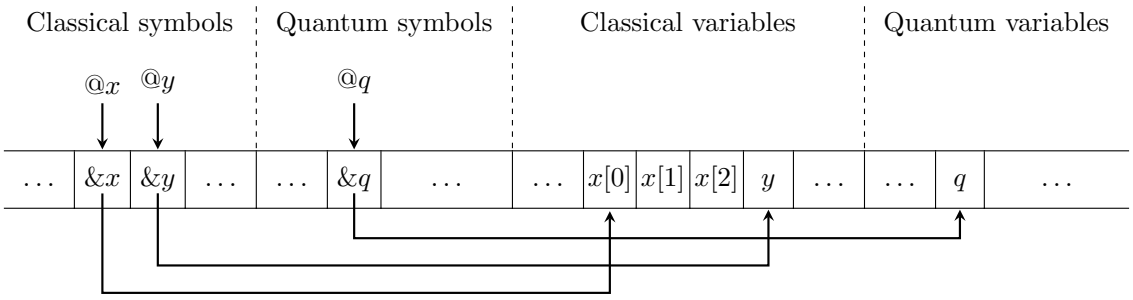


FIGURE 5.6: Example of the symbol table and memory allocation of variables in the QRAM.

5.1.4 Mid-Level to Low-Level Translation

Now we are ready to describe the last pass in which the mid-level program $\mathcal{P}_m$ obtained in the previous sections is translated into a program $\mathcal{P}_l$ in the low-level language **QINS** and thus executable on the quantum register machine. Recall that instructions in the mid-level language can take variables and labels as inputs, while the instructions in the low-level language **QINS** do not. To this end, in this pass, we will use load instructions `ld`, `ldr`, and `fetr` in **QINS** to first load variables from the QRAM into registers, and then execute the instruction, followed by storing the results back to the QRAM. Also, the mid-level language has additional instructions `push`, `pop`, and `brc` that need to be further translated.

Let us use low$\{i\}$ to denote the mid-to-low-level translation of an instruction i. We describe the mid-to-low-level translation through a series of examples, as shown in Figure 5.7. We further explain them as follows.

- The translation of `uni(U,q[x])` shows how to handle inputs containing subscripted quantum variables. As introduced in Section 5.1.3, we use @x to denote the address of the name x (in the symbol table section of the QRAM; see Section 4.2.3). The word at @x stores the address &x of the variable x (in the variable section). Lines 1–2 load the value of x into free register r_2. To obtain the address of $q[x]$, we add the address $\&q = \&(q[0])$ and the value of x, in Lines 3–4. Line 5 loads the value of $q[x]$ into free register r_4, on which the instruction `uni(U,r4)` is executed. Lines 7–11 reverse the effects of Lines 1-5.

- In the translation of `bra(P.ent)`, recall that the classical variable $P.ent$ corresponds to some procedure identifier P. Here, Lines 1–3 loads the value of $P.ent$ into free register r_2. Note that Line 2 uses `fetr` instead of `ldr` to preserve the copy of $P.ent$ in the QRAM for recursive procedure calls. Lines 4–5 calculate in r_2 the offset of

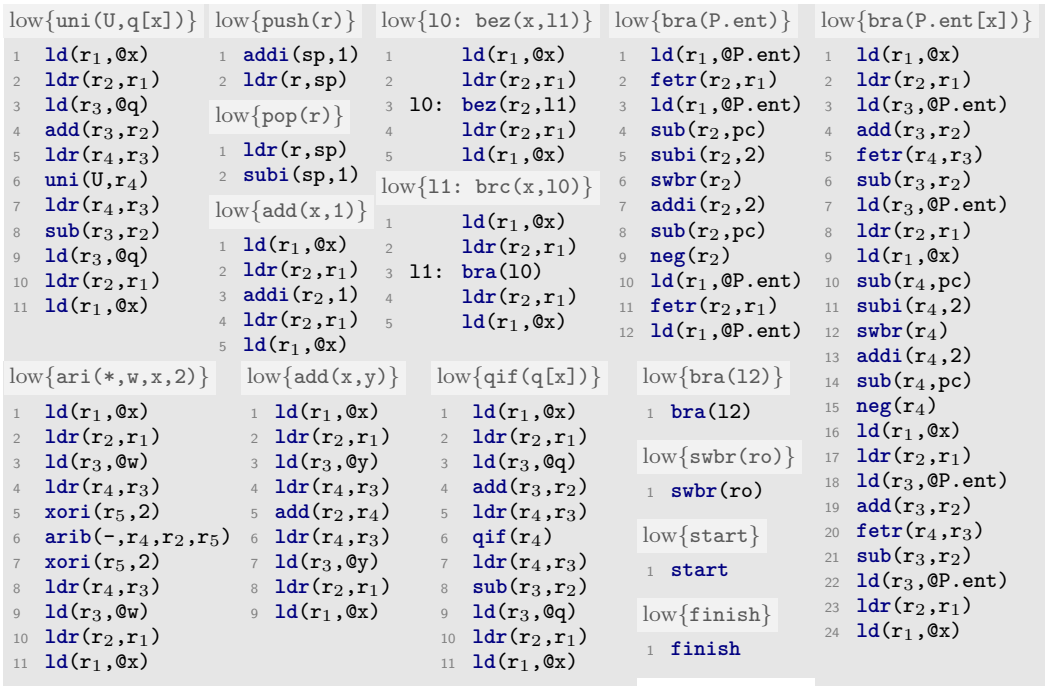


FIGURE 5.7: Examples of the mid-to-low-level translation. Here, all registers r_i are free registers.

P.ent from the address of Line 6. When the branching occurs after Line 6, note that registers r_1 and r_2 are cleared. Lines 7–12 are similar.

- The translations of `push` and `pop` are rather simple. Note that they are reversible, e.g., if an element is pushed into the stack, the original register r will be cleared. In the translation low{`push(r)`}, Line 1 first increments the stack pointer *sp* by 1. Then, Line 2 loads the topmost stack element (whose address is specified by *sp*) into free register r. The translation low{`pop(r)`} works similarly.

- In the translation low{`l0: bez(x,l1)`}, Lines 1–2 first load the value of variable x into free register r_2. Then, Line 3 branches to label $l0$, conditioned on $x \neq 0$. Finally, Lines 4–5 reverse the effects of Lines 1–2.

  Note that instruction `l1: brc(x,l0)` is used in pair with instruction `l0: bez(x,l1)`. So, the free registers r_1 and r_2 used in the translations low{`l1: brc(x,l0)`} and low{`l0: bez(x,l1)`} are also related (the same). This correspondence promises that registers r_1, r_2 are properly cleared (hence become free again) after being used.

- In the translation low{`add(x,1)`}, Lines 1–2 first load the value of variable x into free register r_2. Then, Line 3 adds 1 to r_2 to obtain $x + 1$. Finally, Lines 4–5 reverse the effects of Lines 1–2. Note that we use `addi` in Line 3, because 1 is an immediate number.

In comparison, in the translation low{`add(x,y)`}, we use `add` in Line 5, because both x, y are variables and loaded into registers.

- In the translation low{`ari(*,w,x,2)`}, Lines 1–4 first load the values of w and x into free registers r_2 and r_4, respectively. Then, Line 5 prepares the immediate number 2 in free register r_5. Line 6 performs the binary arithmetic operation $*$ and puts the result $x * 2$ into register r_4. Finally, Lines 7–11 reverse the effects of Lines 1–4

- For instructions like `bra(l2)`, `swbr(ro)`, `start`, and `finish` that are already in **QINS**, no further translation is needed.

- In the translation low{`bra(P.ent[x])`}, Lines 1–2 first load the value of x into free register r_2. Then, the base address $\&P.ent$ of classical array $P.ent$ is loaded into r_3 by Line 3, which is added to r_2 by Line 4 to obtain the address $\&P.ent[x]$ of the subscripted variable $P.ent[x]$. Line 5 loads the value of $P.ent[x]$ to r_4, where `fetr` instead of `ldr` is used to preserve the copy of $P.ent[x]$ in the QRAM for supporting recursive procedure calls. Lines 6–9 reverse the effects of Lines 1–4.

  Lines 10–11 calculate in r_4 the offset of $P.ent[x]$ from the address of Line 12, which branches to the declaration of the subscripted procedure $P[x]$. Note that after Line 12, all previously used registers are cleared and become free again. Finally, Lines 13–24 reverse the effects of Lines 1–12.

For illustration, let us apply the mid-to-low-level translation to the quantum multiplexor program in Figure 5.5 after the high-to-mid-level translation. This yields the full compiled program in Figure 5.8, which is also annotated to show the correspondence with the mid-level program in Figure 5.5. As usual, for simplicity of presentation, manual optimisation has been done on the translated program.

To end the compilation, we need to replace every label l in the compiled program by the offset of the address of where l is defined from where l is used. The compiled program is not yet loaded into the QRAM but stored classically for later partial evaluation (see Section 5.2).

5.2 Partial Evaluation of Quantum Control Flow

At the end of the last section, a compiled program in the low-level language **QINS** is obtained. To execute the compiled program on the quantum register machine, we need to first perform a partial evaluation of quantum control flow to generate a data structure called qif table, which will be loaded into the QRAM at runtime. In this section, we carefully describe this partial evaluation, which depends only on the classical inputs but not the quantum inputs.

```
1            start          ; start
2            ld(r1,@n)      ; load n
3            ldr(r2,r1)     ; load n
4            addi(sp,1)     ; push n
5            ldr(r2,sp)     ; push n
6            ldr(r2,r1)     ; store n
7            ld(r1,@n)      ; store n
8            bra(Pmain.ent) ; call Pmain(n)
9            ld(r1,@n)      ; load n
10           ldr(r2,r1)     ; load n
11           ldr(r2,sp)     ; pop n
12           subi(sp,1)     ; pop n
13           ldr(r2,r1)     ; store n
14           ld(r1,@n)      ; store n
15           finish         ; finish
16 Pmain.beg: bra(Pmain.end)
17 Pmain.ent: swbr(ro)
18           neg(ro)
19           ld(r1,@y3)     ; load y3
20           ldr(r2,r1)     ; load y3
21           ldr(r2,sp)     ; pop y3
22           subi(sp,1)     ; pop y3
23           ld(r3,@n)      ; load n
24           ldr(r4,r3)     ; load n
25           swap(r4,r2)    ; swap(n,y3)
26           addi(sp,1)     ; push y3
27           ldr(r2,sp)     ; push y3
28           ldr(r4,r3)     ; store n
29           ld(r3,@n)      ; store n
30           ldr(r2,r1)     ; store y3
31           ld(r1,@y3)     ; store y3
32           addi(sp,1)     ; push ro
33           ldr(ro,sp)     ; push ro
34           ld(r1,@n)      ; load n
35           ldr(r2,r1)     ; load n
36           addi(sp,1)     ; push n
37           ldr(r2,sp)     ; push n
38           ldr(r2,r1)     ; store n
39           ld(r1,@n)      ; store n
40           ld(r1,@z)      ; load z
41           ldr(r2,r1)     ; load z
42           addi(sp,1)     ; push z
43           ldr(r2,sp)     ; push z
44           ldr(r2,r1)     ; store z
45           ld(r1,@z)      ; store z
46           ld(r1,@P.ent)  ; copy P.ent
47           fetr(r2,r1)    ; copy P.ent
48           ld(r1,@P.ent)  ; copy P.ent
49           sub(r2,pc)     ; compute br
50           subi(r2,2)     ; compute br
51           swbr(r2)       ; bra(P.ent)
52           addi(r2,2)     ; compute P.ent
53           sub(r2,pc)     ; compute P.ent
54           neg(r2)        ; compute P.ent
55           ld(r1,@P.ent)  ; uncopy P.ent
56           fetr(r2,r1)    ; uncopy P.ent
57           ld(r1,@P.ent)  ; uncopy P.ent
58           ld(r1,@z)      ; load z
59           ldr(r2,r1)     ; load z
60           ldr(r2,sp)     ; pop z
61           subi(sp,1)     ; pop z
62           ldr(r2,r1)     ; store z
63           ld(r1,@z)      ; store z
64           ld(r1,@n)      ; load n
65           ldr(r2,r1)     ; load n
66           ldr(r2,sp)     ; pop n
67           subi(sp,1)     ; pop n
68           ldr(r2,r1)     ; store n
69           ld(r1,@n)      ; store n
70           ldr(ro,sp)     ; pop ro
71           subi(sp,1)     ; pop ro
72           ld(r1,@y3)     ; reverse 19-31
73           ldr(r2,r1)     ; reverse 19-31
74           ldr(r2,sp)     ; reverse 19-31
75           subi(sp,1)     ; reverse 19-31
76           ld(r3,@n)      ; reverse 19-31
77           ldr(r4,r3)     ; reverse 19-31
78           swap(r4,r2)    ; reverse 19-31
79           addi(sp,1)     ; reverse 19-31
80           ldr(r2,sp)     ; reverse 19-31
81           ldr(r4,r3)     ; reverse 19-31
82           ld(r3,@n)      ; reverse 19-31
83           ldr(r2,r1)     ; reverse 19-31
84           ld(r1,@y3)     ; reverse 19-31
85 Pmain.end: bra(Pmain.beg)
86 P.beg:    bra(P.end)
87 P.ent:    swbr(ro)
88           neg(ro)
89           ld(r1,@y2)     ; load y2
90           ldr(r2,r1)     ; load y2
91           ldr(r2,sp)     ; pop y2
92           subi(sp,1)     ; pop y2
93           ld(r3,@y1)     ; load y1
94           ldr(r4,r3)     ; load y1
95           ldr(r4,sp)     ; pop y1
96           subi(sp,1)     ; pop y1
97           ld(r5,@k)      ; load k
98           ldr(r6,r5)     ; load k
99           ld(r7,@x)      ; load x
100          ldr(r8,r7)     ; load x
101          swap(r6,r4)    ; swap(k,y1)
102          swap(r8,r2)    ; swap(x,y2)
103          addi(sp,1)     ; push y1
104          ldr(r4,sp)     ; push y1
105          addi(sp,1)     ; push y2
106          ldr(r2,sp)     ; push y2
107          ldr(r2,r1)     ; store y2
108          ld(r1,@y2)     ; store y2
109          ldr(r4,r3)     ; store y1
110          ld(r3,@y1)     ; store y1
111          ldr(r6,r5)     ; store k
112          ld(r5,@k)      ; store k
113          ldr(r8,r7)     ; store x
114          ld(r7,@x)      ; store x
115          addi(sp,1)     ; push ro
116          ldr(ro,sp)     ; push ro
117          ld(r1,@k)      ; load k
118          ldr(r2,r1)     ; load k
119 l0:      bez(r2,l2)     ; bez(k,l2)
```

FIGURE 5.8: Full compiled quantum multiplexor program, after the mid-to-low-level translation.

```
120 ldr(r2,r1)          ; store k
121 ld(r1,@k)           ; store k
122 ld(r1,@w1)          ; load w1
123 ldr(r2,r1)          ; load w1
124 ld(r3,@k)           ; load k
125 ldr(r4,r3)          ; load k
126 xori(r5,1)          ; prepare 1
127 arib(-,r2,r4,r5)    ; arib(-,w1,k,1)
128 ldr(r4,r3)          ; store k
129 ld(r3,@k)           ; store k
130 xori(r5,1)          ; clear 1
131 ld(r3,@k0)          ; load k0
132 ldr(r4,r3)          ; load k0
133 swap(r4,r2)         ; swap(k0,w1)
134 ldr(r4,r3)          ; store k0
135 ld(r3,@k0)          ; store k0
136 addi(sp,1)          ; push w1
137 ldr(r2,sp)          ; push w1
138 ldr(r2,r1)          ; store w1
139 ld(r1,@w1)          ; store w1
140 ld(r1,@w2)          ; load w2
141 ldr(r2,r1)          ; load w2
142 ld(r3,@x)           ; load x
143 ldr(r4,r3)          ; load x
144 xori(r5,2)          ; prepare 2
145 arib(*,r2,r4,r5)    ; arib(*,w2,x,2)
146 ldr(r4,r3)          ; store x
147 ld(r3,@x)           ; store x
148 xori(r5,2)          ; clear 2
149 ld(r3,@x0)          ; load x0
150 ldr(r4,r3)          ; load x0
151 swap(r4,r2)         ; swap(x0,w2)
152 ldr(r4,r3)          ; store x0
153 ld(r3,@x0)          ; store x0
154 addi(sp,1)          ; push w2
155 ldr(r2,sp)          ; push w2
156 ldr(r2,r1)          ; store w2
157 ld(r1,@w2)          ; store w2
158 ld(r1,@w3)          ; load w3
159 ldr(r2,r1)          ; load w3
160 ld(r3,@x0)          ; load x0
161 ldr(r4,r3)          ; load x0
162 xori(r5,1)          ; prepare 1
163 arib(+,r2,r4,r5)    ; arib(+,w3,x0,1)
164 ldr(r4,r3)          ; store x0
165 ld(r3,@x0)          ; store x0
166 xori(r5,1)          ; clear 1
167 ld(r3,@x1)          ; load x1
168 ldr(r4,r3)          ; load x1
169 swap(r4,r2)         ; swap(x1,w3)
170 ldr(r4,r3)          ; store x1
171 ld(r3,@x1)          ; store x1
172 addi(sp,1)          ; push w3
173 ldr(r2,sp)          ; push w3
174 ldr(r2,r1)          ; store w3
175 ld(r1,@w3)          ; store w3
176 ldr(r1,@k)          ; load k
177 ldr(r2,r1)          ; load k
178 ld(r3,@q)           ; compute &q[k]
179 add(r3,r2)          ; compute &q[k]
180     ldr(r4,r3)      ; load q[k]
181     sub(r3,r2)      ; uncompute &q[k]
182     ld(r3,@q)       ; uncompute &q[k]
183     ldr(r2,r1)      ; store k
184     ldr(r1,@k)      ; store k
185     qif(r4)         ; qif(q[k])
186 b0: bnz(r4,b2)      ; bnz(q[k],b2)
187     ldr(r1,@k)      ; store q[k]
188     ldr(r2,r1)      ; store q[k]
189     ld(r3,@q)       ; store q[k]
190     add(r3,r2)      ; store q[k]
191     ldr(r4,r3)      ; store q[k]
192     sub(r3,r2)      ; store q[k]
193     ld(r3,@q)       ; store q[k]
194     ldr(r2,r1)      ; store q[k]
195     ldr(r1,@k)      ; store q[k]
196     ld(r1,@k0)      ; load k0
197     ldr(r2,r1)      ; load k0
198     addi(sp,1)      ; push k0
199     ldr(r2,sp)      ; push k0
200     ldr(r2,r1)      ; store k0
201     ld(r1,@k0)      ; store k0
202     ld(r1,@x0)      ; load x0
203     ldr(r2,r1)      ; load x0
204     addi(sp,1)      ; push x0
205     ldr(r2,sp)      ; push x0
206     ldr(r2,r1)      ; store x0
207     ld(r1,@x0)      ; store x0
208     ld(r1,@P.ent)   ; copy P.ent
209     fetr(r2,r1)     ; copy P.ent
210     ld(r1,@P.ent)   ; copy P.ent
211     sub(r2,pc)      ; compute br
212     subi(r2,2)      ; compute br
213     swbr(r2)        ; bra(P.ent)
214     addi(r2,2)      ; compute P.ent
215     sub(r2,pc)      ; compute P.ent
216     neg(r2)         ; compute P.ent
217     ld(r1,@P.ent)   ; uncopy P.ent
218     fetr(r2,r1)     ; uncopy P.ent
219     ld(r1,@P.ent)   ; uncopy P.ent
220     ld(r1,@x0)      ; load x0
221     ldr(r2,r1)      ; load x0
222     ldr(r2,sp)      ; pop x0
223     subi(sp,1)      ; pop x0
224     ldr(r2,r1)      ; store x0
225     ld(r1,@x0)      ; store x0
226     ld(r1,@k0)      ; load k0
227     ldr(r2,r1)      ; load k0
228     ldr(r2,sp)      ; pop k0
229     subi(sp,1)      ; pop k0
230     ldr(r2,r1)      ; store k0
231     ld(r1,@k0)      ; store k0
232     ldr(r1,@k)      ; load k
233     ldr(r2,r1)      ; load k
234     ld(r3,@q)       ; compute &q[k]
235     add(r3,r2)      ; compute &q[k]
236     ldr(r4,r3)      ; load q[k]
237     sub(r3,r2)      ; uncompute &q[k]
238     ld(r3,@q)       ; uncompute &q[k]
```

FIGURE 5.8: Full compiled quantum multiplexor program, after the mid-to-low-level translation (cont.)

```
239      ldr(r2,r1)      ; store k
240      ldr(r1,@k)      ; store k
241 b1:  bra(b3)         ; brc(q[k],b3)
242 b2:  bra(b0)         ; brc(q[k],b0)
243      ldr(r1,@k)      ; store q[k]
244      ldr(r2,r1)      ; store q[k]
245      ld(r3,@q)       ; store q[k]
246      add(r3,r2)      ; store q[k]
247      ldr(r4,r3)      ; store q[k]
248      sub(r3,r2)      ; store q[k]
249      ld(r3,@q)       ; store q[k]
250      ldr(r2,r1)      ; store q[k]
251      ldr(r1,@k)      ; store q[k]
252      ld(r1,@k0)      ; load k0
253      ldr(r2,r1)      ; load k0
254      addi(sp,1)      ; push k0
255      ldr(r2,sp)      ; push k0
256      ldr(r2,r1)      ; store k0
257      ld(r1,@k0)      ; store k0
258      ld(r1,@x1)      ; load x1
259      ldr(r2,r1)      ; load x1
260      addi(sp,1)      ; push x1
261      ldr(r2,sp)      ; push x1
262      ldr(r2,r1)      ; store x1
263      ld(r1,@x1)      ; store x1
264      ld(r1,@P.ent)   ; copy P.ent
265      fetr(r2,r1)     ; copy P.ent
266      ld(r1,@P.ent)   ; copy P.ent
267      sub(r2,pc)      ; compute br
268      subi(r2,2)      ; compute br
269      swbr(r2)        ; bra(P.ent)
270      addi(r2,2)      ; compute P.ent
271      sub(r2,pc)      ; compute P.ent
272      neg(r2)         ; compute P.ent
273      ld(r1,@P.ent)   ; uncopy P.ent
274      fetr(r2,r1)     ; uncopy P.ent
275      ld(r1,@P.ent)   ; uncopy P.ent
276      ld(r1,@x1)      ; load x1
277      ldr(r2,r1)      ; load x1
278      ldr(r2,sp)      ; pop x1
279      subi(sp,1)      ; pop x1
280      ldr(r2,r1)      ; store x1
281      ld(r1,@x1)      ; store x1
282      ld(r1,@k0)      ; load k0
283      ldr(r2,r1)      ; load k0
284      ldr(r2,sp)      ; pop k0
285      subi(sp,1)      ; pop k0
286      ldr(r2,r1)      ; store k0
287      ld(r1,@k0)      ; store k0
288      ldr(r1,@k)      ; load k
289      ldr(r2,r1)      ; load k
290      ld(r3,@q)       ; compute &q[k]
291      add(r3,r2)      ; compute &q[k]
292      ldr(r4,r3)      ; load q[k]
293      sub(r3,r2)      ; uncompute &q[k]
294      ld(r3,@q)       ; uncompute &q[k]
295      ldr(r2,r1)      ; store k
296      ldr(r1,@k)      ; store k
297 b3:  bez(r4,b1)      ; bez(q[k],b1)
298      fiq(r4)         ; fiq(q[k])
299      ldr(r1,@k)      ; store q[k]
300      ldr(r2,r1)      ; store q[k]
301      ld(r3,@q)       ; store q[k]
302      add(r3,r2)      ; store q[k]
303      ldr(r4,r3)      ; store q[k]
304      sub(r3,r2)      ; store q[k]
305      ld(r3,@q)       ; store q[k]
306      ldr(r2,r1)      ; store q[k]
307      ldr(r1,@k)      ; store q[k]
308      ld(r1,@k)       ; load k
309      ldr(r2,r1)      ; load k
310 l1:  bra(l3)         ; brc(k,l3)
311 l2:  bra(l0)         ; brc(k,l0)
312      ldr(r2,r1)      ; store k
313      ld(r1,@k)       ; store k
314      ld(r1,@x)       ; load x
315      ldr(r2,r1)      ; load x
316      ld(r3,@Q.ent)   ; compute &Q.ent[x]
317      add(r3,r2)      ; compute &Q.ent[x]
318      fetr(r4,r3)     ; copy Q.ent[x]
319      sub(r3,r2)      ; uncompute &Q.ent[x]
320      ld(r3,@Q.ent)   ; uncompute &Q.ent[x]
321      ldr(r2,r1)      ; store x
322      ld(r1,@x)       ; store x
323      sub(r4,pc)      ; compute br
324      subi(r4,2)      ; compute br
325      swbr(r4)        ; bra(Q.ent[x])
326      addi(r4,2)      ; compute Q.ent[x]
327      sub(r4,pc)      ; compute Q.ent[x]
328      neg(r4)         ; compute Q.ent[x]
329      ld(r1,@x)       ; uncopy Q.ent[x]
330      ldr(r2,r1)      ; uncopy Q.ent[x]
331      ld(r3,@Q.ent)   ; uncopy Q.ent[x]
332      add(r3,r2)      ; uncopy Q.ent[x]
333      fetr(r4,r3)     ; uncopy Q.ent[x]
334      sub(r3,r2)      ; uncopy Q.ent[x]
335      ld(r3,@Q.ent)   ; uncopy Q.ent[x]
336      ldr(r2,r1)      ; uncopy Q.ent[x]
337      ld(r1,@x)       ; uncopy Q.ent[x]
338      ld(r1,@k)       ; load k
339      ldr(r2,r1)      ; load k
340 l3:  bnz(r2,l1)      ; bnz(k,l1)
341      ldr(r2,r1)      ; store k
342      ld(r1,@k)       ; store k
343      ld(r1,@w3)      ; reverse 158-175
344      ldr(r2,r1)      ; reverse 158-175
345      ldr(r2,sp)      ; reverse 158-175
346      subi(sp,1)      ; reverse 158-175
347      ld(r3,@x1)      ; reverse 158-175
348      ldr(r4,r3)      ; reverse 158-175
349      swap(r4,r2)     ; reverse 158-175
350      ldr(r4,r3)      ; reverse 158-175
351      ld(r3,@x1)      ; reverse 158-175
352      xori(r5,1)      ; reverse 158-175
353      ld(r3,@x0)      ; reverse 158-175
354      ldr(r4,r3)      ; reverse 158-175
355      arib(+,r2,r4,r5) ; reverse 158-175
356      xori(r5,1)      ; reverse 158-175
357      ldr(r2,r1)      ; reverse 158-175
```

FIGURE 5.8: Full compiled quantum multiplexor program, after the mid-to-low-level translation (cont.)

```
358  ld(r1,@w3)          ; reverse 158-175
359  ldr(r4,r3)          ; reverse 158-175
360  ld(r3,@x0)          ; reverse 158-175
361  ld(r1,@w2)          ; reverse 140-157
362  ldr(r2,r1)          ; reverse 140-157
363  ldr(r2,sp)          ; reverse 140-157
364  subi(sp,1)          ; reverse 140-157
365  ld(r3,@x0)          ; reverse 140-157
366  ldr(r4,r3)          ; reverse 140-157
367  swap(r4,r2)         ; reverse 140-157
368  ldr(r4,r3)          ; reverse 140-157
369  ld(r3,@x0)          ; reverse 140-157
370  xori(r5,2)          ; reverse 140-157
371  ld(r3,@x)           ; reverse 140-157
372  ldr(r4,r3)          ; reverse 140-157
373  arib(*,r2,r4,r5)    ; reverse 140-157
374  xori(r5,2)          ; reverse 140-157
375  ldr(r2,r1)          ; reverse 140-157
376  ld(r1,@w2)          ; reverse 140-157
377  ldr(r4,r3)          ; reverse 140-157
378  ld(r3,@x)           ; reverse 140-157
379  ld(r1,@w1)          ; reverse 122-139
380  ldr(r2,r1)          ; reverse 122-139
381  ldr(r2,sp)          ; reverse 122-139
382  subi(sp,1)          ; reverse 122-139
383  ld(r3,@k0)          ; reverse 122-139
384  ldr(r4,r3)          ; reverse 122-139
385  swap(r4,r2)         ; reverse 122-139
386  ldr(r4,r3)          ; reverse 122-139
387  ld(r3,@k0)          ; reverse 122-139
388  xori(r5,1)          ; reverse 122-139
389  ld(r3,@k)           ; reverse 122-139
390  ldr(r4,r3)          ; reverse 122-139
391  arib(*,r2,r4,r5)    ; reverse 122-139
392  xori(r5,1)          ; reverse 122-139
393  ldr(r2,r1)          ; reverse 122-139
394  ld(r1,@w1)          ; reverse 122-139
395  ldr(r4,r3)          ; reverse 122-139
396  ld(r3,@k)           ; reverse 122-139
397  ldr(ro,sp)          ; pop ro
398  subi(sp,1)          ; pop ro
399  ld(r1,@y2)          ; reverse 75-100
400  ldr(r2,r1)          ; reverse 75-100
401  ldr(r2,sp)          ; reverse 75-100
402  subi(sp,1)          ; reverse 75-100
403  ld(r3,@y1)          ; reverse 75-100
404  ldr(r4,r3)          ; reverse 75-100
405  ldr(r4,sp)          ; reverse 75-100
406  subi(sp,1)          ; reverse 75-100
407  ld(r5,@k)           ; reverse 75-100
408  ldr(r6,r5)          ; reverse 75-100
409  ld(r7,@x)           ; reverse 75-100
410  ldr(r8,r7)          ; reverse 75-100
411  swap(r6,r4)         ; reverse 75-100
412  swap(r8,r2)         ; reverse 75-100
413  addi(sp,1)          ; reverse 75-100
414  ldr(r4,sp)          ; reverse 75-100
415  addi(sp,1)          ; reverse 75-100
416  ldr(r2,sp)          ; reverse 75-100
417  ldr(r2,r1)          ; reverse 75-100
418                      ld(r1,@y2)      ; reverse 75-100
419                      ldr(r4,r3)      ; reverse 75-100
420                      ld(r3,@y1)      ; reverse 75-100
421                      ldr(r6,r5)      ; reverse 75-100
422                      ld(r5,@k)       ; reverse 75-100
423                      ldr(r8,r7)      ; reverse 75-100
424                      ld(r7,@x)       ; reverse 75-100
425  P.end:              bra(P.beg)
426  Q0.beg:             bra(Q0.end)
427  Q.ent[0]:           swbr(ro)
428                      neg(ro)
429                      addi(sp,1)      ; push ro
430                      ldr(ro,sp)      ; push ro
431                      mid{C0}
432                      uncp{C0}
433                      ldr(ro,sp)      ; pop ro
434                      subi(sp,1)      ; pop ro
435  Q0.end:             bra(Q0.beg)
436                      ...
437  QN-1.beg:           bra(QN-1.end)
438  Q.ent[N-1]:         swbr(ro)
439                      neg(ro)
440                      addi(sp,1)      ; push ro
441                      ldr(ro,sp)      ; push ro
442                      mid{CN-1}
443                      uncp{CN-1}
444                      ldr(ro,sp)      ; pop ro
445                      subi(sp,1)      ; pop ro
446  QN-1.end:           bra(QN-1.beg)
```

FIGURE 5.8: Full compiled quantum multiplexor program, after the mid-to-low-level translation (cont.)

5.2.1 The Synchronisation Problem

Programs with only classical control flow can be straightforwardly executed without partial evaluation. However, for programs with quantum control flow, there is an obstruction known as the synchronisation problem [78, 239–246] (see further discussion in Section 5.5.1). In our case, it means in executing the statement

$$\mathbf{qif}[q](|0\rangle \to C_0)\square(|1\rangle \to C_1)\mathbf{fiq},$$

C_0 and C_1 can take different numbers of instruction cycles to terminate. Consequently, the arrival times of two control flows (corresponding to two branches, in superposition) at the **fiq** are asynchronous, and hence they cannot be correctly merged into one control flow, in the same cycle. Another way to view the synchronisation problem is from the (QIF) rule in Figure 3.6. The problem occurs when $(C_0, \sigma, |\theta_0\rangle) \to^{k_0} (\downarrow, \sigma', |\theta_0'\rangle)$ and $(C_1, \sigma, |\theta_1\rangle) \to^{k_1} (\downarrow, \sigma', |\theta_1'\rangle)$ for some $k_0 \neq k_1$.

The synchronisation problem becomes more complicated for general quantum recursive programs. Note that C_0 and C_1 can further contain quantum recursion, and the number of nested procedure calls involved cannot be determined before hand. The program might not even terminate. How to deal with the probably unbounded quantum recursion?

Our solution is by partial evaluation of the quantum control flow. When the classical inputs are given (while the quantum inputs remained unknown), we can check whether the compiled program $\mathcal{P}$ terminates in some practical (manually set) running time $T_{\text{prac}}(\mathcal{P})$. If the program terminates in $T_{\text{prac}}(\mathcal{P})$ cycles, for every **qif** statement, we can count the number of cycles for executing C_0 and C_1, as well as determine the structure of nested quantum branching induced by nested procedure calls. These can be gathered into a classical data structure called *qif table*, which will be used later in quantum superposition at runtime to synchronise two quantum branches in every **qif** statement. Note that this process is only dependent on the classical inputs but *independent* of the quantum inputs, and it does not change the static program text (compared to [78]). Also, our partial evaluation is different from those (e.g., [62, 261]) that aim at optimising the programs.

Along with generating the qif table, given the classical inputs, we can also determine the sizes of all arrays and allocate the addresses for variables (including determining the symbol table). This task is simple and we will not describe its details.

5.2.2 Qif Table

Now let us introduce the notion of qif table, storing the history information of quantum branching for an execution of the compiled program $\mathcal{P}$, within a given practical running

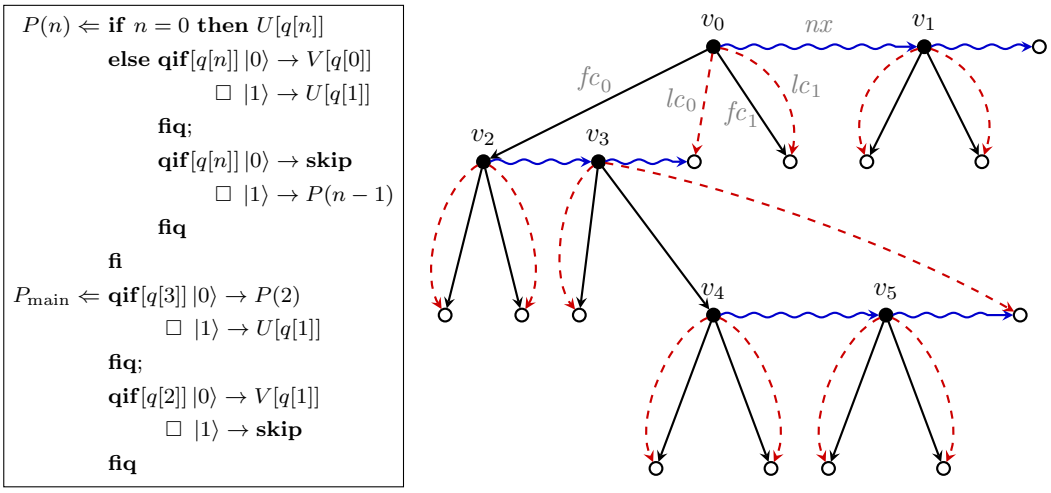


FIGURE 5.9: Example of a program in **RQC**$^{++}$ and its corresponding qif table. In the qif table, we only show the links fc_i (colored in black), lc_i (colored in red and dashed) and nx (colored in blue and squiggled), while w, cf, cl and pr are omitted for simplicity. The correspondence between the nodes on the RHS and the instantiations of **qif** statements on the LHS is as follows: (1) v_0: instantiation of the first **qif**...**fiq** in P_{main}. (2) v_1: instantiation of the second **qif**...**fiq** in P_{main}. (3) v_2 and v_4: the first and second instantiations of the first **qif**...**fiq** in $P(n)$. (4) v_3 and v_5: the first and second instantiations of the second **qif**...**fiq** in $P(n)$.

time $T_{\text{prac}}(\mathcal{P})$. The qif table is a classical data structure that will be used *in quantum superposition* at runtime.

Nodes and Links in Qif Table

Definition 5 (Qif table)**.** A qif table is composed of linked nodes. There are two types of nodes in the qif table. Each node of type • represents an instantiation of **qif**...**fiq**; i.e., an execution running through the `qif` to the corresponding `fiq` once. Nodes of type ∘ are ancilla nodes for the qif table to be *reversibly used*. Each node v of type • records the following information:

1. (Next link $v.nx$): If v has a *continuing* non-nested instantiation v' of **qif**...**fiq**, then $v.nx = v'$. Otherwise, we set $v.nx = v''$ for some node v'' of type ∘, enabling the qif table to be reversibly used (no matter whether v has a continuing instantiation of **qif**...**fiq**) in later execution.

2. (First children links $v.fc_i$) and (Last children links $v.lc_i$) for $i \in \{0,1\}$: If v has *enclosed* nested instantiations of **qif**...**fiq**, then $v.fc_0$ and $v.fc_1$ links to the first two children nodes, representing the first two enclosed instantiations of **qif**...**fiq** (corresponding to branches $|0\rangle$ and $|1\rangle$ from v, respectively). Moreover, $v.lc_0$ and $v.lc_1$ links to the last two children nodes, which are the two next nodes (specified by the next link

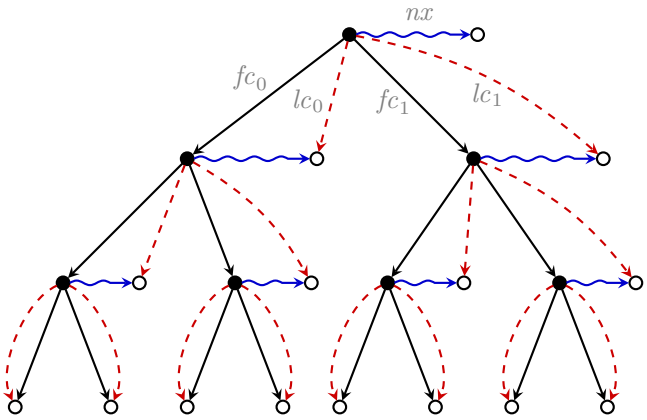


FIGURE 5.10: The qif table for the quantum multiplexor program in Figure 4.1, when $n = 3$.

nx and of type $\circ$) of the last two enclosed instantiations of **qif** ... **fiq** (corresponding to branches $|0\rangle$ and $|1\rangle$ from v, respectively).

Otherwise, $v.fc_0 = v.lc_0 = v'$ and $v.fc_1 = v.lc_1 = v''$ for some nodes v', v'' of type $\circ$.

Further, each node v of either type $\bullet$ or $\circ$ records the following information:

1. (Wait counter $v.w$): It stores the number of cycles to wait at node v.
2. ($v.pr$): pr is the inverse link of nx. If $v.nx = v'$, then $v'.pr = v$.
3. ($v.cf$): cf is the inverse link of fc_0 and fc_1. If $v.fc_0 = v_0$ and $v.fc_1 = v_1$, then $v_0.cf = v_1.cf = v$.
4. ($v.cl$): cl is the inverse link of lc_0 and lc_1. If $v.lc_0 = v_0$ and $v.lc_1 = v_1$, then $v_0.cl = v_1.cl = v$.

In Figure 5.9, we give an example of a program and its corresponding qif table. We only show the links fc_i, lc_i, and nx, and omit w, cf, cl, and pr for simplicity of presentation. The partial evaluation should be done on the compiled program, but for clarity we present the original program written in **RQC**$^{++}$. We also show the correspondence between nodes in the qif table and the instantiations of **qif** statements in the program. It is easy to verify that the links are consistent with Definition 5.

For illustration, we show the qif table for the quantum multiplexor program as another example in Figure 5.10. The original program written in **RQC**$^{++}$ was previously presented in Figure 4.1, while the full compiled program was shown in Figure 5.8. Here, we choose $n = 3$. For the quantum multiplexor program, the qif table is just a tree of depth 3, because there are only nested instantiations of the **qif** statements.

Additionally, we remark that to store the qif table in the QRAM, we need to encode all links and counter recorded at a node. The simplest way is to store them into a tuple,

where links like $v.nx$ records the base address of the tuple of the corresponding node. Further discussion can be found in Section 5.2.4.

5.2.3 Generation of Qif Table

For a compiled program $\mathcal{P}$, we fix a practical running time $T_{\text{prac}}(\mathcal{P})$. The partial evaluation is performed by multiple parallel processes. We classically emulate the execution of the compiled program, neglecting all quantum inputs and unitary gates. Whenever a `qif` is met, the current process forks into two sub-processes, each continuing the evaluation of the corresponding quantum branch. Whenever a `fiq` is met, the current process waits for its pairing sub-process, and collects information from both sub-processes to merge into one process. Every process only goes into a single quantum branch and therefore contains no quantum superposition.

For each process, we maintain the following classical information. We have 6 system registers pc, ins, br, sp, v, t, and a constant number of user registers. Here, v points to the current node in the qif table, and t is a counter that records the number of instructions already executed. We also have a classical memory M storing classical variables and the stack. Let M_i be the value stored at the memory location i.

The algorithm for partial evaluation of quantum control flow and generation of the qif table is presented as Algorithm 1. The major part of the function QEVA is the loop between Lines 5–25, which consists of three stages that also appear in the execution in Section 5.3. The first and the last stages are similar to their classical counterparts. The handling of instructions `qif` or `fiq` in the stage **(Decode & Execute)** is highlighted. Algorithm 1 returns a timeout error if t exceeds the practical running time $T_{\text{prac}}(\mathcal{P})$. Otherwise, we obtain the actual running time $t = T_{\text{exe}}(\mathcal{P})$ for later use in Section 5.3.

Further detailed explanation is as follows. Algorithm 1 is run by multiple parallel processes. We start with one initial process evaluating the function QEVA. At the beginning, the program counter pc is initialised to the starting address of the compiled main program (i.e., the address of instruction `start`) and the counter t is initialised to 0. A new node v is created in the qif table as the initial node. In the generation of the qif table, all new nodes created are of type $\circ$, which can later become nodes of type $\bullet$ if needed. The function QEVA then repeats Lines 5–25, as long as the running time t does not exceed the practical running time, i.e., $t \leq T_{\text{prac}}(\mathcal{P})$. The repetition consists of the following three stages, which also appear in the execution on quantum register machine (see Section 5.3).

1. **(Fetch)**:

   In this stage, register ins is updated with the current instruction M_{pc} whose address is specified by register pc. The counter t then increments by 1.

2. **(Decode & Execute)**:

Algorithm 1 Partial evaluation of quantum control flow and generation of qif table.

function QEVA
  Initialise $pc \leftarrow$ starting address of the compiled main program and $t \leftarrow 0$
  Create an initial node v
  while $t \leq T_{\text{prac}}(\mathcal{P})$ **do**
    (Fetch): Let $ins \leftarrow M_{pc}$ and $t \leftarrow t + 1$
    (Decode & Execute):
    if $ins =$ `qif(q)` **then** ▷ *creation of quantum branching*
      Create nodes v_0 and v_1. Set $v.fc_i, v.lc_i \leftarrow v_i$ and $v_i.cf, v_i.cl \leftarrow v$ ▷ *for the enclosed branches*
      Fork into two sub-processes QEVA_0 and QEVA_1. For QEVA_i, set $v \leftarrow v_i$
    else if $ins =$ `fiq(q)` **then** ▷ *join of quantum branching*
      Wait for the pairing sub-process QEVA' with $v'.cl = v.cl = \hat{v}$ for some parent $\hat{v}$
      $\hat{t} \leftarrow \max\{t, t'\}$, $v.w \leftarrow \hat{t} - t$, and $v'.w \leftarrow \hat{t} - t'$ ▷ *v', t' are corresponding v, t in QEVA'.*
      Merge with the pairing sub-process QEVA' by letting $t \leftarrow \hat{t}$ and $v \leftarrow \hat{v}$
      Create node u. Set $v.nx \leftarrow u$ and $u.pr \leftarrow v$. ▷ *for the continuing branch*
      Suppose $v.cl = \hat{u}$ and $\hat{u}.lc_x = v$ for some $\hat{u}$ and x. Let $u.cl \leftarrow \hat{u}$, $\hat{u}.lc_x \leftarrow u$, and $v.cl \leftarrow 0$
      Update $v \leftarrow u$
    else if $ins =$ `finish` **then** ▷ *termination*
      return t
    else if $ins \notin \{$`uni(G,r)`, `unib(G,`r_1`,`r_2`)`$\}$ **then** ▷ *neglect quantum gates*
      Update registers and M according to Figure 4.4a
    (Branch):
    if $br \neq 0$ **then**
      Let $pc \leftarrow pc + br$
    else
      Let $pc \leftarrow pc + 1$
  return Timeout error

In this stage, we classically emulate the behaviour of the quantum register machine, given the classical inputs. In particular, we neglect unitary gate instructions `uni` and `unib` as they involve the quantum inputs.

(a) If $ins =$ `qif(q)`, then we meet a creation of quantum branching, controlled by the quantum coin q. In this case, two new nodes v_0 and v_1 are created in the qif table, and are then linked with the current node v. Since v_0 and v_1 are from the first enclosed nested instantiation of **qif** ... **fiq** of v, by Definition 5, we have $v.fc_i = v.lc_i = v_i$ for $i = 0, 1$. The inverse links are created similarly. Next, we fork the current process QEVA into two sub-processes QEVA_0 and QEVA_1, which go into the quantum branches $q = 0$ and $q = 1$ (corresponding to $|0\rangle \rightarrow \ldots$ and $|1\rangle \rightarrow \ldots$ in the **qif** statement), respectively. For each QEVA_i, the current node v is updated by the children node v_i.

(b) If $ins =$ `fiq(q)`, then we meet a join of quantum branching, controlled by the quantum coin q. This implies the current process is a sub-process forked

from some parent process. So, we wait for the pairing sub-process QEVA′ with the same parent node; i.e., $v'.cl = v.cl = \hat{v}$ for some $\hat{v}$, where v' denotes the current node of QEVA′, and $\hat{v}$ denotes the parent node of v and v'. Let $\hat{t}$ be the maximum $\max\{t, t'\}$ of the numbers of instructions in executing the two quantum branches (corresponding to $q = 0, 1$). Given $\hat{t}$, we can calculate $v.w = \hat{t} - t$ and $v'.w = \hat{t} - t'$, the numbers of instruction cycles one needs to wait at the nodes v and v', respectively. Note that one of $v.w$ and $v'.w$ will be 0. These wait counter information will be used to synchronise the two quantum branches at runtime (see Section 5.3).

After collecting the information in two quantum branches, the current process QEVA will be merged with the pairing process QEVA′ into one process by updating t with $\hat{t}$ and v with $\hat{v}$.

Finally, we need to create a new node u in the qif table for the continuing quantum branch. We link the nodes v and u via nx and pr. According to Definition 5, node u will be updated as the new last children node for the parent node $\hat{u}$ of the current node v. Then, we move the current node from v to u. Note that the default type of u is $\circ$, and it can become $\bullet$ if later we meet a continuing non-nested instantiation of **qif** ... **fiq**.

(c) If *ins* = `finish`, then we have finished the execution of the program, and can return the counter t as the actual running time.

(d) Otherwise, if *ins* is any instruction other than the unitary gate instructions `uni` and `unib`, we update the involved registers and memory locations in M according to Figure 4.4a.

3. **(Branch)**:

   In this stage, we update the program counter *pc* according to the branch offset *br*. As previously mentioned in Section 4.2.1, if $br = 0$, then we increment *pc* by 1; if $br \neq 0$, then we update *pc* with $pc + br$.

If the program $\mathcal{P}$ does not terminate within the practical time $T_{\text{prac}}(\mathcal{P})$, Algorithm 1 returns a timeout error. Otherwise, it returns the actual running time t, denoted by $T_{\text{exe}}(\mathcal{P})$, which will be used in the execution on quantum register machine (see Section 5.3).

It is worth stressing again that in the partial evaluation, only those quantum variables q involved in the quantum control flow (specifically, `qif(q)` and `fiq(q)`) and all classical variables need to be evaluated; while all other quantum variables are ignored. The parallel processes for running Algorithm 1 are completely classical. One can also make use of this fact to dynamically maintain the classical registers and memory M for every evaluation process, in order to save the space complexity, but we will not discuss the details here for simplicity.

5.2.4 Memory Allocation of Qif Table

In the previous sections, we have defined a data structure called qif table to address the synchronisation problem and described how to generate it. The qif table generated from the partial evaluation needs to be loaded into the QRAM at runtime, as it will be used in superposition during the execution. To store a qif table in the QRAM, we need an encoding of every node and its relevant information (links and counter). The simplest way to encode is to gather all information of a node into a 9-tuple

$$(v.w, v.nx, v.fc_0, v.fc_1, v.lc_0, v.lc_1, v.pr, v.cf, v.cl). \tag{5.3}$$

Here, we assume every node v is identified by its base address of the above tuple in the QRAM. For nodes of type $\circ$, some of the entries in Equation (5.3) might be empty and set to 0. In this way, the whole qif table is encoded into an array of tuples, each corresponding to a node within. Accessing the information of a node can be done in a way similar to accessing an array element. For example, if every entry in Equation (5.3) occupies a word, then given the base address a of a tuple that corresponds to a node v, one can access the information $v.nx$ by the address $a + 1$. In Section 5.3, the quantum register machine might need to access the value of $v.nx$ (see e.g., Algorithm 3), which in this case can be fetched into a free register r for later computation, by applying the unitary $U_{fet}(r, a + 1, mem)$.

It is worth mentioning that more compact ways (with smaller space complexity) of encoding the qif table other than the straightforward encoding in Equation (5.3) exist but are not discussed here, for simplicity of presentation.

5.3 Execution on Quantum Register Machine

The execution is the last step of implementing quantum recursive programs and the only step that concerns the quantum inputs and requires quantum hardware. At this point, the original program written in the language **RQC**$^{++}$ is compiled into a low-level program $\mathcal{P}$ composed of instructions in **QINS**. We are promised that $\mathcal{P}$ terminates and has running time $T_{\text{exe}}(\mathcal{P})$, and we have generated its corresponding qif table that can be used to solve the synchronisation problem during the execution. Along the way, we have also set up a symbol table and allocated memory locations for classical and quantum variables. All these instructions and data are now loaded into the QRAM, according to the layout described in Section 4.2.3.

Algorithm 2 Execution on quantum register machine.

Unitary U_{main}
  Initialise registers according to Section 4.2.1
  for $t = 1, \ldots, T_{\text{exe}}$ **do**
    Apply the unitary U_{cyc} (defined below)
Unitary U_{cyc}
  (Set wait flag): Conditioned on *qifw*, set the wait flag in *wait*:
    Perform $\sum_{w,z} |w\rangle\langle w|_{qifw} \otimes |z \oplus [w > 0]\rangle\langle z|_{wait}$
  (Execute or wait): Conditioned on *wait*, apply the unitary U_{exe} (defined below), or wait and decrement the value in *qifw*:
    Perform $|0\rangle\langle 0|_{wait} \otimes U_{\text{exe}} + \sum_{z \neq 0, w} |z\rangle\langle z|_{wait} \otimes |w-1\rangle\langle w|_{qifw} \otimes \mathbb{1}$
  (Clear wait flag): Conditioned on *qifw* and *qifv*, uncompute the wait flag in *wait*:
    Perform $\sum_{w,v,z} |w\rangle\langle w|_{qifw} \otimes |v\rangle\langle v|_{qifv} |z \oplus [w < v.w]\rangle\langle z|_{wait}$
Unitary U_{exe}
  (Fetch): Apply the unitary $U_{\text{fet}}(ins, pc, mem)$ ▷ U_{fet} *is defined in Definition 4.*
  (Decode & Execute): Apply the unitary U_{dec} ▷ U_{dec} *is defined in Figure 4.4b.*
  (Unfetch): Apply the unitary $U_{\text{fet}}(ins, pc, mem)$ again
  (Branch): Update *pc*, conditioned on *br*:
    Apply $U_{+}(pc, br)$ ▷ U_{+} *performs the mapping* $|x\rangle|y\rangle \mapsto |x+y\rangle|y\rangle$.
    Apply $\circ(br)$-$\sum_x |x+1\rangle\langle x|_{pc}$ ▷ $\circ(\cdot)$-U *is defined in Section 4.3.*

5.3.1 Unitary U_{cyc} and Unitary U_{exe}

Algorithm 2 presents the execution on quantum register machine. The execution on quantum register machine consists of repeated cycles, each performing the unitary U_{cyc}. From the partial evaluation in Algorithm 1, we already obtained the running time $T_{\text{exe}}(\mathcal{P})$ of the compiled program $\mathcal{P}$, so we can fix the number of repetitions to be $T_{\text{exe}} = T_{\text{exe}}(\mathcal{P})$. Before the repetitions of U_{cyc}, we also need to initialise system registers *pc*, *sp*, *qifv* according to Section 4.2.1. All other registers are initialised to $|0\rangle$.

For each instruction cycle, in U_{cyc}, we need to decide whether to wait (i.e., skip the current instruction cycle) or execute, according to the wait counter information stored in the current node (specified by register *qifv*) in the qif table. To reversibly implement this procedure, we exploit two additional registers *qifw* and *wait*, and unitary U_{cyc} is designed to consist of the following three stages.

1. **(Set wait flag)**:

   In this stage, we check whether the value w in the wait counter register *qifw* is > 0, which records the number of cycles to be skipped at the current node. At this point, we are promised that the wait flag register *wait* is in state $|0\rangle$. If the wait counter $w > 0$, then we set a wait flag in register *wait* by flipping it to state $|1\rangle$, which indicates that the machine needs to wait at the current node.

2. **(Execute or wait)**:

In this stage, conditioned on the wait flag in *wait*, we decide whether to wait (i.e., skip the current cycle), or apply the unitary U_{exe} (defined in Algorithm 2 and explained later) to execute. If the wait flag is 0, then we apply U_{exe}. If the wait flag is 1, then we decrement the wait counter w in register *qifw* by 1.

3. **(Clear wait flag)**:

   In this stage, we clear the wait flag in register *wait*, conditioned on the value w in register *qifw* and the counter information $v.w$, where v is the value of register *qifv* that specifies the current node in the qif table. Specially, if $w < v.w$, which implies that we have set the wait flag in the first stage, then we need to clear the wait flag. Note that $v.w$ is stored in the qif table, and we need first to apply the unitary U_{fet} (see Definition 4) to fetch it into some free register before using it, of which details are omitted for simplicity. After this stage, we are guaranteed that the wait flag register *wait* is in state $|0\rangle$.

As a subroutine of U_{cyc}, the unitary U_{exe} consists of the following four stages, inspired by the design of classical reversible processor (e.g., [237]).

1. **(Fetch)**:

   In this stage, we fetch into register *ins* the current instruction, whose address is specified by the program counter *pc*. At this point, we are promised that *ins* is in state $|0\rangle$. So, we simply apply the QRAM fetch unitary $U_{\text{fet}}(ins, pc, mem)$ defined in Definition 4.

2. **(Decode & Execute)**:

   In this stage, we decode the current instruction in *ins*, and execute it by applying the unitary

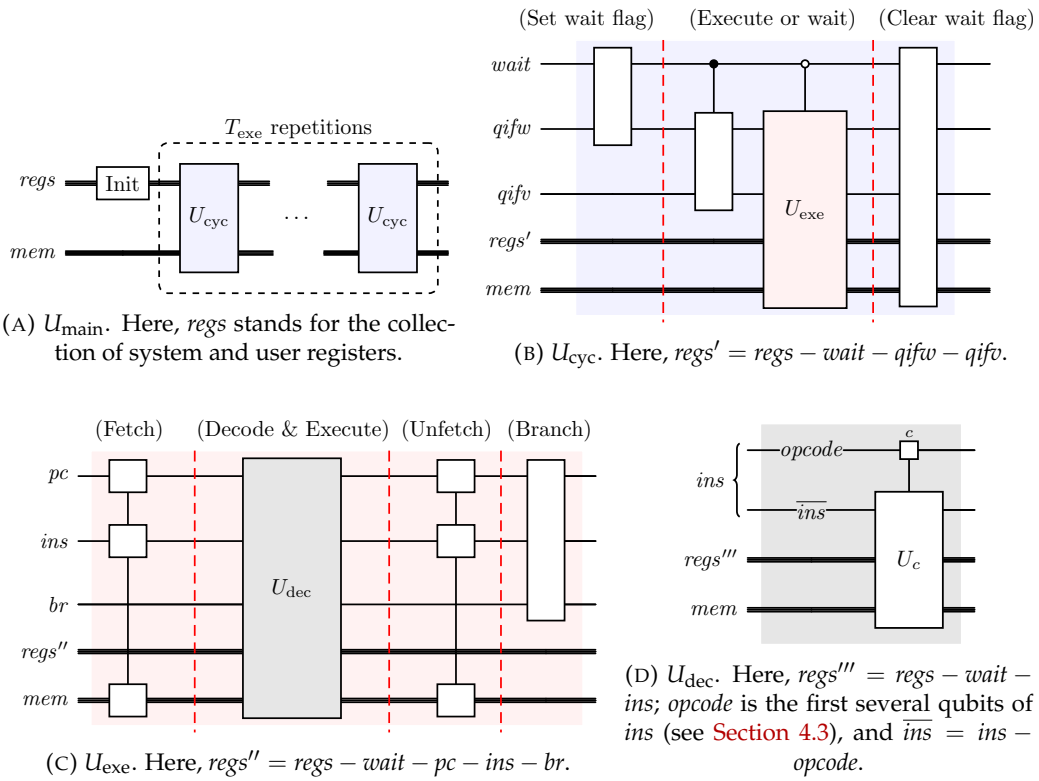
   $$U_{\text{dec}} = \sum_c |c\rangle\langle c| \otimes \sum_d |d\rangle\langle d| \otimes U_{c,d}$$

   previously defined in Figure 4.4b. The full list of unitaries $U_{c,d}$ for implementing different instructions was previously presented in Figure 4.5. What remains unspecified in Figure 4.5 are the unitaries U_{qif} and U_{fiq} for qif instructions `qif` and `fiq`, which are defined in Algorithm 3 and will be explained in detail later in Section 5.3.2.

3. **(Unfetch)**:

   In this stage, we clear the register *ins* by applying the unitary $U_{\text{fet}}(ins, pc, mem)$ again. After this stage, register *ins* is guaranteed to be in state $|0\rangle$.

4. **(Branch)**:



(A) U_{main}. Here, *regs* stands for the collection of system and user registers.

(B) U_{cyc}. Here, $regs' = regs - wait - qifw - qifv$.

(C) U_{exe}. Here, $regs'' = regs - wait - pc - ins - br$.

(D) U_{dec}. Here, $regs''' = regs - wait - ins$; *opcode* is the first several qubits of *ins* (see Section 4.3), and $\overline{ins} = ins - opcode$.

FIGURE 5.11: Visualisation of the execution on quantum register machine in the form of quantum circuits. Here, unitaries U_{main}, U_{cyc}, and U_{exe} are defined in Algorithm 2, while U_{dec} is defined in Figure 4.4b. A single wire in these quantum circuits stands for a quantum word, or a part of a quantum word, while a bundled wire stands for multiple quantum words.

In this stage, we update the program counter *pc* according to the branch offset register *br*. Specifically, if the value y in *br* is non-zero, then we add the offset y to the value in *pc*; otherwise, we simply increment the value in *pc* by 1.

To sum up, in Figure 5.11, we visualise the execution on quantum register machine (described by Algorithm 2) in the form of quantum circuits, which could be more familiar to the quantum computing community.

5.3.2 Unitaries for Executing Qif Instructions

It remains to define the unitaries U_{qif} and U_{fiq} unspecified in Figure 4.4a. They are for executing qif instructions and used in defining the unitary U_{dec} in Figure 4.4b (see also the full list of unitaries $U_{c,d}$ for implementing different instructions in Figure 4.5). We present their constructions in Algorithm 3. Additional remarks are as follows.

- **Unitary $U_{\mathbf{qif}}(q)$.**

Algorithm 3 The unitaries U_{qif} and U_{fiq} in Figure 4.4b.

1: **Unitary** $U_{\text{qif}}(q)$
2: Conditioned on q, move $qifv$ to its first children node in the qif table via the links fc_0 and fc_1; i.e., perform the following series of unitaries:
$V_{fc} = \sum_{v,x,u} |v\rangle\langle v|_{qifv} \otimes |x\rangle\langle x|_q \otimes |u \oplus v.fc_x\rangle\langle u|_r$, where r is a free register
$V_{cf} = \sum_{v,u} |v \oplus u.cf\rangle\langle v|_{qifv} \otimes |u\rangle\langle u|_r$
$U_{\text{swap}}(r, qifv)$, which also clears register r ▷ U_{swap} *is defined in Section 4.3.*
3: Update $qifw$ with the wait counter information corresponding to $qifv$; i.e., perform:
$\sum_{w,v} |v\rangle\langle v|_{qifv} \otimes |w \oplus v.w\rangle\langle w|_{qifw}$
4: **Unitary** $U_{\text{fiq}}(q)$
5: Conditioned on q, move $qifv$ to its parent node in the qif table via the inverse link cl; i.e., perform the following series of unitaries:
$V_{cl} = \sum_{v,u} |v\rangle\langle v|_{qifv} \otimes |u \oplus v.cl\rangle\langle u|_r$, where r is a free register
$V_{lc} = \sum_{v,x,u} |v \oplus u.lc_q\rangle\langle v|_{qifv} \otimes |x\rangle\langle x|_q \otimes |u\rangle\langle u|_r$
$U_{\text{swap}}(r, qifv)$, which also clears register r
6: Move $qifv$ to the next node in the qif table via the link nx; i.e., perform the following series of unitaries:
$V_{nx} = \sum_{v,u} |v\rangle\langle v|_{qifv} \otimes |u \oplus v.nx\rangle\langle u|_r$, where r is a free register
$V_{pr} = \sum_{v,u} |v \oplus u.pr\rangle\langle v|_{qifv} \otimes |u\rangle\langle u|_r$
$U_{\text{swap}}(r, qifv)$, which also clears register r

When the current instruction is `qif(q)`, we first move the current node specified by $qifv$ to its first children node corresponding to the quantum coin q. Specifically, conditioned on the value x of the quantum coin q, for the current node v in register $qifv$, we first compute into some free register r its first children $v.fc_x$ (corresponding to x); then use the inverse link cf to clear the garbage data v. A following swap unitary U_{swap} finishes this move of the current node. Note that the free register r is cleared at this point.

Then, we update the wait counter register $qifw$ with the counter information $v.w$ stored in the qif table, where v is the current node in register $qifv$. Here, we are promised that $qifw$ is initially in state $|0\rangle$, because U_{qif} is used as a subroutine of U_{exe}, which will only be performed by U_{cyc} when the wait counter $qifw$ is in state $|0\rangle$.

Note that in the above, the information fc_x, cf, and w are stored in the qif table, and need to be fetched using U_{fet} into free registers before being used (see also Section 5.2.4), of which details are omitted for simplicity.

- **Unitary** $U_{\textbf{fiq}}(q)$.

  When the current instruction is `fiq(q)`, we first move the current node specified by $qifv$ to its parent node. Specifically, similar to the construction of U_{qif}, for the current node v in register $qifv$, we first compute into some free register r its parent $v.cl$; then conditioned on the value x of the quantum coin q, clear the garbage data v using

the link lc_x. A following swap unitary U_{swap} finishes this first move of the current node.

Then, we continue to move the current node to its next node. Specifically, we first compute into some free register r the next node $v.nx$, where v is the current node in register *qifv*. Next, we clear the garbage data v using the inverse link pr, followed by a swap unitary U_{swap}. After the above procedure, the register r is also cleared.

Similar to the case of U_{qif}, the information cl, lc_x, nx, and pr are stored in the qif table, and need to be fetched using U_{fet} into free registers before being used (see also Section 5.2.4), of which details are omitted for simplicity.

Now we remark on how the qif table as a classical data structure is used *in quantum superposition* during the execution on quantum register machine. Recall that at runtime, the value in register *qifv* indicates the current node in the qif table. Register *qifv* can be in a quantum superposition state, in particular, entangled with the quantum coin q (as well as other register and the QRAM) when instruction `qif(q)` is executed (see Algorithm 3). For example, after the unitary $U_{\text{qif}}(q)$ is performed, the state of the quantum register machine can be $\frac{1}{\sqrt{2}}|0\rangle_q|v_1\rangle_{qifv}|\psi_0\rangle + \frac{1}{\sqrt{2}}|1\rangle_q|v_2\rangle_{qifv}|\psi_1\rangle$, where $|\psi_0\rangle$ and $|\psi_1\rangle$ are states of the remaining quantum registers and the QRAM. In this way, the information in the qif table is used in quantum superposition.

5.4 Examples

In this section, we further illustrate the compilation process through three more examples. For simplicity of presentation, we will only show the high-level transformations and the high-to-mid-level translation of these examples, while omitting the rather lengthy mid-to-low-level translation. These examples are from [37] and previously presented in Section 3.3. For readability, let us recall their high-level programs here.

Generation of the GHZ State

The first example is the generation of the Greenberger-Horne-Zeilinger (GHZ) state [209]. Let us recall the following program in $\mathbf{RQC}^{++}$ to generate the GHZ state [37], which involves recursive procedure calls but no **qif** statements:

$$\begin{aligned} P_{\text{main}}(n) \Leftarrow\ & \textbf{if } n = 1 \\ & \quad \textbf{then } H[q[n]] \\ & \quad \textbf{else } P_{\text{main}}(n-1); CNOT[q[n-1], q[n]] \\ & \textbf{fi}. \end{aligned} \tag{5.4}$$

The GHZ state generation program after the high-level transformations of Equation (5.4) is presented below:

$$
\begin{aligned}
P_{\text{main}}(n) \Leftarrow\ & n_0 := n - 1; \\
& \textbf{if } n_0 \textbf{ then} \\
& \quad P_{\text{main}}(n_0); \\
& \quad CNOT[q[n_0], q[n]] \\
& \textbf{else } H[q[n]] \\
& \textbf{fi}
\end{aligned}
$$

The high-to-mid-level translation is shown in Figure 5.12. As usual, we have done some manual optimisation to improve the presentation.

```
1             start             ; start
2             push(n)           ; call P_main(n)
3             bra(P_main.ent)   ; call P_main(n)
4             pop(n)            ; call P_main(n)
5             finish            ; finish
6  P_main.beg: bra(P_main.end)
7  P_main.ent: swbr(ro)
8             neg(ro)
9             pop(y1)           ; init n
10            swap(n,y1)        ; init n
11            push(y1)          ; init n
12            push(ro)
13            arib(-,w1,n,1)    ; n0 := n-1
14            swap(n0,w1)       ; n0 := n-1
15            push(w1)          ; n0 := n-1
16 10:        bez(n0,12)        ; if n0 then ...
17            push(n0)          ; call P_main(n0)
18            bra(P_main.ent)   ; call P_main(n0)
19            pop(n0)           ; call P_main(n0)
20            unib(CNOT,q[n0],q[n]) ; CNOT[q[n0],q[n]]
21 11:        brc(n0,13)
22 12:        brc(n0,10)        ; else ...
23            uni(H,q[n])       ; H[q[n]]
24 13:        bnz(n0,11)
25            pop(w1)           ; uncp n0
26            swap(n0,w1)       ; uncp n0
27            arib(-,w1,n,1)    ; uncp n0
28            pop(ro)
29            pop(y1)           ; init n
30            swap(n,y1)        ; init n
31            push(y1)          ; init n
32 P_main.end: bra(P_main.beg)
```

FIGURE 5.12: Full high-to-mid-level translation of the GHZ state generation program.

Multi-Controlled Gate

The second example is the multi-controlled gate. Let us recall the following program in **RQC**$^{++}$ to implement the multi-controlled gate [37], which illustrates both **qif** statements and recursive procedure calls:

$$
\begin{aligned}
P_{\text{main}}(n) &\Leftarrow P(1,n) \\
P(m,n) &\Leftarrow \textbf{if } m = n \\
&\qquad \textbf{then } U[q[n]] \\
&\qquad \textbf{else qif}[q[m]]\,|0\rangle \rightarrow \textbf{skip} \\
&\qquad\qquad\quad \square \quad |1\rangle \rightarrow P(m+1,n) \\
&\qquad \textbf{fiq} \\
&\textbf{fi}.
\end{aligned}
\tag{5.5}
$$

The multi-controlled gate program after applying the high-level transformations is presented below:

$$
\begin{aligned}
P_{\text{main}}(n) &\Leftarrow y := 1; P(y,n) \\
P(m,n) &\Leftarrow x := n - m; \\
&\qquad \textbf{if } x \textbf{ then} \\
&\qquad\quad m_0 := m + 1; \\
&\qquad\quad \textbf{qif}[q[m]]\,|0\rangle \rightarrow \textbf{skip} \\
&\qquad\qquad\quad \square \quad |1\rangle \rightarrow P(m_0,n) \\
&\qquad\quad \textbf{fiq} \\
&\qquad \textbf{else } U[q[n]] \\
&\qquad \textbf{fi}
\end{aligned}
\tag{5.6}
$$

The full high-to-mid-level translation of the program is shown in Figure 5.13, which is annotated to show the correspondence with the program in Equation (5.6).

Quantum State Preparation

The third example is the quantum state preparation. Let us recall the following program in $\mathbf{RQC}^{++}$ to generate the state $|\psi\rangle$ [37]:

```
1            start          ; start
2            push(n)        ; call P_main(n)
3            bra(P_main.ent) ; call P_main(n)
4            pop(n)         ; call P_main(n)
5            finish         ; finish
6  P_main.beg: bra(P_main.end)
7  P_main.ent: swbr(ro)
8            neg(ro)
9            pop(y3)        ; init n
10           swap(n,y3)     ; init n
11           push(y3)       ; init n
12           push(ro)
13           ari(+,w3,1)    ; y := 1
14           swap(y,w3)     ; y := 1
15           push(w3)       ; y := 1
16           push(y)        ; call P(y,n)
17           push(n)        ; call P(y,n)
18           bra(P.ent)     ; call P(y,n)
19           pop(n)         ; call P(y,n)
20           pop(y)         ; call P(y,n)
21           pop(w3)        ; uncp y
22           swap(y,w3)     ; uncp y
23           ari(+,w3,1)    ; uncp y
24           pop(ro)
25           pop(y3)        ; init n
26           swap(n,y3)     ; init n
27           push(y3)       ; init n
28 P_main.end: bra(P_main.beg)
29 P.beg:    bra(P.end)
30 P.ent:    swbr(ro)
31           neg(ro)
32           pop(y2)        ; init m,n
33           pop(y1)        ; init m,n
34           swap(m,y1)     ; init m,n
35           swap(n,y2)     ; init m,n
36           push(y1)       ; init m,n
37           push(y2)       ; init m,n
38           push(ro)
39           arib(-,w1,n,m) ; x := n-m
40           swap(x,w1)     ; x := n-m
41           push(w1)       ; x := n-m
42 l0:       bez(x,l2)      ; if x then ...
43           arib(+,w2,m,1) ; m0 := m+1
44           swap(m0,w2)    ; m0 := m+1
45           push(w2)       ; m0 := m+1
46           qif(q[m])      ; qif(q[m]) ...
47 b0:       bnz(q[m],b2)   ; |0⟩ → ...
48 b1:       brc(q[m],b3)
49 b2:       brc(q[m],b0)   ; |1⟩ → ...
50           push(m0)       ; call P(m0,n)
51           push(n)        ; call P(m0,n)
52           bra(P.ent)     ; call P(m0,n)
53           pop(n)         ; call P(m0,n)
54           pop(m0)        ; call P(m0,n)
55 b3:       bez(q[m],b1)
56           fiq(q[m])      ; fiq(q[m]) ...
57 l1:       brc(x,l3)
58 l2:       brc(x,l0)      ; else ...
59           uni(U,q[n])    ; U[q[n]]
60 l3:       bnz(x,l1)
61           pop(w2)        ; uncp x,m0
62           swap(m0,w2)    ; uncp x,m0
63           arib(+,w2,m,1) ; uncp x,m0
64           pop(w1)        ; uncp x,m0
65           swap(x,w1)     ; uncp x,m0
66           arib(-,w1,n,m) ; uncp x,m0
67           pop(ro)
68           pop(y2)        ; init m,n
69           pop(y1)        ; init m,n
70           swap(m,y1)     ; init m,n
71           swap(n,y2)     ; init m,n
72           push(y1)       ; init m,n
73           push(y2)       ; init m,n
74 P.end:    bra(P.beg)
```

FIGURE 5.13: Full high-to-mid-level translation of the multi-controlled gate program.

$$
\begin{aligned}
P_{\text{main}}(n) &\Leftarrow P(0,n,0)\\
P(k,n,x) &\Leftarrow \textbf{if } k \neq n \textbf{ then}\\
&\qquad Q(k,x);\\
&\qquad \textbf{qif}[q[k]]\ |0\rangle \to P(k+1,n,2x)\\
&\qquad\quad \square\ |1\rangle \to P(k+1,n,2x+1)\\
&\qquad \textbf{fiq}\\
&\quad \textbf{fi}\\
Q(k,x) &\Leftarrow C,
\end{aligned}
\tag{5.7}
$$

where C is a quantum circuit that performs $U_{k,x}[q[k+1]]$, and $U_{k,x}$ is defined in Equation (3.5). (In practice, when the elementary gate set is simple, e.g., $\{H,S,T,CNOT\}$, what C performs is only an approximation of $U_{k,x}$, and C depends on the explicit α_j (for

$j \in [N]$).)

The quantum state preparation program after applying the high-level transformations is presented below:

$$
\begin{aligned}
P_{\text{main}}(n) &\Leftarrow z := 0; P(z, n, z) \\
P(k, n, x) &\Leftarrow y := n - k; \\
&\quad \textbf{if } y \textbf{ then} \\
&\quad\quad Q(k, x); \\
&\quad\quad k_0 := k + 1; \\
&\quad\quad x_0 := 2x; \\
&\quad\quad x_1 := x_0 + 1; \\
&\quad\quad \textbf{qif}[q[k]]\ |0\rangle \to P(k_0, n, x_0) \\
&\quad\quad\quad\ \square\ \ |1\rangle \to P(k_0, n, x_1) \\
&\quad\quad \textbf{fiq} \\
&\quad \textbf{fi} \\
Q(k, x) &\Leftarrow C,
\end{aligned}
\tag{5.8}
$$

The full high-to-mid-level translation of the program is presented in Figure 5.14, where the annotation shows the correspondence with the program in Equation (5.8).

5.5 Discussion

5.5.1 Related Work

Automatic Uncomputation

Automatic uncomputation is an ideal feature for quantum programming languages. Silq [26] is the first language that supports automatic uncomputation, which was further investigated in [27, 28, 208, 262–264]. Silq focuses on uncomputation for quantum programs lifted from classical ones, or in their terminology, lifted functions (whose semantics can be described classically and preserves the input). Later works like [28, 264] also considered uncomputation of quantum programs but they do not support quantum recursion. In comparison, **RQC**$^{++}$ supports quantum recursion. The automatic uncomputation in our implementation (see Section 5.2.1) focuses on the classical variables, which are used solely for specifying the control (not data) of the quantum programs (see Chapter 3).

The Synchronisation Problem

In Section 5.2, we have described the partial evaluation of quantum control flow, which is used to address (within a practical running time) the synchronisation problem (see Section 5.2.1) for executing programs with quantum control flow. In the following, we discuss some related works on the synchronisation problem.

The synchronisation problem dates back to Deutsch's definition of quantum Turing machine [238]. In particular, Myers [239] pointed out a problem in Deutsch's original definition: for a universal quantum Turing machine that allows inputs to be in quantum superposition, different quantum branches are asynchronous and can have different termination times, where some branch might never even halt. Consequently, one cannot check the halting of a quantum Turing machine (according to Deutsch's definition) by directly measuring a flag qubit because the measurement can destroy the superposition of different quantum branches.

The later definition of a quantum Turing machine by Bernstein and Vazirani [240] circumvented this problem by an explicit halting scheme. It requires all quantum branches to be synchronised: if any quantum branch terminates the computation at some time T, then all other branches also terminate at T. A number of subsequent discussions [241–246] have also explored the synchronisation problem and the halting schemes of quantum Turing machine. Recently, in [78], they studied the synchronisation problem in general transition systems and formalised synchronisation as a condition.

Addressing (or circumventing) the synchronisation problem is often by restricting the inputs to be a specific subset such that different quantum branches always synchronise. For example, in [240], the synchronisation condition is explicitly stated in their definition of quantum Turing machine. In [246], they extend the definition of quantum Turing machine in [240], and show a conversion of a quantum Turing machine from the extended definition to the standard one by inserting meaningless symbols to synchronise different quantum branches. In [78], they impose structures on the program in their low-level language by borrowing the techniques of paired branch instructions from the classical reversible languages [234–237], and manually inserting `nop` (no operation) into the low-level programs to synchronise different quantum branches.

In this chapter, our implementation of quantum recursive programs addressed the synchronisation problem by exploiting the structures of the programs imposed by the language **RQC**$^{++}$ (in particular, the **qif** statement for managing quantum control flow). Moreover, our compilation ensures that the compiled low-level programs inherit such structures (via paired instructions `qif` and `fiq` in **QINS**; see Section 4.3). Then, we use the partial evaluation of quantum control flow to generate the qif table (see Section 5.2), a data structure that records the history information of quantum branching within a practical running time. This practical time can either be determined by the expected running

time or constrained by the capability of the quantum hardware, and we will conclude with a timeout error if the program does not terminate within the time. Qif table is used later (in quantum superposition) to synchronise different quantum branches at runtime (see Section 5.3.2). Our handling of uncomputation is automatic. In contrast, the technique of manually inserting nop (like in [78, 246]) is not extendable to handle our case, because the length of (dynamic) computation generated by quantum recursion cannot be pre-determined from the (static) program text.

5.5.2 Summary

In summary, we described a comprehensive implementation process of quantum recursive programs, consisting of three steps: compilation, partial evaluation, and execution. The compilation step transformed high-level programs in **RQC**$^{++}$ into low-level instructions in **QINS**, through high-level transformations, high-to-mid-level translation, and mid-to-low-level translation. The compiled program inherits the structure of the original program, and in particular, the structure of quantum branches generated by the **qif** statements.

Next, the partial evaluation step analysed the quantum control flow information of the compiled program, given only the classical inputs. The results were collected into the data structure, the qif table, which will be loaded into the QRAM alongside the compiled program to handle the synchronisation problem at runtime.

Finally, we described the execution step, where a fixed unitary is applied per instruction cycle. The synchronisation is handled using information from the qif table. The fixed unitary is carefully designed to uncompute all intermediate results, and will be eventually implemented by elementary quantum circuits. The efficiency of the above implementation will be analysed in the next chapter.

```
1            start            ; start
2            push(n)          ; call P_main(n)
3            bra(P_main.ent)  ; call P_main(n)
4            pop(n)           ; call P_main(n)
5            finish           ; finish
6  P_main.beg: bra(P_main.end)
7  P_main.ent: swbr(ro)
8            neg(ro)
9            pop(y4)          ; init n
10           swap(n,y4)       ; init n
11           push(y4)         ; init n
12           push(ro)
13           push(z)          ; call P(z,n,z)
14           push(n)          ; call P(z,n,z)
15           push(z)          ; call P(z,n,z)
16           bra(P.ent)       ; call P(z,n,z)
17           pop(z)           ; call P(z,n,z)
18           pop(n)           ; call P(z,n,z)
19           pop(z)           ; call P(z,n,z)
20           pop(ro)
21           pop(y4)          ; init n
22           swap(n,y4)       ; init n
23           push(y4)         ; init n
24 P_main.end: bra(P_main.beg)
25 P.beg:    bra(P.end)
26 P.ent:    swbr(ro)
27           neg(ro)
28           pop(y3)          ; init k,n,x
29           pop(y2)          ; init k,n,x
30           pop(y1)          ; init k,n,x
31           swap(k,y1)       ; init k,n,x
32           swap(n,y2)       ; init k,n,x
33           swap(x,y3)       ; init k,n,x
34           push(y1)         ; init k,n,x
35           push(y2)         ; init k,n,x
36           push(y3)         ; init k,n,x
37           push(ro)
38           arib(-,w1,n,k)   ; y := n-k
39           swap(y,w1)       ; y := n-k
40           push(w1)         ; y := n-k
41 l0:       bez(y,l2)        ; if y then ...
42           push(k)          ; call Q(k,x)
43           push(x)          ; call Q(k,x)
44           bra(Q.ent)       ; call Q(k,x)
45           pop(x)           ; call Q(k,x)
46           pop(k)           ; call Q(k,x)
47           arib(+,w2,k,1)   ; k0 := k+1
48           swap(k0,w2)      ; k0 := k+1
49           push(w2)         ; k0 := k+1
50           arib(*,w3,x,2)   ; x0 := 2x
51           swap(x0,w3)      ; x0 := 2x
52           push(w3)         ; x0 := 2x
53           arib(+,w4,x0,1)  ; x1 := x0+1
54           swap(x1,w4)      ; x1 := x0+1
55           push(w4)         ; x1 := x0+1
56           qif(q[k])        ; qif(q[k]) ...
57 b0:       bnz(q[k],b2)     ; |0⟩ → ...
58           push(k0)         ; call P(k0,n,x0)
59           push(n)          ; call P(k0,n,x0)
60           push(x0)         ; call P(k0,n,x0)
61           bra(P.ent)       ; call P(k0,n,x0)
62           pop(x0)          ; call P(k0,n,x0)
63           pop(n)           ; call P(k0,n,x0)
64           pop(k0)          ; call P(k0,n,x0)
65 b1:       brc(q[k],b3)
66 b2:       brc(q[k],b0)     ; |1⟩ → ...
67           push(k0)         ; call P(k0,n,x1)
68           push(n)          ; call P(k0,n,x1)
69           push(x1)         ; call P(k0,n,x1)
70           bra(P.ent)       ; call P(k0,n,x1)
71           pop(x1)          ; call P(k0,n,x1)
72           pop(n)           ; call P(k0,n,x1)
73           pop(k0)          ; call P(k0,n,x1)
74 b3:       bez(q[k],b1)
75           fiq(q[k])        ; fiq(q[k]) ...
76 l1:       brc(y,l3)
77 l2:       brc(y,l0)        ; else ...
78 l3:       bnz(y,l1)
79           pop(w4)          ; uncp x1
80           swap(x1,w4)      ; uncp x1
81           arib(+,w4,x0,1)  ; uncp x1
82           pop(w3)          ; uncp x0
83           swap(x0,w3)      ; uncp x0
84           arib(*,w3,x,2)   ; uncp x0
85           pop(w2)          ; uncp k0
86           swap(k0,w2)      ; uncp k0
87           arib(+,w2,k,1)   ; uncp k0
88           pop(w1)          ; uncp y
89           swap(y,w1)       ; uncp y
90           arib(-,w1,n,k)   ; uncp y
91           pop(ro)
92           pop(y3)          ; init k,n,x
93           pop(y2)          ; init k,n,x
94           pop(y1)          ; init k,n,x
95           swap(k,y1)       ; init k,n,x
96           swap(n,y2)       ; init k,n,x
97           swap(x,y3)       ; init k,n,x
98           push(y1)         ; init k,n,x
99           push(y2)         ; init k,n,x
100          push(y3)         ; init k,n,x
101 P.end:   bra(P.beg)
102 Q.beg:   bra(Q.end)
103 Q.ent:   swbr(ro)
104          neg(ro)
105          push(ro)
106          mid{C}
107          uncp{C}
108          pop(ro)
109 Q.end:   bra(Q.beg)
```

FIGURE 5.14: Full high-to-mid-level translation of the quantum state preparation program.

Chapter 6

Efficiency Analysis

Having established a systematic framework for implementing quantum recursive programs, this chapter analyses the efficiency. The analysis focuses on the parallel time complexity for the partial evaluation and the execution.

As a bonus, our implementation also offers *automatic parallelisation*. For implementing certain algorithmic subroutines, such as the quantum multiplexor (see Section 4.1.1), we can even obtain exponential parallel speed-up (over the straightforward implementation) from this automatic parallelisation. Our framework steps towards a *top-down* design of efficient quantum algorithms: programmers can focus on designing high-level quantum recursive programs, leaving the machine to automatically realise the parallelisation (whose quality still depends on the program structure).

6.1 Overview of Results

This section provides an overview of the efficiency analysis of our implementation framework in Chapter 5, focusing on the partial evaluation (Section 5.2) and execution (Section 5.3) steps. We first state their complexities in terms of elementary operations on registers and the QRAM and then refine this analysis into parallel time complexity, measured by the standard (classical and quantum) circuit depth. Before presenting the results, recall that $T_{\mathrm{exe}}(\mathcal{P})$ corresponds to the execution time of the longest quantum branch in a program $\mathcal{P}$.

1. **Partial Evaluation**: Algorithm 1 takes $O(T_{\mathrm{exe}}(\mathcal{P}))$ classical parallel elementary operations. Here, "elementary" means the operation only involves a constant number of memory locations in the classical RAM (as the partial evaluation is performed classically). "Parallel" means multiple elementary operations performed simultaneously are counted as one parallel elementary operation, like in the standard parallel computing. The intuition for this complexity is that in the partial evaluation, each of the classical parallel processes only evaluates one quantum branch.

2. **Execution**: Algorithm 2 takes $O(T_{\text{exe}}(\mathcal{P}))$ quantum elementary operations, including on registers and QRAM accesses (see Definition 4). The intuition was previously presented in Section 4.1.1.

The above complexities are in terms of elementary operations. As mentioned in Section 4.1, the implementation will be eventually quantum circuits. Therefore, we need to translate elementary operations into quantum circuits. The overall (classical and quantum) parallel time complexity of Algorithms 1 and 2 will be

$$O\big(T_{\text{exe}}(\mathcal{P}) \cdot \big(T_{\text{reg}} + T_{\text{QRAM}}\big)\big),$$

where T_{reg} and T_{QRAM} are parallel time complexities for elementary operations on registers and QRAM accesses, as aforementioned. Here, we assume that classical elementary operations are cheaper than their quantum counterparts.

For concreteness, let us return to the example of quantum multiplexor program $\mathcal{P}$ in Figure 4.1. Recall that the programs after high-level transformations and high-to-mid-level translation were presented in Figure 5.3. For illustration, we provide a proof sketch of Theorem 1, whose full proof will be presented in Section 6.5.1.

Proof Sketch of Theorem 1. Since each C_x only consists of T_x quantum unitary gates, the number of instructions in the compiled program of $P[x] \Leftarrow C_x$ will be $O(T_x)$. As a result, the whole compiled quantum multiplexor program (presented earlier in Section 5.1.4) contains $\Theta\Big(\sum_{x\in[N]} T_x\Big)$ instructions. It is easy to verify that $T_{\text{exe}}(\mathcal{P}) = O\Big(\max_{x\in[N]} T_x + n\Big)$.

Let us determine T_{reg} and T_{QRAM} by implementing the quantum register machine in the more common quantum circuit model. We can calculate the size N_{QRAM} of the QRAM and the word length L_{word} for implementing $\mathcal{P}$. In particular, taking $N_{\text{QRAM}} = \Theta(\sum_x T_x)$ is sufficient. To see this, we can calculate that the sizes of the program, symbol table, and variable sections are upper bounded by $\Theta(\sum_x T_x)$. The size of the qif table is $\Theta(2^n)$. The size of the stack is upper bounded by $\Theta(\sum_x T_x) + \Theta(n)$. To store an address in such QRAM, taking $L_{\text{word}} = \Theta(\log N_{\text{QRAM}})$ is sufficient.

By lifting results from classical parallel circuits for elementary arithmetic [265–267], we have $T_{\text{reg}} = O\Big(\log^2 L_{\text{word}}\Big)$. By extending existing circuit QRAM constructions (e.g., [218, 233]), we have $T_{\text{QRAM}} = O(\log N_{\text{QRAM}} + \log L_{\text{word}})$. The above calculations are carried out in terms of parallel time complexity, i.e., quantum circuit depth. Combining the above arguments leads to Theorem 1. □

6.2 Efficiency of Partial Evaluation

In this section, let us analyse the time complexity of the partial evaluation step described by Algorithm 1, in terms of classical parallel elementary operations. Our goal is to show that Algorithm 1 can be implemented using $O(T_{\text{exe}}(\mathcal{P}))$ classical parallel elementary operations.

To clarify what is meant by "elementary" here, recall that all registers, the memory, and the qif table in Algorithm 1 are simulated on a classical computer and stored in a classical RAM. We regard any operation that involves a constant number of memory locations in the classical RAM as an elementary operation. For example, in Algorithm 1, an operation on classical registers (including system registers pc, ins, br, sp, v, t and user registers), an access to the classical memory M, or creation of a node in the qif table will all be regarded as a classical elementary operation.

The meaning of "parallel" here is the common one in parallel computing. Multiple elementary operations performed simultaneously are counted as one parallel elementary operation. The parallelism can appear within a single process; e.g., when a process forks into two sub-processes, its data can be copied in a parallel way, for initialising both sub-processes. The parallelism can also appear among multiple processes; e.g., when two sub-processes run simultaneously before being merged, they can perform operations in parallel. We will not bother further going down to the rigorous details.

To prove our goal, note that Algorithm 1 terminates without timeout error if and only if $t = T_{\text{exe}}(\mathcal{P})$. So, it suffices to verify that every step of a process (among other parallel processes) executing QEVA can be implemented using $O(1)$ classical elementary operations, as follows.

- Consider those steps in Algorithm 1 only involving registers, the memory, and the qif table; e.g., $t \leftarrow 0$ in Line 2, $ins \leftarrow M_{pc}$ in Line 5, and the creation of nodes v_0, v_1 in Line 8. It is easy to verify that each of them only involves a constant number of memory locations in the classical RAM, and therefore can be implemented by $O(1)$ elementary operations.

- Consider those steps in Algorithm 1 involving forking and merging of sub-processes, in particular, Line 9 and Line 13. For the forking of a parent process into two sub-processes, we need to create a copy of all registers and the memory of the parent process. This can be done using $O(1)$ classical parallel elementary operations. Similarly, for the merging of two pairing sub-processes, taking the data from any of the two sub-processes and updating registers according to Line 13 can be done using $O(1)$ classical parallel elementary operations.

6.3 Efficiency of Execution

In this section, let us analyse the time complexity of the execution step described by Algorithm 2, in terms of quantum elementary operations. Our goal is to show that the unitary U_{main} for executing a compiled program $\mathcal{P}$ can be implemented using $O(T_{\text{exe}}(\mathcal{P}))$ elementary operations on registers (see Definition 3) and elementary QRAM accesses (see Definition 4).

To prove our goal, first note that the time complexity of U_{main} is dominated by the repeated applications of unitary U_{cyc}. It suffices to show that unitary U_{cyc} can be implemented using $O(1)$ elementary operations on registers and QRAM accesses.

Implementing U_{cyc} consists of the following steps. Let us analyse each of them.

1. (Set wait flag) in Line 6.

   It is easy to see that the unitary

   $$\sum_{w,z} |w\rangle\langle w|_{qifw} \otimes |z \oplus [w > 0]\rangle\langle z|_{wait}$$

   can be performed using $O(1)$ elementary operations on registers.

2. (Execute or wait) in Line 7.

   To implement the unitary

   $$|0\rangle\langle 0|_{wait} \otimes U_{\text{exe}} + \sum_{z \neq 0, w} |z\rangle\langle z|_{wait} \otimes |w-1\rangle\langle w|_{qifw} \otimes \mathbb{1},$$

   we first apply $\bullet(wait)$-$U_{-}(qifw, r)$, where U_{-} and $\bullet(\cdot)$-U are defined in Definition 3, and r is a free register initialised to $|1\rangle$. Then, we apply $\circ(wait)$-U_{exe}, where $\circ(\cdot)$-U is defined in Definition 3, and U_{exe} will be shown to be implementable using $O(1)$ elementary operations on registers and QRAM accesses below. Clearing garbage data is simple and also takes $O(1)$ elementary operations.

3. (Clear wait flag) in Line 8.

   To implement the unitary

   $$\sum_{w,v,z} |w\rangle\langle w|_{qifw} \otimes |v\rangle\langle v|_{qifv} |z \oplus [w < v.w]\rangle\langle z|_{wait},$$

   suppose that we use the simplest memory allocation of qif table aforementioned in Section 5.2.4. We first apply $U_{\text{fet}}(r, qifv, mem)$ (defined in Definition 4) to fetch the information $v.w$ into a free register r, where v is the value in register $qifv$. Given the information $v.w$ in r, it is easy to see that using $O(1)$ elementary operations on

registers we can compute $z \oplus [w < v.w]$ in register *wait*. Clearing garbage data is simple and also takes $O(1)$ elementary operations.

Next, we show that the unitary U_{exe} can be implemented using $O(1)$ elementary operations on registers and QRAM accesses. Implementing U_{exe} consists of the following steps.

1. (Fetch) in Line 10.

   This is a simple application of U_{fet}.

2. (Decode & Execute) in Line 11.

   This is an application of U_{dec}, which will be shown to be implementable using $O(1)$ elementary operations on registers and QRAM accesses below.

3. (Unfetch) in Line 12.

   This is a simple application of U_{fet} again.

4. (Branch) in Line 13.

   This step can be done by first applying the unitary U_+ defined in Definition 3, followed by $\circ(br)$-$U_+(pc, r)$, where r is a free register initialised to $|1\rangle$. Clearing garbage data is simple and also takes $O(1)$ elementary operations.

It remains to show that U_{dec} can be implemented using $O(1)$ elementary operations on registers and QRAM accesses. Recall that U_{dec} is a quantum multiplexor $\sum_c |c\rangle\langle c| \otimes \sum_d |d\rangle\langle d| \otimes U_{c,d}$ (see Figure 4.4b), where the full list of unitaries $U_{c,d}$ for all instructions was presented in Figure 4.5. It it easy to verify that every $U_{c,d}$ can be implemented by using $O(1)$ elementary operations on registers and QRAM accesses. The unitary U_{dec} can then be implemented by a sequential composition of controlled-$U_{c,d}$:

$$|c\rangle\langle c| \otimes |d\rangle\langle d| \otimes U_{c,d} + \sum_{c' \neq c, d' \neq d} |c'\rangle\langle c'| \otimes |d'\rangle\langle d'| \otimes \mathbb{1} \tag{6.1}$$

for all c, d, each using $O(1)$ elementary operations on registers and QRAM accesses. Here, c ranges over possible values of the section *opcode* in *ins* (i.e., the names of all instructions), and d ranges over possible values of other sections in *ins*. Since there are only 22 distinct instructions, $O(1)$ distinct (user and system) registers, and $O(1)$ distinct parameters (in the section *para* of *ins*; see Figure 4.4b) due to $|\mathcal{G}| = O(1)$ (see Section 3.2.2) and $|\mathcal{OP}| = O(1)$ (see Definition 3), the number of unitaries Equation (6.1) in the sequential composition is also $O(1)$. Hence, we conclude that unitary U_{dec} can be implemented using $O(1)$ elementary operations.

6.4 Quantum Circuit Complexity for Elementary Operations

In Section 6.2, we have shown that the partial evaluation for a compiled program $\mathcal{P}$ can be done using $O(T_{\text{exe}}(\mathcal{P}))$ classical parallel elementary operations. In Section 6.3, we have shown that the execution of $\mathcal{P}$ on quantum register machine can be done using $O(T_{\text{exe}}(\mathcal{P}))$ quantum elementary operations, including on registers (see Definition 3) and QRAM accesses (see Definition 4). These costs are in terms of (parallel) elementary operations. An immediate result now is that the overall parallel time complexity for implementing $\mathcal{P}$ is

$$O\big(T_{\text{exe}}(\mathcal{P}) \cdot \big(T_{\text{reg}} + T_{\text{QRAM}}\big)\big), \tag{6.2}$$

where T_{reg} and T_{QRAM} are the parallel time complexities for implementing an elementary operations on register and QRAM access, respectively. Here, we use the assumption that classical elementary operations are cheaper than their quantum counterparts.

The parallel time complexities T_{reg} and T_{QRAM} are further determined by how quantum elementary operations on registers and QRAM accesses are implemented at the quantum circuit level. In particular, we present the following two lemmas. The first lemma shows the quantum circuit complexity for implementing elementary operations on registers, which implies $T_{\text{reg}} = O(\log^2 L_{\text{word}})$ if we consider the parallel time complexity.

Lemma 3 (Quantum circuit complexity for elementary operations on registers)**.** *Suppose that $\mathcal{OP}$ is any subset of the following set of operators: addition, subtraction, multiplication, division, cosine, sine, arctangent, exponentiation, logarithm, maximum, minimum, factorial.*[1] *Then, every elementary operation on registers defined in Definition 3 can be implemented by a quantum circuit of depth $O\left(\log^2 L_{\text{word}}\right)$ and size $O\big(L_{\text{word}}^4\big)$.*

Proof. Let us verify every elementary operation on registers in Definition 3, one by one. For simplicity of presentation, we change the order of items in Definition 3.

- (Reversible versions of possibly irreversible arithmetic): According to Lemma 5 (see Section 6.6 at the end of this chapter), elementary arithmetic in $\mathcal{OP}$ can be implemented by quantum circuits with the desired properties in Lemma 3.
- (Swap): The implementation of U_{swap} is trivially a single layer of $O(L_{\text{word}})$ parallel swap gates.
- (Unitary gate): The unitary gates in the fixed set $\mathcal{G}$ only act on one or two qubits, and can be trivially implemented.
- (Reversible elementary arithmetic):

[1] Here, since we assume every number is stored in a word, each composed of L_{word} bits, these operators are actually approximated with error $2^{-L_{\text{word}}}$. See Lemma 5 for more details.

- The implementation of $U_{\oplus}$ is simply by a single layer of $O(L_{\text{word}})$ parallel CNOT gates.
- The unitary $U_{+}(r_1, r_2)$ can be implemented by first performing the mapping

$$|x\rangle_{r_1} |y\rangle_{r_2} |z\rangle_a \mapsto |x\rangle_{r_1} |y\rangle_{r_2} |z \oplus (x+y)\rangle_a , \tag{6.3}$$

with a being an ancilla register, followed by $U_{\text{swap}}(r_1, a)$, and finally performing the mapping

$$|x\rangle_{r_1} |y\rangle_{r_2} |z\rangle_a \mapsto |x\rangle_{r_1} |y\rangle_{r_2} |z \oplus (x-y)\rangle_a . \tag{6.4}$$

Note that Equations (6.3) and (6.4) are elementary operations of type (Reversibly versions of possibly irreversible arithmetic) discussed above, and therefore can be implemented by a quantum circuit of depth $O\left(\log^2 L_{\text{word}}\right)$ and size $O(L_{\text{word}}^4)$. We also have shown that U_{swap} can be implemented by a quantum circuit of depth $O(1)$ and size $O(L_{\text{word}})$. Hence, U_{+} can be implemented by a quantum circuit with the desired properties in Lemma 3.

- The implementation of U_{-} is similar to that of U_{+}.
- For U_{neg}, if we encode an integer by recording its sign in the first bit, then implementing $U_{\text{neg}}(r)$ is trivially applying an X gate on the first qubit of r.

- (Controlled versions): Recall that we only have a constant number of registers. Suppose that an elementary operation U can be implemented by a quantum circuit Q of depth $O\left(\log^2 L_{\text{word}}\right)$ and size $O(L_{\text{word}}^4)$. Then, the controlled version $\circ(r)$-U can be implemented by first applying $\circ(r)$-$X(a)$, with a being an ancilla qubit, followed by the single-controlled $c(a)$-U, and finally $\circ(r)$-$X(a)$ again. Here, $\circ(r)$-$X(a)$ is actually a multi-controlled Pauli X gate, which is known to be implementable by a quantum circuit of depth $O(\log L_{\text{word}})$ and size $O(L_{\text{word}})$ (similar to, e.g., Corollary 2.5 in [184]). The single-controlled $c(a)$-U can be implemented by replacing every gate in Q by its single-controlled version, which gives a quantum circuit of depth $O\left(\log^2 L_{\text{word}}\right)$ and size $O(L_{\text{word}}^4)$. The implementation of $\bullet(r)$-U is similar.

  Since we only have $O(1)$ registers, it is easy to see by induction that the controlled versions of any elementary operation can be implemented by a quantum circuit with the desired properties in Lemma 3.

□

The second lemma shows the quantum circuit complexity for implementing elementary QRAM accesses, which implies $T_{\text{QRAM}} = O(\log N_{\text{QRAM}} + \log L_{\text{word}})$ if we consider

the parallel time complexity. Using quantum circuits to implement QRAM is actually a well-studied topic, for which the reader is referred to [248] for a detailed review.

Lemma 4 (Quantum circuit complexity for elementary QRAM accesses)**.** *The elementary QRAM accesses defined in Definition 4 can be implemented by quantum circuits of*

- *depth* $O(\log N_{\mathrm{QRAM}} + \log L_{\mathrm{word}})$*; and*
- *size* $O(L_{\mathrm{word}} \cdot N_{\mathrm{QRAM}} \log N_{\mathrm{QRAM}})$.

Proof. We first show how to implement U_{ld}. Suppose that $N = N_{\mathrm{QRAM}}$ and $L = L_{\mathrm{word}}$. Existing circuit QRAM constructions (e.g., [218, 233]) show that a QRAM of N qubits can be implemented by a quantum circuit of depth $O(\log N)$ and size $O(N \log N)$. To adapt this to a QRAM of N quantum words, we can use L circuit QRAMs in parallel.

Specifically, the implementation of $U_{\mathrm{ld}}(r, a, mem)$ is as follows.

1. We first perform the unitary mapping

$$|x\rangle_a |y_1\rangle_{a_1} \dots |y_{L-1}\rangle_{a_{L-1}} \mapsto |x\rangle_a |y_1 \oplus x\rangle_{a_1} \dots |y_{L-1} \oplus x\rangle_{a_{L-1}}, \tag{6.5}$$

   where each of $a_1, \dots, a_{L-1}$ is composed of L ancilla qubits and initialised to $|0\rangle$. Here, Equation (6.5) prepares L copies of the address in a (in the computational basis), and can be implemented by a quantum circuit of depth $O(\log L)$ and size $O(L^2)$ (see e.g., [268]).

2. Suppose that mem_i is composed of the i^{th} qubit of every quantum word in our QRAM mem (composed of quantum words). Let r_i be the i^{th} qubit of the target register r, and let $a = a_0$. Then, we apply $U_{\mathrm{ld}}(r_i, a_i, mem_i)$ for $i = 0$ to $L - 1$ in parallel, each of which is implemented by a circuit QRAM composed of qubits, from previous works (e.g., [218, 233]). In total, this step can be implemented by a quantum circuit of depth $O(\log N)$ and size $O(LN \log N)$.

3. Finally, we perform the unitary mapping Equation (6.5) again to clear the garbage data.

It is easy to verify that the above implementation satisfies the properties in Lemma 4. □

It turns out that T_{reg} and T_{QRAM} given by Lemmas 3 and 4 are actually small enough compared to the overall parallel time complexity for implementing quantum recursive programs. In the next section, we will make this more concrete by analysing the example of the quantum multiplexor program.

6.5 Automatic Parallelisation

As a bonus, our implementation in Chapter 5 offers automatic parallelisation. The intuition was previously pointed out in Section 4.1.1: (1) with quantum control flow, the quantum register machine can go through quantum branches in superposition; and (2) with recursive procedure calls, the program can generate exponentially many quantum branches (as each instantiation of the **qif** statement creates two quantum branches).

6.5.1 Proof of Theorem 1

For concreteness, let us analyse the complexity for implementing the quantum multiplexor program on the quantum register machine. We provide the full proof of Theorem 1. Recall that Theorem 1 shows that we can obtain an exponential parallel speed-up (over the straightforward implementation) for implementing the quantum multiplexor.

Proof of Theorem 1. To determine the parallel time complexity, i.e., quantum circuit depth for implementing the quantum multiplexor program (see Figure 4.1) on quantum register machine, it suffices to determine the terms $T_{\text{exe}}(\mathcal{P})$, T_{reg} and T_{QRAM} in Equation (6.2), where $\mathcal{P}$ denotes the compiled program in Figure 5.8.

- To determine $T_{\text{exe}}(\mathcal{P})$, let us analyse the partial evaluation (described by Algorithm 1) of $\mathcal{P}$. Note that Lines 1–424 of $\mathcal{P}$ are independent of n, in the sense that no matter how large n is, this part of the program text is fixed.

  Before branching to each procedure $Q[x]$ (that describes unitary U_x in the quantum multiplexor Equation (4.1)) for $x \in [N]$, the value of the counter t is only determined by the number of nesting layers of **qif** ... **fiq** instantiations, up to a constant factor. As the qif table of the quantum multiplexor program is a tree, as shown in Figure 5.10, the number of nesting layers is exactly the depth of the tree, which is equal to n.

  After branching to procedures $Q[x]$ for $x \in [N]$, the value of the counter t is determined by the time complexity of executing every $Q[x]$. Recall that Theorem 1 assumes each U_x is composed of T_x elementary unitary gates, i.e., the procedure body C_x of $Q[x]$ is composed of T_x unitary gate instructions. So, the number of cycles to execute every C_x is $O(T_x)$. As Algorithm 1 is run by parallel processes, each going into a single quantum branch, the overall parallel running time is $O\left(\max_{x\in[N]} T_x\right)$.

  Combining the above together, we have

$$T_{\text{exe}}(\mathcal{P}) = O\left(\max_{x\in[N]} T_x + n\right).$$

- To determine T_{reg} and T_{QRAM}, it suffices to determine the word length L_{word} and the QRAM size N_{QRAM} for executing the compiled program $\mathcal{P}$. Let us first determine how large N_{QRAM} is needed, by calculating the sizes (counted by the number of quantum words) of all sections in the QRAM, as follows.

  - Program section: The compiled program $\mathcal{P}$ was presented in Figure 5.8, which consists of $\Theta\left(\sum_{x\in[N]} T_x\right)$ instructions.
  - Symbol table section: The size of the symbol table is upper bounded by the number of variables. By our assumption, every C_x is composed of unitary gate instructions, so the number of variables is further upper bounded by $\Theta\left(\sum_{x\in[N]} T_x\right)$, which is the number of instructions in $\mathcal{P}$.
  - Variable section: The size of the variable section is upper bounded by $\Theta\left(\sum_{x\in[N]} T_x\right)$, as mentioned above.
  - Qif table section: The size of the qif table is easily seen to be $\Theta(2^n)$, by noting that the qif table of $\mathcal{P}$ is a simple tree shown in Figure 5.10.
  - Stack section: The size of the stack is upper bounded by the number of variables and the number of nesting layers of **qif** ... **fiq** instantiations, up to a constant factor. The former is upper bounded by $\Theta\left(\sum_{x\in[N]} T_x\right)$, and the latter is upper bounded by $\Theta(n)$. Hence, the overall size is upper bounded by $\Theta\left(\sum_{x\in[N]} T_x\right)$.

  To summarise, taking $N_{\text{QRAM}} = \Theta\left(\sum_{x\in[N]} T_x\right)$ is sufficient to implement $\mathcal{P}$. Now for the word length L_{word}, we only need it to be large enough to store an address of a memory location in the QRAM, as required by the formats of instructions in **QINS** (see Figure 4.4b). So, taking $L_{\text{word}} = O(\log(N_{\text{QRAM}}))$ is enough. Finally, by Lemmas 3 and 4, we can calculate

$$T_{\text{reg}} = O\left(\log^2 \log N_{\text{QRAM}}\right) = O\left(\log^2 T_{\text{QRAM}}\right),$$
$$T_{\text{QRAM}} = O(\log N_{\text{QRAM}}) = O\left(n + \log\left(\max_{x\in[N]} T_x\right)\right).$$

By inserting the above $T_{\text{exe}}(\mathcal{P})$, T_{reg} and T_{QRAM} into Equation (6.2), we obtain the overall parallel time complexity $\widetilde{O}\left(n \cdot \max_{x\in[N]} T_x + n^2\right)$, as desired. □

6.6 Parallel Quantum Circuits for Elementary Arithmetic

In this section, we provide the detailed statement of a result about parallel quantum circuits for elementary arithmetics, which was previously used in the efficiency analysis in

Chapter 6. In particular, we restate Lemma 2.6 in [184], which translates several classical results in parallel computing [265–267] into the quantum setting: elementary arithmetic functions can be efficiently approximated by classical circuits of low depth. The translation is by introducing garbage data as in standard reversible computing [253, 254]. Although these classical results are more refined, the following lemma only provides simple upper bounds for simplicity.

Lemma 5 (Parallel quantum circuits for elementary arithmetics, restating Lemma 2.6 in [184])**.** *Suppose f is one of the following elementary arithmetic functions: addition, subtraction, multiplication, division, modulo, cosine, sine, arctangent, exponentiation, logarithm, maximum, minimum, factorial. Then, the unitary*

$$\sum_{\widetilde{x},\widetilde{y},z=0}^{2^L-1} |\widetilde{x},\widetilde{y}\rangle\langle\widetilde{x},\widetilde{y}| \otimes |z \oplus \widetilde{f}(\widetilde{x},\widetilde{y})\rangle\langle z|$$

can be implemented by a quantum circuit of depth $O\left(\log^2 L\right)$ and size $O\left(L^4\right)$, where $\widetilde{x},\widetilde{y}$ are certain proper representations of x, y, and $\widetilde{f}(\widetilde{x},\widetilde{y})$ is an approximation of $f(x,y)$ with error 2^{-L}. Here, for unary f, the input y is omitted. Further, if f is addition, subtraction or multiplication (modulo 2^L), the depth can be $O(\log L)$ and the error can be 0.

It is worth mentioning the quantum circuit in Lemma 5 can use ancilla qubits. The number of ancilla qubits can be trivially upper bounded by the size $O\left(L^4\right)$ of the quantum circuit.

6.7 Discussion

6.7.1 Related Work

Automatic parallelisation

Numerous efforts have been devoted to parallelisation of quantum circuits of specific patterns, e.g., [184, 228, 268–279]. Other than the quantum circuit model, the measurement-based quantum computing [280] is also shown to provide certain benefits for parallelisation [281–286]. These techniques of parallelisation are at the low level. In contrast, the automatic parallelisation from our implementation is at the high level: the quantum register machine automatically exploits parallelisation opportunities in the structures of the high-level quantum recursive programs.

Implementation of Quantum Multiplexor

In Theorem 1, we have shown that via the quantum register machine, we can automatically obtain a parallel implementation of the quantum multiplexor. Now we further

compare it with previous efficient implementations in [228, 229]. Recall that a quantum multiplexor can be described by the unitary

$$U = \sum_{x \in [N]} |x\rangle\langle x| \otimes U_x,$$

where $N = 2^n$ and n is the number of control qubits. Let us assume all U_x act on the last m data qubits. For general U_x, it is shown in [228, Algorithm 4 and 5] and [229, Lemma 7] that U can be implemented by a quantum circuit of depth $O\left(n + \max_{x \in [N]} T_x\right)$ and size $O\left(\sum_{x \in [N]} T_x\right)$, where each T_x is the time for implementing controlled-U_x. The idea of their construction is similar to that of the bucket-brigade QRAM [218, 231–233].

In particular, the bucket-brigade QRAM uses a binary tree structure to route the address qubits (see Section 4.2.3) to the corresponding memory location. The construction in [228, 229] uses a similar structure to route the n control qubits and a single data qubit to the location specified by the control qubits. Such structure is repeated m times in parallel to route the total m data qubits, and the first n control qubits also need to be copied (in the computational basis) m times in parallel. At each location $x \in [N]$, the quantum circuit for implementing U_x is placed. Finally, a reverse routing process is applied to retrieve the control and data qubits. In this way, if the n control qubits are in superposition, then the m data qubits can travel through a superposition of paths to pass through unitaries U_x and therefore realise the quantum multiplexor U.

Unlike in [228, 229] where data flows in quantum superposition, in our implementation of the quantum multiplexor, the m data qubits do not move. The control structure in U is completely captured by the quantum multiplexor program $\mathcal{P}$ in Figure 4.1, and the quantum register machine essentially realises the quantum control flow: it executes all quantum programs C_x (that describe unitaries U_x) in quantum superposition. Indeed, the quantum register machine also exploits the circuit QRAM [218, 231–233] (see also Section 6.4) at the low-level, to which the high-level programmer is oblivious.

From the perspective of design (as aforementioned in Sections 4.1.1 and 4.1.2), compared to the manual design of the rather involved quantum circuits in [228, 229], Theorem 1 is automatically obtained from our implementation of quantum recursive programs. In our framework, the programmer only needs to design at a high-level (in particular, the quantum multiplexor program in Figure 4.1), and needs not to know the explicit construction of QRAM. Not only restricted to the quantum multiplexor, the automatic parallelisation provided by our framework also works for general quantum recursive programs.

6.7.2 Conclusion and Open Questions

In this chapter, we analysed the efficiency of our implementation of quantum recursive programs proposed in the previous chapters. We also showed that this implementation offers automatic parallelisation, which can yield exponential parallel speed-up (over the straightforward implementation) for implementing some important quantum algorithmic subroutines like the quantum multiplexor.

This efficiency was developed on foundations established in the preceding chapters: (1) The architecture quantum register machine that provides instruction-level support for quantum control flow and recursive procedure calls at the same time; and (2) A comprehensive implementation process of quantum recursive programs on the quantum register machine, covering compilation, partial evaluation, and execution.

To conclude Part I of this thesis, let us list several topics for future research. Firstly, an immediate next step is to develop a software that realises our implementation of quantum recursive programs for actual execution on future quantum hardware. Moreover, one can consider certifying such software implementation, analogous to recent verified quantum compilers, e.g., [57, 61–63, 287]. Secondly, our implementation is designed to be simple for clarity. It is worth extending the features of the quantum register machine and further optimise the steps in the compilation, partial evaluation, and execution. Thirdly, it is interesting to see what other quantum algorithms (except those considered in [37] and here) can be written in quantum recursive programs and benefit (with possible speed-up) from the efficient implementation of the quantum register machine.

Part II

Access Control

Chapter 7

Access Control Threatened by Quantum Entanglement

Access control is a cornerstone of computer security, preventing unauthorised access to resources. The two chapters of Part II of this thesis investigate this concept in the new context of quantum operating systems by addressing the question: Do security guarantees of access control still hold when existing operating systems integrate quantum computing? This chapter provides a negative answer to this question. We present the first explicit scenario of a *security breach*, demonstrating how the security guarantees of a classical access control system fail when straightforwardly adapted to the quantum setting. We identify that the threat is *fundamentally from quantum entanglement*, highlighting the necessity of developing new quantum access control models.

7.1 Introduction

7.1.1 Motivation

A fundamental issue in computer security is how to control access to resources in computer systems. Initially proposed with the seminal concept of explicitly managing *rights* granted to a *subject* to access an *object*, the access matrix model [6, 175, 288–290] has served as the standard core model of access control. Over time, according to different security requirements, it has evolved into various sophisticated access control models, such as discretionary [291, 292], mandatory [293–297], and role-based access control [298–301], along with their further extensions, which are now widely deployed in modern operating systems.

On the other hand, the rapid emergence of quantum computing technology has drawn increasing attention to the security of quantum computer systems. For example, to protect user privacy when using untrusted quantum computing servers, numerous efforts have been devoted to delegated quantum computation (and further, blind

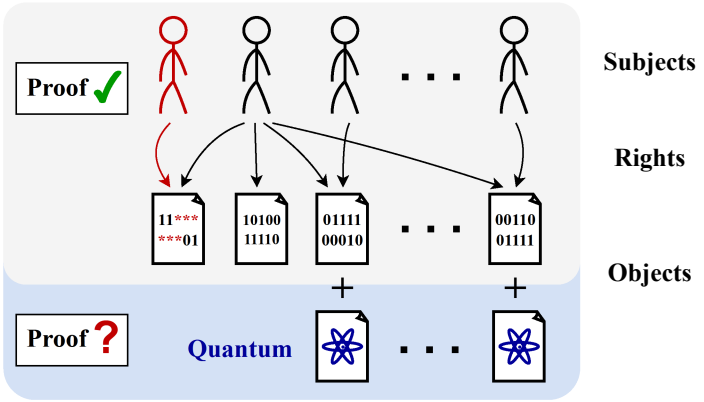


FIGURE 7.1: Do the security guarantees of access control still hold when existing operating systems integrate quantum computing?

quantum computation) [302–315], as well as quantum computer trusted execution environment [316–319] in recent years. Protecting security against hardware and side channel attacks in quantum computers has also attracted much attention [320–322]. The first attempt to access control in quantum systems was made by [116] through quantum information flow security. In the context of quantum internet, specific control of entanglement accessibility was also studied [323].

Still, a significant question remains open: whether the security guarantees of access control still hold when existing operating systems integrate quantum computing. More precisely:

Question 1. *Suppose a classical system earns your trust through a proof that its access control mechanism can protect your confidential information from other users. One day, the system is upgraded to integrate new quantum computing services and the access control remains unchanged. Should you still trust the security of the system?*

This question is becoming increasingly important as IBM Quantum and other researchers are actively exploring quantum-centric supercomputing, which integrates high-performance computers with quantum computing [79–83]. Definitely, we hope the hybrid systems remain secure.

However, the answer to Question 1 is arguably *no*. Our first aim is to show a security breach can occur in this case, through an explicit scenario. This highlights the necessity of developing new models of access control for quantum operating systems, which is our second aim.

7.1.2 Overview

More concretely, in the remainder of this chapter, we reveal a threat from quantum entanglement to access control. Background on access control is provided in Section 7.2. Then, in Section 7.3, we present the first explicit scenario of a security breach when a classically secure access control system is straightforwardly adapted to the quantum setting. The key insight behind this scenario is that quantum entanglement violates Mermin inequality [176], a multipartite variant of the celebrated Bell inequality [324–328]. Consequently, the threat we reveal is information-theoretic (i.e., without computational assumptions) and fundamentally quantum. Since entanglement is believed to be the source of quantum advantages [329] for many quantum algorithms [12–14, 216], our scenario highlights the importance of developing models of quantum access control against threats from entanglement, which will be studied in Chapter 8. For readability, proof details are deferred to Section 7.4.

7.2 Background

7.2.1 Access Control

We start by introducing the framework of access control used in this part of the thesis, which adopts ideas and concepts from the modern access (usage) control framework UCON [330–332].

Subjects, Objects, and Rights

A *subject* (e.g., user, process) can access an *object* (e.g., file, directory, register) by exercising a *right* (e.g., `read`, `write`). Here, we restrict our subjects to be users, objects to be (classical and quantum) registers, and rights to be abilities to perform certain operations on registers. Unless explicitly specified, classical and quantum registers are initialised to 0 and $|0\rangle$, respectively. Let $\mathbf{Sub}$, $\mathbf{Obj}$, and $\mathbf{Rt}$ denote the sets of subjects, objects, and rights, respectively.

Attributes

An *attribute* is a (partial) function with domain $\mathbf{Sub}$, $\mathbf{Obj}$ or $\mathbf{Sub} \times \mathbf{Obj}$. Attributes can be used to enforce access control rules. Standard attributes [331, 333] are only functions with domain $\mathbf{Sub}$ and $\mathbf{Obj}$, known as subject attributes and object attributes, respectively. Here, we slightly extend this notion for convenience of exposition.

A widely used attribute is the access matrix, initially proposed in [175] and later refined by [288–290].

FIGURE 7.2: To exercise right r on object o, subject s sends a request (s, o, r) to the access control system.

Definition 6 (Access matrix)**.** An access matrix is a function $M_{\text{acc}} : \mathbf{Sub} \times \mathbf{Obj} \to \mathcal{P}(\mathbf{Rt})$.

Here, $M_{\text{acc}}[s, o]$ records rights granted to subject s to access object o.

Requests

When subject s attempts to exercise right r on object o, it issues an access *request* (s, o, r) to the access control system. The request will then be handled by the access control system according to a set of rules (to be introduced below), as illustrated in Figure 7.2. Let $\mathbf{Req} = \mathbf{Sub} \times \mathbf{Obj} \times \mathbf{Rt}$ denote the set of requests.

Rules

A *rule* describes how the system handles a request $(s, o, r) \in \mathbf{Req}$. The most basic rule in access control is the authorisation rule. Upon receiving a request, the system will determine whether to authorise the received access request according to a function *Auth*.

Definition 7 (Authorisation)**.** An authorisation rule is a function $Auth : \mathbf{Req} \to \{true, false\}$.

The simplest authorisation rule is based on the access matrix: $Auth(s, o, r) \equiv r \in M_{\text{acc}}[s, o]$.

Another rule we consider (see Section 8.3) is the post-update rule. After a request is authorised, and before the next request is handled, several post-update operations can be performed on attributes according to the partial function *Post*, for future authorisation decisions.

Definition 8 (Post-update)**.** A post-update rule is a partial function *Post* such that for request $(s, o, r) \in \mathbf{Req}$ and attribute $f \in \mathbf{Attr}$, $Post(s, o, r)(f) = f'$ for some f' of the same function type as f.

Intuitively, after authorising a request (s, o, r), rule *Post* updates f to f'. For example, let $u \in \mathbf{Obj}$ be a fixed object. For an attribute $f : \mathbf{Obj} \to \{0, 1\}$, a possible post-update rule can be $Post(s, o, r)(f) \equiv f'_u$, where $f'_u[u] = 1 - f[u]$ and $f'_u[o] = f[o]$ for $o \neq u$. This *Post* means after a request (s, o, r) is authorised, $f[u]$ is updated to be $1 - f[u]$ and other $f[o]$ remain unchanged.

Systems

Finally, let us use a 5-tuple $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule})$ to denote an access control system, where **Attr** and **Rule** denote the sets of attributes and rules, respectively. When the context is clear, we simply say system instead of access control system.

7.2.2 Execution Model

Next we describe the execution of an access control system, which involves concurrency. We assume the requests in the system are atomic. During an execution, the system receives a sequence of requests from subjects and enforces the access control rules accordingly. To describe the non-deterministic ordering of requests made by different subjects, we use the notion of a scheduler (like in e.g., [334, 335]).

Definition 9 (Scheduler)**.** A scheduler of the system is a function $S : \bigcup_{k=0}^{\infty} \mathbf{Req}^k \to \mathbf{Sub}$.

Intuitively, given any finite sequence of requests, the scheduler S determines the next subject s to make a request. A scheduler can be an adversary: to prove a safety property that something bad (e.g., security breach) never happens in a system, we need to consider it against all schedulers.

To describe valid sequences of requests under a scheduler, we introduce the notion of a history.

Definition 10 (History)**.** Given a scheduler S of the system, a history is a (finite or infinite) sequence of access requests $\alpha = \alpha(0), \alpha(1), \ldots$ such that for all $t \in \mathbb{N}$, if $\alpha(t) = (s, o, r)$ then $s = S(\alpha(0), \ldots, \alpha(t-1))$. Further, a history α is said to be *authorised* if $Auth(\alpha(t)) = \mathit{true}$ for all $t \in \mathbb{N}$.

The scheduler S alone does not fully determines the history of an execution. While it determines the next subject s to make a request (s, o, r), the object o and the right r in this request are determined by the behaviour of the subject s, which is specified by a program P_s (or any other computational model). Let us collect all programs P_s for $s \in \mathbf{Sub}$ and the initial state of objects into a program P. Then, we can use (S, P) to denote an execution of the system.

Each execution (S, P) generates a history (or a probabilistic distribution over histories, if P is probabilistic). The actual generation is determined by the explicit semantics of the program and requests. For example, consider a system with $\mathbf{Sub} = \{s\}$ and $\mathbf{Obj} = \{o\}$. Suppose program $P_s \equiv o := o + 1$, and o is initialised to 0. In this case, if $\mathbf{Rt} = \{\texttt{read}, \texttt{write}\}$, then the history generated could be $(s, o, \texttt{read}), (s, o, \texttt{write})$; if $\mathbf{Rt} = \{\texttt{inc}\}$, where `inc` means the ability to increment the value of the register by 1, then the history generated could be $(s, o, \texttt{inc})$.

For simplicity, we do not bother formalising such generation, because our focus is the access control system. Nevertheless, we can define the equivalence between two systems with respect to authorised histories (see Definition 10).

Definition 11 (Equivalent systems)**.** Two systems $\mathcal{A}$ and $\mathcal{A}'$ are said to be equivalent, denoted by $\mathcal{A} \simeq \mathcal{A}'$, if for any program P of concern and any scheduler S:

- (S, P) can generate (valid) histories in both $\mathcal{A}$ and $\mathcal{A}'$; and
- The histories generated by (S, P) in $\mathcal{A}$ are authorised iff the histories generated by (S, P) in $\mathcal{A}'$ are authorised.

An *access control model* is a family of systems. An important metric to evaluate an access control model is its *flexibility*. While in general, the flexibility cannot be characterised by a quantity, we can compare the flexibility of two models.

Definition 12 (Flexibility)**.** An access control model M is said to be less flexible than another M', denoted by $\mathsf{M} \leq \mathsf{M}'$, if for any system $\mathcal{A} \in \mathsf{M}$, there exists a system $\mathcal{A}' \in \mathsf{M}'$ such that $\mathcal{A} \simeq \mathcal{A}'$. Further, M is said to be strictly less flexible than M', denoted by $\mathsf{M} < \mathsf{M}'$, if $\mathsf{M} \leq \mathsf{M}'$ and $\mathsf{M}' \not\leq \mathsf{M}$.

7.3 Scenario: Threat from Quantum Entanglement

To answer Question 1, we reveal a threat from quantum entanglement by presenting an explicit scenario of a security breach when a classically secure access control system is straightforwardly adapted to the quantum setting. As computer security typically concerns the worst case, this threat demonstrates the inadequacy of existing access control models for quantum systems.

The key insight behind this scenario is that *entanglement (even without communication) violates Mermin inequality* [176], a multipartite variant of the celebrated Bell inequality [324–328]. This implies that the threat we reveal is *fundamentally quantum*, rather than coming from computational assumptions or side channels.

7.3.1 Problem Setting

Let us consider a system $\mathcal{S} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule})$ with

- $\mathbf{Sub} = \{u, v, w_1, \ldots, w_n\}$,
- $\mathbf{Obj} = \{A, B, C_1, \ldots, C_n, M_{\text{acc}}\}$,
- $\mathbf{Rt} = \{\texttt{read}, \texttt{write}, \texttt{flip}, \texttt{all}\}$,
- $\mathbf{Attr} = \{M_{\text{acc}}, L\}$, and

	The program P_v
	Initial: $M_{\text{acc}} = M_0$
1	Write $M_{\text{acc}} \leftarrow M_1$
2	Generate uniformly at random an n-bit string $x = (x_1, \ldots, x_n) \in \{x \in \{0,1\}^n : \lvert x\rvert \bmod 2 = 0\}$
3	For $j = 1$ to n, write $C_j^1 \leftarrow x_j$, the first bit of C_j
4	Read $a \leftarrow A$ and calculate $b = \left(\frac{\lvert x\rvert}{2} \bmod 2\right) \oplus a$
5	Write $B \leftarrow b$
6	Write $M_{\text{acc}} \leftarrow M_2$

FIGURE 7.3: The program P_v that describes the behaviour of user v. Here, matrices M_0, M_1 and M_2 are shown in Figures 7.4 to 7.6, respectively.

- $\mathbf{Rule} = \{Auth\}$.

Here, $L : \mathbf{Sub} \to \mathbf{Int}$ with $\mathbf{Int}$ being the set of (bounded) integers, and $Auth(s, o, r) \equiv r \in M_{\text{acc}}[s, o]$.

The ingredients of this system are explained as follows.

- In **Sub**: $u, v, w_1, \ldots, w_n$ are all users.
- In **Obj**: A, B are bit registers and $C_1, \ldots, C_n$ are integer registers. By a slight abuse of notation, M_{acc} represents an integer register[1] storing the access matrix M_{acc}.
- In **Rt**: `read` and `write` correspond to standard read and write operations. Exercising `flip` means changing every bit 0 to 1 and 1 to 0 in a register. The right `all` means full access, allowing one to perform any operations.
- The **Attr** consists of only two elements: (i) the access matrix M_{acc} in Definition 6; and (ii) an attribute $L : \mathbf{Sub} \to \mathbf{Int}$. Here, for each user $s \in \mathbf{Sub}$, $L[s]$ denotes the local memory of s, used to store temporary results for exercising rights `read` and `write`.[2] Only s can access $L[s]$. It should be noticed that L is not in **Obj** and thus not guarded by the access control.

The behaviour of v is fixed and shown as a program P_v in Figure 7.3. We should notice that it is actually a probabilistic program, as in Line 2, v samples from a random

[1] Here, using an integer register to store the whole matrix M_{acc} is solely for simplifying the presentation of results in Section 7.3. In practice and later in Section 8.1, we actually use multiple register (or memory locations) to store a matrix (that represents an attribute), where each register (or location) can store an entry of the matrix.

[2] In the literature, the local memory is often not explicitly stated as an attribute. In this thesis, we include L as an attribute for the following two reasons: L is useful in the statement and analysis of system security (see Theorem 2); and whether L is classical or quantum in a system with quantum objects needs to be explicitly specified (see Sections 7.3.3 and 8.1).

	A	B	C_1	C_2	$\dots$	C_n	M_{acc}
u	`all`						
v							`all`
w_1			`all`	`all`	$\dots$	`all`	
w_2			`all`	`all`	$\dots$	`all`	
$\vdots$			$\vdots$	$\vdots$	$\ddots$	$\vdots$	
w_n			`all`	`all`	$\dots$	`all`	

FIGURE 7.4: Matrix M_0.

	A	B	C_1	C_2	$\dots$	C_n	M_{acc}
u							
v	`read`	`write`	`all`	`all`	$\dots$	`all`	`all`
w_1		`flip`	`all`				
w_2		`flip`		`all`			
$\vdots$		$\vdots$			$\ddots$		
w_n		`flip`				`all`	

FIGURE 7.5: Matrix M_1.

distribution. We explain what accesses are permitted when M_{acc} is M_0, M_1, and M_2 in P_v, respectively:

- $M_{\text{acc}} = M_0$: User u can write 1 bit of confidential information into A. Other users $w_1, \dots, w_n$ can communicate through the shared $C_1, \dots, C_n$ and devise a strategy.
- $M_{\text{acc}} = M_1$: User v can read the secret of u from A and access B, $C_1, \dots, C_n$. Each user w_j can only access C_j and flip B. These w_j cannot communicate with each other, but they can exploit any pre-determined strategy.

	A	B	C_1	C_2	$\dots$	C_n	M_{acc}
u							
v							`all`
w_1		`read`	`all`				
w_2		`read`		`all`			
$\vdots$		$\vdots$			$\ddots$		
w_n		`read`				`all`	

FIGURE 7.6: Matrix M_2.

- $M_{\text{acc}} = M_2$: Each w_j can access C_j and read B. These w_j still cannot communicate with each other.

Interpretation

Intuitively, in the context of Question 1, u is the user concerned about the security, v is a system user with trusted and fixed behaviour, and $w_1, \ldots, w_n$ are other users of the system. Our *security policy* is to prevent the confidential information of user u from leaking to other users $w_1, \ldots, w_n$.

7.3.2 Security in the Classical Case

We can prove: if all objects in system $\mathcal{S}$ (see Section 7.3.1) are classical, then the amount of information leaked from u to any other w_j is exponentially small in n. As a notation convention, for a register X, we use $X(t)$ to represent its value at time t.

Theorem 2 (Security in the classical case)**.** *Let $n \geq 5 \in \mathbb{N}$. If all objects in system $\mathcal{S}$ are classical, then the confidential information of user u can only leak with negligible probability. Specifically, for any execution (S, P) with P_v described in Figure 7.3, any time $t_u, t_w \in \mathbb{N}$, and any $j \in [n]$, the mutual information*

$$I\big(A(t_u); \text{Obs}\big(w_j, t_w\big)\big) \leq 2^{-(n-7)/2}, \tag{7.1}$$

where $\text{Obs}\big(w_j, t\big) := \big\{o \in \textbf{Obj} : \texttt{read} \in M_{\text{acc}}\big[w_j, o\big](t)\big\} \cup \big\{L\big[w_j\big](t)\big\}$ is what w_j can observe at time t.

Intuition

In the context of Question 1, this proof guarantees that user u can safely write confidential information into the system, without (significantly) leaking it to other users $w_1, \ldots, w_n$. Even for a small system with 100 users, any other user can only obtain approximately 10^{-14} bits of confidential information, an amount practically negligible.

Techniques

The proof of Theorem 2 essentially relies on the following variant of Mermin inequality [176].

Lemma 6 (A variant of Mermin inequality [176])**.** *Let $n \in \mathbb{N}$ be a fixed number. Let $\mathcal{X}_b := \big\{x \in \{0,1\}^n : |x| \bmod 2 = b\big\}$, where $b \in \{0,1\}$. Let $\mathcal{Y} = \{0,1\}^n$. For any fixed $b \in \{0,1\}$, consider random variable $X = X_1, \ldots, X_n$ chosen uniformly at random from $\mathcal{X}_b$, any*

$Y = Y_1, \ldots, Y_n$ in $\mathcal{Y}$, and any $\Lambda = \Lambda_1, \ldots, \Lambda_n$ independent of X such that

$$\mathbf{Pr}[Y = y \mid X = x, \Lambda = \lambda] = \prod_{j=1}^{n} \mathbf{Pr}\left[Y_j = x_j \,\middle|\, X_j = x_j, \Lambda_j = \lambda_j\right],$$

Then we have

$$\left|\mathbf{E}\left[(-1)^{|X|/2+|Y|+b/2}\right]\right| \leq 2^{-n/2+1}. \tag{7.2}$$

The original Mermin inequality in [176] is the special case of $b = 0$ in Lemma 6. Mermin inequality extends the celebrated Bell inequality [324–328] to the n-party case and reveals the fundamental difference between classical and quantum mechanics.

For readability, we only provide a proof sketch of Theorem 2 below. The full proof is deferred to Section 7.4.3.

Proof sketch of Theorem 2. Intuitively, in system $\mathcal{S}$ (see Section 7.3.1), the "best" strategy for users w_j to learn the confidential information of u is learning the value $\frac{|x|}{2} \bmod 2$ in Figure 7.3 and then taking the $\oplus$ operation with b in Figure 7.3 to exactly recover a. However, the behaviours of all w_j are guarded by the access matrix M_{acc}, and this strategy turns out to only have a negligible success probability, essentially due to the variant of Mermin inequality in Lemma 6.

Now we explain how to formalise the above intuition. Consider any execution (S, P). By analysing how M_{acc} constrains information flow, proving Equation (7.1) can be first reduced to proving the special case of $t_u = t_1$ and $t_w \geq t_2 + 1$, where t_1 and t_2 are time points after the write requests in Lines 1 and 6 of P_v (see Figure 7.3) are issued, respectively. Denote $C_j, L\left[w_j\right]$ by D_j. Using the symmetry of M_{acc} (with respect to different w_j), we can further reduce our goal to proving

$$\frac{\mathbf{Pr}[A(t_1) = a \mid B(t_w) = b, D_1(t_w) = d]}{\mathbf{Pr}[A(t_1) = a]} \approx 1 \tag{7.3}$$

for any bit a, b and integer d. Here, the degree of the approximation $\approx$ is related to the RHS of Equation (7.1).

The remaining analysis largely relies on techniques in probabilistic graphical models. First, we identify several time points and random variables of concern. For example, for each $j \in [n]$, let $t_{v,j}$ be the time point after the write request in Line 3 of P_v is issued. Then, $C_j^1\left(t_{v,j}\right)$ is equal to the value x_j chosen by v in P_v. Second, we analyse relations between these random variables, based on the program P_v, matrices M_0, M_1, M_2, and temporal ordering of requests. These relations are abstracted into a graph and used to break down Equation (7.3), through decomposition of joint probability distributions, into terms closer to the form Equation (7.2) in Lemma 6. Finally, we can obtain an upper

bound $2^{-(n-7)/2}$ on the degree of approximation in Equation (7.3), and the conclusion follows. □

7.3.3 Security Breach in the Quantum Case

To answer Question 1, let us examine the scenario when a system integrates quantum computing services, which, in general, augments the access to a classical register with extra access to a quantum register. For our system S (see Section 7.3.1), we consider its registers $C_1, \ldots, C_n$ to now be quantum registers (which can simulate classical-quantum registers).

Lifting to Quantum

We need to consider how to properly lift[3] system $\mathcal{S}$ to the quantum setting; i.e., how to interpret attributes and rules in $\mathcal{S}$. Specifically, let us focus on how to lift a request $(s, X, \texttt{all})$ to the quantum setting. One might first try to interpret it as: "user s can perform any quantum operation on quantum register X". However, this interpretation forbids any quantum entanglement between registers. Since entanglement is believed to be the source of quantum advantages (e.g., [329]) for many quantum algorithms [12–14, 216], such lifting is definitely an unsatisfactory choice.

The remaining natural lifting is to interpret:

- (LF) Request $(s, X, \texttt{all})$ means user s can perform any quantum operation on the composite system of quantum register X and the local memory $L[s]$ of s.

This lifting (LF) implicitly assumes the local memories of subjects are also quantum; i.e., $L : \mathbf{Sub} \to \mathcal{H}_{\mathbf{Int}}$ with $\mathcal{H}_{\mathbf{Int}}$ being the Hilbert space lifted from **Int**. Under (LF), entanglement can be generated between registers. For example, in system $\mathcal{S}$, when $M_{\text{acc}} = M_0$, user w_1 can generate an EPR state $\frac{1}{\sqrt{2}}\left(|0\rangle_{C_1}|1\rangle_{C_2} + |1\rangle_{C_1}|1\rangle_{C_2}\right)$ in C_1 and C_2 (technically, their first qubits), by first performing a Hadamard H gate on C_1, followed by a *CNOT* gate on C_1 and $L[w_1]$, and finally a *SWAP* gate between C_2 and $L[w_1]$. However, (LF) also turns out to be unsatisfactory, because it can actually lead to a *security breach*: the security guaranteed by Theorem 2 will be broken in the quantum case.

Theorem 3 (Security breach in the quantum case)**.** *If $C_1, \ldots, C_n$ in system $\mathcal{S}$ are quantum registers and we adopt the lifting (LF), then the confidential information of user u can be leaked with certainty in the worst case. Specifically, there exists an execution (S, P) with P_v described in Figure 7.3 such that the mutual information*

$$I(A(t_1); \text{Obs}(w_1, t_2)) = 1,$$

[3]The terms "adapt" and "lift" will be used interchangeably.

	The program P_{w_j}
1	If $j = 1$, prepare the state $\|\mathrm{GHZ}(n)\rangle_{C^2} = \frac{1}{\sqrt{2}}\left(\|0\rangle_{C_1^2}\ldots\|0\rangle_{C_n^2} + \|1\rangle_{C_1^2}\ldots\|1\rangle_{C_n^2}\right)$
2	If $C_j^1 = 1$, apply the phase gate $\sqrt{Z}$ to C_j^2
3	Apply the Hadamard gate H to C_j^2
4	Measure C_j^2 in the computational basis to obtain outcome b_j
5	If $b_j = 1$, flip B

FIGURE 7.7: The program P_{w_j} that describes the behaviour of each user w_j, in an attempt to learn the confidential information of user u. Here, C_j^k represents the k^{th} qubit of C_j, and $C^2 = C_1^2, \ldots, C_n^2$.

where t_1, t_2 are time points after the write requests in Line 1 and 6 in Figure 7.3 are issued, respectively. Here, $\mathrm{Obs}(\cdot,\cdot)$ *is defined in Theorem 2.*

Entanglement — Source of Threats

Theorems 2 and 3 together reveal a ***threat fundamentally from entanglement***. It is worth noting that the security breach in Theorem 3 is *NOT* due to an additional communication channel created by entanglement, as the access matrix M_{acc} of the system does not change. Indeed, it is well-known that entanglement cannot enable information transmission between users without direct communication. Instead, the insecurity proof relies on how entanglement violates Mermin inequality [176]. This implies the threat has a *quantum nature* and is not restricted to the specific system considered here.

Proof of Theorem 3. Note that

$$\mathrm{Obs}(w_1, t_2) = B(t_2), C_j(t_2), L[w_1](t_2).$$

It suffices to show there exists an execution (S, P) such that $\mathbf{Pr}[B(t_2) = A(t_1)] = 1$. The program P (in particular, P_{w_j}) we construct exactly follows the quantum strategy for Mermin n-player game [176, 336], which leads to a violation of Mermin inequality in the quantum setting.

Let us first construct the program P. The program P_{w_j} that describes the behaviour of each w_j is shown in Figure 7.7. Note that when $M_{\text{acc}} = M_1$, given the lifting (LF), Line 1 in Figure 7.7 can be executed by (a) swapping the content of C_j for each $j \in [n]$ into the local memory $L[w_1]$; (b) preparing the state $|\mathrm{GHZ}(n)\rangle$ in $L[w_1]$; and (c) swapping back the content of $L[w_1]$ to C_j for each j, which moves the GHZ state to C^2. Without loss of generality, let P_u consist of a single write $A \leftarrow a$, where $a \in \{0, 1\}$ is the confidential information of u.

Next we construct the scheduler S. We take $t_1 = 2$ and $t_2 = 8n + 5$. S is defined such that for $t \in \mathbb{N}$, $S(\alpha(0), \ldots, \alpha(t-1)) = s(t)$. Below, we specify $s(t)$ and describe the associated user behaviours at time t.

- $s(0) = u$: write 1 bit of confidential information into A.
- $s(1) = v$: execute Line 1 in Figure 7.3 to modify M_{acc}.
- $s(2) = \ldots = s(2n+1) = w_1$: execute Line 1 in Figure 7.7.
- $s(2n+2) = \ldots = s(3n+3) = v$: execute Lines 2–5 in Figure 7.3.
- For $k = 0$ to 4, and $j \in [n]$, $s((k+3)n + j + 3) = w_j$: execute Lines 2–5 in Figure 7.7.
- $s(8n+4) = v$: execute Line 6 in Figure 7.3 to modify M_{acc}.
- $s(8n+5) = w_1$: read the value in B.

In the above, we implicitly fix how to generate requests from the program P (see Definition 10 and discussion there). The time points above (e.g., $2n+1$, $3n+3$) are chosen regarding this specific generation. For example, w_1 executes Line 2 in Figure 7.7 through two requests $\left(w_1, C_j^1, \texttt{read}\right)$, $\left(w_1, C_j^2, \texttt{all}\right)$, at time $t = 3n+4$ and $t = 3n+5$.

Let us verify that the execution (S, P) constructed above yields $\mathbf{Pr}[B(t_2) = A(t_1)] = 1$. In our system, only C^2 will be in quantum superposition. As operations on C^1 are actually classical, C^1 can be still regarded as a classical random variable, for simplicity. Let $E := \left|C^1(3n+4)\right|/2$ and

$$F := \left|\left\{t \in [3n+4, 8n+3] : \alpha(t) = \left(w_j, B, \texttt{flip}\right), j \in [n]\right\}\right| \bmod 2.$$

From P_v in Figure 7.3 and P_{w_j} in Figure 7.7, we have $B(t_2) = E \oplus F \oplus A(t_1)$.

Now it suffices to show $\mathbf{Pr}[E = F] = 1$. For $b \in \{0, 1\}$, define

$$|\psi_b\rangle := \frac{1}{\sqrt{2}} H^{\otimes n}\left(|0\rangle^{\otimes n} + (-1)^b |1\rangle^{\otimes n}\right) = \frac{1}{2^{(n-1)/2}} \sum_{|y| \bmod 2 = b} |y\rangle .$$

It is easy to see that the state of $C^2(6n+4)$ (before each w_j executes Line 4 in Figure 7.7) is $|\psi_E\rangle$. Thus, we obtain

$$\mathbf{Pr}[F = b \,|\, E = b] = \sum_{|y| \bmod 2 = b} |\langle y|\psi_b\rangle|^2 = 1. \qquad \square$$

7.4 Proof Details

In this section, we present the proof details for the security of system $\mathcal{S}$ described in Section 7.3.

7.4.1 Background on Probabilistic Graphical Models

We first briefly introduce the notations and tools in probabilistic graphical models, which will be used in Section 7.4.3 for proving Theorem 2. The reader is referred to the textbook [337] for a more thorough introduction.

Probabilities

For a random variable A, we use $\mathbf{Pr}[A = a] \in [0, 1]$ to denote the probability of A taking the value a. It holds that $\sum_a \mathbf{Pr}[A = a] = 1$. The joint probability

$$\mathbf{Pr}[A = a, B = b] = \mathbf{Pr}[A = a \cap B = b]$$

denotes the probability of A taking the value a and another random variable B taking the value b. For simplicity, sometimes we simply write A, B for the random variable (A, B).

Let the conditional probability $\mathbf{Pr}[A = a \mid B = b]$ be the probability of A taking the value a given that B takes the value b. The joint probability $\mathbf{Pr}[A = a, B = b]$ can be calculated by

$$\mathbf{Pr}[A = a, B = b] = \mathbf{Pr}[A = a \mid B = b] \cdot \mathbf{Pr}[B = b]. \tag{7.4}$$

By summing over a, we have the following decomposition of $\mathbf{Pr}[A = a]$ by conditioning on different $B = b$:

$$\mathbf{Pr}[A = a] = \sum_b \mathbf{Pr}[A = a \mid B = b] \cdot \mathbf{Pr}[B = b]. \tag{7.5}$$

A useful generalisation of Equation (7.4) is the following *chain rule*:

$$\begin{aligned} &\mathbf{Pr}[A_1 = a_1, \ldots, A_n = a_n] \\ =\ &\mathbf{Pr}[A_n = a_n \mid A_{n-1} = a_{n-1}, \ldots, A_1 = a_1] \\ &\qquad \ldots \mathbf{Pr}[A_2 = a_2 \mid A_1 = a_1] \, \mathbf{Pr}[A_1 = a_1]. \end{aligned} \tag{7.6}$$

If $\mathbf{Pr}[A = a \mid B = b] = \mathbf{Pr}[B = b]$ whenever $\mathbf{Pr}[B = b] > 0$, then A and B are said to be *independent*, denoted by $A \perp\!\!\!\perp B$. More generally, if

$$\mathbf{Pr}[A = a \mid B = b, C = c] = \mathbf{Pr}[B = b \mid C = c]$$

whenever $\mathbf{Pr}[B = b, C = c] > 0$, then A and B are said to be *conditionally independent* given C, denoted by $A \perp\!\!\!\perp B \mid C$. Intuitively, it means that if we know the value of C, we cannot learn extra information about A from learning the value of B. We mention two useful properties of conditional independence:

- (Symmetry) $A \perp\!\!\!\perp B \mid C$ implies $B \perp\!\!\!\perp A \mid C$.

- (Decomposition) $A \perp\!\!\!\perp B, C \mid D$ implies $A \perp\!\!\!\perp B \mid D$.

The concept of conditional independence plays an important role in probabilistic graphical models.

As usual, we use $\mathbf{E}[A] = \sum_a a \cdot \mathbf{Pr}[A = a]$ to denote the expectation of A.

Probabilistic Graphical Models

A graph can be used to represent the relations between multiple random variables. Each vertex represents a random variable. A *directed* edge between random variables represents a *causal relation*; i.e., $A \to B$ means that A can influence B. A *bidirected* edge between random variables represent a *mutual dependence*, often due to an unobserved common cause; i.e., $A \leftrightarrow B$ means that A and B are dependent. For example, in Figure 7.8, the directed edge $A(t_1) \to B(t_v)$ comes from that $B(t_v)$ is written by user v, who reads the secret $A(t_1)$ written by user u; the undirected edges between $X_1, \ldots, X_n$ comes from that the values of these X_j are randomly drawn by user v from a distribution (see Figure 7.3). Recall that in Figure 7.8, vertices within a gray area are fully connected by bidirected edges.

Based on the graph structure, the joint probability of random variables corresponding to all vertices can be decomposed into a product of conditional probabilities, In particular, we can refine the chain rule in Equation (7.6) to

$$\begin{aligned} &\mathbf{Pr}[A_1 = a_1, \ldots, A_n = a_n] \\ =&\prod_j \mathbf{Pr}\left[A_j = a_j \,\middle|\, \forall k, (A_k \to A_j \wedge k < j) \Rightarrow (A_k = a_k)\right], \end{aligned} \tag{7.7}$$

where $A_k \to A_j$ denotes a directed edge. We have used this decomposition rule to decompose Equation (7.15) into Equation (7.16) in Section 7.4.3.

Moreover, the graph structure allow us to conveniently infer the conditional independence of random variables. Let X, Y, Z be sets of random variables. X is said to be d-separated from Y by Z if, for any (undirected) path p from a node $A \in X$ to a node $B \in Y$, one of the following conditions hold:

- p contains $C \to D \to E$ or $C \leftarrow D \to E$ with $D \in Z$; or
- p contains $C \to D \leftarrow E$ such that for any F, if $D \to^* F$, then $F \notin Z$.

If X is *d-separated* from Y by Z, then we have $X \perp\!\!\!\perp Y \mid Z$. In Section 7.4.3, we have derived several conditional independence relations from Figure 7.8 using this notion.

7.4.2 Proof of Lemma 6

Now we provide a proof of the variant of Mermin inequality in Lemma 6 for completeness. The proof idea is almost the same as the one in [176].

Proof of Lemma 6. First note that

$$\mathbf{E}\Big[(-1)^{|X|/2+|Y|+b/2}\Big] = \sum_{\lambda} \mathbf{Pr}[\Lambda=\lambda]\,\mathbf{E}\Big[(-1)^{|X|/2+|Y|+b/2}\,\Big|\,\Lambda=\lambda\Big].$$

To prove the target inequality Equation (7.2), it suffices to prove that

$$\Big|\mathbf{E}\Big[(-1)^{|X|/2+|Y|+b/2}\,\Big|\,\Lambda=\lambda\Big]\Big| \le 2^{-n/2+1}. \tag{7.8}$$

Let us consider the quantity

$$F_b := \begin{cases} \mathrm{Re}\Big(\prod_{j\in[n]}\big(\mathbf{E}\Big[(-1)^{Y_j}\,\Big|\,X_j=0,\Lambda_j=\lambda_j\Big]+ \\ \qquad i\,\mathbf{E}\Big[(-1)^{Y_j}\,\Big|\,X_j=1,\Lambda_j=\lambda_j\Big]\big)\Big), & b=0, \\ -\mathrm{Im}\Big(\prod_{j\in[n]}\big(\mathbf{E}\Big[(-1)^{Y_j}\,\Big|\,X_j=0,\Lambda_j=\lambda_j\Big]+ \\ \qquad \mathbf{E}\Big[(-1)^{Y_j}\,\Big|\,X_j=1,\Lambda_j=\lambda_j\Big]\big)\Big), & b=1. \end{cases} \tag{7.9}$$

Since each term $\mathbf{E}\Big[(-1)^{Y_j}\,\Big|\,X_j=a,\Lambda_j=\lambda_j\Big] \in [-1,1]$, it is easy to see that

$$|F_b| \le \left(\sqrt{2}\right)^n = 2^{n/2}. \tag{7.10}$$

On the other hand, by calculation, we obtain

$$F_b = \sum_{x\in\mathcal{X}_b} (-1)^{|x|/2+b/2} \prod_j \mathbf{E}\Big[(-1)^{Y_j}\,\Big|\,X_j=x_j,\Lambda_j=\lambda_j\Big]. \tag{7.11}$$

Since

$$\begin{aligned} &\mathbf{E}\Big[(-1)^{Y_j}\,\Big|\,X_j=x_j,\Lambda_j=\lambda_j\Big] \\ &= \mathbf{Pr}\big[Y_j=0\,\big|\,X_j=x_j,\Lambda_j=\lambda_j\big] - \mathbf{Pr}\big[Y_j=1\,\big|\,X_j=x_j,\Lambda_j=\lambda_j\big], \end{aligned}$$

we further have

$$\textit{Equation (7.11)} = \sum_{x\in\mathcal{X}_b} (-1)^{|x|/2+b/2} \sum_{y\in\mathcal{Y}} (-1)^{|y|} \mathbf{Pr}[Y = y \mid X = x, \Lambda = \lambda] \tag{7.12}$$

$$= \sum_{x\in\mathcal{X}_b} \mathbf{E}\Big[(-1)^{|x|/2+|Y|+b/2} \,\Big|\, X = x, \Lambda = \lambda\Big]. \tag{7.13}$$

As Λ is independent of X, $\mathbf{Pr}[X = x \mid \Lambda = \lambda] = \mathbf{Pr}[X = x] = \frac{1}{2^{n-1}}$ for any $x \in \mathcal{X}$. Consequently,

$$\begin{aligned}\textit{Equation (7.13)} &= 2^{n-1} \sum_{x\in\mathcal{X}_b} \mathbf{E}\Big[(-1)^{|x|/2+|Y|+b/2} \,\Big|\, X = x, \Lambda = \lambda\Big] \mathbf{Pr}[X = x \mid \Lambda = \lambda] \\ &= 2^{n-1}\, \mathbf{E}\Big[(-1)^{|X|/2+|Y|+b/2} \,\Big|\, \Lambda = \lambda\Big].\end{aligned}$$

Finally, combining the above with Equation (7.10) yields Equation (7.8). □

7.4.3 Proof of Theorem 2

Finally, we present the full proof of Theorem 2.

Proof of Theorem 2. Let us fix any scheduler S of the system. Since we allow the program P to be probabilistic (e.g., P_v in Figure 7.3 is probabilistic), the value of any register can be regarded as a random variable. For example, $A(t)$, the value of the register A at time t, is a random variable. Similarly, $\alpha(t)$, the request at time t in the history, can also be seen as a random variable.

Let us first identify several time points $t_1, t_{v,j}, t_v, t_2$ with respect to program P_v in Figure 7.3, by supposing:

- At $t = t_1 - 1$: v issues the write request in Line 1
- At $t = t_{v,j} - 1$: v issues the write request for j in Line 3.
- At $t = t_v - 1$: v issues the write request in Line 5.
- At $t = t_2 - 1$: v issues the write request in Line 6.

For convenience, let us denote $D_j := C_j, L[w_j]$. Note that from M_0, M_1, M_2, we have

$$\mathrm{Obs}(w_j, t) = \begin{cases} D_j(t), & t \le t_2, \\ B(t), D_j(t), & t \ge t_2 + 1. \end{cases}$$

where we denote a set by an ordered list and will use this convention throughout the proof.

We can restrict $t_u = t_1$ and $t_w \geq t_2 + 1$ in Theorem 2. This is because if $t_u \neq t_1$ and $A(t_u) \neq A(t_1)$, or if $t_w \leq t_2$, then there is no information flow from $A(t_u)$ to $\mathrm{Obs}(w_j, t_w)$ and thus $A(t_u) \perp\!\!\!\perp \mathrm{Obs}(w_j, t_w)$. Moreover, since the access matrix M_{acc} (taking values in M_0, M_1, M_2) is always symmetric for all w_j, we can only prove for the case $j = 1$ without loss of generality. Now proving Theorem 2 reduces to proving

$$I(A(t_1); B(t_w), D_1(t_w)) \leq 2^{-(n-7)/2} \tag{7.14}$$

for any $t_w \geq t_2 + 1$.

We identify and define all random variables of concern in our proof as follows.

- $A(t_1)$ stores the secret information written by u into A.
- Let $X_j := C_j^1(t_{v,j})$, where C_j^1 denotes the first bit of C_j. Let $\Lambda_j := C_j^{\bar{1}}(t_{v,j}), L[w_j](t_{v,j})$, where $C_j^{\bar{1}}$ denotes the remaining bits (except for the first bit) of C_j.
- $B(t_v)$ stores the information written by v into B, which also encodes the secret information of u.
- For each $j \in [n]$, let

  $$Y_j := \left|\left\{t \in [t_v + 1, t_2 - 1] : \alpha(t) = (w_j, \texttt{flip}, B)\right\}\right| \bmod 2$$

  denote the parity of the number of `flip` exercised by w_j on B for $t \in [t_v + 1, t_2 - 1]$.
- $B(t_2)$ is obtained from $B(t_v)$ after each w_j exercises a number of `flip`.
- $B(t_w)$ and $D_j(t_w)$ contain all information accessible to w_j at time $t_w \geq t_2 + 1$.

For convenience, we also define the following notations:

- Let $X := X_1, \ldots, X_n$.
- Let $X_{\bar{j}} := X_1, \ldots, X_{j-1}, X_{j+1}, \ldots, X_n$ for $j \in [n]$.
- Let $X' = X_{\bar{1}}$.

The above notations apply when X is replaced by Y or Λ.

By the program P_v of v described in Figure 7.3, the change of access matrix $M_{\mathrm{acc}}(t)$ in Figures 7.4 to 7.6, and the temporal order of requests, the relations between concerned random variables can be summarised in the probabilistic graphical model in Figure 7.8.

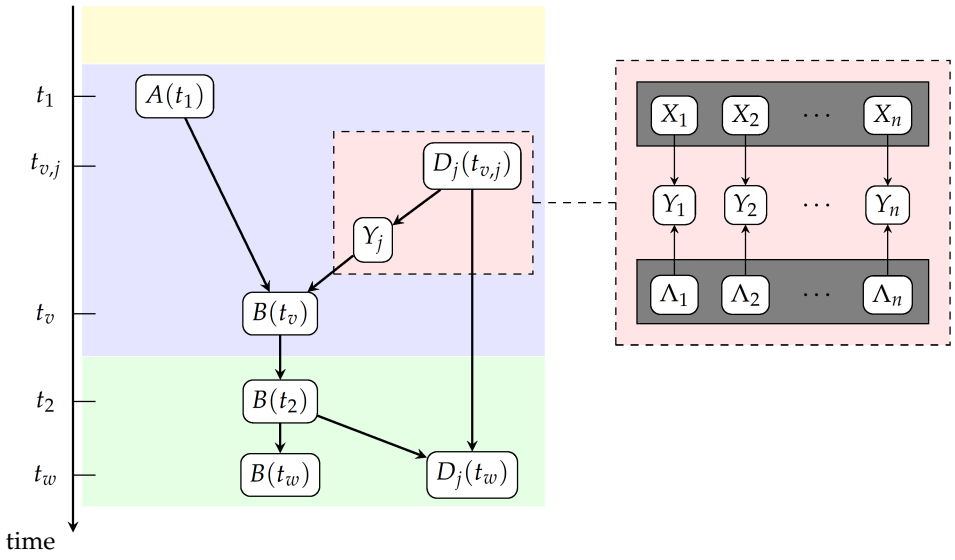


FIGURE 7.8: Probabilistic graphical model of concerned random variables in the system described in Section 7.3.1. As usual, a directed edge represents a causal relation, and a bidirected edge represents a mutual dependence. The LHS depicts the relations between $A(t_1), D_j(t_{v,j}), Y_j, B(t_v), B(t_2), B(t_w), D_j(t_w)$. The RHS depicts the relations between X, Y, Λ, where nodes in each gray area (e.g., $X_1, \ldots, X_n$) are fully connected (by bidirected edges).

Let us also fix some $a, b \in \{0,1\}$ and integer d. From Figure 7.8, we can decompose the joint probability distribution of these concerned random variables as

$$\mathbf{Pr}[A(t_1) = a, B(t_v) = b_1, B(t_2) = b, X = x, \Lambda = \lambda, Y = y, D_1(t_w) = d] \tag{7.15}$$

$$= \mathbf{Pr}[A(t_1) = a]\, \mathbf{Pr}[B(t_v) = b_1 \mid A(t_1) = a, X = x]\, \mathbf{Pr}[X = x, Y = y, \Lambda = \lambda]$$
$$\mathbf{Pr}[B(t_2) = b \mid B(t_v) = b_1, Y = y]\, \mathbf{Pr}[D_1(t_w) = d \mid B(t_2) = b, X = x, \Lambda = \lambda]. \tag{7.16}$$

Additionally, from Figure 7.8, the following conditional independence relations hold:

- $D_1(t_w) \perp\!\!\!\perp X_{\bar{1}}, \Lambda_{\bar{1}} \mid B(t_2), X_1, \Lambda_1$;
- $X \perp\!\!\!\perp \Lambda$; and
- For any $j \in [n]$,

$$Y_j \perp\!\!\!\perp X_{\bar{j}}, Y_{\bar{j}}, \Lambda_{\bar{j}} \mid X_j, \Lambda_j \text{ and } Y_{\bar{j}} \perp\!\!\!\perp X_j, Y_j, \Lambda_j \mid X_{\bar{j}}, \Lambda_{\bar{j}}. \tag{7.17}$$

By the fixed program P_v of user v in Figure 7.3, we further have:

- $\mathbf{Pr}[X = x] = \frac{1}{2^{n-1}}$ for $x \in \{0,1\}^n$ with $|x| \bmod 2 = 0$.

- $B(t_v) = A(t_1) \oplus \left(\frac{|X|}{2} \bmod 2\right)$. As a result,

$$\mathbf{Pr}[B(t_v) = b_1 \mid A(t_1) = a, X = x] \neq 0$$

  iff $(-1)^{b_1} = (-1)^{a+|x|}$.

- $B(t_v) = B(t_2) \oplus \bigoplus_j Y_j$. As a result,

$$\mathbf{Pr}[B(t_2) = b \mid B(t_v) = b_1, Y = y] \neq 0$$

  iff $(-1)^{|y|} = (-1)^{b+b_1}$.

Combining the above observations, Equation (7.16) can be simplified as

$$\begin{aligned}&\mathbf{Pr}[A(t_1) = a]\mathbb{1}\Big[(-1)^{b_1} = (-1)^{a+|x|}\Big]\,\mathbf{Pr}[X = x, Y = y, \Lambda = \lambda]\\ &\mathbb{1}\Big[(-1)^{|y|} = (-1)^{b+b_1}\Big]\,\mathbf{Pr}[D_1(t_w) = d \mid B(t_2) = b, X_1 = x_1, \Lambda_1 = \lambda_1].\end{aligned} \tag{7.18}$$

To prove our goal in Equation (7.14), let us start with calculating the quantity

$$\mathbf{Pr}[A(t_1) = a, B(t_w) = b, D_1(t_w) = d] \tag{7.19}$$

$$= \sum_{x,y,\lambda,b_1} \mathbf{Pr}\left[A(t_1) = a, B(t_v) = b_1, B(t_w) = b, X = x, \Lambda = \lambda, Y = y, D_1(t_w) = d\right] \tag{7.20}$$

$$\begin{aligned}&= \mathbf{Pr}[A(t_1) = a] \sum_{x,y,\lambda} \mathbb{1}\Big[(-1)^{|x|/2+|y|+a+b} = 1\Big]\,\mathbf{Pr}[X = x, Y = y, \Lambda = \lambda]\\ &\qquad \mathbf{Pr}[D_1(t_w) = d \mid B(t_2) = b, X_1 = x_1, \Lambda_1 = \lambda_1],\end{aligned} \tag{7.21}$$

where in the last equality we replace the joint probability distribution Equation (7.15) by Equation (7.18).

Using the conditions in Equation (7.17) gives the term

$$\begin{aligned}&\mathbf{Pr}[X = x, Y = y, \Lambda = \lambda]\\ &= \mathbf{Pr}\left[X' = x', Y' = y', \Lambda' = \lambda' \,\middle|\, X_1 = x_1, Y_1 = y_1, \Lambda_1 = \lambda_1\right]\mathbf{Pr}[X_1 = x_1, Y_1 = y_1, \Lambda_1 = \lambda_1]\\ &= \mathbf{Pr}\left[X' = x', Y' = y', \Lambda' = \lambda' \,\middle|\, X_1 = x_1, \Lambda_1 = \lambda_1\right]\mathbf{Pr}[X_1 = x_1, Y_1 = y_1, \Lambda_1 = \lambda_1].\end{aligned}$$

Consequently, Equation (7.21) can be rewritten as

$$\begin{aligned}&\mathbf{Pr}[A(t_1) = a] \sum_{x,y,\lambda} \mathbb{1}\Big[(-1)^{|x|/2+|y|+a+b} = 1\Big]\,\mathbf{Pr}[X_1 = x_1, Y_1 = y_1, \Lambda_1 = \lambda_1]\\ &\qquad \mathbf{Pr}\left[X' = x', Y' = y', \lambda' = \lambda' \,\middle|\, X_1 = x_1, \Lambda_1 = \lambda_1\right]\\ &\qquad\qquad \mathbf{Pr}[D_1(t_w) = d \mid B(t_2) = b, X_1 = x_1, \Lambda_1 = \lambda_1].\end{aligned} \tag{7.22}$$

For convenience, let us define

$$f_a(x_1,y_1,\lambda_1) := \sum_{x',y',\lambda'} \mathbb{1}\Big[(-1)^{|x|/2+|y|+a+b}=1\Big] \\ \mathbf{Pr}\big[X'=x', Y'=y', \Lambda'=\lambda' \,\big|\, X_1=x_1, \Lambda_1=\lambda_1\big], \tag{7.23}$$

$$g := \frac{4\,\mathbf{Pr}[D_1(t_w)=d, B(t_2)=b]}{\sum_{x_1,\lambda_1}\mathbf{Pr}[\Lambda_1=\lambda_1]\,\mathbf{Pr}[D_1(t_w)=d \,|\, B(t_2)=b, X_1=x_1, \Lambda_1=\lambda_1]}. \tag{7.24}$$

Then, using the technical Lemmas 7 and 8, we can rewrite Equation (7.22) as

$$\mathbf{Pr}[A(t_1)=a] \sum_{x_1,y_1,\lambda_1} f_a(x_1,y_1,\lambda_1)\,\mathbf{Pr}[X_1=x_1, Y_1=y_1, \Lambda_1=\lambda_1] \\ \mathbf{Pr}[D_1(t_w)=d \,|\, B(t_2)=b, X_1=x_1, \Lambda_1=\lambda_1] \tag{7.25}$$

$$= \mathbf{Pr}[A(t_1)=a]\left(\frac{1}{2}+\delta\right) \sum_{x_1,\lambda_1} \mathbf{Pr}[X_1=x_1]\,\mathbf{Pr}[\Lambda_1=\lambda_1] \\ \mathbf{Pr}[D_1(t_w)=d \,|\, B(t_2)=b, X_1=x_1, \Lambda_1=\lambda_1] \tag{7.26}$$

$$= \mathbf{Pr}[A(t_1)=a](1+2\delta)\,\mathbf{Pr}[D_1(t_w)=d, B(t_2)=b]g^{-1} \tag{7.27}$$

$$= (1+2\delta)(1+\epsilon)^{-1}\,\mathbf{Pr}[A(t_1)=a]\,\mathbf{Pr}[D_1(t_w)=d, B(t_2)=b] \tag{7.28}$$

for some $|\delta| \le 2^{-(n-1)/2}$ and $|\epsilon| \le 2^{-(n-3)/2}$. Here, Equation (7.26) comes from Lemma 7 and $X_1 \perp\!\!\!\perp \Lambda_1$; Equation (7.27) comes from $\mathbf{Pr}[X_1=x_1]=\frac{1}{2}$; and Equation (7.28) comes from Lemma 8.

All the above together yield $\mathbf{Pr}[A(t_1)=a, B(t_w)=b, D_1(t_w)=d] = \textit{Equation (7.28)}$. Now we are ready to compute

$$\frac{\mathbf{Pr}[A(t_1)=a \,|\, B(t_w)=b, D_1(t_w)=d]}{\mathbf{Pr}[A(t_1)=a]} \\ = \frac{\mathbf{Pr}[A(t_1)=a, B(t_w)=b, D_1(t_w)=d]}{\mathbf{Pr}[A(t_1)=a]\,\mathbf{Pr}[B(t_w)=b, D_1(t_w)=d]} \\ = (1+2\delta)(1+\epsilon)^{-1},$$

which can be upper bounded by $\frac{1+2^{-(n-3)/2}}{1-2^{-(n-3)/2}} \le 1+2^{-(n-7)/2}$. Finally, using the inequality $\log(1+z) \le z$ and the definition of mutual information leads to Equation (7.14). □

In the following are two technical lemmas used in the proof of Theorem 2. Intuitively, Lemma 7 says $f_a(x_1,y_1,\lambda_1)$ is close to $1/2$, and Lemma 8 says g is close to 1.

Lemma 7. *Let $f_a(x_1,y_1,\lambda_1)$ be defined as in Equation (7.23). Then, for any $x_1,y_1 \in \{0,1\}$,*

$$\left| f_a(x_1,y_1,\lambda_1) - \frac{1}{2} \right| \le 2^{-(n-1)/2}. \tag{7.29}$$

Proof. Using Equation (7.17) and $X \perp\!\!\!\perp \Lambda$, we have

$$\begin{aligned}&\mathbf{Pr}\big[X' = x', Y' = y', \Lambda' = \lambda' \,\big|\, X_1 = x_1, \Lambda_1 = \lambda_1\big] \\ =\ &\mathbf{Pr}\big[Y' = y' \,\big|\, X' = x', \Lambda' = \lambda'\big]\, \mathbf{Pr}\big[X' = x' \,\big|\, X_1 = x_1\big]\, \mathbf{Pr}\big[\Lambda' = \lambda' \,\big|\, \Lambda_1 = \lambda_1\big].\end{aligned}$$

Let X'', Λ'' be random variables such that

$$\begin{aligned}\mathbf{Pr}\big[X'' = x'\big] &= \mathbf{Pr}\big[X' = x' \,\big|\, X_1 = x_1\big] \\ \mathbf{Pr}\big[\Lambda'' = \lambda'\big] &= \mathbf{Pr}\big[\Lambda' = \lambda' \,\big|\, \Lambda_1 = \lambda_1\big].\end{aligned}$$

It is easy to see that the probability distribution of X'' is uniform over the set

$$\Big\{x' \in \{0,1\}^{n-1} : \big|x'\big| \bmod 2 = x_1\Big\}.$$

In this case, we can rewrite Equation (7.23) as

$$f_a(x_1, y_1, \lambda_1) = \mathop{\mathbf{Pr}}_{\Lambda''}\Big[(-1)^{|X''|/2+|Y'|+a+b+x_1/2+y_1} = 1\Big], \tag{7.30}$$

where we use the subscript Λ'' to indicate this hidden random variable.

Let us write $X'' = X''_2 \ldots X''_n$ and the same convention applies to Λ''. Similar to Equation (7.17), we have $Y_j \perp\!\!\!\perp X''_{\bar{j}}, Y''_{\bar{j}}, \Lambda''_{\bar{j}} \mid X''_j, \Lambda''_j$, and consequently

$$\begin{aligned}\mathbf{Pr}\big[Y' = y' \,\big|\, X'' = x', \Lambda'' = \lambda'\big] &= \prod_{j\geq 2} \mathbf{Pr}\big[Y_j = y_j \mid X'' = x', \Lambda'' = \lambda'\big] \\ &= \prod_{j\geq 2} \mathbf{Pr}\Big[Y_j = y_j \,\Big|\, X''_j = x_j, \Lambda''_j = \lambda_j\Big].\end{aligned}$$

Hence, the conditions in Lemma 6 are satisfied. By Mermin inequality in Lemma 6, we have

$$\Big|\mathbf{E}\Big[(-1)^{|X''|/2+|Y'|+x_1/2}\Big]\Big| \leq 2^{-(n-1)/2+1}.$$

Note that

$$\begin{aligned}&\mathbf{Pr}\Big[(-1)^{|X''|/2+|Y'|+x_1/2} = 1\Big] - \mathbf{Pr}\Big[(-1)^{|X''|/2+|Y'|+x_1/2} = -1\Big] \\ =\ &\mathbf{E}\Big[(-1)^{|X''|/2+|Y'|+x_1/2}\Big]\end{aligned}$$

and

$$\mathbf{Pr}\Big[(-1)^{|X''|/2+|Y'|+x_1/2} = 1\Big] + \mathbf{Pr}\Big[(-1)^{|X''|/2+|Y'|+x_1/2} = -1\Big] = 1.$$

Therefore, we can derive for any $a \in \{0,1\}$:

$$\left|\mathbf{Pr}\left[(-1)^{|X''|/2+|Y'|+x_1/2+a+b+y_1}=1\right]-\frac{1}{2}\right| \leq 2^{-(n-1)/2}, \tag{7.31}$$

and Equation (7.29) immediately follows from Equation (7.30). □

Lemma 8. *Let g be defined as in Equation (7.24). Then, we have*

$$|g-1| \leq 2^{-(n-3)/2}. \tag{7.32}$$

Proof. Note that

$$\begin{aligned}
&\mathbf{Pr}[B(t_2)=b \mid X_1=x_1, \Lambda_1=\lambda_1] \\
&= \mathbf{Pr}\left[(-1)^{|X|/2+|Y|+A(t_1)+b}=1 \,\middle|\, X_1=x_1, \Lambda_1=\lambda_1\right] \\
&= \sum_{a',y_1} \mathbf{Pr}\left[(-1)^{|X'|/2+|Y'|+a'+b+x_1/2+y_1}=1 \middle| A(t_1)=a', Y_1=y_1, X_1=x_1, \Lambda_1=\lambda_1\right] \\
&\qquad\qquad \mathbf{Pr}\left[A(t_1)=a', Y_1=y_1 \mid X_1=x_1, \Lambda_1=\lambda_1\right] \\
&= \sum_{a',y_1} f_{a'}(x_1,y_1,\lambda_1)\, \mathbf{Pr}\left[A(t_1)=a', Y_1=y_1 \mid X_1=x_1, \Lambda_1=\lambda_1\right].
\end{aligned}$$

Thus, using Lemma 7, we have

$$\left|\mathbf{Pr}[B(t_2)=b \mid X_1=x_1, \Lambda_1=\lambda_1]-\frac{1}{2}\right| \leq 2^{-(n-1)/2}.$$

Next, by $X_1 \perp\!\!\!\perp \Lambda_1$, we can write

$$\begin{aligned}
&\mathbf{Pr}[D_1(t_w)=d, B(t_2)=b] \\
&= \sum_{x_1,\lambda_1} \mathbf{Pr}[D_1(t_w)=d \mid B(t_2)=b, X_1=x_1, \Lambda_1=\lambda_1] \\
&\qquad \mathbf{Pr}[B(t_2)=b \mid X_1=x_1, \Lambda_1=\lambda_1]\, \mathbf{Pr}[X_1=x_1]\, \mathbf{Pr}[\Lambda_1=\lambda_1].
\end{aligned}$$

Combining the above with $\mathbf{Pr}[X_1=x_1]=\frac{1}{2}$ and the definition of g in Equation (7.24), our goal Equation (7.32) easily follows. □

7.5 Discussion

7.5.1 Related Work

Security and Bell-Type Inequalities

The violation of Bell-type inequalities, such as the Mermin inequality [176] used in this chapter, essentially reflects the exotic nature of quantum mechanics. This property has been exploited in a number of security protocols that utilise quantum properties. For example, the celebrated E91 protocol proposed in [338] modifies the Bell test to detect eavesdropping and securely generate private keys for cryptography. This technique was later greatly extended into a line of works on device-independent quantum cryptography [339–347]. Similar ideas have also been employed in randomness expansion [344, 348–352] and randomness amplification [353–356]. Most of the above works focus on quantum cryptography and leverage the quantum entanglement as an advantage for enhancing security. In contrast, our focus here is the access control of quantum computer systems. We identify entanglement as a source of security threats, and in Chapter 8 we will propose new access control models to protect against such threats.

7.5.2 Summary

In this chapter, we revealed a threat from quantum entanglement to access control. This work was motivated by the critical question, especially with the rise of quantum-centric supercomputing: do the security guarantees of access control still hold when existing operating systems integrate with quantum computing? We provided a negative answer to this question, by presenting an explicit scenario of a security breach when a classically secure access control system is straightforwardly adapted to the quantum setting. The source of the threat is quantum entanglement — our proofs essentially relies on the fact that entanglement can violate Mermin inequality, a variant of the Bell inequality. This highlights the necessity of developing new quantum access control models, which will be the subject of the next chapter.

Chapter 8

Protection against Threats from Entanglement

The threat revealed by the explicit scenario from the last chapter is fundamentally from quantum entanglement. To protect against this threat, we propose several new access control models that allow explicit control over the generation of entanglement or the entanglement itself as a resource. We rigorously analyse the following metrics of these models: (a) *security* against threats from entanglement; (b) *flexibility* regarding the granularity of specifying the access control; and (c) *efficiency* regarding the space and time complexity for implementation.

8.1 Core Model of Quantum Access Control

Through the previous explicit scenario, we have seen that the security guarantees of access control can be threatened if a system is not properly adapted to the quantum setting. While the system $\mathbb{S}$ studied in Section 7.3 is specific, we have identified that the threat *fundamentally stems from quantum entanglement* (see the last paragraph of Section 7.3.3), which is indispensable to quantum computing. In this section, we study how to handle threats from entanglement.

In classical access control, usually a request (s, o, r) only involves a single object o, which is sufficient in most practical scenarios. However, quantum operations on multiple objects (registers) can generate entanglement between them even when they were initially in a separable state. These quantum operations should be explicitly controlled to protect the security of quantum computer systems. For this purpose, we extend the set **Obj** to include every quantum subsystem (consisting of multiple real quantum objects) as a virtual object, as suggested in [116]. More precisely, suppose $\mathbf{Obj}_{\mathrm{c}}$ and $\mathbf{Obj}_{\mathrm{q}}$ are the sets of real classical and quantum objects, respectively. Then the set of objects considered in this section is $\mathbf{Obj} = \mathbf{Obj}_{\mathrm{c}} \cup \mathcal{P}_+\left(\mathbf{Obj}_{\mathrm{q}}\right)$, where $\mathcal{P}_+(\cdot)$ stands for the set of all non-empty subsets.

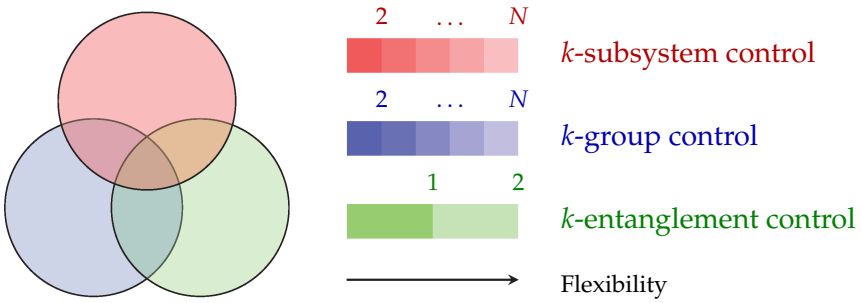


FIGURE 8.1: Comparison of flexibility of different quantum access control models (see Theorems 10 and 11).

Additionally, in this section, we restrict the local memories of subjects to be classical; i.e., we only consider $L : \mathbf{Sub} \to \mathbf{Int}$ (instead of $L : \mathbf{Sub} \to \mathcal{H}_{\mathbf{Int}}$). As shown in Theorem 3, allowing quantum local memories is likely to introduce uncontrollable quantum entanglement that may lead to security breach. Note that avoiding implicit local quantum memory is equivalent to managing all quantum objects explicitly in the access control, and thus does not affect the computational power of the system being protected.

Consequently, in a quantum access control system, we have $\mathbf{Rt} = \mathbf{Rt}_c \cup \mathbf{Rt}_q$, where $\mathbf{Rt}_c$ and $\mathbf{Rt}_q$ consist of abilities to perform operations on classical registers and quantum subsystems, respectively. Note that if $s \in \mathbf{Sub}$ performs a quantum measurement on quantum registers, the classical outcomes will be stored into the local memory $L[s]$.

We summarise these conventions in the following definition for clarity.

Definition 13 (Core model of quantum access control)**.** The core specifications of quantum access control include:

- **Sub** is a set of users. $\mathbf{Obj} = \mathbf{Obj}_c \cup \mathcal{P}_+\left(\mathbf{Obj}_q\right)$, where $\mathbf{Obj}_c$ and $\mathbf{Obj}_q$ are sets of classical and quantum registers.
- The local memories $L : \mathbf{Sub} \to \mathbf{Int}$ of subjects are classical.
- The classical part of the access control is guarded by the access matrix $M_c : \mathbf{Sub} \times \mathbf{Obj}_c \to \mathcal{P}(\mathbf{Rt}_c)$.

8.1.1 Overview

All models of quantum access control to be introduced later are refinements of the core model in Definition 13. To handle threats from quantum entanglement, we introduce two types of models. In Section 8.2, we consider explicitly controlling quantum operations on subsystems of multiple quantum registers; in Section 8.3, we consider explicitly controlling the resource of quantum entanglement.

	Security	Efficiency
Straightforward lifting (Section 7.3)	✗	$O\big(M \cdot (N_c + N_q)\big)$ space $O(x)$ time
k-subsystem control (Section 8.2.1)	✓	$O\Big(M \cdot \Big(N_c + \sum_{j=1}^{k} \binom{N_q}{k}\Big)\Big)$ space $O(x)$ time
k-group control (Section 8.2.2)	✓	$O\big(M \cdot (N_c + N_q)\big)$ space $O(x)$ time
k-entanglement control ($k = 1, 2$; Section 8.3)	✓	$O\Big(M \cdot \Big(N_c + N_q^k\Big)\Big)$ space $O\big(x + xN_q(k-1)\big)$ time

TABLE 8.1: Comparison of security and efficiency of different quantum access control models. Here, we assume $|\mathbf{Sub}| = M$, $|\mathbf{Obj}_c| = N_c$, $\left|\mathbf{Obj}_q\right| = N_q$, and the request has length x.

To evaluate and compare these models, we consider the following three metrics [357, 358]:

- **Security**, in this chapter, concerns whether the model can protect against threats from entanglement by properly managing entanglement between objects; in particular, by forbidding certain entanglement through the specification of attributes. For concreteness, we will check the security of a model by verifying if system $\mathcal{S}$ (see Section 7.3.1) can be lifted to this model while retaining the security guarantee in Theorem 2.
- **Flexibility** is related to the granularity of specifying the access control, and thus how well the model can support the principle of least privilege [359]. In this chapter, we compare the flexibility of different models by Definition 12.
- **Efficiency** measures the space complexity for implementing the model and the time complexity for handling a request.

All of the proposed models are secure, but their flexibility and efficiency vary. In practice, the choice of which model to use depends on the *specific requirements* about the flexibility and efficiency. For visualisation, in Table 8.1, we compare the security and efficiency of the proposed models, and in Figure 8.1 we compare the flexibility.

8.2 Control of Quantum Operations

8.2.1 Subsystem Control

Subsystem control has been initially studied in [116]. The original observation in [116] is that having full access to a composite subsystem of quantum registers A and B is not

the same as the combination of separate accesses to A and to B. Thus, they proposed to regard every quantum subsystem of multiple quantum registers as a virtual object, as mentioned at the beginning of Section 8.1. In our terminology, they define the authorisation rule via an access matrix $M : \mathbf{Sub} \times \mathbf{Obj} \to \mathcal{P}(\mathbf{Rt})$, where $\mathbf{Obj} = \mathbf{Obj}_{\mathrm{c}} \cup \mathcal{P}_{+}\left(\mathbf{Obj}_{\mathrm{q}}\right)$ is as defined in the core model Definition 13. In the following, we slightly extend this idea to k-subsystem control, which offers a better trade-off between flexibility and efficiency.

Definition 14 (k-subsystem control)**.** Suppose that $1 \leq k \leq \left|\mathbf{Obj}_{q}\right|$. The k-subsystem control model, denoted by SUBSYS^{k}, extends Definition 13 by letting $\mathbf{Attr} = \{M_{\mathrm{c}}, M_{\mathrm{q}}, L\}$, $\mathbf{Rule} = \{Auth\}$, $M_{\mathrm{q}} : \mathbf{Sub} \times \mathcal{P}_{\leq k}\left(\mathbf{Obj}_{\mathrm{q}}\right) \to \mathcal{P}(\mathbf{Rt}_{\mathrm{q}})$, and

$$
\begin{aligned}
Auth(s, o, r) \equiv\ & p_{\mathrm{c}} \wedge p_{\mathrm{q}}, \text{ where:} \\
& p_{\mathrm{c}} \equiv o \in \mathbf{Obj}_{\mathrm{c}} \to r \in M_{\mathrm{c}}[s, o], \\
& p_{\mathrm{q}} \equiv o \in \mathcal{P}_{+}\left(\mathbf{Obj}_{\mathrm{q}}\right) \to |o| \leq k \wedge r \in M_{\mathrm{q}}[s, o].
\end{aligned}
$$

Here, $\mathcal{P}_{\leq k}(\cdot)$ denotes the set of non-empty subsets of cardinality $\leq k$.

In the authorisation rule, p_{c} says that if o is a classical register, then we check if $r \in M_{\mathrm{c}}[s, o]$; and p_{q} says that if o is a quantum subsystem involving $\leq k$ registers, then we check if $r \in M_{\mathrm{q}}[s, o]$. Compared to [116] (equivalent to setting $k = \left|\mathbf{Obj}_{\mathrm{q}}\right|$), Definition 14 only authorises requests involving subsystem of size $\leq k$, which achieves better efficiency by reducing the space complexity of storing the attribute M_{q}, as will be shown later in Theorem 5. Further comparison with [116] can be found in Section 8.5.1.

Typical choices of k include $k = 2$ and $k = \left|\mathbf{Obj}_{\mathrm{q}}\right|$. Note that the case $k = 1$ forbids any entanglement between quantum registers (see the first unsatisfactory lifting in Section 7.3.3).

Security

The k-subsystem control model provides the most direct control over quantum operations performed on multiple registers and the entanglement generated between them. Therefore, it can protect against threats from entanglement.

Theorem 4 (Security of k-subsystem control)**.** *For $2 \leq k \leq \left|\mathbf{Obj}_{\mathrm{q}}\right|$, system $\mathcal{S}$ in Section 7.3.1 can be lifted to a system in SUBSYS^{k} such that the security guarantee in Theorem 2 holds.*

Again, while the security in Theorem 4 (and in subsequent theorems) is stated for the specific system $\mathcal{S}$, the access control model can be employed to protect against *any threat from entanglement*. This is because, within the model, entanglement can be explicitly forbidden through the specification of attributes. For example, setting $M_{\mathrm{q}}[s, \{X, Y\}] = \emptyset$ forbids user s from generating new entanglement between registers X and Y.

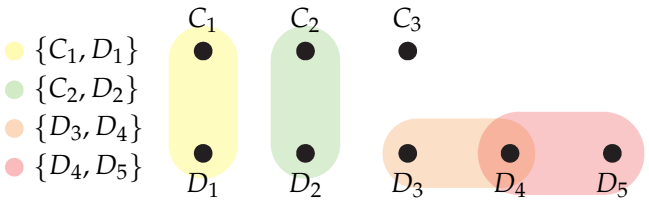


FIGURE 8.2: Illustration of a system in the 2-subsystem control model (see the proof of Theorem 4; take $n = 3$). Each 2-subsystem accessible by some user with right `all` is colored.

Proof of Theorem 4. We only prove the case $k = 2$. The proof for other k is similar. For illustration of the model's flexibility, let us assume several additional quantum registers, say $\mathbf{Obj}_{\mathrm{q}} = \{C_1, \ldots, C_n, D_1, \ldots, D_5\}$; and we only show one possible way of lifting to this model. To prove the security, it suffices to verify that no entanglement can be generated among $C_1, \ldots, C_n$ in the lifted system.

The lifted system has $\mathbf{Obj}_{\mathrm{c}} = \{A, B, M_{\mathrm{c}}, M_{\mathrm{q}}\}$, constructed as follows.

- Let $M_{\mathrm{c}}[v, M_{\mathrm{c}}] = M_{\mathrm{c}}[v, M_{\mathrm{q}}] = \{\mathtt{all}\}$; i.e., v can modify M_{c} and M_{q}, lifted from $M_{\mathrm{acc}}[v, M_{\mathrm{acc}}] = \{\mathtt{all}\}$ in $\mathcal{S}$.
- For $X \in \{A, B\}$, define $M_{\mathrm{c}}[s, X] = M_{\mathrm{acc}}[s, X]$. For $X \in \{C_1, \ldots, C_n\}$, let $M_{\mathrm{q}}[s, \{X\}] = M_{\mathrm{acc}}[s, X]$.

  For $X \in \{D_1, \ldots, D_5\}$, let $M_{\mathrm{q}}[s, \{X\}] = \{\mathtt{all}\}$. Accordingly, modify Line 1 and 6 of Figure 7.3 to write $M_{\mathrm{c}}[s, X]$ and $M_{\mathrm{q}}[s, \{X\}]$ instead of $M_{\mathrm{acc}}[s, X]$.
- Let $M_{\mathrm{q}}[w_1, \{C_1, D_1\}] = M_{\mathrm{q}}[w_2, \{C_2, D_2\}] = M_{\mathrm{q}}[w_3, \{D_3, D_4\}] = M_{\mathrm{q}}[w_3, \{D_4, D_5\}] = \{\mathtt{all}\}$.
- Those $M_{\mathrm{c}}[s, o]$ and $M_{\mathrm{q}}[s, o]$ unspecified above are defined to be $\emptyset$. In particular, we have $M_{\mathrm{q}}[w_j, \{C_l, C_r\}] = \emptyset$ for $l \neq r$, implying that entanglement cannot be generated among $C_1, \ldots, C_n$.

Note that the above lifted system only forbids entanglement generated among $C_1, \ldots, C_n$, but allows entanglement generated between C_1 and D_1, C_2 and D_2, D_3 and D_4, and D_4 and D_5. For illustration, the lifted system is visualised in Figure 8.2. □

Efficiency

Now we analyse the efficiency of the k-subsystem control, including space and time complexities. Here and throughout this chapter, the space complexity for implementing an access control model is measured by the number of classical memory locations (each capable of storing a bounded integer) to store all the attributes. The time complexity

for handling a request is measured by the number of elementary operations (including arithmetic, logical, and memory access operations) in the standard word RAM model.

Theorem 5 (Efficiency of k-subsystem control)**.** *Suppose* $|\mathbf{Sub}| = M$, $|\mathbf{Obj}_\mathrm{c}| = N_\mathrm{c}$ *and* $\left|\mathbf{Obj}_\mathrm{q}\right| = N_\mathrm{q}$*, then the* k*-subsystem control model uses* $O\left(M \cdot \left(N_\mathrm{c} + \sum_{j=1}^{k} \binom{N_\mathrm{q}}{k}\right)\right)$ *space for access control, and it takes* $O(x)$ *time to authorise a request of length* x*.*

Compared to the original idea in [116], our Theorem 5, together with Theorem 10 presented later, demonstrates a trade-off between flexibility and efficiency. In particular, taking smaller k in the k-subsystem control model leads to greater efficiency but reduced flexibility (see Theorem 10). For example, focusing on the dependence on N_q, then for $k = 2$, the space complexity is $O\left(N_\mathrm{q}^2\right)$. However, for $k = N_\mathrm{q}$, equivalent to the case originally suggested by [116], the space complexity is $O\left(2^{N_\mathrm{q}}\right)$, which is exponentially large.

Another point worth mentioning is that the space or time complexity in Theorem 5 (and subsequent theorems) is regarding the worst case. We do not bother considering more efficient data structures [360] (like ACL or capability lists [175]) to store the attributes, which is left for future research (see also Section 4.4).

Proof of Theorem 5. The space complexity for implementing k-subsystem control is dominated by that for storing the attributes M_c and M_q in Definition 14. The matrix representation of M_c has $|\mathbf{Sub}|$ rows and $|\mathbf{Obj}_\mathrm{c}|$ columns, while that of M_q has $|\mathbf{Sub}|$ rows and $\left|\mathcal{P}_{\leq k}\left(\mathbf{Obj}_\mathrm{q}\right)\right| = \sum_{j=1}^{k} \binom{N_\mathrm{q}}{k}$ columns.

The time complexity for handling an access request $(s, o, r) \in \mathbf{Req}$ is dominated by, according to the authorisation rule in Definition 14, reading the whole request and checking the size of the subsystem $o \subseteq \mathbf{Obj}_\mathrm{q}$, which scales as the length of the request. □

8.2.2 Group Control

The space complexity for implementing k-subsystem control (even for the smallest nontrivial $k = 2$) could be formidable when the number N_q of quantum objects is large. In practical classical systems, the number of objects can be in the tens of millions [357]. While it may take a long time to build quantum computers at such a scale, we can still consider models with lower space requirements, such as the following k-group control model.

Definition 15 (k-group control)**.** Suppose that $1 \leq k \leq \left|\mathbf{Obj}_q\right|$. The k-group control model, denoted by GRP^k, extends Definition 13 by setting $\mathbf{Attr} = \left\{M_\mathrm{c}, M_\mathrm{q}, G, L\right\}$, $\mathbf{Rule} =$

$\{Auth\}$, $M_q : \mathbf{Sub} \times \mathbf{Obj}_q \to \mathcal{P}(\mathbf{Rt}_q)$, $G : \mathbf{Obj}_q \to [k]$, and

$$\begin{aligned} Auth(s,o,r) &\equiv p_c \wedge p_q, \text{ where :} \\ p_c &\equiv o \in \mathbf{Obj}_c \to r \in M_c[s,o], \\ p_q &\equiv o \in \mathcal{P}_+\left(\mathbf{Obj}_q\right) \to (\forall X, Y \in o : G[X] = G[Y]) \wedge \left(\forall X \in o : r \in M_q[s,X]\right). \end{aligned}$$

Explanation

The attribute G assigns a group label to every real quantum object. In the authorisation rule, p_c is standard; and p_q says that if o is a quantum subsystem, then the request is authorised only if all quantum registers in o has the same group label, and the right r appears in $M_q[s,X]$ for any quantum register $X \in o$. Note that the attribute M_q in Definition 15 is different from that in Definition 14: M_q in the k-group control model has a smaller domain.

Definition 15 can be slightly modified (by adding a group label 0) to abstract the notion of entangling zone employed in some architectures of quantum hardware [16], where two-qubit gates can only apply on qubits in the entangling zone.

Security

The k-group control model also provides explicit control over quantum operations performed on multiple registers, through specifying the attributes G and M_q. Thus, it can protect against threats from entanglement.

Theorem 6 (Security of k-group control)**.** *Let n be as defined in Section 7.3.1. For $n+1 \leq k \leq \left|\mathbf{Obj}_q\right|$, system $\mathcal{S}$ can be lifted to a system in GRP^k such that the security guarantee in Theorem 2 holds.*

Proof. We only prove the case $k = n+1$. The proof for other k is similar. Like in the proof of Theorem 4, let us assume several additional quantum registers, say $\mathbf{Obj}_q = \{C_1, \ldots, C_n, D_1, \ldots, D_5\}$; and we only show one possible way of lifting to this model. To prove that the lifted system retains the security guarantee in Theorem 2, it suffices to verify that no entanglement is permitted to be generated among $C_1, \ldots, C_n$.

The lifted system has $\mathbf{Obj}_c = \{A, B, M_c, M_q, G\}$, constructed as follows.

- Let $M_c[v, M_c] = M_c[v, M_q] = \{\texttt{all}\}$.
- For $X \in \{A, B\}$, define $M_c[s,X] = M_{acc}[s,X]$. For $X \in \{C_1, \ldots, C_n\}$, let $M_q[s,X] = M_{acc}[s,X]$. For $X \in \{D_1, \ldots, D_5\}$, let $M_q[s,X] = \{\texttt{all}\}$. Accordingly, modify Line 1 and 6 of P_v in Figure 7.3 to write $M_c[s,X]$ and $M_q[s,X]$ instead of $M_{acc}[s,X]$.

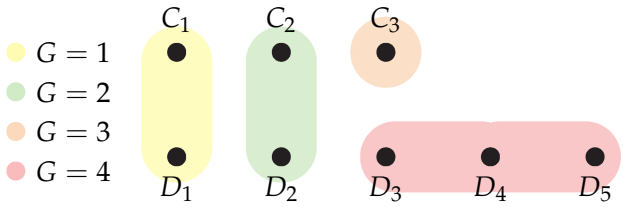


FIGURE 8.3: Illustration of a system in the $(n+1)$-group control model (see the proof of Theorem 6; take $n = 3$). Each group with the same label (specified by G) is colored.

- Let $G[C_1] = G[D_1] = 1$, $G[C_2] = G[D_2] = 2$, $G[C_j] = j$ for $j > 2$, and $G[D_2] = G[D_3] = G[D_4] = n+1$.

By Definition 15, the above lifted system forbids entanglement generated among $C_1, \ldots, C_n$, but allows entanglement generated between C_1 and D_1, C_2 and D_2, and among D_3, D_4, and D_5. For illustration, the lifted system is visualised in Figure 8.3. □

Efficiency

Now we analyse the efficiency of the k-group control model. Focusing on the dependence on N_q, its space complexity is $O(N_q)$, which is much smaller than that of the k-subsystem control model.

Theorem 7 (Efficiency of k-group control)**.** *Suppose that* $|\mathbf{Sub}| = M$, $|\mathbf{Obj}_c| = N_c$, *and* $\left|\mathbf{Obj}_q\right| = N_q$, *then the k-group control model uses* $O\big(M \cdot (N_c + N_q)\big)$ *space for access control, and it takes* $O(x)$ *time to handle a request of length x.*

Proof. Similar to the proof of Theorem 5, the space complexity is dominated by that for storing the attributes M_c and M_q in Definition 15. The matrix representation of M_c has $|\mathbf{Sub}|$ rows and $|\mathbf{Obj}_c|$ columns, while that of M_q has $|\mathbf{Sub}|$ rows and $\left|\mathbf{Obj}_q\right|$ columns.

The time complexity is dominated by, according to the authorisation rule in Definition 15, checking if all $X \in o$ have the same group label. This can be done by (a) picking an $X \in o$; (b) scanning other $Y \in o$; (c) checking if $G[X] = G[Y]$. The conclusion immediately follows. □

8.3 Control of Entanglement

The subsystem control and group control models in Section 8.2 offer explicit control over quantum operations on multiple quantum registers that can generate entanglement. However, within these models, it is not possible to explicitly control entanglement as a resource: e.g., we cannot make a specification to "forbid any entanglement to exist between

quantum registers A and B'' after entanglement has already been established between A and B, because no information about existing entanglement is recorded. Thus, we propose the following model to control the resource of entanglement.

Definition 16 (1-entanglement control)**.** The 1-entanglement control model, denoted by ENT^1, extends Definition 13 by letting $\mathbf{Attr} = \{M_c, M_q, M_e, D, L\}$, $\mathbf{Rule} = \{Auth, Post\}$, $M_q : \mathbf{Sub} \times \mathbf{Obj}_q \to \mathcal{P}(\mathbf{Rt}_q)$, $M_e, D : \mathbf{Obj}_q \to \{true, false\}$, and

$$
\begin{aligned}
Auth(s,o,r) &\equiv p_c \wedge p_e \wedge p_q, \text{ where:}\\
&\quad p_c \equiv o \in \mathbf{Obj}_c \to r \in M_c[s,o],\\
&\quad p_e \equiv o = M_e[X] \to (\neg D[X] \wedge M_e[X] \to r = \texttt{read}),\\
&\quad p_q \equiv o \in \mathcal{P}_+\left(\mathbf{Obj}_q\right) \to \left(\forall X \in o : r \in M_q[s,X]\right) \wedge \left(|o| > 1 \to \bigwedge_{X \in o} M_e[X]\right),\\
Post(s,o,r) &\equiv \textbf{if } o \in \mathcal{P}_+\left(\mathbf{Obj}_q\right) \textbf{ then}\\
&\qquad \textbf{if } r = \texttt{measure} \textbf{ then}\\
&\qquad\quad \textbf{for } X \in o \textbf{ do } D[X] := true \textbf{ od}\\
&\qquad \textbf{else if } |o| > 1 \textbf{ then}\\
&\qquad\quad \textbf{for } X \in o \textbf{ do } D[X] := false \textbf{ od}\\
&\qquad \textbf{fi}\\
&\quad \textbf{fi}
\end{aligned}
$$

Here, $\texttt{measure} \in \mathbf{Rt}_q$ means the ability to perform a complete measurement (see Chapter 2). Recall that *Post* denotes the post-update rule (see Definition 8).

Explanation

In Definition 16, we introduce two attributes M_e and D. For quantum register $X \in \mathbf{Obj}_q$, $M_e[X]$ represents whether X is *permitted to be entangled* with other quantum registers; and $D[X]$ represents whether X is *ensured to be disentangled* from other quantum registers. More precisely, $D[X] = true$ means X is ensured to be disentangled, and $D[X] = false$ means X is potentially entangled. The authorisation and post-update rules are explained as follows.

- For the authorisation rule, p_c is standard. p_e maintains consistency: if a register X is potentially entangled ($D[X] = false$), then its attribute $M_e[X]$ becomes read-only. This prevents from the situation where X is entangled ($D[X] = false$) but is not permitted to be ($M_e[X] = false$). Finally, p_q states that to exercise right r on a quantum subsystem o, r must appear in $M_q[X]$ for all $X \in o$; and if o involves multiple registers, then all $X \in o$ must be permitted to be entangled.

- The post-update rule updates the attribute D after an authorised request. If the request performs a complete measurement on a quantum subsystem, then every register within it is ensured to be disentangled. Otherwise, if the subsystem involves multiple registers, the registers within can be potentially entangled (in the worst case).

It is worth noting that the attribute D only provides *approximate knowledge* of existing entanglement. As a result, a register X might be disentangled even when $D[X] = \mathit{false}$. In this case, the authorisation rule forces some user to perform a redundant measurement on X simply to update $D[X]$ before $M_e[X]$ can be set to *false*. Nevertheless, we argue that achieving accurate control over entanglement (without approximation) is impractical, as this requires tracking the explicit state of quantum registers. Further, such state-tracking is generally considered beyond the scope of access control.

Another point about the post-update rule is its use of complete measurement to ensure disentanglement. An open question here is whether other weaker conditions exist that can ensure disentanglement (see also Section 4.4).

We can further refine Definition 16 into the following model that records more information about existing entanglements.

Definition 17 (2-entanglement control)**.** The 2-entanglement control model, denoted by ENT^2, extends Definition 13 as follows. Let $\mathbf{Attr} = \{M_c, M_q, M_e, D, L\}$, $\mathbf{Rule} = \{\mathit{Auth}, \mathit{Post}\}$, where $M_q : \mathbf{Sub} \times \mathbf{Obj}_q \to \mathcal{P}(\mathbf{Rt}_q)$, $M_e, D : \mathcal{P}_2\left(\mathbf{Obj}_q\right) \to \{\mathit{true}, \mathit{false}\}$, and

$$
\begin{aligned}
\mathit{Auth}(s,o,r) \equiv\ & p_c \wedge p_e \wedge p_q, \text{ where:} \\
& p_c \equiv o \in \mathbf{Obj}_c \to r \in M_c[s,o], \\
& p_e \equiv o = M_e[X,Y] \to (\neg D[X,Y] \wedge M_e[X,Y] \to r = \texttt{read}), \\
& p_q \equiv o \in \mathcal{P}_+\left(\mathbf{Obj}_q\right) \to (\forall X \in o : r \in M_q[s,X]) \wedge (|o| > 1 \to \bigwedge_{X \neq Y \in o} M_e[X,Y]) \\
\mathit{Post}(s,o,r) \equiv\ & \textbf{if } o \in \mathcal{P}_+\left(\mathbf{Obj}_q\right) \textbf{ then} \\
& \quad \textbf{if } r = \texttt{measure} \textbf{ then} \\
& \quad\quad \textbf{for } X \in o \wedge Y \in \mathbf{Obj}_q \textbf{ do } D[X,Y] := \mathit{true} \textbf{ od} \\
& \quad \textbf{else if } |o| > 1 \textbf{ then} \\
& \quad\quad \textbf{for } X \neq Y \in o \textbf{ do } D[X,Y] := \mathit{false} \textbf{ od} \\
& \quad \textbf{fi} \\
& \textbf{fi}
\end{aligned}
$$

Here, $\mathcal{P}_2(\cdot)$ denotes the set of subsets of cardinality 2.

Compare to ENT^1 in Definition 16, we extend the attributes M_e and E to be functions on $\mathcal{P}_2\left(\mathbf{Obj}_\mathrm{q}\right)$. Specifically, $M_\mathrm{e}[X, Y]$ represents whether X, Y are allowed to be entangled; and $D[X, Y]$ represents whether X is ensured to be disentangled from Y.

The authorisation and post-update rules in Definition 17 are similar to but more fine-grained (regarding entanglement between two quantum registers) than those in Definition 16. Note that in the post-update rule, we modify $D[X, Y]$ to be *true* for all $Y \in \mathbf{Obj}_\mathrm{q}$ when X is completely measured.

In the above, we only define k-entanglement control for $k = 1, 2$. A similar definition for higher k is possible, but it seems less useful due to the following intuitive reason. ENT^2 is more flexible than ENT^1 because it records "whether two quantum registers can be entangled", which is more fine-grained than "whether one quantum register can be entangled with others". For example, saying "X_1, X_2 are entangled" is more fine-grained than saying "X_1 is entangled with some register and X_2 is also entangled". However, when we consider $k = 3$, it is unclear whether saying "X_1, X_2, X_3 are entangled" is more fine-grained than saying "X_1, X_2 are entangled and X_1, X_3 are also entangled".

Security

The entanglement control model provides explicit control over entanglement as a resource, and therefore offers protection against threats from entanglement.

Theorem 8 (Security of k-entanglement control)**.** *System $\mathcal{S}$ in Section 7.3.1 can be lifted to a system in* ENT^1 *(or* ENT^2*) such that the security guarantee in Theorem 2 holds.*

Proof. We only prove the theorem for ENT^1, and the proof for ENT^2 is similar. Like in the proof of Theorem 4, let us assume several additional quantum registers, say $\mathbf{Obj}_\mathrm{q} = \{C_1, \ldots, C_n, D_1, \ldots, D_5\}$; and we only show one possible way of lifting.

The lifted system has $\mathbf{Obj}_\mathrm{c} = \{A, B, M_\mathrm{c}, M_\mathrm{q}, M_\mathrm{e}, D\}$, constructed as follows.

- Let $M_\mathrm{c}[v, M_\mathrm{c}] = M_\mathrm{c}[v, M_\mathrm{q}] = M_\mathrm{c}[v, M_\mathrm{e}] = \{\mathtt{all}\}$.
- For $X \in \{A, B\}$, we define $M_\mathrm{c}[s, X] = M_\mathrm{acc}[s, X]$. For $X \in \{C_1, \ldots, C_n\}$, let $M_\mathrm{q}[s, X] = M_\mathrm{acc}[s, X]$. For $X \in \{D_1, \ldots, D_5\}$, let $M_\mathrm{q}[s, X] = \{\mathtt{all}\}$. Accordingly, modify Line 1 and 6 of P_v in Figure 7.3 to write $M_\mathrm{c}[s, X]$ and $M_\mathrm{q}[s, X]$ instead of $M_\mathrm{acc}[s, X]$.
- For $j \in [n]$, let $M_\mathrm{e}[C_j]$ be initialised to 1 (bit representation of *true*). We add the following line before Line 1 of P_v: For $j \in [n]$, measure C_j in the computational basis and flip $M_\mathrm{e}[C_j]$. This new line forbids future entanglement among $C_1, \ldots, C_n$.

Before v modifies each $M_\mathrm{e}[C_j]$ to 0, according to the authorisation rule, $D[C_j]$ has to be 1, meaning C_j is ensured to be disentangled from other quantum registers. Meanwhile, $M_\mathrm{e}[D_l] = 1$, so each D_l is permitted to be entangled. □

Efficiency

Finally, let us analyse the efficiency of the entanglement control model.

Theorem 9 (Efficiency of k-entanglement control)**.** *Suppose* $|\mathbf{Sub}| = M$, $|\mathbf{Obj}_c| = N_c$ *and* $\left|\mathbf{Obj}_q\right| = N_q$*, then the k-entanglement control model uses* $O\left(M \cdot \left(N_c + N_q^k\right)\right)$ *space for access control for* $k = 1, 2$*, and it takes* $O\big(x + xN_q(k-1)\big)$ *time to handle a request of length* x*.*

Proof. The space complexity is dominated by that for storing the attributes M_c, M_q, M_e, and D in Definitions 16 and 17. The matrix representation of M_c has $|\mathbf{Sub}|$ rows and $|\mathbf{Obj}_c|$ columns, while that of M_q has $|\mathbf{Sub}|$ rows and $\left|\mathbf{Obj}_q\right|$ columns. M_e and D have $\left|\mathcal{P}_{\leq k}\left(\mathbf{Obj}_q\right)\right| = O\left(N_q^k\right)$ rows and 1 column, for $k = 1, 2$.

The time complexity is dominated by the first for-loop in the post-update rule. For $k = 1$ (see Definition 16), the loop goes through every $X \in o$ and has time complexity $O(x)$. For $k = 2$ (see Definition 17), the loop goes through every $X \in o$ and $Y \in \mathbf{Obj}_q$ and has time complexity $O\big(x \cdot N_q\big)$. □

8.4 Comparison of Flexibility

Now we compare the flexibility of different models introduced in Sections 8.2 and 8.3. The comparison results were previously visualised in Figure 8.1. In practice, one can also consider a hybrid of these models to achieve a better trade-off between flexibility and efficiency.

8.4.1 Flexibility Hierarchy

Our first theorem shows any model in SUBSYS, GRP, and ENT becomes more flexible as the parameter k increases.

Theorem 10 (Flexibility hierarchy)**.** *For* $\mathsf{M} \in \{\mathsf{SUBSYS}, \mathsf{GRP}\}$ *and any* $k \geq 2$*, or* $\mathsf{M} = \mathsf{ENT}$ *and* $k = 2$*, we have* $\mathsf{M}^{k-1} < \mathsf{M}^k$*.*

For simplicity of presentation, let us prove Theorem 10 by proving three lemmas for $\mathsf{M} = \mathsf{SUBSYS}$, $\mathsf{M} = \mathsf{GRP}$, and $\mathsf{M} = \mathsf{ENT}$, respectively. Firstly, the $\mathsf{M} = \mathsf{SUBSYS}$ part of Theorem 10 can be restated as the following lemma.

Lemma 9. *For any* $k \geq 2$, $\mathsf{SUBSYS}^{k-1} < \mathsf{SUBSYS}^k$.

Proof.

1. We first prove $\mathsf{SUBSYS}^k \not\leq \mathsf{SUBSYS}^{k-1}$. The proof idea is using the existence of quantum operations acting non-trivially on k quantum registers. For concreteness,

let us consider QFT_k, the quantum Fourier transform on k qubits, and use $\texttt{QFT}_k$ to denote the right to implement a QFT_k quantum circuit.

Let us consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{SUBSYS}^k$, where $\mathbf{Sub} = \{u, v\}$, $\mathbf{Obj}_\mathrm{c} = \emptyset$, $\mathbf{Obj}_\mathrm{q} = \{X_1, \ldots, X_k\}$, $\mathbf{Rt}_\mathrm{c} = \emptyset$, and $\mathbf{Rt}_\mathrm{q} = \{\texttt{QFT}_k\}$. Attributes $M_\mathrm{c}, M_\mathrm{q}$ are initialised as follows. Since $\mathbf{Obj}_\mathrm{c} = \emptyset$, we set $M_\mathrm{c} = \emptyset$. Denote subsystem $q = \mathbf{Obj}_\mathrm{q}$. For $s \in \mathbf{Sub}, o \subseteq \mathbf{Obj}_\mathrm{q}$:

$$M_\mathrm{q}[s, o] = \begin{cases} \{\texttt{QFT}_k\}, & s = u \wedge o = q, \\ \emptyset, & o.w. \end{cases} \tag{8.1}$$

Assume for contradiction that there exists another system

$$\mathcal{A}' = \left(\mathbf{Sub}, \mathbf{Obj}', \mathbf{Rt}', \mathbf{Attr}', \mathbf{Rule}'\right) \in \mathsf{SUBSYS}^{k-1}$$

with $M'_\mathrm{c}, M'_\mathrm{q} \in \mathbf{Attr}'$ such that $\mathcal{A}' \simeq \mathcal{A}$. We can further assume that $\mathbf{Obj}'_\mathrm{c} = \emptyset$ and $\mathbf{Rt}'_\mathrm{c} = \emptyset$, because otherwise $\mathcal{A}$ and $\mathcal{A}'$ will be obviously inequivalent. As a result, M'_q cannot be dynamically modified.

Consider an execution (S, P) with $P_u \equiv QFT_k[q]$ and $P_v \equiv \perp$, where $\perp$ denotes termination without doing anything. By our construction of $\mathcal{A}$, the history generated by (S, P) in $\mathcal{A}$ is simply $(u, q, \texttt{QFT}_k)$ and is authorised.

Meanwhile, a request accessing quantum register o in $\mathcal{A}'$ is only authorised if $|o| \leq k - 1$, according to the authorisation rule in Definition 14. Since QFT_k non-trivially acts on all k quantum registers, the history α generated by (S, P) in $\mathcal{A}'$ contains more than one requests. The above implies that $\alpha(0) = (u, o, r)$, where $o \subseteq q$ is a quantum register with $|o| \leq k - 1$ and $r \neq \texttt{QFT}_k$ is the ability to perform some quantum circuit $U \neq QFT_k$. As we assume $\mathcal{A} \simeq \mathcal{A}'$, α is also authorised.

Now we consider another execution (S, P') with $P'_u \equiv U[o]$ and $P'_v \equiv \perp$. The history generated by (S, P) in $\mathcal{A}'$ is (u, o, r), which is therefore authorised as a prefix of the authorised history α. However, (S, P') cannot generate a valid history in $\mathcal{A}$ because $r \notin \mathbf{Rt} = \{\texttt{QFT}_k\}$. Hence, we obtain a contradiction and the conclusion follows.

2. Next, we prove that $\mathsf{SUBSYS}^{k-1} \leq \mathsf{SUBSYS}^k$.

   Suppose that $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{SUBSYS}^{k-1}$ with $M_\mathrm{c}, M_\mathrm{q} \in \mathbf{Attr}$. Then, we can define another system $\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}', \mathbf{Rule}) \in \mathsf{SUBSYS}^k$ with $M'_\mathrm{c}, M'_\mathrm{q} \in \mathbf{Attr}$ such that $M'_\mathrm{c} = M_\mathrm{c}$ and for any $s \in \mathbf{Sub}, o \subseteq \mathbf{Obj}_\mathrm{q}$:

$$M'_\mathrm{q}[s, o] = \begin{cases} M_\mathrm{q}[s, o], & |o| \leq k - 1, \\ \emptyset, & o.w. \end{cases}$$

It is easy to see that $\mathcal{A} \simeq \mathcal{A}'$ from this construction.

□

Secondly, the $\mathsf{M} = \mathsf{GRP}$ part of Theorem 10 can be restated as the following lemma.

Lemma 10. *For any $k \geq 2$, $\mathsf{GRP}^{k-1} < \mathsf{GRP}^{k}$.*

Proof.

1. We first prove that $\mathsf{GRP}^{k} \not\leq \mathsf{GRP}^{k-1}$. The proof idea is by noticing that a system in GRP^{k} can assign all quantum registers into k groups, while a system in GRP^{k-1} can only assign them into $k-1$ groups. Using the pigeonhole principle, there will be two quantum registers that belong to different groups in the former system, and to the same group in the latter system. Then, intuitively, we can show that the latter system authorises strictly more requests than the former.

   Let us consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{GRP}^{k}$, where $\mathbf{Rt}_{\mathrm{c}} = \emptyset$, $\mathbf{Rt}_{\mathrm{q}} = \{\texttt{CNOT}\}$, $\mathbf{Sub} = \{u, v\}$, $\mathbf{Obj}_{\mathrm{c}} = \emptyset$, and $\mathbf{Obj}_{\mathrm{q}} = \{X_1, \ldots, X_{2k}\}$. Here, $\texttt{CNOT}$ means the ability to perform a *CNOT* gate. Attributes $M_{\mathrm{c}}, M_{\mathrm{q}}, G \in \mathbf{Attr}$ are initialised as follows. Since $\mathbf{Obj}_{\mathrm{c}} = \emptyset$, we set $M_{\mathrm{c}} = \emptyset$. For $s \in \mathbf{Sub}, o \in \mathbf{Obj}_{\mathrm{q}}$:

$$M_{\mathrm{q}}[s, o] = \begin{cases} \{\texttt{CNOT}\}, & s = u, \\ 0, & o.w. \end{cases}$$

   Let $G[X_j] = \lfloor (j+1)/2 \rfloor$ for $j \in [2k]$.

   Assume for contradiction that there exists another system

$$\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}', \mathbf{Rt}', \mathbf{Attr}', \mathbf{Rule}') \in \mathsf{GRP}^{k-1}$$

   with $M'_{\mathrm{c}}, M'_{\mathrm{q}}, G' \in \mathbf{Attr}$ such that $\mathcal{A} \simeq \mathcal{A}'$. We can further assume that $\mathbf{Obj}'_{\mathrm{c}} = \emptyset$ and $\mathbf{Rt}'_{\mathrm{c}} = \emptyset$, because otherwise $\mathcal{A}$ and $\mathcal{A}'$ will be obviously inequivalent. Using similar reasoning to that in the proof of $\mathsf{SUBSYS}^{k} \not\leq \mathsf{SUBSYS}^{k-1}$ in Theorem 10, we can also restrict that $\mathbf{Rt}'_{\mathrm{q}} = \{\texttt{CNOT}\}$.

   Consider an execution (S, P) with

$$P_u \equiv \textbf{for } l \in [k] \textbf{ do } \mathit{CNOT}[X_{2l-1}, X_{2l}] \textbf{ od}$$

   and $P_v \equiv \bot$. By our construction of $\mathcal{A}$, the history generated by (S, P) in $\mathcal{A}$ is

$$(u, \{X_1, X_2\}, \texttt{CNOT}), \ldots, (u, (X_{2k-1}, X_{2k}), \texttt{CNOT})$$

and authorised according to Definition 15. Since we assume $\mathcal{A} \simeq \mathcal{A}'$, the history generated by (S, P) in $\mathcal{A}'$ is also authorised, which implies $\texttt{CNOT} \in M'_{\mathrm{q}}[u, X_j]$ for $j \in [2k]$.

Observe that by the pigeonhole principle, there must exist distinct $j_1, j_2, j_3 \in [2k]$ such that $G'[X_{j_1}] = G'[X_{j_2}] = G'[X_{j_3}]$ and $G[X_{j_1}] \neq G[X_{j_2}]$.

Now consider another execution (S, P') with $P'_u \equiv CNOT[X_{j_1}, X_{j_2}]$ and $P'_v \equiv \perp$. The histories generated by (S, P') in $\mathcal{A}$ and $\mathcal{A}'$ are the same $(u, \{X_{j_1}, X_{j_2}\}, \texttt{CNOT})$. By our construction of $\mathcal{A}$, this history is unauthorised in $\mathcal{A}$ as $G[X_{j_1}] \neq G[X_{j_2}]$ (see the authorisation rule in Definition 15). However, it is authorised in $\mathcal{A}'$ because $G'[X_{j_1}] = G'[X_{j_2}]$. Hence, we obtain a contradiction and the conclusion follows.

2. Next we prove that $\mathsf{GRP}^{k-1} < \mathsf{GRP}^k$.

   Suppose that $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{GRP}^{k-1}$ with $M_{\mathrm{c}}, M_{\mathrm{q}}, G \in \mathbf{Attr}$. Then, we can define $\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}', \mathbf{Rule}) \in \mathsf{GRP}^k$ with $M'_{\mathrm{c}}, M'_{\mathrm{q}}, G' \in \mathbf{Attr}$, such that $M'_{\mathrm{c}} = M_{\mathrm{c}}$, $M'_{\mathrm{q}} = M_{\mathrm{q}}$, and $G'[o] = G[o]$ for any $o \in \mathbf{Obj}_{\mathrm{q}}$. It is easy to see that $\mathcal{A} \simeq \mathcal{A}'$ in this case.

□

Thirdly, the $\mathsf{M} = \mathsf{ENT}$ part of Theorem 10 can be restated as the following lemma.

Lemma 11. $\mathsf{ENT}^1 < \mathsf{ENT}^2$.

Proof.

1. First we prove $\mathsf{ENT}^2 \not\leq \mathsf{ENT}^1$. The proof idea is by observing that the attribute M_{e} of a system in ENT^2 records whether two quantum registers can be entangled, while M_{e} of a system in ENT^1 only records whether a quantum register can be entangled with others. Therefore, a system in ENT^2 has a more fine-grained control of entanglement than a system in ENT^1.

   Let us consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{ENT}^2$, where $\mathbf{Rt}_{\mathrm{c}} = \emptyset$, $\mathbf{Rt}_{\mathrm{q}} = \{\texttt{CNOT}, \texttt{measure}\}$, $\mathbf{Sub} = \{u, v\}$, $\mathbf{Obj}_{\mathrm{c}} = \emptyset$, and $\mathbf{Obj}_{\mathrm{q}} = \{X_1, X_2, X_3, X_4\}$. Here, `measure` means the ability to perform a complete measurement. Attributes $M_{\mathrm{c}}, M_{\mathrm{q}}, M_{\mathrm{e}}, D \in \mathbf{Attr}$ are initialised as follows. As $\mathbf{Obj}_{\mathrm{c}} = \emptyset$, we set $M_{\mathrm{c}} = \emptyset$. For $s \in \mathbf{Sub}, o \in \mathbf{Obj}_{\mathrm{q}}$: $M_{\mathrm{q}}[s, o] = \{\texttt{CNOT}, \texttt{measure}\}$ and $D[o] = \mathit{true}$. For $s \in \mathbf{Sub}, o \in \mathcal{P}_2\left(\mathbf{Obj}_{\mathrm{q}}\right)$:

$$M_{\mathrm{e}}[o] = \begin{cases} \mathit{true}, & o = \{X_1, X_2\} \vee o = \{X_3, X_4\}, \\ \mathit{false}, & o.w. \end{cases}$$

   Assume for contradiction that there exists another system

$$\mathcal{A}' = \left(\mathbf{Sub}, \mathbf{Obj}', \mathbf{Rt}', \mathbf{Attr}', \mathbf{Rule}'\right) \in \mathsf{ENT}^1$$

with $M'_c, M'_q, M'_e, D' \in \mathbf{Attr}'$ such that $\mathcal{A} \simeq \mathcal{A}'$. We can further assume that $\mathbf{Obj}'_c = \emptyset$ and $\mathbf{Rt}'_c = \emptyset$, because otherwise $\mathcal{A}$ and $\mathcal{A}'$ will be obviously inequivalent. Using similar reasoning to that in the proof of $\mathsf{SUBSYS}^k \not\leq \mathsf{SUBSYS}^{k-1}$ in Theorem 10, we can also restrict that $\mathbf{Rt}'_q = \{\texttt{CNOT}, \texttt{measure}\}$.

Consider an execution (S, P) with

$$P_u \equiv CNOT[X_1, X_2]; CNOT[X_3, X_4]$$

and $P_v \equiv \bot$. By our construction of $\mathcal{A}$, the history generated by (S, P) in $\mathcal{A}$ is

$$(u, \{X_1, X_2\}, \texttt{CNOT}), (u, \{X_3, X_4\}, \texttt{CNOT})$$

and authorised according to Definition 17.

On the other hand, since we assume $\mathcal{A} \simeq \mathcal{A}'$, the history generated by (S, P) in $\mathcal{A}'$ is also authorised. By Definition 16, this implies that $M'_e[o] = \mathit{true}$ for $o \in \{X_1, X_2, X_3, X_4\}$ (meaning any quantum register in $\mathcal{A}'$ can be entangled with others) and

$$\texttt{CNOT} \in M'_q[u, X_1] \cap M'_q[u, X_3].$$

Now consider another execution (S, P') with $P'_u \equiv CNOT[X_1, X_3]$ and $P'_v \equiv \bot$. The histories generated by (S, P') in $\mathcal{A}$ and $\mathcal{A}'$ are the same $(u, \{X_1, X_3\}, \texttt{CNOT})$. By our construction of $\mathcal{A}$, this history is unauthorised in $\mathcal{A}$ because $M_e[X_1, X_3] = \mathit{false}$ (see the authorisation rule in Definition 17). However, it is authorised in $\mathcal{A}'$ due to $M'_e[X_1] = M'_e[X_2] = \mathit{true}$ and $\texttt{CNOT} \in M'_q[u, X_1] \cap M'_q[u, X_3]$ (see the authorisation rule in Definition 16). Hence, we obtain a contradiction and the conclusion follows.

2. Next we prove $\mathsf{ENT}^1 \leq \mathsf{ENT}^2$.

   Consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{ENT}^1$ with $M_c, M_q, M_e, D \in \mathbf{Attr}$. Then, we can define $\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}', \mathbf{Rule}') \in \mathsf{ENT}^2$ with $M'_c, M'_q, M'_e, D' \in \mathbf{Attr}'$ such that $M'_c = M_c$, $M'_q = M_q$, and for any $o_1 \neq o_2 \in \mathbf{Obj}_q$: $M'_e[o_1, o_2] = M_e[o_1] \wedge M_e[o_2]$ and $D'[o_1, o_2] = D[o_1] \vee D[o_2]$. It is easy to see that $\mathcal{A} \simeq \mathcal{A}'$.

□

8.4.2 Flexibility of Different Models

Our second theorem presents a comparison between the flexibility of subsystem control, group control, and entanglement control. Let $\mathsf{SUBSYS} = \bigcup_k \mathsf{SUBSYS}^k$, and define GRP and ENT similarly. Let $\mathsf{SUBSYS}^{<N} = \bigcup_k \mathsf{SUBSYS}^k \cap \left\{\mathcal{A} : \mathcal{A} \text{ has } \left|\mathbf{Obj}_q\right| > k\right\}$ be the set of systems using k-subsystem control, where k is less than $\left|\mathbf{Obj}_q\right|$.

Theorem 11 (Comparison of Flexibility)**.** *The flexibility of* SUBSYS, GRP, ENT *can be compared as follows.*

1. $\mathsf{SUBSYS} \not\leq \mathsf{GRP}, \mathsf{ENT}$.
2. $\mathsf{GRP} \not\leq \mathsf{ENT}, \mathsf{SUBSYS}^{<N}$ *and* $\mathsf{GRP} \leq \mathsf{SUBSYS}$.
3. $\mathsf{ENT} \not\leq \mathsf{SUBSYS}, \mathsf{GRP}$.

For simplicity of presentation, let us prove Theorem 11 by proving the following three lemmas, for Items 1 to 3, respectively. Firstly, Item 1 of Theorem 11 can be restated as the following lemma.

Lemma 12. $\mathsf{SUBSYS} \not\leq \mathsf{GRP}, \mathsf{ENT}$.

Proof.

1. We first prove that $\mathsf{SUBSYS} \not\leq \mathsf{GRP}$. The proof idea is similar to that for proving $\mathsf{ENT}^2 \not\leq \mathsf{ENT}^1$. Intuitively, the attribute M_q in a system in SUBSYS records information about subsystems which consists of multiple quantum registers, while the attributes M_q, G in a system in GRP only records information about each individual quantum register. In some cases, the former provides a more fine-grained control of quantum operations than the latter.

   Let us consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{SUBSYS}$, where $\mathbf{Rt}_\mathrm{c} = \emptyset$, $\mathbf{Rt}_\mathrm{q} = \{\mathtt{CNOT}\}$, $\mathbf{Sub} = \{u, v\}$, $\mathbf{Obj}_\mathrm{c} = \emptyset$, and $\mathbf{Obj}_\mathrm{q} = \{X_1, X_2, X_3\}$. Attributes $M_\mathrm{c}, M_\mathrm{q} \in \mathbf{Attr}$ are initialised as follows. As $\mathbf{Obj}_\mathrm{c} = \emptyset$, we set $M_\mathrm{c} = \emptyset$. For $s \in \mathbf{Sub}, o \subseteq \mathbf{Obj}_\mathrm{q}$.

$$M_\mathrm{q}[s, o] = \begin{cases} \{\mathtt{CNOT}\}, & s = u \wedge (o = \{X_1, X_2\} \vee o = \{X_2, X_3\}), \\ \emptyset, & o.w. \end{cases}$$

   Assume for contradiction that there exists another system

$$\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}', \mathbf{Rt}', \mathbf{Attr}', \mathbf{Rule}') \in \mathsf{GRP}$$

   with $M'_\mathrm{c}, M'_\mathrm{q}, G' \in \mathbf{Attr}'$ such that $\mathcal{A}' \simeq \mathcal{A}$. We can further assume that $\mathbf{Obj}'_\mathrm{c} = \emptyset$ and $\mathbf{Rt}'_\mathrm{c} = \emptyset$, because otherwise $\mathcal{A}$ and $\mathcal{A}'$ will be obviously inequivalent. Using similar reasoning to that in the proof of $\mathsf{SUBSYS}^k \not\leq \mathsf{SUBSYS}^{k-1}$ in Theorem 10, we can also restrict that $\mathbf{Rt}'_\mathrm{q} = \{\mathtt{CNOT}\}$.

   Consider an execution (S, P) with

$$P_u \equiv CNOT[X_1, X_2]; CNOT[X_2, X_3]$$

and $P_v \equiv \bot$. By our construction of $\mathcal{A}$, the history generated by (S, P) in $\mathcal{A}$ is

$$(u, \{X_1, X_2\}, \texttt{CNOT}), \{u, \{X_2, X_3\}, \texttt{CNOT}\}$$

and authorised. Since we assume $\mathcal{A} \simeq \mathcal{A}'$, the history generated by (S, P) in $\mathcal{A}'$ is also be authorised. By the authorisation rule in Definition 15, this implies that $\texttt{CNOT} \in M'_{\mathrm{q}}[X_1] \cap M'_{\mathrm{q}}[X_3]$ and $G'[X_1] = G'[X_2] = G'[X_3]$.

Now we consider another execution (S, P'), with $P'_u \equiv CNOT[X_1, X_3]$ and $P'_v \equiv \bot$. The histories generated by (S, P') in $\mathcal{A}$ and $\mathcal{A}'$ are the same $(u, \{X_1, X_3\}, \texttt{CNOT})$. By our construction of $\mathcal{A}$, this history is unauthorised in $\mathcal{A}$ as $\texttt{CNOT} \notin M_{\mathrm{q}}[u, \{X_1, X_3\}]$ (see the authorisation rule in Definition 14). However, it is authorised in $\mathcal{A}'$ because $\texttt{CNOT} \in M'_{\mathrm{q}}[X_1] \cap M'_{\mathrm{q}}[X_3]$ and $G'[X_1] = G'[X_3]$ (see the authorisation rule in Definition 15). Hence, we obtain a contradiction and the conclusion follows.

2. Next we prove that $\mathsf{SUBSYS} \not\leq \mathsf{ENT}$. The proof idea is essentially using the difference between control of quantum operations and control of entanglement. In particular, a system in SUBSYS controls whether a quantum operation is authorised or unauthorised, and does not force a measurement to be applied before modifying M_{q}. In contrast, a system in ENT controls whether entanglement is permitted to exist between quantum registers, and a measurement has to be applied if there is no guarantee of disentanglement, before we modify M_{e} to disentangle two objects.

   Specifically, let us prove $\mathsf{SUBSYS} \not\leq \mathsf{ENT}^2$. Consider a system

$$\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{SUBSYS},$$

   where $\mathbf{Rt}_{\mathrm{c}} = \{\texttt{read}, \texttt{write}\}$, $\mathbf{Rt}_{\mathrm{q}} = \{\texttt{H}, \texttt{CNOT}, \texttt{measure}\}$, $\mathbf{Sub} = \{u, v\}$, $\mathbf{Obj}_{\mathrm{c}} = \{M_{\mathrm{q}}\}$, and $\mathbf{Obj}_{\mathrm{q}} = \{X_1, X_2, X_3\}$. Here, $\mathbf{Obj}_{\mathrm{c}} = \{M_{\mathrm{q}}\}$ implies that M_{q} can be dynamically modified. Attributes $M_{\mathrm{c}}, M_{\mathrm{q}} \in \mathbf{Attr}$ are initialised as follows. For $s \in \mathbf{Sub}, o \in \mathbf{Obj}_{\mathrm{c}}$:

$$M_{\mathrm{c}}[s, o] = \begin{cases} \{\texttt{read}, \texttt{write}\}, & s = u, \\ \emptyset, & o.w. \end{cases}$$

   For $s \in \mathbf{Sub}, o \subseteq \mathbf{Obj}_{\mathrm{q}}$: $M_{\mathrm{q}}[s, o] = \{\texttt{H}, \texttt{CNOT}\}$.

   Assume for contradiction that there exists another system

$$\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}', \mathbf{Rt}', \mathbf{Attr}', \mathbf{Rule}') \in \mathsf{ENT}^2$$

   with $\mathbf{Obj}' = \mathbf{Obj}'_{\mathrm{c}} \cup \mathbf{Obj}_{\mathrm{q}}$ and $M'_{\mathrm{c}}, M'_{\mathrm{q}}, M'_{\mathrm{e}}, D' \in \mathbf{Attr}'$ such that $\mathcal{A}' \simeq \mathcal{A}$. Using similar reasoning to that in the proof of $\mathsf{SUBSYS}^k \not\leq \mathsf{SUBSYS}^{k-1}$ in Theorem 10, we can further restrict that $\mathbf{Rt}'_{\mathrm{q}} = \{\texttt{H}, \texttt{CNOT}, \texttt{measure}\}$.

Consider an execution (S, P). Here, the scheduler S is defined by $S(\alpha(0), \ldots, \alpha(t-1)) = s(t)$, where $s(0) = s(1) = s(2) = u$ and $s(3) = s(4) = v$. The program P is defined by

$$P_u \equiv H[X_1]; CNOT[X_1, X_2]; forbid(v, \{X_1, X_2\})$$

and $P_v \equiv \perp$, where $forbid(v, \{X_1, X_2\})$ means to modify attributes such that future request $(v, \{X_1, X_2\}, r)$ will be unauthorised for any right r. By our construction of $\mathcal{A}$, the history α generated by (S, P) in $\mathcal{A}$ is

$$(u, \{X_1\}, \mathtt{H}), (u, \{X_1, X_2\}, \mathtt{CNOT}), \{u, M_q[v, \{X_1, X_2\}], \mathtt{read}\}, (u, M_q[v, \{X_1, X_2\}], \mathtt{write})$$

and authorised.

On the other hand, since we assume $\mathcal{A}' \simeq \mathcal{A}$, the history α' generated by (S, P) in $\mathcal{A}'$ is also authorised. Note that according to the authorisation rule in Definition 17, whether the future request $(v, \{X_1, X_2\}, r)$ will be authorised in $\mathcal{A}'$ is determined by the attributes $M'_e[X_1, X_2]$, $M'_q[v, X_1]$, and $M'_q[v, X_2]$. As P_u contains $forbid(v, \{X_1, X_2\})$, the above implies the following two cases:

- Either there exists $t_3 \in \mathbb{N}$ such that $\alpha'(t_3) = (u, M'_e[X_1, X_2], r)$ for some right r that modifies (e.g., `write`) $M'_e[X_1, X_2]$ to *false*. In this case, after $CNOT[X_1, X_2]$ in P_u is executed, the quantum state of X_1, X_2 becomes

  $$\frac{1}{\sqrt{2}}\left(|0\rangle_{X_1}|0\rangle_{X_2} + |1\rangle_{X_1}|1\rangle_{X_2}\right),$$

  and we also have $D'[X_1, X_2] = false$ at some time $t_1 < t_3$ (meaning that X_1, X_2 are not ensured to be disentangled), due to the post-update rule in Definition 17. Moreover, according to Definition 17, this implies that there exists $t_2 \in (t_1, t_3)$ with $\alpha'(t_2) = (s, X, \mathtt{measure})$ for some $s \in \mathbf{Sub}$ and $X \in \{X_1, X_2\}$. However, such α' cannot be generated from program P, because P does not contain measurement.
- Or there exists $t \in \mathbb{N}$ such that $\alpha'(t) = \left(u, M'_q[v, X], r\right)$ for some right r that modifies (e.g., `write`) $M'_q[v, X]$ and removes `CNOT` from it, where $X \in \{X_1, X_2\}$. In this case, after the final request of α', we have $\mathtt{CNOT} \notin M'_q[v, X]$.
  Now consider another execution (S, P') with $P'_u = P_u$ and

  $$P'_v \equiv CNOT[X_2, X_3]; CNOT[X_1, X_3].$$

  By the construction of the scheduler S, the history generated by (S, P') in $\mathcal{A}$ is $\beta = \alpha, (v, \{X_2, X_3\}, \mathtt{CNOT}), (v, \{X_1, X_3\}, \mathtt{CNOT})$, where α is the history generated by (S, P) in $\mathcal{A}$ previously. Since after α, we still have $\mathtt{CNOT} \in M_q[v, \{X_1, X_3\}] \cap$

$M_q[v, \{X_2, X_3\}]$, the history β is authorised in $\mathcal{A}$ (see the authorisation rule in Definition 14). Similarly, the history generated by (S, P') in $\mathcal{A}'$ is

$$\beta' = \alpha', (v, \{X_2, X_3\}, \texttt{CNOT}), (v, \{X_1, X_3\}, \texttt{CNOT}),$$

where α' is the history generated by (S, P) in $\mathcal{A}'$ previously. However, β' is unauthorised in $\mathcal{A}'$, because after α', we have $\texttt{CNOT} \notin M'_q[v, X]$ for some $X \in \{X_1, X_2\}$ (see the authorisation rule in Definition 17).

In either case, we obtain a contradiction and the conclusion follows.

□

Secondly, Item 2 of Theorem 11 can be restated as the following lemma.

Lemma 13. $\mathsf{GRP} \not\leq \mathsf{ENT}, \mathsf{SUBSYS}^{<N}$ *and* $\mathsf{GRP} \leq \mathsf{SUBSYS}$,

Proof.

1. We first prove that $\mathsf{GRP} \not\leq \mathsf{SUBSYS}^{<N}$. The proof idea is by noticing that a system in $\mathsf{SUBSYS}^{<N}$ only allows quantum operations on subsystem of size $< \left|\mathbf{Obj}_q\right|$, while a system in GRP can allow quantum operations on subsystem of size $\left|\mathbf{Obj}_q\right|$.

   Let us consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{GRP}$, where $\mathbf{Rt}_c = \emptyset$, $\mathbf{Rt}_q = \{\texttt{QFT}_k\}$, $\mathbf{Sub} = \{u, v\}$, $\mathbf{Obj}_c = \emptyset$, and $\mathbf{Obj}_q = \{X_1, \ldots, X_k\}$. Here, $\texttt{QFT}_k$ means the ability to perform a quantum Fourier transform circuit QFT_k on k qubits. Attributes $M_c, M_q, G \in \mathbf{Attr}$ are initialised as follows. Since $\mathbf{Obj}_c = \emptyset$, we set $M_c = \emptyset$. For any $s \in \mathbf{Sub}, o \in \mathbf{Obj}_q$:

   $$M_q[s, o] = \begin{cases} \texttt{QFT}_k, & s = u, \\ \emptyset, & o.w. \end{cases}$$

   and $G[o] = 1$.

   Consider another system $\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}', \mathbf{Attr}', \mathbf{Rule}') \in \mathsf{SUBSYS}^{<N}$ with $M'_c, M'_q \in \mathbf{Attr}'$. Suppose that $\mathcal{A}' \in \mathsf{SUBSYS}^{k'}$ for some $k' < \left|\mathbf{Obj}_q\right| = k$.

   Consider an execution (S, P) with $P_u \equiv QFT_k$ and $P_v \equiv \perp$. By our construction of $\mathcal{A}$, the history generated by (S, P) in $\mathcal{A}$ is simply $(u, \{X_1, \ldots, X_k\}, \texttt{QFT}_k)$ and authorised. Now consider the history α generated by (S, P) in $\mathcal{A}'$. There are two cases: either α contains multiple requests like in the proof of $\mathsf{SUBSYS}^k \not\leq \mathsf{SUBSYS}^{k-1}$ in Theorem 10, and then we can derive $\mathcal{A} \not\approx \mathcal{A}'$; or $\alpha = (u, \{X_1, \ldots, X_k\}, \texttt{QFT}_k)$, which is unauthorised due to the authorisation rule in Definition 14 and $k = \left|\mathbf{Obj}_q\right| > k'$. In either case, $\mathcal{A} \not\approx \mathcal{A}'$ and the conclusion follows.

2. Next we prove $\mathsf{GRP} \not\leq \mathsf{ENT}$. Like in the proof of $\mathsf{SUBSYS} \not\leq \mathsf{ENT}$ in Lemma 12, the idea is essentially using the difference between control of quantum operations and control of entanglement. Specifically, let us prove $\mathsf{GRP} \not\leq \mathsf{ENT}^2$.

   Consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{GRP}$, where $\mathbf{Rt}_\mathrm{c} = \{\mathtt{read}, \mathtt{write}\}$, $\mathbf{Rt}_\mathrm{q} = \{\mathtt{H}, \mathtt{CNOT}, \mathtt{measure}\}$, $\mathbf{Sub} = \{u, v\}$, $\mathbf{Obj}_\mathrm{c} = \{G\}$, and $\mathbf{Obj}_\mathrm{q} = \{X_1, X_2, X_3, X_4\}$. Note that $\mathbf{Obj}_\mathrm{c} = \{G\}$ means G can be dynamically modified. Attributes $M_\mathrm{c}, M_\mathrm{q}, G \in \mathbf{Attr}$ are initialised as follows. For $s \in \mathbf{Sub}, o \in \mathbf{Obj}_\mathrm{c}$:

$$M_\mathrm{c}[s, o] = \begin{cases} \{\mathtt{read}, \mathtt{write}\}, & s = u, \\ \emptyset, & o.w. \end{cases} \tag{8.2}$$

   For $s \in \mathbf{Sub}, o \in \mathbf{Obj}_\mathrm{q}$: $M_\mathrm{q}[s, o] = \{\mathtt{H}, \mathtt{CNOT}\}$. Let $G[X_1] = G[X_2] = G[X_3] = 1$ and $G[X_4] = 2$.

   Assume for contradiction that there exists another system

$$\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}', \mathbf{Rt}', \mathbf{Attr}', \mathbf{Rule}') \in \mathsf{ENT}^2$$

   with $\mathbf{Obj}' = \mathbf{Obj}'_\mathrm{c} \cup \mathbf{Obj}_\mathrm{q}$ and $M'_\mathrm{c}, M'_\mathrm{q}, M'_\mathrm{e}, D' \in \mathbf{Attr}'$ such that $\mathcal{A}' \simeq \mathcal{A}$. Using similar reasoning to that in the proof of $\mathsf{SUBSYS}^k \not\leq \mathsf{SUBSYS}^{k-1}$ in Theorem 10, we can further restrict that $\mathbf{Rt}'_\mathrm{q} = \{\mathtt{H}, \mathtt{CNOT}, \mathtt{measure}\}$.

   Consider an execution (S, P). Here, the scheduler S is defined by $S(\alpha(0), \ldots, \alpha(t-1)) = s(t)$, where $s(0) = s(1) = s(2) = u$ and $s(3) = s(4) = v$. The program P is defined by

$$P_u \equiv H[X_1]; CNOT[X_1, X_2]; newgrp(X_1, X_4)$$

   and $P_v \equiv \bot$, where $newgrp(X_1)$ means to modify attributes such that X_1, X_4 are put into a new group and any quantum operation on X_1 or X_4 should act within this group; i.e., for future request (s, o, r), if $o \cap \{X_1, X_4\} \neq \emptyset$, then $o \subseteq \{X_1, X_4\}$. By our construction of $\mathcal{A}$, the history generated by (S, P) in $\mathcal{A}$ is

$$\begin{aligned} &(u, \{X_1\}, \mathtt{H}), (u, \{X_1, X_2\}, \mathtt{CNOT}), (u, G[X_1], \mathtt{read}), \\ &\quad (u, G[X_1], \mathtt{write}), (u, G[X_4], \mathtt{read}), (u, G[X_4], \mathtt{write}) \end{aligned}$$

   and authorised.

   The remaining reasoning is similar to the proof of $\mathsf{SUBSYS} \not\leq \mathsf{ENT}^2$ in Lemma 12. Since we assume $\mathcal{A} \simeq \mathcal{A}'$, the history α' generated by (S, P) in $\mathcal{A}'$ is also authorised. This implies initially $M'_\mathrm{q}[X_1, X_2] = \mathit{true}$. According to the authorisation rule in Definition 15, whether the future request (s, o, r) with $o \cap \{X_1, X_4\} \neq \emptyset$ will be authorised is determined by the attributes $M'_\mathrm{e}[X_1, X]$ for $X \neq X_1$, $M'_\mathrm{e}[X_4, X]$ for

$X \neq X_4$, $M'_q[X_1]$ and $M'_q[X_4]$. As P_u contains $newgrp(X_1, X_4)$, which forbids future request like $(v, \{X_1, X_2\}, r)$, the above implies the following two cases:

- Either there exists $t \in \mathbb{N}$ such that $\alpha'(t) = (u, M'_e[X_1, X_2], r)$ for some right r that modifies $M'_e[X_1, X_2]$ to *false*. In this case, using the same reasoning as in the proof of SUBSYS $\not\leq$ ENT^2, we can derive $\mathcal{A} \not\simeq \mathcal{A}'$.
- Or there exists $t \in \mathbb{N}$ such that $\alpha'(t) = \left(u, M'_q[v, X], r\right)$ for some right r that modifies (e.g., `write`) $M'_q[v, X]$ and removes `CNOT` from it, where $X \in \{X_1, X_2\}$. In this case, after the final request of α', we have $\texttt{CNOT} \notin M'_q[v, X]$.
  Now consider another execution (S, P') with $P'_u = P_u$ and

  $$P'_v \equiv CNOT[X_1, X_4]; CNOT[X_2, X_3].$$

  By the construction of the scheduler S, the history generated by (S, P') in $\mathcal{A}$ is

  $$\beta = \alpha, (v, \{X_1, X_4\}, \texttt{CNOT}), (v, \{X_2, X_3\}, \texttt{CNOT}).$$

  Since after α, we still have $\texttt{CNOT} \in M_q[v, Y]$ for $Y \in \{X_1, X_2, X_3, X_4\}$, $G[X_1] = G[X_4]$, and $G[X_2] = G[X_3]$, the history β is authorised in $\mathcal{A}$ (see the authorisation rule in Definition 15). Similarly, the history β' generated by (S, P') in $\mathcal{A}'$ is

  $$\beta' = \alpha', (v, \{X_1, X_4\}, \texttt{CNOT}), (v, \{X_2, X_3\}, \texttt{CNOT}).$$

  However, β' is unauthorised in $\mathcal{A}'$, because after α', we have $\texttt{CNOT} \notin M'_q[v, X]$ for some $X \in \{X_1, X_2\}$ (see the authorisation rule in Definition 17), .

In either case, we obtain a contradiction and the conclusion follows.

3. Finally we prove GRP $\leq$ SUBSYS.

   Consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in$ GRP with $M_c, M_q, G \in \mathbf{Attr}$. Then, we can define $\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}', \mathbf{Rule}') \in$ SUBSYS, where $M'_c, M'_q \in \mathbf{Attr}'$ are defined by $M'_c = M_c$, and for $s \in \mathbf{Sub}, o \subseteq \mathbf{Obj}_q$:

   $$M'_q[s, o] = \begin{cases} \bigcap_{X \in o} M_q[s, X], & \forall X, Y \in o, G[X] = G[Y], \\ \emptyset, & o.w. \end{cases}$$

   It is easy to see that $\mathcal{A} \simeq \mathcal{A}'$ from this construction.

□

Finally, Item 3 of Theorem 11 can be restated as the following lemma.

Lemma 14. ENT $\not\leq$ SUBSYS, GRP.

Proof. Let us only prove $\mathsf{ENT}^1 \not\leq \mathsf{SUBSYS}$. Then, $\mathsf{ENT} \not\leq \mathsf{GRP}$ easily follows from $\mathsf{GRP} \leq \mathsf{SUBSYS}$ in Lemma 13. The proof idea is essentially using the difference between control of quantum operations and control of entanglement. In particular, ENT uses attribute D to record approximate knowledge of disentanglement, allowing a system in ENT to make authorisation decisions based on this extra information about existing entanglement. In contrast, a system in SUBSYS cannot (even approximately) distinguish whether entanglement has been established or not, and its authorisation rule is independent of the quantum state.

Let us consider a system $\mathcal{A} = (\mathbf{Sub}, \mathbf{Obj}, \mathbf{Rt}, \mathbf{Attr}, \mathbf{Rule}) \in \mathsf{ENT}$, where $\mathbf{Sub} = \{u, v\}$, $\mathbf{Obj}_\mathrm{c} = \{M_\mathrm{e}\}$, $\mathbf{Obj}_\mathrm{q} = \{X_1, X_2\}$, $\mathbf{Rt}_\mathrm{c} = \{\mathtt{read}, \mathtt{write}\}$, and $\mathbf{Rt}_\mathrm{q} = \{\mathtt{CNOT}, \mathtt{measure}\}$. Here, `CNOT` means the ability to perform a *CNOT* gate, and `measure` means the ability to perform a computational basis measurement. $M_\mathrm{e} \in \mathbf{Obj}_\mathrm{c}$ implies that attribute M_e can be dynamically modified by users. Attributes $M_\mathrm{c}, M_\mathrm{q}, M_\mathrm{e}, D \in \mathbf{Attr}$ in $\mathcal{A}$ are initialised as follows. For $s \in \mathbf{Sub}, o \in \mathbf{Obj}_\mathrm{c}$:

$$M_\mathrm{c}[s, o] = \begin{cases} \{\mathtt{read}, \mathtt{write}\}, & s = u, \\ \emptyset, & o.w. \end{cases}$$

For $s \in \mathbf{Sub}, o \subseteq \mathbf{Obj}_\mathrm{q}$: $M_\mathrm{q}[s, o] = \{\mathtt{CNOT}, \mathtt{measure}\}$, $M_\mathrm{e}[o] = \mathit{true}$, and $D[o] = \mathit{true}$.

Assume for contradiction that there exists another system

$$\mathcal{A}' = (\mathbf{Sub}, \mathbf{Obj}', \mathbf{Rt}', \mathbf{Attr}', \mathbf{Rule}') \in \mathsf{SUBSYS}$$

with $M'_\mathrm{c}, M'_\mathrm{q} \in \mathbf{Attr}'$ such that $\mathcal{A}' \simeq \mathcal{A}$. Note that we assume $\mathcal{A}'$ has the same **Sub** as that of $\mathcal{A}$ because otherwise they will be obviously inequivalent.

Consider an execution (S, P) with $P_u \equiv \mathit{disent}(X_1)$ and $P_v \equiv \perp$, where $\mathit{disent}(X_1)$ means to modify attributes such that quantum register X_1 is disentangled from others, and $\perp$ denotes termination without doing anything. By our construction of $\mathcal{A}$, the history generated by (S, P) in $\mathcal{A}$ is

$$(u, M_\mathrm{e}[X_1], \mathtt{read}), (u, M_\mathrm{e}[X_1], \mathtt{write})$$

and is authorised. Note that during the execution, the value of $M_\mathrm{e}[X_1]$ will be modified from *true* to *false*, and the value of $D[X_1]$ is always *true*.

Consider another execution (S, P') with

$$P'_u \equiv H[X_1]; \mathit{CNOT}[X_1, X_2]; \mathit{disent}(X_1)$$

and $P'_v \equiv \bot$. The history generated by (S, P') in $\mathcal{A}$ is

$$(u, \{X_1\}, \texttt{H}), (u, \{X_1, X_2\}, \texttt{CNOT}), (u, M_{\text{e}}[X_1], \texttt{read}), (u, M_{\text{e}}[X_1], \texttt{write}).$$

This history is unauthorised because the post-update rule in Definition 16 modifies $D[X_1]$ and $D[X_2]$ to *false* after the second request, when the quantum state of X_1, X_2 becomes $\frac{1}{\sqrt{2}}\left(|0\rangle_{X_1}|0\rangle_{X_2} + |1\rangle_{X_1}|1\rangle_{X_2}\right)$, which is entangled. Then, the last request modifying $M_{\text{e}}[X_1]$ will be denied by the authorisation rule.

On the other hand, suppose that the histories generated by (S, P) and (S, P') in $\mathcal{A}'$ are α and α', respectively. Since we assume $\mathcal{A} \simeq \mathcal{A}'$, by Definition 11, α is authorised and α' is unauthorised. Observe that α is a suffix of α': we have $\alpha' = \beta, \alpha$ for some sequence β of requests generated from executing $H[X_1]; CNOT[X_1, X_2]$ in P'_u. This is because the authorisation rule of $\mathcal{A}'$ (see Definition 14) is based on attributes $M'_{\text{c}}, M'_{\text{q}}$, which are unchanged by β. Further, this implies α' should be authorised, because the prefix β does not change $M'_{\text{c}}, M'_{\text{q}}$ and will not affect whether the suffix α is authorised. Hence, we obtain a contradiction and the conclusion follows. □

8.5 Discussion

8.5.1 Related Work

Quantum Access Control

The work [116] first studied access control in quantum computing from the perspective of information-flow security. Their observation that rights should be specified for quantum subsystems motivated our Definition 13 and the $k = \left|\mathbf{Obj}_{\text{q}}\right|$ case of Definition 14, as mentioned in Section 8.1. However, they did not provide any explicit scenario demonstrating how *entanglement can leak secret information beyond direct communication*. In contrast, our Chapter 7 presented the first explicit scenario of how a classically secure access control system becomes insecure when adapted to the quantum setting, with a rigorous proof. Revealing this threat from entanglement enabled us to design new quantum access control models in Section 8.1. Other related work [323] has studied entanglement accessibility in the context of the quantum internet, which is a distinct topic from access control in computer security we address here.

Quantum Operating Systems

Operating systems are a major area where access control mechanisms have been extensively studied and implemented. In quantum computing, numerous efforts have been

devoted to tackling specific issues relevant to operating systems. These include task decomposition (due to the scarcity of qubits in existing quantum hardware, and typically via quantum circuit cutting, e.g., [89–96]), job scheduling (e.g., [100–103]), multiprogramming (e.g., [101, 104–109]), memory management (e.g., [111–115]), and concurrency (e.g., [36, 119, 120, 127–129, 142–144, 182]). Additionally, there are efforts on more holistic approaches to designing quantum operating systems [84–88]. It can be expected that quantum access control will become increasingly crucial to the security of quantum and classical-quantum hybrid computer systems when various quantum operating systems are deployed in the future.

8.5.2 Conclusion and Open Questions

This chapter addressed the threat from quantum entanglement to access control, first identified in Chapter 7. We proposed three new quantum access control models, including subsystem control, group control, and entanglement control. Our analysis confirmed that all three models are secure against threats from entanglement, yet they offer a trade-off between flexibility and efficiency. This allows the system designers to select or hybridise models based on specific practical needs.

This work is merely a first step towards access control for quantum computers. To conclude Part II of this thesis, we list several topics for future research. Firstly, to prevent security breaches, an immediate next step is to integrate new access control mechanisms into the design of future classical-quantum hybrid systems (including quantum-centric supercomputing systems [79–83]). This involves further refining the quantum access control models proposed in this thesis to accommodate practical requirements. Secondly, analogous to classical access control [360], designing more efficient data structures to store the models' attributes remains an important open problem, which could potentially leverage unique quantum properties. Thirdly, our entanglement control model only uses complete measurements to ensure disentanglement, which is a coarse-grained approximation. Exploring finer-grained approximations that offer greater flexibility (possibly at the cost of efficiency) is a promising future direction.

Part III

Concurrency

Chapter 9

Atomicity: From Classical to Quantum

Atomic actions are a fundamental concept in distributed computing. Nearly all distributed computing models, including classical and quantum, build upon the assumption that actions are atomic. While this atomicity assumption has well-established guarantees in the classical setting, its foundation in quantum computing remains largely unexamined.

The three chapters of Part III of this thesis focus on the *rigorous basis* for atomicity in distributed quantum computing. This chapter begins our investigation by demonstrating the non-triviality of guaranteeing atomicity in the quantum setting, even for the atomicity of local actions. We identify the unique challenges posed by quantum entanglement and the measurement problem, and provide an overview of how they will be addressed in the subsequent chapters.

9.1 Introduction

Distributed Systems and Atomic Actions

A distributed system is a collection of processes that take actions concurrently. Nearly all models of distributed systems build upon the **atomicity assumption**: that all actions are atomic. Atomic actions are considered indivisible, meaning that for any two atomic actions a and b (potentially from different processes), a temporal order exists: either a precedes b, or b precedes a. In reality, however, actions can be non-atomic, where two actions a and b can be concurrent with no defined temporal order at all [10]. The atomicity assumption is crucial because it significantly reduces the *non-determinism* from concurrency, which simplifies the modeling and analysis of a distributed system. For example, consider a trivial system composed of two processes: one executes $x := 100$ and the other executes $y := x$, with x and y both initialised to 0. Without the atomicity assumption, even such a simple system can exhibit complicated behaviour. As the first process updates variable x from 0 to 100, the second process might read a corrupted intermediate

value from x (e.g., 4) and write it into y. However, if we assume all actions are atomic, the system's behaviour can be modeled as a simple interleaving of atomic actions, and consequently the value of y is guaranteed to be either 0 or 100.

From Classical to Quantum

In classical computing, the atomicity assumption is supported by well-established guarantees at both the hardware and software levels. At the hardware level, modern architectures often provide atomic instructions for actions on the memories, such as the compare-and-swap instruction in the x86 architecture. At the software level, sophisticated methods exist to construct high-level atomic actions, either from low-level atomic actions (e.g., via semaphores [4]), or from non-atomic actions (e.g., building atomic registers from safe registers [178, 179]). Further discussion of this extensive work can be found in Section 9.4.1.

The need for a similar foundation in distributed quantum computing is growing. As aforementioned in Chapter 1, numerous efforts are underway to build distributed quantum computers [361]. From the software perspective, nearly all existing distributed quantum models (e.g., quantum process algebra [127–129], quantum versions of the LOCAL [130–132, 134] and CONGEST model [136–141], and distributed quantum programming [36, 142–144]) and algorithms (e.g., for distributed coordination problems [119–122] and graph problems [123–126]) either implicitly or explicitly rely on the atomicity assumption.

Despite its widespread use, even the concept of atomicity itself has not been rigorously studied in the quantum setting. This critical gap in our understanding leads us to the following question:

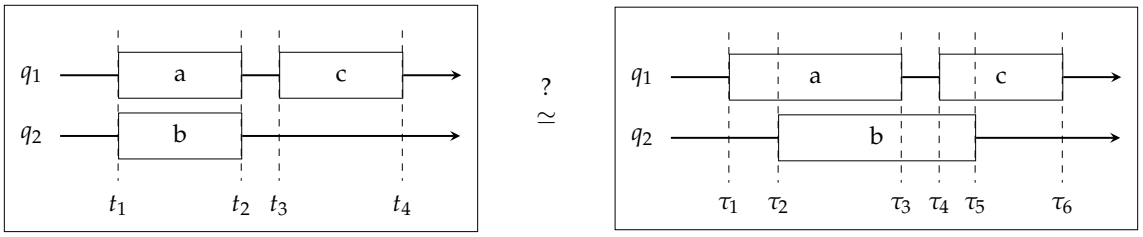
Question 2. *Can the atomicity assumption be rigorously guaranteed in distributed quantum systems?*

Note that actions can be classified into two types:

- *Local actions* that never overlap in space-time with other actions.
- *Non-local actions* that may overlap in space-time with other actions.

Indeed, even guaranteeing the atomicity of local actions — the ostensibly simpler case — is non-trivial in the quantum setting, as a series of motivating examples in Section 9.2 will demonstrate. Further, classical guarantees cannot be naively applied in the quantum setting, as we will discuss in Section 9.4.1.

Therefore, this chapter, along with the subsequent ones, will partially answer Question 2 by establishing a rigorous guarantee for the atomicity of local actions in distributed quantum systems. More precisely, we prove:



Synchronous S_1 Asynchronous S_1'

FIGURE 9.1: Proving the synchronous system S_1 is equivalent to the asynchronous system S_1'. Here, q_1, q_2 are qubits, and a, b, c are actions. The arrow of time is from the left to the right.

Theorem 12 (Atomicity of local quantum actions; informal version of Theorem 14)**.** *Any distributed quantum system S is equivalent to another system S' in which all local actions are atomic.*

9.2 Motivating Examples — A Simpler Task

Let us start with a series of examples, each building upon the previous, which reveal the subtleties involved in proving the seemingly simple Theorem 12. For the purpose of this illustration, we temporarily restrict our focus to *deterministic* quantum systems that are *terminating* and contain *no* non-local actions. The equivalence of two such systems S and S', denoted by $S \simeq S'$, is naturally defined as follows: (a) they implement the same logical quantum circuits; and (b) for any input quantum state, they produce the identical output state.

Further, we consider an even **simpler task**: proving the equivalence between *synchronous* and *asynchronous* [1] distributed quantum systems that implement the same logical quantum circuits. In a synchronous system, the time intervals of actions are aligned, while in an asynchronous system they are not. Since any system with atomic actions is inherently synchronous, the task considered here is weaker than proving our main goal in Theorem 12.

Example 6. In Figure 9.1, we present two real-time distributed quantum systems S_1 and S_1' implementing the same logical quantum circuits. We can think of them as a space-time diagram. Here, q_1 and q_2 are two qubits, spanning the space; and a, b, and c are actions, each specifying the qubits it acts on and the real-time interval it spans. System S_1 is an ideal synchronous implementation, and S_1' is an asynchronous implementation in reality.

[1] In Part III of this thesis, we adopt the non-degenerate global-time model [177, Section 3.1]: (a) there exists a global time; and (b) in a process, no starting or finishing time of two actions are identical. Note that the existence of global (physical) time is not equivalent to a global (logical) clock. Different processes in a system can use different logical clocks.

Difficulty posed by Example 6 Now let us attempt to show the real S_1' is equivalent to the ideal S_1. In Figure 9.1, we have identified several time instants $t_1, \ldots, t_4$ (resp., $\tau_1, \ldots, \tau_6$), for S_1 (resp., S_1'). We use $|\psi(t)\rangle$ (resp., $|\psi'(t)\rangle$) to denote the quantum state of S_1 (resp., S_1') at time t. Then our goal, according to the previous informal definition of equivalence, is to show: for any $|\psi(t_1)\rangle = |\psi'(\tau_1)\rangle$, we have $|\psi(t_4)\rangle = |\psi'(\tau_6)\rangle$.

The difficulty in showing this goal lies in the following observation.

- For the synchronous S_1, we can derive $|\psi(t_4)\rangle = (ca \otimes b)\,|\psi(t_1)\rangle$ by identifying the relations: $|\psi(t_2)\rangle = (a \otimes b)\,|\psi(t_1)\rangle$, $|\psi(t_3)\rangle = |\psi(t_2)\rangle$, and $|\psi(t_4)\rangle = (c \otimes \mathbb{1})\,|\psi(t_3)\rangle$. Here, by a slight abuse of notation, we use the same notation for an action and the operation it performs.

- For the asynchronous S_1': we can only obtain relations like $|\psi'(\tau_3)\rangle = (a \otimes b')\,|\psi'(\tau_1)\rangle$, for some unknown partial action b' of b. In this case, even relating $|\psi'(\tau_1)\rangle$ and $|\psi'(\tau_6)\rangle$ using a, b, and c is difficult.

Solution to Example 6 The previous difficulty essentially comes from *quantum entanglement*. Unlike in the classical case, we cannot write $|\psi'(\tau)\rangle$ as a product state $|\psi_1'(\tau)\rangle\,|\psi_2'(\tau)\rangle$ if $|\psi(\tau)\rangle$ is entangled.

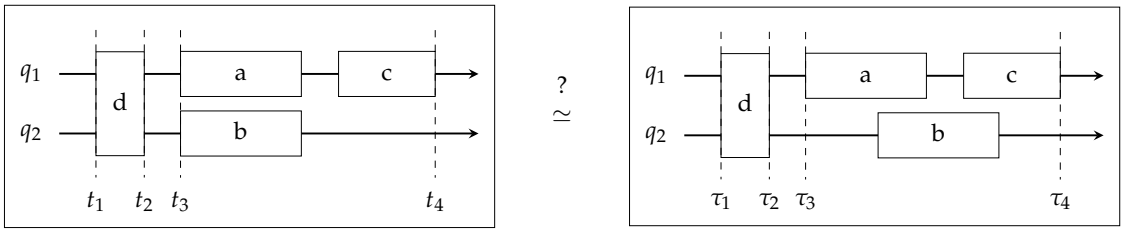
There is a way to circumvent this difficulty. The Hilbert space of two qubits is spanned by the computational basis states $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, and by linearity of unitary operators it suffices to verify: for any $|\psi(t_1)\rangle = |\psi'(\tau_1)\rangle$ in the computational basis states, $|\psi(t_4)\rangle = |\psi'(\tau_6)\rangle$. It is easy to see the state remains a product state at all time. Suppose that $|\psi'(\tau)\rangle = |\psi_1'(\tau)\rangle\,|\psi_2'(\tau)\rangle$, where $|\psi_1'\rangle$ and $|\psi_2'\rangle$ are states of the first and second qubits, respectively. Then, we have

- $|\psi_1'(\tau_6)\rangle = c\,|\psi_1'(\tau_4)\rangle = c\,|\psi_1'(\tau_3)\rangle = ca\,|\psi_1'(\tau_1)\rangle$; and

- $|\psi_2'(\tau_6)\rangle = |\psi_2'(\tau_5)\rangle = b\,|\psi_2'(\tau_2)\rangle = b\,|\psi_2'(\tau_1)\rangle$.

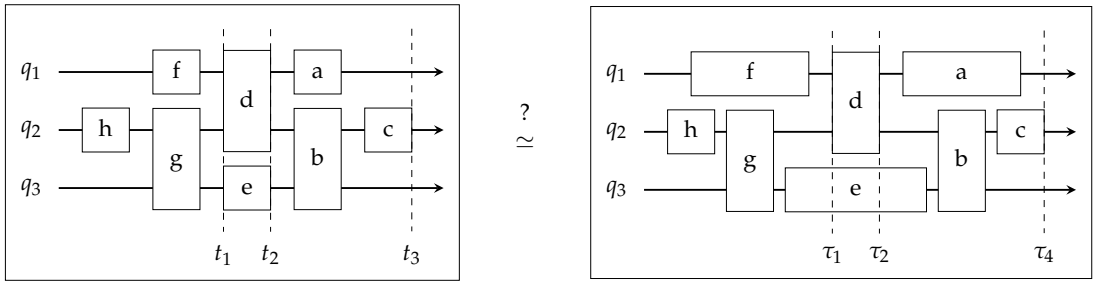
Combined with the previous observation that $|\psi(t_4)\rangle = (ca \otimes b)\,|\psi(t_1)\rangle$, we finish the verification and can conclude $S_1 \simeq S_1'$.

Example 7. Next we consider a harder example, for which the solution to Example 6 no longer works. In Figure 9.2, we present a synchronous system S_2 and an asynchronous S_2' and want to prove their equivalence. Compared to S_1 and S_1' in the previous Example 6, S_2 and S_2' contain an additional action d on both q_1, q_2 at the beginning. Specifically, let d perform a unitary gate for the EPR state preparation; i.e., $d\,|xy\rangle = |\beta_{xy}\rangle = \frac{1}{\sqrt{2}}(|0\rangle\,|y\rangle + (-1)^x\,|1\rangle\,|1-y\rangle)$ for $x, y \in \{0,1\}$.



Synchronous S_2 Asynchronous S_2'

FIGURE 9.2: Proving the synchronous system S_2 is equivalent to the asynchronous system S_2'.



Synchronous S_3 Asynchronous S_3'

FIGURE 9.3: Proving the synchronous system S_3 is equivalent to the asynchronous system S_3'.

Difficulty posed by Example 7 Problem arises if we attempt to prove $S_2 \simeq S_2'$ using the same idea from Example 6 by examining all computational basis input states. Specifically, in system S_2', if we start from a computational basis state $|\psi'(t_1)\rangle = |xy\rangle$, then the intermediate state $|\psi'(\tau_2)\rangle = |\beta_{xy}\rangle$ will be entangled. Because S_2' (when restricted to the time region $[\tau_3, \tau_4]$) contains a similar structure to S_1', we met the same issue as in Example 6: it is hard to relate $|\psi'(\tau_4)\rangle$ and $|\psi'(\tau_3)\rangle$ in S_2'.

Solution to Example 7 There is still a way to circumvent the above new difficulty. The Hilbert space of two qubits is also spanned by the EPR basis states $\{|\beta_{00}\rangle, |\beta_{01}\rangle, |\beta_{10}\rangle, |\beta_{11}\rangle\}$. Therefore, by linearity, we can verify $S_2 \simeq S_2'$ by examining all inputs from EPR basis states. In this case, $|\psi(t_2)\rangle = d\,|\psi(t_1)\rangle$ and $|\psi'(\tau_2)\rangle = d\,|\psi'(\tau_1)\rangle$ become product states by action d, and we reduce to a similar situation to Example 6.

Example 8. Finally, let us examine an even harder example, for which the solution to Example 7 no longer works. In Figure 9.3, we present a synchronous system S_3 and an asynchronous S_3' and want to prove their equivalence. Here, g, d, and b all perform the EPR preparation unitary gate.

Difficulty posed by Example 8 Note that our construction forces the entanglement to appear at some points in S'_3, no matter what the input quantum state is. In particular, one of the states $|\psi'(\tau_1)\rangle$ and $|\psi'(\tau_2)\rangle$ must be entangled, due to the action d. Without loss of generality, suppose $|\psi'(\tau_2)\rangle$ is entangled. In this case, as S'_3 (when restricted to the time region $[\tau_2, \tau_3]$) contains a similar structure to S'_1, the difficulty posed by Example 6 appears again. For Example 8, we are unaware of any simple solution.

Summary Through Examples 6 to 8, we have seen the challenges inherent in the seemingly simple task of proving the equivalence between a synchronous and asynchronous distributed quantum systems, even when these systems are highly restricted. Our aim of proving Theorem 12 is only more complicated — in general, distributed quantum systems can be probabilistic, non-terminating, and with non-local actions.

9.3 Overview of Framework

9.3.1 Identification and Resolution of Challenges

This section identifies challenges from unique quantum properties for proving Theorem 12 and provides an overview of how we resolve them in Chapters 10 and 11.

The Challenge from Entanglement

From Section 9.2, we can identify the first challenge stemming from quantum entanglement. In particular, when all objects (e.g., registers) in a system are classical, the real-time system state can be always decomposed into a direct product of object states. This product decomposition allows us to relate real-time states by discrete actions, as was done in the solution to Example 6.

However, when the objects are quantum, the system state can be entangled and can no longer be decomposed into product states (or more generally, separable states). As a result, one must treat the system state as a whole. This makes characterising the state's evolution difficult, as it becomes dependent on how actions are implemented, (e.g., $|\psi'(\tau_1)\rangle$ depends on the unknown partial action b' of b in Example 6).

The Challenge from the Measurement Problem

The second challenge, not illustrated in Section 9.2, comes from the notorious measurement problem: how the quantum state evolves during a quantum measurement is not fully understood. A key consequence is that it is impossible to model when the probabilistic branching occurs during a measurement, which is a continuous, non-instantaneous procedure [362].

This fundamental uncertainty, combined with the following two factors, makes the rigorous modeling of non-atomic distributed quantum systems particularly difficult:

- **Non-termination**: A distributed system is typically non-terminating. A complete model must be able to define the probability of an event involving infinitely many actions (e.g., the event that "the outcomes from some repeated measurement M are always 1").
- **Complex non-determinism**: The model must harmoniously handle two distinct sources of non-determinism: the probabilistic branching and the concurrency (see also [142]).

Resolving Challenges

To resolve the first challenge from entanglement, we propose a novel model of non-atomic distributed quantum systems. The key insight, drawn from quantum physics, is that the real-time state of a quantum system can be well-defined at any moment if we include all effective quantum degrees of freedom — particularly those implicit and uncontrollable ones introduced by measurement devices [362]. (See Section 10.1.2 and the discussion there.) This insight has not been exploited previously in classical concurrency, where the real-time system state during an action is often undefined [363] (see further discussion in Section 9.4.1).

The second challenge, posed by the measurement problem will be partly resolved by our proper modeling of non-atomic distributed quantum systems, and partly resolved by our techniques in proving Theorem 12.

- **Modeling**: We incorporate both probabilistic branching and concurrency in a consistent model, and we carefully keep all definitions insensitive to the implementation details of actions. Consequently, our proofs are ensured to be independent on how quantum operations are physically performed, thereby circumventing the measurement problem (see also Sections 10.1.3 and 10.1.4 and discussions there).
- **Proof Techniques**: To define and analyse the probabilities of events involving infinitely many actions, we resort to measure-theoretic tools for probability (also adopted in probabilistic concurrency [364, 365]). We establish a formal connection between the real-time state of a distributed quantum system and the probabilities of classically observable events in this system, which provides the foundation for proving equivalence between systems.

9.3.2 A Model of Non-atomic Distributed Quantum Systems

This section provides a brief and informal overview of our model of non-atomic distributed quantum systems, which serves as the basis for proving Theorem 12. The formal details will be presented in Chapter 10. We first introduce the core notions of actions, quantum processes, and distributed quantum systems, without making the atomicity assumption.

Action

An *action* is a set of bounded space-time events for performing a quantum operation. It models a *real-time implementation* of a logical operation. An action a has the following key properties.

1. $T[a]$: the time interval that a spans.
2. $q[a]$: the quantum register that a acts on.
3. $\mathcal{E}[a]$: the quantum operation, either a unitary gate or a partial measurement, that a performs.
4. $e[a]$: the quantum environment (i.e., a set of quantum particles) introduced by a. If $\mathcal{E}[a]$ is a unitary, then $e[a] = \emptyset$, as a only acts on $q[a]$ (effectively). If $\mathcal{E}[a]$ is a (partial) measurement, then $e[a]$ includes the quantum degrees of freedom in the measurement device.

Note that $q[a]$ and $\mathcal{E}[a]$ are discrete, logical properties, while $T[a]$ and $e[a]$ are real-time, physical properties.

Quantum Process

A *quantum process* is a collection of countably many actions with a tree structure. Operationally, as time progresses, a quantum process repeatedly takes actions sequentially. The tree structure comes from the probabilistic branching created by quantum measurements. In a measurement, each classical outcome m is associated with a probabilistic branch, or equivalently, an action performing a partial measurement M_m. A quantum process can be represented by a rooted tree of actions, by connecting every two successive actions with a directed edge $\rightarrow$, with the following additional conditions:

1. (Sequentiality) For any $a \rightarrow b$, the time interval $T[a]$ must entirely precede $T[b]$.
2. (Branching) Actions performing partial measurements from the same measurement must be consistent.

3. (Finitely many actions in finite time) For any time t, only a finitely number of actions can have begun before t.

Distributed Quantum System

A *distributed quantum system* is a collection of quantum processes, recursively defined as follows:

1. Any single quantum process is a distributed quantum system.
2. The parallel composition $A \parallel B$ of two distributed quantum systems A and B with effectively separated quantum environments is also a distributed quantum system.

Here, the quantum environment of a system is the union of the environment of all its actions. The condition of effectively separated quantum environments is because the logical specification of a distributed quantum system should be oblivious to the physical implementation of measurements. In this case, processes in a system can only explicitly communicate via quantum registers, but not implicitly via quantum environments.

Local Actions and Atomic Actions

Given a distributed quantum system, we can introduce the notions of local actions and atomic actions.

An action a in a distributed quantum system S is local, if a is space-time-separated from actions in other processes in S. That is, for any action b in any other process in S, either their quantum registers do not overlap ($q[a] \cap q[b] = \emptyset$) or their time intervals do not overlap ($T[a] \cap T[b] = \emptyset$).

A subset D of actions in a distributed quantum system is said to be atomic, if for any two actions a and b in D, either a precedes b in time, or b precedes a. That is, either $T[a] < T[b]$ or $T[b] < T[a]$.

Real-Time Semantics

The *real-time semantics* of a distributed quantum system characterises the real-time evolution of the system state. Since distributed quantum systems are probabilistic, we introduce the notions of partial processes and partial systems.

A quantum process B is called a *partial process* of another quantum process A, if it consists of a rooted path to some action b and the sub-tree rooted at b in A. Intuitively, B is obtained from A by fixing the first several actions up to b, among other probabilistic branches. Similarly, one can define a distributed quantum system to be a *partial system* of another, if each quantum process in the former is a partial process of a corresponding process in the latter.

The real-time semantics of a distributed quantum system S is determined by the real-time evolution of the system state restricted to every partial system of S. Equivalently, we define a function $[\![\cdot]\!]_S(\cdot)$, where for any partial system C and time t, $[\![C]\!]_S(t)$ is the quantum operation that maps the initial partial system state to the state at time t. Note that the challenge from the measurement problem prevents us from explicitly describing $[\![\cdot]\!]_S(t)$ for all t using the logical specification of S. Nevertheless, we identify the following conditions that $[\![\cdot]\!]_S(\cdot)$ must satisfy. Formal details can be found in Chapter 10.

1. (Initial condition) At time 0, the system is in the initial state.
2. (Branching) After a probabilistic branching occurs, the state of a partial system containing all branches is the sum of states of partial systems each containing exactly one branch.
3. (Evolution) Let I be a time interval. If a partial system does not branch before I, then its state evolution within I is uniquely determined by all actions a with $T[a]$ overlapping I.

   Moreover, if the partial system is a parallel composition of two systems with no interaction in I, then the two systems evolve separately. As a special case, if a process in the first system takes a local action a with $T[a] = I$, then the evolution of the first system within I is captured by $\mathcal{E}[a]$.
4. (Trace) The density-operator trace of a partial system state is non-increasing, and becomes a constant if the system is trace-preserving.

To circumvent the challenge from the measurement problem, the statements of the above conditions are kept insensitive to how actions are physically implemented. For example, the condition (Branching) only describes the real-time evolution after a probabilistic branching (from a quantum measurement), but not during a branching. The condition (Evolution) includes a formalisation of the Dijkstra-Lamport condition; i.e., any local action is implemented correctly.

Observable Semantics

The real-time semantics of a distributed quantum system characterises the physical implementation. Based on it, we introduce the observable semantics describing what can be classically observed from the system. Later, observable semantics will be used to formally define the equivalence between systems (see our major aim Theorem 12).

Since the observable events in a distributed quantum system possibly involve infinitely many actions, we use measure-theoretic tools (see Section 10.3) to define their probabilities. As time progresses, a quantum process A only goes into one branch probabilistically, associated with a *maximal path* in the tree structure of A. We denote the set

of all maximal paths in A by $\omega(A)$. By taking a direct product, this notion can be generalised to $\omega(S)$ for a distributed quantum system S.

The set $\{\omega(C) : C \text{ is a partial system of } S\}$ generates (in the sense of σ-algebra; see Section 10.3) the set of observable events. Given an initial quantum state ρ and the real-time semantics of S, we can define a consistent probability measure $\mu_{\rho\to S}$ for observable events. The observable semantics of S is then determined by the function $\mu_{.\to S}$.

System Equivalence

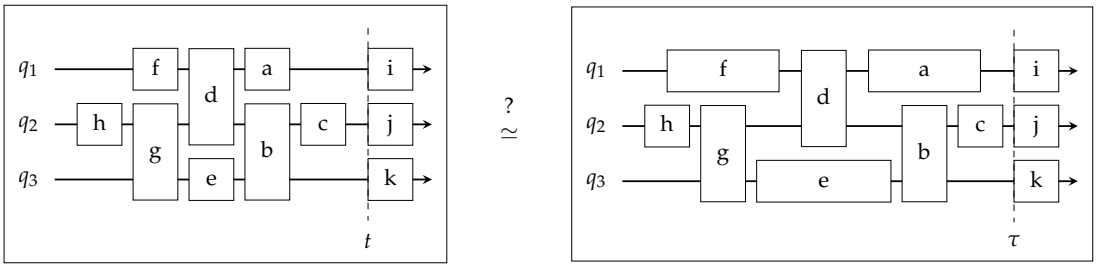
Now we extend the system equivalence informally introduced in Section 9.2. Two distributed quantum systems S_1 and S_2 are said to be equivalent, denoted by $S_1 \simeq S_2$, if (a) they have the same logical specification; i.e., they look the same when physical properties $T[a]$ and $e[a]$ are ignored; and (b) they have the same observable semantics; i.e., they are indistinguishable to any classical observer. This definition of equivalence is used in Theorem 12.

9.3.3 Atomicity of Local Actions

Based upon the established model of non-atomic distributed quantum systems, we are able to prove Theorem 12, thereby providing a rigorous guarantee for the atomicity of local actions in distributed quantum computing. Note that the definition of equivalence in Theorem 12 is based on observable semantics containing all information that a classical observer can learn from the system.

Application to motivating examples

For illustration, we apply Theorem 12 to address the unsolved Example 8 in Section 9.2. As shown in Figure 9.4, we append additional actions i, j, k performing single-qubit measurements to the systems from Example 8, resulting in systems S_4 and S_4'. Since S_4 and S_4' contain only local actions, using Theorem 12, they must have identical observable semantics as their atomic versions are equivalent (which is easy to verify). As the measurements i, j, k can be arbitrarily chosen, it follows from the basic properties of quantum operations that the output states of S_3 and S_3' must also be identical (i.e., $|\psi(t)\rangle = |\psi'(\tau)\rangle$), as desired.



Synchronous S_4 Asynchronous S_4'

FIGURE 9.4: Appending systems in Figure 9.3 with extra single-qubit quantum measurements.

9.4 Discussion

9.4.1 Related Work

Real-Time Semantics v.s. Partial Order Semantics

In the literature of concurrency, the equivalence between distributed systems (i.e., histories in [366] or system executions in [177]) is typically defined using the partial order semantics [10, 11, 177–179, 367], where only the (partial) temporal orders between actions matter. However, a widely adopted but implicit assumption in this approach is: arbitrarily ordering concurrent local actions does not change the partial order semantics (e.g., see [177, Section 3.1], [366, Section 2.1] and [367, Section 4.2.3]). While natural, this assumption lacks a formal guarantee. We are unaware of any prior research formally investigating how the partial order semantics reflects the more fundamental real-time semantics.

Even in the early days, Dijkstra noticed similar subtleties in defining the semantics of a distributed system. He realised that the notion of state is only meaningful at the starting and finishing times of actions [363]. During an action, the state is not well-defined. For example, during the execution of a `write` operation, the real-time value of the register being written cannot be determined. This observation is reminiscent of the challenge from the measurement problem we identified in Section 9.3.1: it is impossible to fully characterise the real-time state evolution in the model.

It turns out that in the classical setting, Dijkstra's concerns can be circumvented, and therefore the implicit assumption for partial order semantics can be guaranteed from the more fundamental real-time semantics. The key insight is: the real-time state of a classical system at time t can always be represented as a direct product of the states of objects (e.g., registers) in this system at time t. For illustration, consider the classical distributed system S_5 composed of two classical registers and two actions in Figure 9.5. The real-time state of S_5 at time t can be expressed as $\rho(t) = (\rho_1(t), \rho_2(t))$, where $\rho_1(t)$

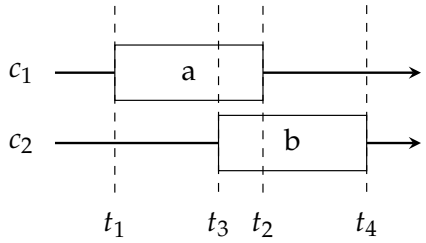


FIGURE 9.5: Classical distributed system S_5.

and $\rho_2(t)$ are states of c_1 and c_2 at time t, respectively. The merit of the product decomposition allows one to define discrete-time states of the system by abstracting real-time states. For S_5, the discrete-time states of the system have the form $\sigma = (\sigma_1, \sigma_2)$, where $\sigma_1 \in \{\rho_1(t_1), \rho_1(t_2), \rho_1(t_4)\}$ and $\sigma_2 \in (\rho_2(t_1), \rho_2(t_3), \rho_2(t_4))$. The discrete-time states are consistent with Dijkstra's observation, e.g., $\rho_1(t)$ for $t \notin \{t_1, t_2\}$ are not determined and therefore not used for defining discrete-time states. To guarantee the implicit assumption for the partial order semantics, we only need to verify that the initial discrete-time state $(\rho_1(t_1), \rho_2(t_1))$ is taken to the same $(\rho_1(t_4), \rho_2(t_4))$ by either

- implementing a followed by b: $(\rho_1(t_1), \rho_2(t_1)) = (\rho_1(t_1), \rho_2(t_3)) \xrightarrow{a} (\rho_1(t_2), \rho_2(t_3)) \xrightarrow{b} (\rho_1(t_2), \rho_2(t_4)) = (\rho_1(t_4), \rho_2(t_4))$; or
- implementing b followed by a: $(\rho_1(t_1), \rho_2(t_1)) = (\rho_1(t_1), \rho_2(t_3)) \xrightarrow{b} (\rho_1(t_1), \rho_2(t_4)) \xrightarrow{a} (\rho_1(t_2), \rho_2(t_4)) = (\rho_1(t_4), \rho_2(t_4))$.

The above verification only relies on the following facts: $\rho_1(t_1)$ is taken to $\rho_1(t_2)$ by action a; $\rho_2(t_3)$ is taken to $\rho_2(t_4)$ by action b; $\rho_1(t_2) = \rho_1(t_4)$; and $\rho_2(t_3) = \rho_2(t_4)$.

In contrast, in the quantum setting, as we identified in Sections 9.2 and 9.3.1, the real-time state of a system cannot be decomposed into a product form, because it can be entangled. The previous reasoning for classical systems no longer works. Therefore, we choose to guarantee the atomicity of local actions (see Theorem 12) directly based on the fundamental real-time semantics (see also Chapter 10).

Classical Guarantees for the Atomicity Assumption

Recall that actions can be classified into local and non-local actions. In the literature of classical concurrency, the atomicity of local actions is implicitly assumed when the partial order semantics is adopted (see the above discussion). Thus, most prior research focuses on providing guarantees for the atomicity of non-local actions, which typically perform operations (e.g., `read`/`write`) on shared registers.

Hardware guarantees: A direct guarantee is provided by modern hardware in the form of atomic instructions, whose implementations can be thought of as hardware-level

atomic actions. For example, the atomic compare-and-swap instruction is provided in the x86 architecture. These lower-level atomic actions can be used to construct higher-level ones. A well-known example is the implementation of the synchronising primitive semaphore using atomic instructions (e.g., test-and-set) [4, 9].

Software guarantees: Another line of research, pioneered by Lamport, focuses on purely software-based guarantees for the atomicity assumption. Lamport introduced the notions of safe and atomic registers [178, 179]. Safe registers are weak and can be implemented without assuming lower-level atomic actions. In contrast, atomic registers are strong, and actions on them are guaranteed to be atomic. It was shown that atomic registers can be constructed from safe registers. This result is closely related to Lamport's foundational work on non-atomic solutions to the mutual exclusion problem [8, 10, 11, 368].

The Role of Entanglement

Entanglement is a manifestation of quantum non-locality at the state level. It is shown in [369] that quantum non-locality can also appear at the operation level: there exist separable quantum operations that cannot be implemented by local actions. Fortunately, this operation-level quantum non-locality does not pose a challenge in our situation, as we are concerned only with the atomicity of local actions and make no assumptions about non-local ones (that may exhibit operation level non-locality).

Importance of Rigorous Basis

Finally, we stress the importance of establishing a rigorous basis for the atomicity assumption in distributed quantum computing. Reasoning about distributed systems is notoriously error-prone, largely due to the immense non-determinism from concurrency. For example, consider the history of the mutual exclusion problem. Dijkstra described his solution to it as "by far the most difficult pieces of program I ever made" [9]. Lamport recalled that in the early years of the field of concurrency, some published concurrent algorithms later turned out to be incorrect [370]. Even for Lamport's celebrated bakery algorithm, its original correctness proof [8] contained a fallacious assumption which was discovered some years later in [10, 11]; and it was only after [371] that two more unnoticed assumptions (in different previous versions of proofs [8, 368, 372]) were found. In [371], it was pointed out "the danger in trying to replace one program with an equivalent one, if the equivalence has not been proved formally". Note that the atomicity assumption is precisely such an assumption of equivalence, reinforcing the need for the formal guarantee provided in this work.

9.4.2 Summary

In this chapter, we examined the atomicity assumption in distributed quantum computing. Our major aim was to rigorously guarantee the atomicity of local actions, which is widely assumed in nearly all models of distributed quantum systems. Through a series of motivating examples, we demonstrated that even the simpler task of proving the equivalence of synchronous and asynchronous distributed quantum systems is highly non-trivial. We identified two fundamental challenges for achieving our aim: one from quantum entanglement and the other from the measurement problem. Finally, we provided an overview of our solution: a novel model of non-atomic distributed quantum systems, which will be formally presented in the next chapter.

Chapter 10

A Model of Distributed Quantum Systems

This chapter presents a formal model of non-atomic distributed quantum systems, which serves as the foundation for guaranteeing the atomicity of local actions in the next chapter. We begin by defining the core notions of actions, processes, and distributed quantum systems. We then define two types of semantics for these systems. The real-time semantics characterises the real-time evolution of the system state, while the observable semantics characterises the probabilities of all classical observable events in the system. Finally, we define the equivalence between distributed quantum systems based on their observable semantics.

10.1 Actions, Processes, and Systems

10.1.1 Quantum Objects

In distributed quantum computing, there are two types of quantum objects.

- *Qubits* are controllable quantum objects. We use **Qbit** to denote the countable set of qubits. A qubit $x \in \mathbf{Qbit}$ has a Hilbert space $\mathcal{H}_x = \mathbb{C}^2$. A quantum register $q \subset \mathbf{Qbit}$ is a set of qubits, which has a Hilbert space $\mathcal{H}_q = \bigotimes_{x \in q} \mathcal{H}_x$.
- *Quantum particles* are uncontrollable quantum objects, introduced by quantum measurements. When a measurement is performed on a quantum register q, qubits in q will evolve together with quantum particles in the measurement device [362]. We use **Qpar** to denote the set of quantum particles. A quantum particle $x \in \mathbf{Qpar}$ has a Hilbert space $\mathcal{H}_x$ of either discrete or continuous dimension depending on x. A quantum environment $e \subset \mathbf{Qpar}$ is a set of quantum particles, which has a Hilbert space $\mathcal{H}_e = \bigotimes_{x \in e} \mathcal{H}_x$.

10.1.2 Actions

In quantum computing, an action is a set of bounded space-time events for performing a quantum operation. It models a physical implementation (e.g., an evolution of physical qubits) of a logical operation (e.g., a logical *CNOT* gate). We are concerned about the following properties of an action a.

1. $T[a] = [x, y]$ for some $x \leq y \in \mathbb{R}_{\geq 0}$: the time interval that a spans.
2. $q[a] \subset \mathbf{Qbit}$: the quantum register that a acts on.
3. $\mathcal{E}[a] \in \mathcal{QO}\left(\mathcal{H}_{q[a]}\right)$: the logical quantum operation that a performs, which is either a unitary gate or a (partial) measurement.
4. $e[a] \subset \mathbf{Qpar}$: the quantum environment introduced by a. If $\mathcal{E}[a]$ is a unitary, then $e[a] = \emptyset$, as a only acts on $q[a]$ effectively. If $\mathcal{E}[a]$ is a (partial) measurement, then $e[a]$ includes the (possibly infinite-dimensional) quantum degrees of freedom in the measurement device.

In the above, $q[a]$ and $\mathcal{E}[a]$ are discrete and logical properties, depending only on the logical specification of a; while $T[a]$ and $e[a]$ are real-time and physical properties, depending on how a is actually implemented. For simplicity, here, we only consider actions that perform quantum operations but not classical operations, as classical computation can be simulated by quantum computation with little overhead.

It is worth noting again why quantum environments are of concern here. To resolve the challenge from entanglement (see Section 9.3.1), we need to define the real-time state of a distributed quantum system at any time. Consequently, our model must take into account all effective quantum degrees of freedom, including those in the uncontrollable quantum environments introduced by measurement devices.

We adopt the following terminologies and notations:

- Let $\mathcal{H}_a := \mathcal{H}_{q[a]} \otimes \mathcal{H}_{e[a]}$ be the Hilbert space that a acts on.
- We use **Act** to denote the set of actions.
- For a countable set $A \subset \mathbf{Act}$ and a time region $X \subset \mathbb{R}_{\geq 0}$, let

$$A \upharpoonright_X = \{a \in A : T[a] \cap X \neq \emptyset\}.$$

- For a countable set $A \subseteq \mathbf{Act}$, let $q[A] = \bigcup_{a \in A} q[a]$. We use the same convention for $e[\cdot]$ and let $\mathcal{H}_A := \mathcal{H}_{q[A]} \otimes \mathcal{H}_{e[A]}$.

10.1.3 Quantum Processes

A quantum process is a collection of countably many actions with a tree structure. From an operational view, a quantum process repeatedly takes actions as time progresses. Two successive actions taken by the quantum process are connected by a relation $\to$. A non-terminating process involves countably many actions. The tree structure comes from the probabilistic branching created by quantum measurements: a quantum measurement produces finitely many probabilistic branches, where each branch corresponds to a classical outcome m, or equivalently, an action performing a partial measurement M_m. More formally, we define:

Definition 18 (Quantum process)**.** A quantum process is a tuple $(A, \to)$, where $A \subseteq \mathbf{Act}$ is a countable set of actions, and $\to$ is a relation on A satisfying the following conditions:

(a) (Rooted tree) $(A, \to)$ is a rooted tree with vertices in A and edges in $\to$.

(b) (Sequentiality) $\forall a, b \in A, a \to b \Rightarrow T[a] < T[b]$.[1]

(c) (Branching) $\forall a, b, c \in A, a \to b \wedge a \to c \Rightarrow q[b] = q[c]$. Moreover, $\forall a \in A, \sum_{b:a\to b} \mathcal{E}[b]$ is a quantum operation.

(d) (Finitely many actions in finite time) $\forall t \in \mathbb{R}_{\geq 0}, A \restriction_{[0,t]}$ is a finite set.

The above definition of quantum process is conceptually the simplest one can think of: it does not presume any computational model.

Now we explain the conditions in Definition 18 as follows: Definition 18 (a) says that the next possible action taken by a quantum process depends on all previous actions. Definition 18 (b) says that a quantum process is sequential. Definition 18 (c) says that all possible next actions taken by a quantum process are consistently from the same quantum operation; in particular, the partial measurement actions in a branching of the tree should correspond to different classical outcomes from the same quantum measurement. Definition 18 (d) says that the number of actions that begin before any time t is finite, which is similar to Axiom A5 in [178].

We adopt the following terminologies and notations:

- When the context is clear, we simply use A to denote $(A, \to)$.
- We use **Proc** to denote the set of quantum processes.
- We say A is *trace-preserving*, if the quantum operation $\sum_{b:a\to b} \mathcal{E}[b]$ in Definition 18 (c) is trace-preserving.

[1] Here, we denote $X < Y$ if $\forall t_x \in X, t_y \in Y, t_x < t_y$ for $X, Y \subseteq \mathbb{R}_{\geq 0}$.

- We say A is *aligned*, if in addition to Definition 18 (c), we have $a \to b \wedge a \to c \Rightarrow \min T[b] = \min T[c] \wedge e[b] = e[c]$.
- We use $\mathrm{root}(A)$ to denote the root of the tree A.
- Let $\to^k$ be the k^{th} composition of $\to$. Let $\to^*$ and $\to^+$ be the Kleene star and Kleene plus of $\to$; i.e., $a \to^* b$ if $\exists k \geq 0, a \to^k b$; and $a \to^+ b$ if $\exists k > 0, a \to^k b$.

We further define the concept of partial quantum process, which can be thought of as a restriction of another process by fixing the first several actions up to some action b.

Definition 19 (Partial quantum process)**.** A process $(B, \to) \in \mathbf{Proc}$ is said to be a partial (quantum) process of another $(A, \to) \in \mathbf{Proc}$, if $\exists b \in B$ such that B consists of a rooted path to b and the sub-tree rooted at b; i.e., $B = \{a \in A : \mathrm{root}(A) \to^* a \to^* b\} \cup \{a \in A : b \to^+ a\}$. In this case, we denote $B = A/b$.

Intuitively, the rooted path to b selects a prefix of probabilistic branches. Note that it is possible that $B = A/a = A/b$ for $a \neq b$, when there is no branching between a and b. We adopt the following terminologies and notations:

- For convenience, we additionally define A to be a partial process of itself. We use $A/*$ to denote the set of all partial processes of A.
- For a quantum process $A \in \mathbf{Proc}$ and a subset $B \subseteq A$, we say that B *has no branching* if $(B, \to^*)$ is a totally ordered set. It is easy to see $\forall a \in A$, $(A/a) \restriction_{[0, \max T[a]]}$ has no branching. It is also obvious that any $C \subseteq B$ has no branching if B has no branching.
- For a trace-preserving quantum process $A \in \mathbf{Proc}$, we say that a partial process $B \in A/*$ is *trace-preserving after time* t, if $B = A/a$ for some a such that: either (a) $t \geq \max T[a]$; or (b) for any $b \in B \restriction_{[t, \max T[a]]}$, $\mathcal{E}[b]$ is trace-preserving. Intuitively, after time t, B includes all future branches. Under this definition, A itself is trace-preserving after time 0.

10.1.4 Non-Atomic Distributed Quantum Systems

A (non-atomic) distributed quantum system is a collection of finitely many quantum processes with non-overlapping quantum environments.

Definition 20 (Distributed quantum system)**.** A (non-atomic) distributed quantum system is recursively defined by the following rules:

1. Any single quantum process A is a distributed quantum system.

2. For two distributed quantum systems A and B with $e[A] \cap e[B] = \emptyset$, their parallel composition C, denoted by $C = A \parallel B$, is also a distributed quantum system. We identify $A \parallel B$ and $B \parallel A$ as the same system, and recursively define $e[C] = e[A] \cup e[B]$.

In the above, the condition $e[A] \cap e[B] = \emptyset$ says the quantum environments of the two systems A and B are effectively separated, and they can only explicitly communicate through the shared quantum register $q[A] \cap q[B]$, but not implicitly through quantum environments $e[A]$ and $e[B]$.[2] We set this condition because the logical specification of a distributed quantum system should be oblivious to the physical implementation of measurements. Similar condition also appears in [371], where different processes have disjoint sets of private variables.

We adopt the following notations and naturally extend some definitions for quantum processes to distributed quantum systems:

- We use **Sys** to denote the set of distributed quantum systems. For convenience, we include $\emptyset \in$ **Sys**, and define $C \parallel \emptyset := C$ for any $C \in$ **Sys**.
- For an action $a \in$ **Act** and a system $C = A \parallel B$, we denote $a \in C$ if $a \in A$ or $a \in B$.
- We say $C = A \parallel B$ is trace-preserving (resp. aligned), if A and B are both trace-preserving (resp. aligned).
- We say C is a partial system of another C', denoted by $C \in C'/*$, if $C = A \parallel B$, $C' = A' \parallel B'$ and $A \in A'/*$ and $B \in B'/*$.
- For $D \subseteq C$, we say D has no branching, if for any $A \in$ **Proc**, $B \in$ **Sys** with $C = A \parallel B$, $A \cap D$ has no branching.
- We say $C = A \parallel B$ is trace-preserving after time t, if A and B are both trace-preserving after time t.

For illustration, in Figure 10.1, we present an example of a distributed quantum system of two processes. Each process has a tree structure. Every segment corresponds to an action, whose projection onto the time axis is the time interval occupied by this action. Every dotted segment represents that there is no action during the time interval. Every branching corresponds to a set of measurement actions. Two partial processes A/a and B/b are highlighted in bold. For illustration, $(A/a) \restriction_{[0,t_2]}$ has no branching, and B/b is trace-preserving after time t_1.

[2] In distributed systems, interprocess communications typically include message passing. As is observed in [10, 178], message passing actually can be modeled by a shared memory. From a physical perspective, message passing is also performing quantum operations on the transmission media. For simplicity, we do not consider message passing.

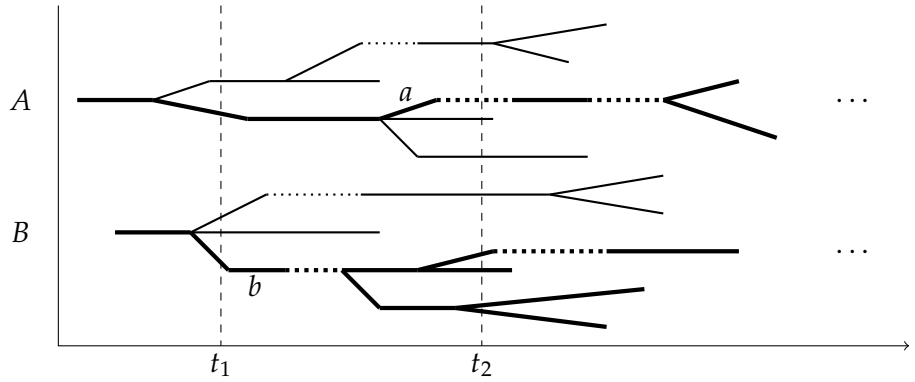


FIGURE 10.1: An example of distributed quantum system $A \parallel B$ of two processes A and B.

10.1.5 Local Actions

Recall that our major aim is to prove Theorem 12, thereby providing a rigorous guarantee for the atomicity of local actions. Now we can define the notions of local actions in a distributed quantum system.

Definition 21 (Local actions)**.** Given a distributed quantum system $C = A \parallel B$ for some process $A \in \mathbf{Proc}$, an action $a \in A$ is said to be local, if $\forall b \in B$, $q[a] \cap q[b] = \emptyset$ or $T[a] \cap T[b] = \emptyset$. Intuitively, a is space-time-separated from actions in other processes.

Dijkstra-Lamport Condition

Next let us revisit an important condition about the correctness of local actions. It is informally stated as: *any local action is implemented correctly*. Intuitively, if an action a is local, then its implementation will not be affected by any other actions, and therefore a correctly transforms the real-time state at time x to time y according to the logical operation $\mathcal{E}[a]$ it performs, given $T[a] = [x, y]$.

This condition seems to be first mentioned by Dijkstra implicitly in [5, 363]. In particular, in [5] he pointed out "the nature of a single sequential process, performing its sequence of actions autonomously, i.e., independent of its surroundings"; and in [363] he observed "the effect of actions is only defined by and describable in a projection of a subspace...each process is related to its own subspace." Later, Lamport first explicitly used this condition to analyse non-atomic `read`/`write`: a `read` that does not overlap any `write` must obtain the correct value [10, 368]. Hence, we call this condition Dijkstra-Lamport condition, which will be incorporated in our formal definition of the real-time semantics of distributed quantum systems later (see Definition 22).

10.2 Real-Time Semantics

In this section, we define the real-time semantics of a trace-preserving distributed quantum system S, characterising the real-time state evolution of all partial systems of S. In particular, it is determined by a function $[\![\cdot]\!]_S(\cdot)$, where for any partial system $C \in S/*$ and time t, $[\![C]\!]_S(t)$ is the quantum operation that maps the initial partial system state of C at time 0 to the state at time t.

Definition 22 (Real-time semantics)**.** For a trace-preserving distributed quantum system $S \in \mathbf{Sys}$, its real-time semantics is a function $[\![\cdot]\!]_S(\cdot)$, where for any partial system $C \in S/*$ and time t, $[\![C]\!]_S(t) \in \mathcal{QO}(\mathcal{H}_C)$ is a quantum operation satisfying the following conditions. Here, we omit the subscript S for simplicity.

(a) (Initialisation) $[\![C]\!](0) = \mathbb{1}$.

(b) (Branching) If $C = A/a \parallel B$ for some process $A \in \mathbf{Proc}$, system $B \in \mathbf{Sys}$, and action $a \in A$ with $\max_{b:a\to b} \max T[b] \leq t$,[3] then

$$[\![C]\!](t) = \sum_{b:a\to b} [\![A/b \parallel B]\!](t). \tag{10.1}$$

(c) (Evolution) For any time interval $I = [x, y]$, if $C \restriction_{[0,x)}$ has no branching, then

$$[\![C]\!](y) = \mathcal{F} \circ [\![C]\!](x-)^{4} \tag{10.2}$$

for some quantum operation $\mathcal{F} \in \mathcal{QO}(\mathcal{H}_C)$ uniquely determined by $C \restriction_I$.

Moreover, suppose $C = A \parallel B$ for some $A, B \in \mathbf{Sys}$. If $q[A \restriction_I] \cap q[B \restriction_I] = \emptyset$, then $\mathcal{F} = \mathcal{F}_A \otimes \mathcal{F}_B$ for some quantum operations $\mathcal{F}_A$ and $\mathcal{F}_B$ uniquely determined by $A \restriction_I$ and $B \restriction_I$, respectively.

(Dijkstra-Lamport) In particular, $\mathcal{F}_A = \mathcal{E}[a]$ if $A \restriction_I = \{a\}$ for some local a and $I = T[a]$; and $\mathcal{F}_A = \mathbb{1}$ if $A \restriction_I = \emptyset$.

(d) (Trace) For any initial quantum state ρ, $\mathrm{tr}([\![C]\!](t)(\rho))$ is non-increasing with respect to t. Moreover, if C is trace-preserving after time t_0, then for any $t > t_0$, $\mathrm{tr}([\![C]\!](t_0)(\rho)) = \mathrm{tr}([\![C]\!](t)(\rho))$.

Let us explain the conditions in Definition 22 as follows:

- Definition 22 (a) says that at time 0 the system does nothing.

[3] We use the convention that on $\mathbb{R}_{\geq 0}$, $\min \emptyset := +\infty$ and $\max \emptyset := 0$.

[4] For any function $f(x)$, we use $f(x-) := \lim_{x'\to x-} f(x')$ to denote the left limit of f at x.

- Definition 22 (b) says that at any time after a branching in a process, the partial system state of C (containing all branches) is the sum of states of partial systems each containing exactly one branch.

- Definition 22 (c) is the most important condition. The first part means the real-time evolution of the partial system state of C in a time interval $I = [x, y]$ is uniquely determined by all actions in C that overlap I, provided that C does not branch before x. The second part further implies that if C is a parallel composition of two systems A and B with no interaction in I, then the two systems evolve separately. The third part is the Dijkstra-Lamport condition (see Section 10.1.5): if a process in the first system A takes a local action a with time interval I, then the action correctly performs the logical operation $\mathcal{E}[a]$. It is also natural that if a process has no action in I, it does nothing.

- Definition 22 (d) says the trace of the state evolved according to C is non-increasing, and becomes constant after time t_0, if C is trace-preserving after time t_0.

The conditions in Definition 22 are the simplest we can think of. There are many possible functions $[\![\cdot]\!]_S(\cdot)$ satisfying Definition 22, as Definition 22 does not determine the full real-time state evolution (which is similar to our observation in Example 6). In the following, when we talk about the real-time semantics of S, we arbitrarily pick one $[\![\cdot]\!]_S(\cdot)$ that satisfies Definition 22.

Now we explain how the definition of real-time semantics in Definition 22 circumvents the challenge from the measurement problem identified in Section 9.3.1. The conditions in Definition 22 are carefully kept insensitive to how actions are implemented. For example, Definition 22 (b) only constrains those states at time t after all partial measurement actions (in a branching) finish. It also does not depend on how measurements are implemented.

Finally, we make the following remarks:

- Note that in Definition 22, the quantum operation $[\![C]\!](t)$ acts on the Hilbert space $\mathcal{H}_C$. This means the real-time state contains all quantum degrees of freedom, including those explicit in quantum registers and those implicit in quantum environments.

- Given a function $[\![\cdot]\!]_S(\cdot)$ satisfying Definition 22, it also satisfies Definition 22 (c) when the time interval is replaced by $I = [x, y)$. To see this, we can take a limit of y in the original Definition 22 (c), in which case Equation (10.2) still holds except that y is replaced by $y-$. Similar statements hold for the cases $I = (x, y]$ and $I = (x, y)$ by replacing $C \restriction_{[0,x)}$ with $C \restriction_{[0,x]}$ and $x-$ with x in Definition 22 (c).

- Since any partial quantum system C is trace-preserving after some sufficiently large time t_C, according to Definition 22 (d), we have $\mathrm{tr}([\![C]\!](t_C)(\rho)) = \lim_{t\to\infty} \mathrm{tr}([\![C]\!](t)(\rho))$.

10.3 Background on Measure-Theoretic Probability

Before proceeding to the observable semantics of distributed quantum systems, let us briefly introduce the measure-theoretic probability, which will be used to define the probabilities of classical observable events in a system. A more detailed introduction can be found in the textbook [373].

The measure-theoretic probability is needed because the state space of classical observable events is as large as $\mathbb{R}$. In particular, for a non-terminating distributed quantum system, an observable event can involve countably many classical outcomes from quantum measurements; e.g., the event can be "the outcomes from repeated measurements from process A are always 1". More generally, the probabilities of any safety (i.e., something bad never happens) or liveness (i.e., something good eventually happens) properties are of great concern in the design and analysis of distributed systems.

Let us model the space of elementary events, the set of observable events, and finally a probability measure. Let Ω be a set of elementary events. Consider a class of subsets $\mathcal{A} \subset 2^\Omega$. We first define the notions of a semiring and a σ-algebra.

Definition 23 (Semiring)**.** A class of sets $\mathcal{A} \subset 2^\Omega$ is a semiring if $\mathcal{A}$ satisfies the following properties:

1. $\emptyset \in \mathcal{A}$.
2. For any $X, Y \in \mathcal{A}$, the difference $X - Y$ is a finite disjoint union of sets in $\mathcal{A}$.
3. For any $X, Y \in \mathcal{A}$, $X \cap Y \in \mathcal{A}$.

Definition 24 (σ-algebra)**.** A class of sets $\mathcal{A} \subset 2^\Omega$ is a σ-algebra if $\mathcal{A}$ satisfies the following properties:

1. $\Omega \in \mathcal{A}$.
2. For any $X \in \mathcal{A}$, $X^\complement \in \mathcal{A}$.
3. For any $\{X_k\}_{k\in\mathbb{N}}$ with $X_k \in \mathcal{A}$, the countable union $\bigcup_{k\in\mathbb{N}} X_k \in \mathcal{A}$.

A σ-algebra satisfies the natural properties of a set of observable events, on which one can define a probability measure consistently. Consider the following properties of set functions.

Definition 25. Let $\mathcal{A} \subset 2^\Omega$ and $\mu : \mathcal{A} \to [0, 1]$ be a set function. We say that

- μ is additive if $\mu(X) = \sum_{k=1}^{K} \mu(X_k)$ for any $X \in \mathcal{A}$ and finitely many $X_1, X_2, \ldots, X_K \in \mathcal{A}$ with $X = \biguplus_{k=1}^{K} X_k$ and $K \in \mathbb{N}$.
- μ is σ-additive if $\mu(X) = \sum_{k\in\mathbb{N}} \mu(X_k)$ for any $X \in \mathcal{A}$ and countably many $X_1, X_2, \ldots \in \mathcal{A}$ with $X = \biguplus_{k\in\mathbb{N}} X_k$.
- μ is σ-subadditive if $\mu(X) \leq \sum_{k\in\mathbb{N}} \mu(X_k)$ for any $X \in \mathcal{A}$ and countably many $X_1, X_2, \ldots \in \mathcal{A}$ with $X \subset \bigcup_{k\in\mathbb{N}} X_k$.

Then, a probability measure is defined as follows.

Definition 26 (Probability measure)**.** Let $\mathcal{A} \subseteq 2^{\Omega}$ be a σ-algebra, and $\mu : \mathcal{A} \to [0,1]$ be a set function. We say μ is a probability measure on $\mathcal{A}$ if μ is σ-additive, $\mu(\emptyset) = 0$ and $\mu(\Omega) = 1$.

An important lemma in probability theory is the measure extension lemma, enabling one to extend a properly defined set function on a semiring $\mathcal{A}$ to the σ-algebra $\sigma(\mathcal{A})$ generated by $\mathcal{A}$, as follows.

Lemma 15 (Carathéodory's measure extension, special case of Theorem 1.53 in [373])**.** *Let $\mathcal{A} \subset 2^{\Omega}$ be a semiring with $\Omega \in \mathcal{A}$, and $\sigma(\mathcal{A})$ be the σ-algebra generated by $\mathcal{A}$. Let $\mu : \mathcal{A} \to [0,1]$ be an additive, σ-subadditive function on $\mathcal{A}$ with $\mu(\emptyset) = 0$ and $\mu(\Omega) = 1$, then μ has a unique extension to a probability measure $\tilde{\mu} : \sigma(\mathcal{A}) \to [0,1]$, and*

$$\tilde{\mu}(X) = \inf\left\{ \sum_{k\in\mathbb{N}} \mu(X_k) : X \subset \bigcup_{k\in\mathbb{N}} X_k \wedge \forall k, X_k \in \mathcal{A} \right\}.$$

10.4 Observable Semantics

Based on the real-time semantics defined in Section 10.2, we can define the observable semantics of a distributed quantum system S, characterising the probabilities of all classically observable events in S. To make the precise definition, we first identify the elementary observable events in a system, each corresponding to a maximal path, defined as follows.

Definition 27 (Maximal path)**.** For a tree $(A, \to)$, a subset $B \subseteq A$ is called a path if $B = \{a_1, a_2, \ldots\}$ and $a_j \to a_{j+1}$ for all j. A path $B \subseteq A$ is maximal if there does not exist another path $C \subseteq A$ such that $B \subsetneq C$. It is easy to see that every maximal path is rooted.

For a quantum process $A \in \mathbf{Proc}$, let $\omega(A)$ denote the set of maximal paths in the tree A. For a distributed quantum system $C = A \parallel B$, let $\omega(C) := \omega(A) \times \omega(B)$.[5]

[5]Here, note that the Cartesian product $\times$ implicitly assumes an order between A and B. In defining $\omega(C)$, we assume a specific order is chosen and used consistently.

For a distributed quantum system $S \in \mathbf{Sys}$, let

$$\mathcal{S} := \{\omega(C) : C \in S/*\} \cup \{\emptyset\} \subseteq \mathcal{P}(\omega(S))$$

be the class of sets of maximal paths in partial systems of S (together with an empty set). Here, $\mathcal{P}(X)$ denotes the power set of X. The following lemma shows that $\mathcal{S}$ is a semiring (see Section 10.3). For readability, the proof is deferred to Section 10.5.1.

Lemma 16. *For a distributed quantum system S, the class $\mathcal{S} \subset \mathcal{P}(\omega(S))$ forms a semiring.*

Let $\sigma(\mathcal{S}) \subset \mathcal{P}(\omega(S))$ be the σ-algebra generated by $\mathcal{S}$. Intuitively, $\sigma(\mathcal{S})$ is the set of classically observable events to which we can assign probabilities. Then, we can define a probability measure on $\sigma(\mathcal{S})$ by the following lemma, based on the real-time semantics of S. For readability, the proof is deferred to Section 10.5.2.

Lemma 17. *For a trace-preserving distributed quantum system $S \in \mathbf{Sys}$ and an initial quantum state $\rho \in \mathcal{D}\left(\mathcal{H}_{q[S]}\right)$ with $\mathrm{tr}(\rho) = 1$, there exists a unique probability measure $\mu_{\rho \to S} : \sigma(\mathcal{S}) \to [0,1]$ such that for any partial system $C \in S/*$,*

$$\mu_{\rho \to S} \circ \omega(C) = \lim_{t \to \infty} \mathrm{tr}(\llbracket C \rrbracket_S(t)(\rho)). \tag{10.3}$$

From Lemma 17, we can define the observable semantics of S.

Definition 28 (Observable semantics)**.** The observable semantics of a distributed quantum system S is a function $\mu_{\cdot \to S}(\cdot)$ satisfying Equation (10.3).

Finally, based on the observable semantics, we can define the equivalence between distributed quantum systems. To this end, we introduce the isomorphism between two systems. Intuitively, two systems are isomorphic if they have the same logical specification, which ignores the physical properties of the systems (i.e., $T[a]$ and $e[a]$ for every action a within the systems).

Definition 29 (Isomorphism)**.** For two trace-preserving distributed quantum systems $S, S' \in \mathbf{Sys}$, a bijection $\gamma : S \to S'$ is an isomorphism if

1. γ preserves $\to$: $\forall a, b \in S, a \to b \Leftrightarrow \gamma(a) \to \gamma(b)$.
2. γ preserves q and $\mathcal{E}$: $\forall a \in S, q[a] = q[\gamma(a)] \wedge \mathcal{E}[a] = \mathcal{E}[\gamma(a)]$.

By a slight abuse of notation, for an isomorphism $\gamma : S \to S'$ and any set X, we recursively define $\gamma(X) := \{\gamma(Y) : Y \in X\}$. Then for $X \in \sigma(\mathcal{S})$, we have $\gamma(X) \in \sigma(\mathcal{S}')$. Now we define the equivalence between systems as follows.

Definition 30 (Equivalent systems)**.** Two trace-preserving distributed quantum systems $S, S' \in \mathbf{Sys}$, are equivalent, denoted by $S \simeq S'$, if there exists an isomorphism $\gamma : S \to S'$ such that: for any state $\rho \in \mathcal{D}\left(\mathcal{H}_{q[S]}\right)$, $\mu_{\rho \to S} = \mu_{\rho \to S'} \circ \gamma$.

Note that $q[S] = q[S']$ because γ is an isomorphism. By our definition of observable semantics in Definition 28, equivalent systems are indistinguishable to any classical observer.

10.5 Proof Details

In this section, we present the proof details of Lemmas 16 and 17. To this end, we begin by analysing the structure of the set of classically observable events in a distributed quantum system.

Technical Lemmas about Partial Processes

We first prove two useful lemmas about partial processes. It is easy to derive them from the tree structure of quantum processes and Definition 19.

Lemma 18. *Given any partial process $A/a \in A/*$ for some quantum process $A \in$ **Proc** and action $a \in A$, if $\{b : a \to b\} \neq \emptyset$, then we have $\omega(A/a) = \biguplus_{b:a\to b} \omega(A/b)$.*

Lemma 19. *Given any two partial processes $A/a, A/b \in A/*$ for some quantum process $A \in$ **Proc** and actions $a, b \in A$, if $\omega(A/a) \cap \omega(A/b) \neq \emptyset$, then $a \to^* b \vee b \to^* a$.*

Decomposition Lemmas about Partial Systems

Now we show how to decompose a partial system into "finer" ones. We first define a relation $\rightsquigarrow$, induced from the relation $\to$ in Definition 18.

Definition 31 (Relation $\rightsquigarrow$)**.** Given a trace-preserving distributed quantum system $S \in$ **Sys**, define a relation $\rightsquigarrow \subseteq \mathcal{P}(S/*) \times \mathcal{P}(S/*)$ on the powerset of $S/*$ such that $X \rightsquigarrow Y$ iff $X - Y = \{A/a \parallel B\}$ for some trace-preserving $A \in$ **Proc**, $a \in A$ and $B \in$ **Sys**; and $Y - X = \{A/b \parallel B : a \to b\}$.

As usual, we use $\rightsquigarrow^*$ and $\rightsquigarrow^+$ to denote the Kleene star and Kleene plus of $\rightsquigarrow$, respectively. The following lemma shows that if a finite set of partial systems is decomposed from another w.r.t. the relation $\rightsquigarrow$, then the unions of their corresponding maximal paths are the same.

Lemma 20. *Given a trace-preserving distributed quantum system $S \in$ **Sys**, and two finite sets $X, Y \subseteq S/*$ of partial systems, if $X \rightsquigarrow^* Y$ and $\forall C, D \in X, \omega(C) \cap \omega(D) = \emptyset$, then $\biguplus_{C\in X} \omega(C) = \biguplus_{C\in Y} \omega(C)$.*[6]

Proof. We prove the lemma by induction on the transitive closure $\rightsquigarrow^*$.

[6]The inverse direction however does not hold. Finding a counter example is left as an exercise.

1. $X = Y$. This case is trivial.

2. $X \rightsquigarrow^* Z$ and $Z \rightsquigarrow Y$ for some finite set $Z \subseteq S/*$. By the induction hypothesis, $\biguplus_{C\in X} \omega(C) = \biguplus_{C\in Z} \omega(C)$. According to Definition 31, $Z - Y = \{A/a \parallel B\}$ for some trace-preserving $A \in \mathbf{Proc}$, $a \in A$ and $B \in \mathbf{Sys}$; and $Y - Z = \{A/b \parallel B : a \to b\}$. By Lemma 18, $\omega(A/a) = \biguplus_{b:a\to b} \omega(A/b)$. Thus, $\biguplus_{C\in Z} \omega(C) = \biguplus_{C\in Y} \omega(C)$. The conclusion immediately follows.

□

Recall that for a quantum process A, any partial process $B \in A/*$ consists of a rooted path to some $a \in A$ and the subtree rooted at a. We define a function $\ell(\cdot)$ such that $\ell(B)$ is the minimal length of such rooted path. It can also be naturally generalised to partial systems.

Definition 32 (Function ℓ)**.** Given a partial process $B \in A/*$ for some trace-preserving quantum process $A \in \mathbf{Proc}$, define $\ell(B) = \min\{r : B = A/a \wedge \mathrm{root}(A) \to^r a \in A\} \in \mathbb{N}$.

Given a partial system $C = A \parallel B$, define $\ell(C) = \max\{\ell(A), \ell(B)\}$.

Our next lemma shows that two partial systems can be decomposed with respect to the relation $\rightsquigarrow$ such that after the decomposition, the intersection of the two sets of partial systems is a unique partial system, whose corresponding maximal paths are exactly the intersection of those of the original two partial systems.

Lemma 21. *Given a trace-preserving distributed quantum system $S \in \mathbf{Sys}$, for any two partial systems $C, D \in S/*$ with $\omega(C) \cap \omega(D) \neq \emptyset$, there exist two corresponding finite sets $X, Y \subseteq S/*$ such that*

1. *$\{C\} \rightsquigarrow^* X$ and $\{D\} \rightsquigarrow^* Y$;*

2. *$X \cap Y = \{E\}$ for some $E \in S/*$ and $\omega(E) = \omega(C) \cap \omega(D)$.*

Proof. We show how to construct X, Y step by step. In the construction, we require the intermediate X, Y to satisfy the following:

$$\{C\} \rightsquigarrow^* X \wedge \{D\} \rightsquigarrow^* Y, \tag{10.4}$$

$$\exists!(E, F) \in X \times Y, \omega(E) \cap \omega(F) \neq \emptyset. \tag{10.5}$$

The construction is as follows.

1. Initially, let $X = \{C\}$ and $Y = \{D\}$.

2. Repeat the following procedure. First pick the unique pair $(E, F) \in X \times Y$ according to Equation (10.5).

If $E = F$, then by Equation (10.4), Equation (10.5) and Lemma 20, it is easy to see that the properties in Lemma 21 are satisfied and we can terminate.

If $E \neq F$, we can write $E = A/a \parallel B$ and $F = A/a' \parallel B'$ for some trace-preserving $A \in \mathbf{Proc}$, $a \neq a' \in A$ and $B, B' \in \mathbf{Sys}$. From Equation (10.5), we have $\omega(A/a) \cap \omega(A/a') \neq \emptyset$. Moreover, by Lemma 19 and $a \neq a'$, either $a \rightarrow^+ a'$ or $a' \rightarrow^+ a$. Without loss of generality, suppose that $a \rightarrow^+ a'$, then let $X' = (X - \{E\}) \cup \{A/b \parallel B : a \rightarrow b\}$ and $Y' = Y$, where X', Y' stand for the new values of X, Y, respectively.

Now we show that Equations (10.4) and (10.5) hold in the above construction. In Step 1, they obviously hold for the initial X, Y. Let us verify Equations (10.4) and (10.5) for X', Y' given that they hold for X, Y, in Step 2:

- Equation (10.4): It simply follows from $X \rightsquigarrow X'$ and $Y \rightsquigarrow^* Y'$.
- Equation (10.5): From above, we have $\omega(E) = \biguplus_{G \in X' - X} \omega(G)$. From Equation (10.5) for X, Y, we also have $\forall E' \in X \cap X', F' \in Y, \omega(E') \cap \omega(F') = \emptyset$ and $\forall F' \neq F \in Y, \omega(E) \cap \omega(F') = \emptyset$. Now it suffices to show that $\exists! G \in X' - X, \omega(G) \cap \omega(F) \neq \emptyset$. Since $a \rightarrow^+ a'$, there exists a unique b' such that $a \rightarrow b' \wedge b' \rightarrow^* a'$. Let $G = A/b' \parallel B$, then $\omega(A/b') \cap \omega(A/a') \neq \emptyset$. From Equation (10.5) for X, Y, $\omega(B) \cap \omega(B') \neq \emptyset$. Hence, $\omega(G) \cap \omega(F) \neq \emptyset$. The uniqueness of G follows from that of b'.

It only remains to show that the above construction terminates. Note that in Step 2, if $X' \neq X$, by construction we have $\forall G \in X' - X$, $\ell(G) \leq \ell(F)$. A similar statement holds for Y. By a simple induction, it can be seen that $\forall G \in (X' - X) \cup (Y' - Y), \ell(G) \leq \max\{\ell(C), \ell(D)\}$. Moreover, consider the set

$$Z = \{G \in S/* : \ell(G) \leq \max\{\ell(C), \ell(D)\}\}.$$

Let us focus on the construction of X. Each time when $X' \neq X$ in Step 2, in $X' - X$ we will visit at least one new element in Z. Since Z is finite, the number of times when $X' \neq X$ should be finite. A similar statement holds for Y. Hence, the above construction terminates. □

Our last lemma in this section shows that if for two finite sets of partial systems, the unions of their corresponding maximal paths are the same, then they can be decomposed into the same set of partial systems w.r.t. the relation $\rightsquigarrow$.

Lemma 22. *Given a trace-preserving distributed quantum system $S \in \mathbf{Sys}$, for any two finite sets $X, Y \subseteq S/*$ of partial systems, if $\biguplus_{C \in X} \omega(C) = \biguplus_{C \in Y} \omega(C)$, then there exists a finite set $Z \subseteq S/*$ of partial systems such that $X \rightsquigarrow^* Z$ and $Y \rightsquigarrow^* Z$.*

Proof. We construct two finite sets W, Z step by step such that finally $W = Z$. In the construction, we require the intermediate W, Z to satisfy:

$$X \rightsquigarrow^* W \wedge Y \rightsquigarrow^* Z. \tag{10.6}$$

The construction is as follows.

1. Initially, let $W = X$ and $Z = Y$.
2. Repeat the following procedure. First pick $C \in W$ and $D \in Z$ such that

$$\omega(C) \cap \omega(D) \neq \emptyset \wedge C \neq D. \tag{10.7}$$

If such C, D do not exist, then by Equation (10.6) and Lemma 20, it is easy to see that $W = Z$ and Z satisfies the properties in Lemma 22, and we can terminate.

Otherwise, by Lemma 21, there exist $X_1, X_2 \subseteq S/*$ such that

(a) $\{C\} \rightsquigarrow^* X_1$ and $\{D\} \rightsquigarrow^* X_2$.

(b) $X_1 \cap X_2 = \{E\}$ for some $E \in S/*$ and $\omega(E) = \omega(C) \cap \omega(D)$.

Let $W' = (W - \{C\}) \cup X_1$ and $Z' = (Z - \{D\}) \cup X_2$, where W', Z' stand for the new values of W, Z, respectively.

Now we show that Equation (10.6) holds in the above construction. In Step 1, it obviously holds for the initial W, Z. Let us verify Equation (10.6) for W', Z' given that it holds for W, Z in Step 2. As $\{C\} \rightsquigarrow^* X_1$ and $\{D\} \rightsquigarrow^* X_2$, we have $W \rightsquigarrow^* W'$ and $Z \rightsquigarrow^* Z'$ by Definition 31. The conclusion immediately follows.

It only remains to show that the above construction terminates. Note that in Step 2, we have $\forall F \in X_1 \cup X_2, \ell(F) \leq \max\{\ell(C), \ell(D)\}$, according to the proof of Lemma 21. Let $r = \max_{F \in X \cup Y} \ell(F)$. By a simple induction, it can be seen that $\forall F \in (W' - W) \cup (Z' - Z), \ell(F) \leq r$. Moreover, consider the set

$$V = \{F \in S/* : \ell(F) \leq r\}.$$

Let us focus on the construction of W. Each time when $W' \neq W$ in Step 2, in $W' - W$ we will visit at least one new element in V. Since V is finite, the number of times when $W' \neq W$ should be finite. A similar statement holds for Z. Hence, the above construction terminates. □

10.5.1 Proof of Lemma 16

Using the previous lemmas, we are ready to prove Lemma 16 by verifying every condition of a semiring in Definition 23.

Proof of Lemma 16. Let us verify the following properties of $\mathcal{S}$:

1. $\emptyset \in \mathcal{S}$.

   This is from the definition of $\mathcal{S}$.

2. For any $X, Y \in \mathcal{S}$, the difference $X - Y$ is a finite disjoint union of sets in $\mathcal{S}$.

   W.l.o.g, suppose that $X \cap Y \neq \emptyset$, $X = \omega(C)$, and $Y = \omega(D)$ for some $C, D \in S/*$. By Lemma 21, there exist two finite sets $W_C, W_D \subseteq S/*$ such that

   (a) $\{C\} \rightsquigarrow^* W_C$ and $\{D\} \rightsquigarrow^* W_D$;

   (b) $W_C \cap W_D = \{E\}$ for some $E \in S/*$ and $\omega(E) = \omega(C) \cap \omega(D)$.

   From Lemma 20, we further have $\omega(C) = \biguplus_{F \in W_C} \omega(F)$. Hence,

   $$X - Y = \omega(C) - \omega(C) \cap \omega(D) = \biguplus_{F \in W_C - \{E\}} \omega(F),$$

   which is a finite disjoint union of sets in $\mathcal{S}$.

3. For any $X, Y \in \mathcal{S}$, $X \cap Y \in \mathcal{S}$.

   W.l.o.g, suppose that $X \cap Y \neq \emptyset$, $X = \omega(C)$, and $Y = \omega(D)$ for some $C, D \in S/*$. The desired property immediately follows from Lemma 21.

By Definition 23, $\mathcal{S}$ is a semiring. □

Technical Lemmas about Semiring $\mathcal{S}$

Next, we prove two lemmas about the semiring $\mathcal{S}$, which are useful in deriving the σ-subadditivity of the function $\mu_{\rho \to S}$ when we prove Lemma 17 in Section 10.5.2. Let us consider a metric d on the set of maximal paths $\omega(A)$ of a quantum process A.

Definition 33 (Metric d)**.** For a quantum process A, we define a metric $d : \omega(A) \times \omega(A) \to \mathbb{R}_{\geq 0}$ such that for any $B, C \in \omega(A)$, $d(B, C) = 2^{-|B \cap C|}$ if $B \neq C$; and $d(B, C) = 0$ otherwise.

It is easy to verify d is indeed a metric. For a quantum process A, $(\omega(A), d)$ forms a metric space. Note that for any partial process $B \in A/*$, the set $\omega(B)$ is open, because $\omega(B) = \left\{ C \in \omega(A) : d(C, C_B) < 2^{-\ell(B)-2} \right\}$ is an open ball of radius $2^{-\ell(B)-2}$ centered at C_B, for any $C_B \in \omega(B)$, where $\ell(\cdot)$ is defined in Definition 32. Similarly, the set $\omega(B)$ is also closed, because $\omega(B) = \left\{ C \in \omega(A) : d(C, C_B) \leq 2^{-\ell(B)-1} \right\}$ is a closed ball of radius $2^{-\ell(B)-1}$ centered at C_B, for any $C_B \in \omega(B)$.

The following lemma shows the compactness of the metric space $(\omega(A), d)$.

Lemma 23. *The metric space $(\omega(A), d)$ is compact.*

Proof. It suffices to show that $(\omega(A), d)$ is complete and totally bounded, as follows.

1. $(\omega(A), d)$ is complete; that is, every Cauchy sequence in $\omega(A)$ converges in $\omega(A)$.

   For any $B \in \omega(A)$, let $a_k(B) \in A$ be such that $\text{root}(A) \to^k a_k(B)$. Consider a Cauchy sequence $B_1, B_2, \ldots \in \omega(A)$; that is, $\forall \epsilon > 0, \exists K \in \mathbb{N}$ such that $\forall k, l > K$, $d(B_k, B_l) < \epsilon$. It is easy to verify by definition that $\forall k \in \mathbb{N}, \exists K \in \mathbb{N}, b_k \in A, \forall l > K, a_k(B_l) = b_k$. Let $B \in \omega(A)$ be such that $a_k(B) = b_k$ for $k \in \mathbb{N}$. Then, $\lim_{k\to\infty} B_k = B$.

2. $(\omega(A), d)$ is totally bounded; that is, $\forall \epsilon > 0$, $\omega(A)$ can be covered by finitely many open balls of radius ϵ.

   For any $\epsilon > 0$, let $m \in \mathbb{N}$ be such that $2^{-m} < \epsilon$. For any $a \in A$ with $\text{root}(A) \to^m a$, let us pick $B_a \in \omega(A)$ such that $a \in B_a$. There are finitely many such B_a. Let $C_a = \{B \in \omega(A) : d(B, B_a) < \epsilon\}$ be the open ball of radius ϵ centered at B_a, then it is easy to see that

   $$\omega(A) \subseteq \bigcup_{a \in A : \text{root}(A) \to^m a} C_a.$$

□

Recall that for any quantum system $C = A \parallel B$, $\omega(C) = \omega(A) \times \omega(B)$. For any trace-preserving quantum system S, we can consider the product topology on $\omega(S)$. As $\omega(S)$ is a product of compact spaces, by Tychonoff theorem, it is also compact. In this case, for any partial system $C \in S/*$, $\omega(C)$ as a finite product of clopen sets is also clopen. Consequently, $\omega(C)$ is also compact.

Finally, we present two useful lemmas.

Lemma 24. *For the semiring $\mathcal{S}$, given $X \subseteq \bigcup_{k\in\mathbb{N}} X_k$ with $X, X_k \in \mathcal{S}$, there exists $K \in \mathbb{N}$ such that $X \subseteq \bigcup_{k\in[K]} X_k$.*

Proof. W.l.o.g., suppose that $X = \omega(C)$ and $X_k = \omega(C_k)$ for some $C, C_k \in S/*$. As shown above, $\omega(C)$ as a compact set has an open cover $\bigcup_{k\in\mathbb{N}} \omega(C_k)$. By the definition of compactness, there exists a finite subcover and the conclusion immediately follows. □

Lemma 25. *For the semiring $\mathcal{S}$, given $X \subseteq \bigcup_{k\in[K]} X_k$ with $X, X_k \in \mathcal{S}$ and $K \in \mathbb{N}$, there exist finite sets $P_k \subseteq \mathcal{S}$ for $k \in [K]$, such that $\forall k \in [K], \biguplus_{Y\in P_k} Y \subseteq X_k$ and $X = \biguplus_{k\in[K]} \biguplus_{Y\in P_k} Y$.*

Proof. For $k \in [K]$, let $Z_k = X \cap X_k - \bigcup_{j<k} X_j$. In this case, $X = \biguplus_{k\in[K]} Z_k$, where $Z_k \subseteq X_k$. As $\mathcal{S}$ is a semiring, Z_k is a finite disjoint union of sets in $\mathcal{S}$, and the conclusion immediately follows. □

The above two lemmas together enable one to derive the σ-subadditivity of a function μ on $\mathcal{S}$ from the additivity of μ.

10.5.2 Proof of Lemma 17

Using the previous lemmas, now we are ready to prove Lemma 17.

Proof of Lemma 17. Given an initial state ρ, define a function $\mu : \mathcal{S} \to [0,1]$ such that $\mu(\emptyset) = 0$, and for $C \in S/*$,

$$\mu \circ \omega(C) = \lim_{t\to\infty} \mathrm{tr}([\![C]\!](t)(\rho)),$$

where we omit the subscript S in $[\![\cdot]\!]_S(\cdot)$ for simplicity. The limit exists because $\mathrm{tr}([\![C]\!](t)(\rho))$ is non-increasing with respect to t, according to Definition 22 (d); and bounded below by 0, because $[\![C]\!](t) \in \mathcal{QO}(\mathcal{H}_C)$ according to Definition 22. In the following we show that μ satisfies the conditions for Lemma 15, and therefore can be uniquely extended to the desired probability measure $\mu_{\rho\to S}$. To this end, we only need to verify the following properties:

1. $\mu(\emptyset) = 0$ and $\mu \circ \omega(S) = 1$.

   From the definition of μ, it is obvious that $\mu(\emptyset) = 0$. From Definitions 22 (a) and 22 (d), it is also easy to obtain $\mu \circ \omega(S) = \mathrm{tr}([\![S]\!](0)(\rho)) = \mathrm{tr}(\rho) = 1$.

2. For any two finite sets $P, Q \subseteq S/*$ with $P \rightsquigarrow^* Q$, we have $\sum_{C\in P} \mu \circ \omega(C) = \sum_{C\in Q} \mu \circ \omega(C)$.

   By a simple induction on the transitive closure $\rightsquigarrow^*$, it suffices to prove this property with $\rightsquigarrow^*$ replaced by $\rightsquigarrow$. In this case, since $P \rightsquigarrow Q$, by Definition 31, we have $P - Q = \{A/a \parallel B\}$ for some trace-preserving $A \in \mathbf{Proc}$, $a \in A$ and $B \in \mathbf{Sys}$; and $Q - P = \{A/b \parallel B : a \to b\}$. Now it suffices to show that

   $$\mu \circ \omega(A/a \parallel B) = \sum_{b:a\to b} \mu \circ \omega(A/b \parallel B). \tag{10.8}$$

   Let us choose any sufficiently large t such that $t \geq \max_{b:a\to b} \max T[b]$ and $A/b \parallel B$ is trace-preserving after time t. By Definition 22 (d) and the definition of μ, we have

   $$\mu \circ \omega(A/a \parallel B) = \mathrm{tr}([\![A/a \parallel B]\!](t)(\rho)),$$
   $$\mu \circ \omega(A/b \parallel B) = \mathrm{tr}([\![A/b \parallel B]\!](t)(\rho)),$$

   for all b with $a \to b$. The above together with Definition 22 (b) yield Equation (10.8).

3. μ is additive; that is, $\mu(X) = \sum_{k=1}^{K} \mu(X_k)$ for any $X \in \mathcal{S}$ and finitely many $X_1, X_2, \ldots, X_K \in \mathcal{S}$ with $X = \biguplus_{k=1}^{K} X_k$ and $K \in \mathbb{N}$.

   W.l.o.g., suppose that $X = \omega(C)$ and $X_k = \omega(C_k)$ for some $C, C_k \in S/*$ and $k \in [K]$. Let $P = \{C_k\}_{k\in[K]} \subseteq S/*$. Since $\omega(C) = \biguplus_{D\in P} \omega(D)$, using Lemma 22, there exists

a finite set $Q \subseteq S/*$ such that $\{C\} \rightsquigarrow^* Q$ and $P \rightsquigarrow^* Q$. By Property 2 just proved above, we have $\mu \circ \omega(C) = \sum_{D \in Q} \mu \circ \omega(D)$ and $\sum_{D \in P} \mu \circ \omega(D) = \sum_{D \in Q} \mu \circ \omega(D)$. Consequently, $\mu(X) = \mu \circ \omega(C) = \sum_{k \in [K]} \mu \circ \omega(C_k) = \sum_{k \in [K]} \mu(X_k)$.

4. μ is σ-subadditive; that is, $\mu(X) \leq \sum_{k \in \mathbb{N}} \mu(X_k)$ for any $X \in \mathcal{S}$ and countably many $X_1, X_2, \ldots \in \mathcal{S}$ with $X \subseteq \bigcup_{k \in \mathbb{N}} X_k$.

   Using Lemmas 24 and 25, there exist $P_1, P_2, \ldots, P_K \subseteq \mathcal{S}$ for some $K \in \mathbb{N}$ such that $\forall k \in [K], \biguplus_{Y \in P_k} Y \subseteq X_k$ and $X = \biguplus_{k \in [K]} \biguplus_{Y \in P_k} Y$. Since $\mathcal{S}$ is a semiring, for any $k \in [K]$, there exists a finite set $Q_k \subseteq \mathcal{S}$ such that $X_k = \biguplus_{Y \in P_k} Y \uplus \biguplus_{Z \in Q_k} Z$. By Property 3 just proved above, this implies $\sum_{Y \in P_k} \mu(Y) \leq \sum_{Y \in P_k} \mu(Y) + \sum_{Z \in Q_k} \mu(Z) = \mu(X_k)$. Finally, we have

$$\mu(X) = \sum_{k \in [K]} \sum_{Y \in P_k} \mu(Y) \leq \sum_{k \in [K]} \mu(X_k) \leq \sum_{k \in \mathbb{N}} \mu(X_k).$$

□

10.6 Discussion

10.6.1 Related Work

Analysis of Non-Atomic Distributed Systems

There are several methods to analyse a non-atomic distributed (classical) system in the literature. In particular, Lamport proposed two methods of reasoning about non-atomic systems. The first, behaviour reasoning, began with [8] and culminated in the celebrated two-arrow model [10, 11, 178, 179, 368]. This method was used to prove the correctness of solutions to the mutual exclusion problem [10, 11, 368] and constructions of atomic registers [178, 179]. The second method, assertional reasoning, is a more formal approach based on discrete-time states and actions. Its power was demonstrated when it was used to discover [371] two unrevealed assumptions in previous correctness proofs [8, 368, 372] of the bakery algorithm [8].

On the other hand, Herlihy and Wing considered linearizability [366], which, in addition to the traditional atomicity, takes into account the real-time orders between actions. A distributed system is linearizable if it is equivalent to another system with atomic actions that also preserve the real-time orders. This notion was later extended to set-linearizability [374] and interval-linearizability [375].

The prior works discussed above all rely on either the partial-order semantics or the notion of a discrete-time state. As we argued in Section 9.4, these classical concepts are

difficult to straightforwardly extend to the quantum setting, due to the challenges identified in Section 9.3.1. In contrast, the model presented in this chapter defines the equivalence between distributed quantum systems based on the observable semantics, which is derived directly from the most fundamental real-time semantics.

10.6.2 Summary

In this chapter, we formally defined a model of non-atomic distributed quantum systems. We introduced the notions of actions, processes and distributed systems in the quantum setting. Building on these notions, we defined two types of semantics of a distributed quantum system. The real-time semantics characterises the real-time evolution of the system state. From this, we further defined the observable semantics that captures all information a classical observer can access, namely, the probabilities of all classical observable events within the system.

As the observable semantics is what concerns an external programmer, we used it to define the equivalence between distributed quantum systems. This notion of equivalence provided the basis for formally stating our major aim of guaranteeing the atomicity of local actions (see also Theorem 12) in the next chapter. The definitions in this chapter were carefully designed to be insensitive to how actions are physically implemented, thereby addressing the challenges identified in Section 9.3.1.

Chapter 11

Atomicity of Local Actions

Using the model of non-atomic distributed quantum systems established in the previous chapter, now we are ready to provide a rigorous guarantee for the atomicity of local actions. To this end, we first prove an intermediate theorem showing that any distributed quantum system is equivalent to another in which local actions are instantaneous. Then, we are able to prove the main theorem: any distributed quantum system is equivalent to another in which local actions are atomic.

11.1 Overview

11.1.1 Atomic Actions

Based on the model developed in Chapter 10, we can prove our main theorem, which provides a rigorous guarantee for the atomicity of local actions. Let us first formalise the notion of atomic action.

Definition 34 (Atomic actions)**.** In a distributed quantum system S, a set $D \subseteq S$ is said to be atomic, if $\forall a, b \in D$ with $a \in A, b \in B$ for some $A, B \in \mathbf{Sys}$ and $S = A \parallel B$, either $T[a] < T[b]$ or $T[b] < T[a]$.

Remark 1. In Definition 34, it is worth noting that the sequentiality condition (either $T[a] < T[b]$ or $T[b] < T[a]$) is only imposed on atomic actions, and nothing is presumed for non-atomic actions. What is the temporal relation between an atomic and a non-atomic action? They can still be concurrent (and not sequential), because the former is "indivisible" and the latter is "divisible".

11.1.2 An Intermediate Theorem

Then, we prove the following intermediate theorem, showing any distributed quantum system is equivalent to another system where local actions are instantaneous.

Theorem 13 (Instantaneous Local Actions)**.** *Given a trace-preserving distributed quantum system $S \in \mathbf{Sys}$, there exists another trace-preserving system $S' \simeq S$ and an isomorphism $\gamma : S \to S'$ such that:*

1. *For any local action $a \in S$, $e[\gamma(a)] = \emptyset$ and $T[\gamma(a)] = \{t_a\}$ for some $t_a \in \mathbb{R}_{\geq 0}$.*
2. *For any non-local action $a \in S$, $\gamma(a) = a$.*

The insight behind Theorem 13 is similar to the shrinking of time intervals in [178, Proposition 1] and [177, Proposition 4]. The difference is that in [177, 178], the equivalence between systems is defined based on the partial order semantics, which implicitly assumes the atomicity of local actions. In contrast, here, our equivalence is based on the observable semantics, derived from the more fundamental real-time semantics. Further comparison can be found in Section 9.4.1.

We give a proof sketch of Theorem 13. For readability, the full proof is deferred to Section 11.2.

Proof sketch of Theorem 13. The proof consists of two steps.

1. Given a system S and a local action $a \in S$, we can change the time interval $T[a]$ of a to an instant $\{t_a\}$ with $t_a \in T[a]$ and obtain a new system S'. Then, $S \simeq S'$.

   To prove $S \simeq S'$, suppose that $T[a] = [x, y]$. For any partial system $C \in S/*$, denote the corresponding partial system (via the isomorphism γ) of S' by C'. By Definition 30 and Lemma 17, it suffices to prove $\mu_{\rho \to S} \circ \omega(C) = \mu_{\rho \to S'} \circ \omega(C')$ for any state ρ and partial system C. By decomposing C using Lemma 26 and Definition 22 (b), the task can be further reduced to the case when $C \restriction_{[0,y]}$ has no branching. Then, we can prove the following equalities step by step, by leveraging Definition 22 (c):

   (a) $[\![C]\!]_S(x-) = [\![C']\!]_{S'}(x-)$;

   (b) $[\![C]\!]_S(y) = [\![C']\!]_{S'}(y)$; and

   (c) $[\![C]\!]_S(t) = [\![C']\!]_{S'}(t)$ for $t > y$.

   In the above, the second equality is the most complicated to prove, where we resort to the Dijkstra-Lamport condition in Definition 22 (c). Taking the trace and $t \to +\infty$ in the last equality leads to the conclusion, according to Lemma 17.

2. Given a system S, we can construct a family of systems $\{S_m\}_{m \in \mathbb{N}}$, such that each S_m is obtained from S_{m-1} through the above Step 1. Then, we have $S \simeq S_1 \simeq S_2 \simeq \ldots$. Taking the limit $S' = \lim_{m \to \infty} S_m$, we can verify that all local actions in S' are instantaneous and $S' \simeq S$.

To prove $S' \simeq S$, for any partial system $C \in S/*$, let us use C_m and C' to denote its corresponding partial systems in $S_m/*$ and $S'/*$, respectively. From Definition 22 (d), we can choose sufficiently large t such that C is trace-preserving after time t. Let us pick m such that $C_m \restriction_{[0,t]} = C' \restriction_{[0,t]}$. Consequently, by Definition 22 (c), $\mathrm{tr}\big([\![C_m]\!]_{S_m}(t)(\rho)\big) = \mathrm{tr}([\![C']\!]_{S'}(t)(\rho))$, which leads to $S' \simeq S_m$ according to Definition 22 (d) and Lemma 17. The conclusion follows from $S \simeq S_m \simeq S'$.

□

11.1.3 Main Theorem

Based on Theorem 13, we can prove our main theorem showing any distributed quantum system is equivalent to another system where local actions are atomic. This establishes a rigorous guarantee for the atomicity of local actions. An example of how this theorem can be applied has been shown in Section 9.3.3.

Theorem 14 (Atomicity of Local Actions)**.** *Given a trace-preserving and aligned distributed quantum system $S \in$ **Sys** with $\forall a \in S, |T[a]| > 0$, there exists another trace-preserving and aligned system $S' \simeq S$ such that local actions in S' are atomic.*

We give a proof sketch of Theorem 14. For readability, the full proof is deferred to Section 11.2.

Proof sketch of Theorem 14. Given Theorem 13, one can replace all local actions in S with instantaneous versions to obtain a system S'. Since $|T[a]| > 0$ for all $a \in S$, it is possible to arrange the instants of these local actions such that they never overlap. The conclusion immediately follows. □

Note that any practical system naturally satisfies the conditions of S in Theorem 14: (a) If a system has a physical implementation, it preserves the probability and is hence trace-preserving. (b) Since partial measurements from the same measurement are consistently performed by the same device, the system is also aligned. (c) Finally, in the real world, no actions are performed instantly, which precisely means $\forall a \in S, |T[a]| > 0$.

11.2 Proof Details

In this section, we present the proof details of Theorems 13 and 14.

11.2.1 Proof of Theorem 13

Before proving the intermediate Theorem 13, we need the following lemma, showing that for any time t, a partial system can be decomposed with respect to the relation $\rightsquigarrow$

(see Definition 31) such that any partial system in the decomposition has no branching when restricted to the time region $[0, t]$.

Lemma 26. *For any trace-preserving distributed quantum system $S \in \mathbf{Sys}$, partial system $C \in S/*$, and $t \in \mathbb{R}_{\geq 0}$, there exists a finite set $P \subseteq S/*$ such that $\{C\} \rightsquigarrow^* P$ and for any $D \in P$, $D \restriction_{[0,t]}$ has no branching.*

Proof. We show how to construct P step by step. In the construction, we require the intermediate P to satisfy

$$\{C\} \rightsquigarrow^* P. \tag{11.1}$$

The construction is as follows.

1. Initially, let $P = \{C\}$.
2. Repeat the following procedure. First pick $D \in P$ such that $D \restriction_{[0,t]}$ has branching. If such D does not exist, then we can terminate. Otherwise, we have $D = A/a \parallel B$ for some $A \in \mathbf{Proc}$, $a \in A$ and $B \in \mathbf{Sys}$, where $(A/a) \restriction_{[0,t]}$ has branching. As a result, $\{b \in A : a \to b\} \neq \emptyset$. Let $P' = (P - D) \cup \{A/b \parallel B : a \to b\}$, where P' stands for the new value of P.

It is easy to see that Equation (11.1) holds in the above construction. It remains to show that the above construction terminates. For any $E \in S/*$, let

$$\phi(E) = \#\Big\{(c, d) : c, d \in E \restriction_{[0,t]} \wedge c \not\to^* d \wedge d \not\to^* c\Big\}$$

be the number of action pairs (c, d) that cannot be ordered by $\to^*$ in $E \restriction_{[0,t]}$. Note that in Step 2, $\sum_{b:a\to b} \phi(A/b \parallel B) < \phi(A/a \parallel B)$. Consequently, $\sum_{E\in P'} \phi(E) < \sum_{E\in P} \phi(E)$. Initially, $\phi(C)$ is finite due to Definition 18. Hence, the above construction terminates. □

Now we prove the intermediate Theorem 13.

Proof of Theorem 13. The proof consists of two steps.

1. Let us fix a local action $a \in S$. Without loss of generality, suppose that $T[a] = [x, y]$ and $x > 0$. Consider another trace-preserving system S' and an isomorphism $\gamma : S \to S'$ such that:

   (a) $e[\gamma(a)] = \emptyset$, and $T[\gamma(a)] = \{t_a\}$ for some $t_a \in T[a]$.

   (b) $\forall b \neq a \in S, \gamma(b) = b$.

   Let us prove $S \simeq S'$.

By Definition 30 and Lemma 17, it suffices to show for any state $\rho \in \mathcal{D}\left(\mathcal{H}_{q[S]}\right)$ and any partial system $C \in S/*$,

$$\mu_{\rho\to S} \circ \omega(C) = \mu_{\rho\to S'} \circ \omega \circ \gamma(C). \tag{11.2}$$

Let us fix a state ρ and a partial system C. By Lemma 26, there exists a finite $P \subseteq S/*$ such that $\{C\} \rightsquigarrow^* P$ and for any $D \in P$, $D \restriction_{[0,y]}$ has no branching. From the proof of Lemma 26, it is easy to see $\{\gamma(C)\} \rightsquigarrow^* \gamma(P)$, and for any $D \in P$, $\gamma(D) \restriction_{[0,y]}$ has no branching. Using Lemma 20 and that $\mu_{\rho\to S}$ and $\mu_{\rho\to S'}$ are probability measures, we have

$$\begin{aligned}\mu_{\rho\to S} \circ \omega(C) &= \sum_{D\in P} \mu_{\rho\to S} \circ \omega(D)\\ \mu_{\rho\to S'} \circ \omega \circ \gamma(C) &= \sum_{D\in P} \mu_{\rho\to S'} \circ \omega \circ \gamma(D).\end{aligned}$$

Now proving Equation (11.2) reduces to proving

$$\mu_{\rho\to S} \circ \omega(D) = \mu_{\rho\to S'} \circ \omega \circ \gamma(D) \tag{11.3}$$

for $D \in S/*$, where $D \restriction_{[0,y]}$ and $\gamma(D) \restriction_{[0,y]}$ have no branching.

If $a \notin D$, then Equation (11.3) is trivial because $D = \gamma(D)$. In the following, we only need to consider $a \in D$. In particular, assume $D = A \parallel B$ for some $A \in \mathbf{Proc}$ and $B \in \mathbf{Sys}$ with $a \in A$.

Note that $D \restriction_I$ and $\gamma(D) \restriction_I$ have no branching for any interval $I \subseteq [0, y]$. Also, $D \restriction_I = \gamma(D) \restriction_I$ for any interval $I \subseteq [0, x) \cup (y, +\infty)$, and $B \restriction_I = \gamma(B) \restriction_I$ for any interval $I \subseteq [x, y]$. In the following, let us check $[\![D]\!]_S(t) = [\![\gamma(D)]\!]_{S'}(t)$ for $t = 0, x-$ and y step by step, each building upon the previous. Some condition checks are obvious and thus omitted for readability. For simplicity of notations, when the context is clear, we omit the subscript S or S' in the real-time semantics $[\![\cdot]\!]_S(\cdot)$ and $[\![\cdot]\!]_{S'}(\cdot)$.

(a) The $t = 0$ case is obvious by Definition 22 (a).

(b) The $t = x-$ case is also simple by taking $I = (0, x)$ in the first part of Definition 22 (c) (see also the remarks there), combined with $D \restriction_I = \gamma(D) \restriction_I$.

(c) The $t = y$ case is more complicated. Let $I_1 = [x, t_a), I_2 = [t_a, t_a] = \{t_a\}$ and $I_3 = (t_a, y]$. Note that $A \restriction_{[x,y]} = \{a\}$ and a is local. By taking $I = [x, y]$ in the second part of Definition 22 (c), we have $[\![D]\!]_S(y) = (\mathcal{E}[a] \otimes \mathcal{F}') \circ [\![\gamma(D)]\!]_{S'}(x-)$, for some $\mathcal{F}'$ uniquely determined by $B \restriction_{[x,y]}$. Alternatively, by taking $I =$

I_1, I_2, I_3 in the second part of Definition 22 (c), we have

$$\begin{aligned}[\![D]\!](t_a-) &= (\mathcal{F}_1 \otimes \mathcal{F}'_1) \circ [\![D]\!](x-)\\ [\![D]\!](t_a) &= (\mathcal{F}_2 \otimes \mathcal{F}'_2) \circ [\![D]\!](t_a-)\\ [\![D]\!](y) &= (\mathcal{F}_3 \otimes \mathcal{F}'_3) \circ [\![D]\!](t_a),\end{aligned}$$

for some quantum operations $\mathcal{F}_b$ and $\mathcal{F}'_b$ uniquely determined by $A \restriction_{I_b}$ and $B \restriction_{I_b}$ with $b \in [3]$, respectively. As a result, $\mathcal{F}' = \mathcal{F}'_3 \circ \mathcal{F}'_2 \circ \mathcal{F}'_1$.
Note that $\gamma(A) \restriction_{[x,t_a)} = \gamma(A) \restriction_{(t_a,y]} = \emptyset$ since a is local. By taking $I = I_1, I_3$ in the second part of Definition 22 (c), combined with $B \restriction_I = \gamma(B) \restriction_I$, we have

$$\begin{aligned}[\![\gamma(D)]\!](t_a-) &= (\mathbb{1} \otimes \mathcal{F}'_1) \circ [\![\gamma(D)]\!](x-)\\ [\![\gamma(D)]\!](y) &= (\mathbb{1} \otimes \mathcal{F}'_3) \circ [\![\gamma(D)]\!](t_a).\end{aligned}$$

By taking $I = I_2$ in the second part of Definition 22 (c), combined with $B \restriction_I = \gamma(B) \restriction_I$, we have $[\![\gamma(D)]\!](t_a) = (\mathcal{E}[\gamma(a)] \otimes \mathcal{F}'_2) \circ [\![\gamma(D)]\!](t_a-)$. Since γ is an isomorphism, $\mathcal{E}[\gamma(a)] = \mathcal{E}[a]$.
The above together yield the conclusion.

For any $t > y$, we also have $[\![D]\!]_S(t) = [\![\gamma(D)]\!]_{S'}(t)$, by taking $I = (y, t]$ in the second part of Definition 22 (c), combined with $D \restriction_I = \gamma(D) \restriction_I$ and the above results.

Finally, from Lemma 17, we have

$$\begin{aligned}\mu_{\rho\to S} \circ \omega(D) &= \lim_{t\to\infty} \mathrm{tr}([\![D]\!]_S(t)(\rho))\\ \mu_{\rho\to S'} \circ \omega \circ \gamma(D) &= \lim_{t\to\infty} \mathrm{tr}([\![\gamma(D)]\!]_{S'}(t)(\rho)),\end{aligned}$$

and Equation (11.3) immediately follows.

2. Now we construct a system S' and an isomorphism $\gamma : S \to S'$ that satisfy the properties in Theorem 13. Since S as a set is countable, we can enumerate all local actions in S as $\{a_1, a_2, \ldots\}$. Consider a set of systems $\{S_m\}_{m\in\mathbb{N}}$ with each $S_m \in \mathbf{Sys}$ and a set of isomorphisms $\{\eta_m\}_{m\in\mathbb{N}}$ with each $\eta_m : S_m \to S_{m+1}$ such that $S_1 = S$ and for any $m > 1$:

   (a) $e[\eta_m(a_m)] = \emptyset$ and $T[\eta_m(a_m)] = \{t_m\}$ for some $t_m \in T[a_m]$.
   (b) $\forall b \neq a_m \in S_m, \eta_m(b) = b$.

   According to the results in Step 1, we have $\forall m \in \mathbb{N}, S_m \simeq S$. Let $\gamma_m = \eta_m \circ \eta_{m-1} \circ \ldots \circ \eta_1$, then $\gamma_m : S \to S_m$ is an isomorphism. Let $\gamma = \lim_{m\to\infty} \gamma_m$ be the point-wise limit of γ_m. It can be seen that γ is an isomorphism and, together with the system $S' = \gamma(S)$, satisfies the first and second properties in Theorem 13.

It remains to show that $S' \simeq S$; that is, by Definition 30, to show for any state ρ and partial system $C \in S/*$,

$$\mu_{\rho\to S} \circ \omega(C) = \mu_{\rho\to S'} \circ \omega \circ \gamma(C). \tag{11.4}$$

As $\forall m \in \mathbb{N}, S_m \simeq S$, proving Equation (11.4) reduces to proving the existence of some $m' \in \mathbb{N}$ such that

$$\mu_{\rho\to S_{m'}} \circ \omega \circ \gamma_{m'}(C) = \mu_{\rho\to S'} \circ \omega \circ \gamma(C). \tag{11.5}$$

To this end, let us first choose a sufficiently large $t \in \mathbb{R}_{\geq 0}$ such that C is trace-preserving after time t. Since for any $a \in C$ and $m \in \mathbb{N}$, $T[\gamma_m(a)] \subseteq T[a]$, we have that $\gamma_m(C)$ and $\gamma(C)$ are also trace-preserving after time t. According to Lemma 17 and Definition 22 (d), proving Equation (11.5) further reduces to proving

$$\mathrm{tr}\Big(\llbracket \gamma_{m'}(C) \rrbracket_{S_{m'}}(t)(\rho)\Big) = \mathrm{tr}(\llbracket \gamma(C) \rrbracket_{S'}(t)(\rho)). \tag{11.6}$$

Let us choose a sufficiently large $m' \in \mathbb{N}$ such that $\gamma_{m'}(C) \restriction_{(0,t]} = \gamma(C) \restriction_{(0,t]}$, which is achievable because the set $C \restriction_{(0,t]}$ is finite and $\gamma_m(C) \restriction_{(0,t]} \subseteq C \restriction_{(0,t]}$ for any $m \in \mathbb{N}$. Now by taking $I = (0, t]$ in Definition 22 (c), we have $\llbracket \gamma_{m'}(C) \rrbracket(t) = \mathcal{F} \circ \llbracket \gamma_{m'}(C) \rrbracket(0)$ and $\llbracket \gamma(C) \rrbracket(t) = \mathcal{F} \circ \llbracket \gamma(C) \rrbracket(0)$ for some quantum operation $\mathcal{F}$ uniquely determined by $\gamma_{m'}(C) \restriction_{(0,t]} = \gamma(C) \restriction_{(0,t]}$. Combined with $\llbracket \gamma_{m'}(C) \rrbracket_{S_{m'}}(0) = \llbracket \gamma(C) \rrbracket_{S'}(0)$ from Definition 22 (a), we have $\llbracket \gamma_{m'}(C) \rrbracket_{S_{m'}}(t) = \llbracket \gamma(C) \rrbracket_{S'}(t)$, and Equation (11.6) immediately follows.

□

11.2.2 Proof of Theorem 14

Finally, we are able to prove our main Theorem 14.

Proof of Theorem 14. Let us construct a system S' that satisfies the properties in Theorem 14. Since S as a set is countable, we can enumerate all local actions in S as $L = \{a_1, a_2, \ldots\}$. Consider a system S' and an isomorphism $\gamma : S \to S'$ such that

1. $\forall m \in \mathbb{N}, e[\gamma(a_m)] = \emptyset$ and $T[\gamma(a_m)] = \{t_m\}$, where t_m are chosen by

   $$t_m \in T[a_m] - \{t_n : n < m, a_n \in A, a_m \in B, S = A \parallel B\}.$$

   The existence of t_m is guaranteed by $|T[a_m]| > 0$.

2. $\forall b \notin L, \gamma(b) = b$.

Moreover, as S is aligned, when choosing the $\{t_m\}_{m\in\mathbb{N}}$ we can also require that for any $b \in S$ and $j,k \in \mathbb{N}$, if $b \to a_j$ and $b \to a_k$ then $t_j = t_k$, which is achievable because $\min T\left[a_j\right] = \min T[a_k]$ and $\left|T\left[a_j\right]\right|, |T[a_k]| > 0$. It is easy to see that S' satisfies the properties in Theorem 14. By the proof of Theorem 13, we also have $S' \simeq S$. □

11.3 Conclusion and Open Questions

In this chapter, we established a rigorous guarantee for the atomicity of local actions in distributed quantum computing, by proving that any system is equivalent to another in which local actions are atomic. To achieve this, we first defined the notion of atomic action based on our model of non-atomic distributed quantum systems in Chapter 10. Then, we proved an intermediate theorem showing any system is equivalent to one where local actions are instantaneous. Building on this, we proved our main theorem without assuming any pre-existing atomic actions.

This work is just one of the first steps to a theory of concurrency in quantum computing. To conclude Part III of this thesis, we list several questions for future research.

1. In this work, we focused only on the atomicity of local actions. An immediate open question is whether the atomicity of non-local actions can also be rigorously guaranteed, particularly through purely software-based methods (as was done in the classical case [178, 179]). Note that classical actions like `read/write` cannot be simulated by a *single* action in quantum computing (i.e., unitary and measurement), preventing a straightforward application of classical results.

2. The systems considered in Part III contained only classical control flow. A compelling question is how to handle concurrency control in the presence of quantum control flow (see Part I), which would seem to involve causal orders in superposition.

3. Verifying sequential quantum systems is of great practical interest (e.g., see [376, 377]). While there have been attempts to verify high-level distributed (or concurrent) quantum systems (e.g., [36, 142]), a promising future direction is to develop techniques to verify low-level models such as the one presented in Chapter 10.

Bibliography

[1] E. W. Dijkstra, "Notes on structured programming," circulated privately, 1970. DOI: 10.26153/tsw/53177. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[2] O. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured programming*. Academic Press Ltd., 1972.

[3] V. A. Alfred, S. L. Monica, R. Sethi, and D. U. Jeffrey, *Compilers: principles, techniques & tools*. Pearson Education, 2007.

[4] E. W. Dijkstra, "The structure of the "THE"-multiprogramming system," in *Proceedings of the First ACM Symposium on Operating System Principles*, ser. SOSP '67, 1967, pp. 10.1–10.6. DOI: 10.1145/363095.363143.

[5] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages: NATO Advanced Study Institute*, Academic Press, 1968, pp. 43–112. DOI: 10.1007/978-1-4757-3472-0_2.

[6] B. W. Lampson, "Dynamic protection structures," in *Proceedings of the November 18-20, 1969, fall joint computer conference*, 1969, pp. 27–38. DOI: 10.1145/1478559.1478563.

[7] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, 1965. DOI: 10.1145/365559.365617.

[8] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, 1974. DOI: 10.1145/361082.361093.

[9] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, pp. 115–138, 1971. DOI: 10.1007/BF00289519.

[10] L. Lamport, "The mutual exclusion problem: Part i— a theory of interprocess communication," *Journal of the ACM*, vol. 33, no. 2, pp. 313–326, 1986. DOI: 10.1145/5383.5384.

[11] L. Lamport, "The mutual exclusion problem: Part ii— statement and solutions," *Journal of the ACM*, vol. 33, no. 2, pp. 327–348, 1986. DOI: 10.1145/5383.5385.

[12] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual IEEE Symposium on Foundations of Computer Science*, ser. FOCS '94, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[13] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, ser. STOC '96, 1996, pp. 212–219. DOI: 10.1145/237814.237866.

[14] S. Lloyd, "Universal quantum simulators," *Science*, vol. 273, no. 5278, pp. 1073–1078, 1996. DOI: 10.1126/science.273.5278.1073.

[15] F. Arute, K. Arya, R. Babbush, *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, pp. 505–510, 2019. DOI: 10.1038/s41586-019-1666-5.

[16] D. Bluvstein, S. J. Evered, A. A. Geim, *et al.*, "Logical quantum processor based on reconfigurable atom arrays," *Nature*, vol. 626, no. 7997, pp. 58–65, 2023. DOI: 10.1038/s41586-023-06927-3.

[17] S. Bravyi, A. W. Cross, J. M. Gambetta, D. Maslov, P. Rall, and T. J. Yoder, "High-threshold and low-overhead fault-tolerant quantum memory," *Nature*, vol. 627, no. 8005, pp. 778–782, 2024. DOI: 10.1038/s41586-024-07107-7.

[18] Google Quantum AI and Collaborators, "Quantum error correction below the surface code threshold," *Nature*, vol. 638, no. 8052, pp. 920–926, 2025. DOI: 10.1038/s41586-024-08449-y.

[19] A. Javadi-Abhari, M. Treinish, K. Krsulich, *et al.*, *Quantum computing with Qiskit*, 2024. arXiv: 2405.08810 [quant-ph].

[20] C. Developers, *Cirq*. Zenodo, 2025. DOI: 10.5281/ZENODO.4062499. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.4062499.

[21] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q#: Enabling scalable quantum computing and development with a high-level DSL," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, 2018. DOI: 10.1145/3183895.3183901.

[22] T. Altenkirch and J. Grattage, "A functional quantum programming language," in *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*, 2005, pp. 249–258. DOI: 10.1109/LICS.2005.1.

[23] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, and F. Chong, "Scaffold: Quantum programming language," 2012.

[24] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: A scalable quantum programming language," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 333–342, 2013. DOI: 10.1145/2499370.2462177.

[25] M. Ying, *Foundations of quantum programming*. Morgan Kaufmann, 2016. DOI: 10.1016/C2014-0-02660-3.

[26] B. Bichsel, M. Baader, T. Gehr, and M. Vechev, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 286–300. DOI: 10.1145/3385412.3386007.

[27] C. Yuan and M. Carbin, "Tower: Data structures in quantum superposition," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 259–288, 2022. DOI: 10.1145/3563297.

[28] F. Voichick, L. Li, R. Rand, and M. Hicks, "Qunity: A unified language for quantum and classical computing," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 921–951, 2023. DOI: 10.1145/3571225.

[29] R. Seidel, S. Bock, R. Zander, M. Petrič, N. Steinmann, N. Tcholtchev, and M. Hauswirth, *Qrisp: A framework for compilable high-level programming of gate-based quantum computers*, 2024. arXiv: 2406.14792 [quant-ph].

[30] E. D'hondt and P. Panangaden, "Quantum weakest preconditions," *Mathematical Structures in Computer Science*, vol. 16, no. 3, pp. 429–451, 2006. DOI: 10.1017/S0960129506005251.

[31] M. Ying, "Floyd-Hoare logic for quantum programs," *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 6, pp. 1–49, 2011. DOI: 10.1145/2049706.2049708.

[32] L. Zhou, N. Yu, and M. Ying, "An applied quantum Hoare logic," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1149–1162. DOI: 10.1145/3314221.3314584.

[33] Y. Feng and M. Ying, "Quantum Hoare logic with classical variables," *ACM Transactions on Quantum Computing*, vol. 2, no. 4, pp. 1–43, 2021. DOI: 10.1145/3456877.

[34] L. Zhou, G. Barthe, J. Hsu, M. Ying, and N. Yu, "A quantum interpretation of bunched logic & quantum separation logic," in *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2021, pp. 1–14. DOI: 10.1109/LICS52264.2021.9470673.

[35] X. Le, S. Lin, J. Sun, and D. Sanan, "A quantum interpretation of separating conjunction for local reasoning of quantum programs based on separation logic," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–27, 2022. DOI: 10.1145/3498697.

[36] M. Ying, L. Zhou, Y. Li, and Y. Feng, "A proof system for disjoint parallel quantum programs," *Theoretical Computer Science*, vol. 897, pp. 164–184, 2022. DOI: 10.1016/j.tcs.2021.10.025.

[37] M. Ying and Z. Zhang, *Verification of recursively defined quantum circuits*, 2024. arXiv: 2404.05934 [quant-ph].

[38] M. Ying, *A practical quantum Hoare logic with classical variables, I*, 2025. arXiv: 2412.09869 [cs.PL].

[39] O. Brunet and P. Jorrand, "Dynamic quantum logic for quantum programs," *International Journal of Quantum Information*, vol. 2, no. 01, pp. 45–54, 2004. DOI: 10.1142/S0219749904000067.

[40] A. Baltag and S. Smets, "LQP: The dynamic logic of quantum information," *Mathematical structures in computer science*, vol. 16, no. 3, pp. 491–525, 2006. DOI: 10.1017/S0960129506005299.

[41] T. Takagi, C. M. Do, and K. Ogata, "Automated quantum program verification in dynamic quantum logic," in *International Workshop on Dynamic Logic*, 2023, pp. 68–84. DOI: 10.1007/978-3-031-51777-8_5.

[42] C. M. Do, T. Takagi, and K. Ogata, "Automated quantum protocol verification based on concurrent dynamic quantum logic," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 6, pp. 1–36, 2025. DOI: 10.1145/3708475.

[43] N. Yu, *Quantum temporal logic*, 2019. arXiv: 1908.00158 [cs.LO].

[44] Y. Peng, M. Ying, and X. Wu, "Algebraic reasoning of quantum programs via non-idempotent Kleene algebra," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 657–670. DOI: 10.1145/3519939.3523713.

[45] K. Singhal, *Quantum Hoare type theory*, 2021. arXiv: 2012.02154 [cs.PL].

[46] C. Yuan, C. McNally, and M. Carbin, "Twist: Sound reasoning for purity and entanglement in quantum programs," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–32, 2022. DOI: 10.1145/3498691.

[47] Y. Chen, K. Chung, O. Lengál, J. Lin, W. Tsai, and D. Yen, "An automata-based framework for verification and bug hunting in quantum circuits," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1218–1243, 2023. DOI: 10.1145/3591270.

[48] P. A. Abdulla, Y. Chen, Y. Chen, L. Holík, O. Lengál, J. Lin, F. Lo, and W. Tsai, "Verifying quantum circuits with level-synchronized tree automata," *Proceedings of the ACM on Programming Languages*, vol. 9, no. POPL, pp. 923–953, 2025. DOI: 10.1145/3704868.

[49] J. Liu, B. Zhan, S. Wang, S. Ying, T. Liu, Y. Li, M. Ying, and N. Zhan, "Formal verification of quantum algorithms using quantum hoare logic," in *International Conference on Computer Aided Verification*, 2019, pp. 187–207. DOI: 10.1007/978-3-030-25543-5_12.

[50] D. Unruh, "Quantum relational hoare logic," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019. DOI: 10.1145/3290346.

[51] D. Unruh, "Post-quantum verification of Fujisaki-Okamoto," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2020, pp. 321–352. DOI: 10.1007/978-3-030-64837-4_11.

[52] M. Barbosa, G. Barthe, X. Fan, B. Grégoire, S. Hung, J. Katz, P. Strub, X. Wu, and L. Zhou, "EasyPQC: Verifying post-quantum cryptography," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021, pp. 2564–2586. DOI: 10.1145/3460120.3484567.

[53] L. Zhou, G. Barthe, P.-Y. Strub, J. Liu, and M. Ying, "CoqQ: Foundational verification of quantum programs," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 833–865, 2023.

[54] Q. Huang, L. Zhou, W. Fang, M. Zhao, and M. Ying, "Efficient formal verification of quantum error correcting programs," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1068–1093, 2025. DOI: 10.1145/3729293.

[55] J. Paykin, R. Rand, and S. Zdancewic, "Qwire: A core language for quantum circuits," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 846–858, 2017. DOI: 10.1145/3009837.3009894.

[56] R. Rand, J. Paykin, and S. Zdancewic, "QWIRE practice: Formal verification of quantum circuits in Coq," *Electronic Proceedings in Theoretical Computer Science*, vol. 266, pp. 119–132, 2018. DOI: 10.4204/eptcs.266.8.

[57] R. Rand, J. Paykin, D.-H. Lee, and S. Zdancewic, "ReQWIRE: Reasoning about reversible quantum circuits," *Electronic Proceedings in Theoretical Computer Science*, vol. 287, pp. 299–312, 2019. DOI: 10.4204/eptcs.287.17.

[58] K. Hietala, R. Rand, S. Hung, L. Li, and M. Hicks, "Proving quantum programs correct," in *12th International Conference on Interactive Theorem Proving*, vol. 193, 2021, 21:1–21:19. DOI: 10.4230/LIPICS.ITP.2021.21.

[59] C. Chareton, S. Bardin, F. Bobot, V. Perrelle, and B. Valiron, "An automated deductive verification framework for circuit-building quantum programs," in *European Symposium on Programming*, 2021, pp. 148–177. DOI: 10.1007/978-3-030-72019-3_6.

[60] A. Bordg, H. Lachnitt, and Y. He, "Certified quantum computation in Isabelle/HOL," *Journal of Automated Reasoning*, vol. 65, no. 5, pp. 691–709, 2020. DOI: 10.1007/s10817-020-09584-7.

[61] K. Hietala, R. Rand, S. Hung, X. Wu, and M. Hicks, "A verified optimizer for quantum circuits," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021. DOI: 10.1145/3434318.

[62] L. Li, F. Voichick, K. Hietala, Y. Peng, X. Wu, and M. Hicks, "Verified compilation of quantum oracles," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 589–615, 2022. DOI: 10.1145/3563309.

[63] R. Tao, Y. Shi, J. Yao, X. Li, A. Javadi-Abhari, A. W. Cross, F. T. Chong, and R. Gu, "Giallar: Push-button verification for the Qiskit quantum compiler," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 641–656. DOI: 10.1145/3519939.3523431.

[64] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Checking equivalence of quantum circuits and states," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, IEEE, 2007, pp. 69–74. DOI: 10.5555/1326073.1326089.

[65] L. Burgholzer and R. Wille, "Advanced equivalence checking for quantum circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 9, pp. 1810–1824, 2020. DOI: 10.1109/TCAD.2020.3032630.

[66] L. Burgholzer and R. Wille, "Improved DD-based equivalence checking of quantum circuits," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2020, pp. 127–132. DOI: 10.1109/ASP-DAC47756.2020.9045153.

[67] X. Hong, Y. Feng, S. Li, and M. Ying, "Equivalence checking of dynamic quantum circuits," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–8. DOI: 10.1145/3508352.3549479.

[68] X. Hong, W. Huang, W. Chien, Y. Feng, M. Hsieh, S. Li, and M. Ying, *Equivalence checking of parameterised quantum circuits*, 2024. arXiv: 2404.18456 [quant-ph].

[69] R. Wille, D. Große, D. M. Miller, and R. Drechsler, "Equivalence checking of reversible circuits," in *2009 39th International Symposium on Multiple-Valued Logic*, 2009, pp. 324–330. DOI: 10.1109/ISMVL.2009.19.

[70] S. Yamashita and I. L. Markov, "Fast equivalence-checking for quantum circuits," in *2010 IEEE/ACM International Symposium on Nanoscale Architectures*, 2010, pp. 23–28. DOI: 10.1109/NANOARCH.2010.5510932.

[71] R. Wille, N. Przigoda, and R. Drechsler, "A compact and efficient SAT encoding for quantum circuits," in *2013 Africon*, 2013, pp. 1–6. DOI: 10.1109/AFRCON.2013.6757630.

[72] T. Peham, L. Burgholzer, and R. Wille, "Equivalence checking of quantum circuits with the ZX-calculus," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 12, no. 3, pp. 662–675, 2022. DOI: 10.1109/JETCAS.2022.3202204.

[73] A. Lehmann, B. Caldwell, B. Shah, and R. Rand, *VyZX: Formal verification of a graphical quantum language*, 2024. arXiv: 2311.11571 [cs.PL].

[74] L. Burgholzer and R. Wille, "The power of simulation for equivalence checking in quantum computing," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218563.

[75] C. M. Do and K. Ogata, "Equivalence checking of quantum circuits based on Dirac notation in Maude," in *International Workshop on Rewriting Logic and its Applications*, 2024, pp. 84–103. DOI: 10.1007/978-3-031-65941-6_5.

[76] M. Ying, N. Yu, and Y. Feng, *Defining quantum control flow*, 2012. arXiv: 1209.4379 [quant-ph].

[77] A. Sabry, B. Valiron, and J. K. Vizzotto, "From symmetric pattern-matching to quantum control," in *Foundations of Software Science and Computation Structures: 21st International Conference, FOSSACS 2018*, 2018, pp. 348–364. DOI: 10.1007/978-3-319-89366-2_19.

[78] C. Yuan, A. Villanyi, and M. Carbin, "Quantum control machine: The limits of control flow in quantum programming," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 1–28, 2024. DOI: 10.1145/3649811.

[79] J. Gambetta, "Quantum-centric supercomputing: The next wave of computing," *IBM Research Blog*, 2022.

[80] J. Gambetta, "Expanding the ibm quantum roadmap to anticipate the future of quantum-centric supercomputing," *IBM Research Blog*, 2022.

[81] Y. Alexeev, M. Amsler, M. A. Barroca, S. Bassini, T. Battelle, D. Camps, D. Casanova, Y. J. Choi, F. T. Chong, C. Chung, *et al.*, "Quantum-centric supercomputing for materials science: A perspective on challenges and future directions," *Future Generation Computer Systems*, vol. 160, pp. 666–710, 2024. DOI: 10.1016/j.future.2024.04.060.

[82] R. Mandelbaum, A. D. Córcoles, and J. Gambetta, "Ibm's big bet on the quantum-centric supercomputer: Recent advances point the way to useful classical-quantum hybrids," *IEEE Spectrum*, vol. 61, no. 9, pp. 24–33, 2024. DOI: 10.1109/MSPEC.2024.10669253.

[83] V. R. Pascuzzi and A. D. Córcoles, *Quantum-centric supercomputing for physics research*, 2024. arXiv: 2408.11741 [quant-ph].

[84] H. Corrigan-Gibbs, D. J. Wu, and D. Boneh, "Quantum operating systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 76–81. DOI: 10.1145/3102980.3102993.

[85] R. Honan, T. W. Lewis, S. Anderson, and J. Cooke, "A quantum computer operating system," in *Algorithms and Architectures for Parallel Processing: 20th International Conference, ICA3PP 2020*, 2020, pp. 415–431. DOI: 10.1007/978-3-030-60239-0_28.

[86] W. Kong, J. Wang, Y. Han, Y. Wu, Y. Zhang, M. Dou, Y. Fang, and G. Guo, *Origin Pilot: A quantum operating system for effecient usage of quantum resources*, 2021. arXiv: 2105.10730 [quant-ph].

[87] E. Giortamis, F. Romão, N. Tornow, and P. Bhatotia, *QOS: A quantum operating system*, 2024. arXiv: 2406.19120 [quant-ph].

[88] C. Delle Donne, M. Iuliano, B. van der Vecht, G. M. Ferreira, H. Jirovská, T. J. W. van der Steenhoven, A. Dahlberg, M. Skrzypczyk, D. Fioretto, M. Teller, *et al.*, "An operating system for executing applications on quantum network nodes," *Nature*, vol. 639, no. 8054, pp. 321–328, 2025. DOI: 10.1038/s41586-025-08704-w.

[89] S. Bravyi, G. Smith, and J. A. Smolin, "Trading classical and quantum computational resources," *Physical Review X*, vol. 6, no. 2, p. 021 043, 2016. DOI: 10.1103/PhysRevX.6.021043.

[90] T. Peng, A. W. Harrow, M. Ozols, and X. Wu, "Simulating large quantum circuits on a small quantum computer," *Physical Review Letters*, vol. 125, no. 15, p. 150 504, 2020. DOI: 10.1103/PhysRevLett.125.150504.

[91] K. Mitarai and K. Fujii, "Constructing a virtual two-qubit gate by sampling single-qubit operations," *New Journal of Physics*, vol. 23, no. 2, p. 023 021, 2021. DOI: 10.1088/1367-2630/abd7bc.

[92] W. Tang, T. Tomesh, M. Suchara, J. Larson, and M. Martonosi, "CutQC: Using small quantum computers for large quantum circuit evaluations," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 473–486. DOI: 10.1145/3445814.3446758.

[93] A. Eddins, M. Motta, T. P. Gujarati, S. Bravyi, A. Mezzacapo, C. Hadfield, and S. Sheldon, "Doubling the size of quantum simulators by entanglement forging," *PRX Quantum*, vol. 3, no. 1, p. 010 309, 2022. DOI: 10.1103/PRXQuantum.3.010309.

[94] P. Li, J. Liu, H. P. Patil, P. Hovland, and H. Zhou, "Enhancing virtual distillation with circuit cutting for quantum error mitigation," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*, IEEE, 2023, pp. 94–101. DOI: 10.1109/ICCD58817.2023.00024.

[95] C. Piveteau and D. Sutter, "Circuit knitting with classical communication," *IEEE Transactions on Information Theory*, vol. 70, no. 4, pp. 2734–2745, 2024. DOI: 10.1109/TIT.2023.3310797.

[96] Z. Li, M. Guo, M. Barad, W. Tang, E. Z. Zhang, and Y. Huang, *A case for quantum circuit cutting for NISQ applications: Impact of topology, determinism, and sparsity*, 2024. arXiv: 2412.17929 [quant-ph].

[97] A. M. Brańczyk, A. Carrera Vazquez, D. J. Egger, *et al.*, *Qiskit addon: Circuit cutting*, https://github.com/Qiskit/qiskit-addon-cutting, 2024. DOI: 10.5281/zenodo.7987997.

[98] C. Ying, B. Cheng, Y. Zhao, *et al.*, "Experimental simulation of larger quantum circuits with fewer superconducting qubits," *Physical Review Letters*, vol. 130, no. 11, p. 110 601, 2023. DOI: 10.1103/PhysRevLett.130.110601.

[99] A. P. Singh, K. Mitarai, Y. Suzuki, K. Heya, Y. Tabuchi, K. Fujii, and Y. Nakamura, "Experimental demonstration of a high-fidelity virtual two-qubit gate," *Physical Review Research*, vol. 6, no. 1, p. 013 235, 2024. DOI: 10.1103/PhysRevResearch.6.013235.

[100] G. S. Ravi, K. N. Smith, P. Murali, and F. T. Chong, "Adaptive job and resource management for the growing quantum cloud," in *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2021, pp. 301–312. DOI: 10.1109/QCE52317.2021.00047.

[101] L. Liu and X. Dou, "QuCloud+: A holistic qubit mapping scheme for single/multi-programming on 2D/3D NISQ quantum computers," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 1, pp. 1–27, 2024. DOI: 10.1145/3631525.

[102] A. Orenstein and V. Chaudhary, "QGroup: Parallel quantum job scheduling using dynamic programming," in *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1, 2024, pp. 990–999. DOI: 10.1109/QCE60285.2024.00118.

[103] J. Li, Y. Song, Y. Liu, J. Pan, L. Yang, T. Humble, and W. Jiang, *QuSplit: Achieving both high fidelity and throughput via job splitting on noisy quantum computers*, 2025. arXiv: 2501.12492 [quant-ph].

[104] P. Das, S. S. Tannu, P. J. Nair, and M. Qureshi, "A case for multi-programming quantum computers," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 291–303. DOI: 10.1145/3352460.3358287.

[105] L. Liu and X. Dou, "QuCloud: A new qubit mapping mechanism for multi-programming quantum computing in cloud environment," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 167–178. DOI: 10.1109/HPCA51647.2021.00024.

[106] Y. Ohkura, T. Satoh, and R. Van Meter, "Simultaneous execution of quantum circuits on current and near-future NISQ systems," *IEEE Transactions on Quantum Engineering*, vol. 3, pp. 1–10, 2022. DOI: 10.1109/TQE.2022.3164716.

[107] S. Resch, A. Gutierrez, J. S. Huh, S. Bharadwaj, Y. Eckert, G. Loh, M. Oskin, and S. Tannu, *Accelerating variational quantum algorithms using circuit concurrency*, 2021. arXiv: 2109.01714 [cs.ET].

[108] S. Niu and A. Todri-Sanial, "Enabling multi-programming mechanism for quantum computing in the NISQ era," *Quantum*, vol. 7, p. 925, 2023. DOI: 10.22331/q-2023-02-16-925.

[109] S. Khadirsharbiyani, M. Sadeghi, M. E. Zarch, J. Kotra, and M. T. Kandemir, "TRIM: crossTalk-awaRe qubIt Mapping for multiprogrammed quantum systems," in *2023 IEEE International Conference on Quantum Software (QSW)*, 2023, pp. 138–148. DOI: 10.1109/QSW59989.2023.00025.

[110] R. Tao, H. Zhu, J. Nieh, J. Yao, and R. Gu, "Quantum virtual machines," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025, pp. 411–428.

[111] J. Heckey, S. Patil, A. JavadiAbhari, A. Holmes, D. Kudrow, K. R. Brown, D. Franklin, F. T. Chong, and M. Martonosi, "Compiler management of communication and parallelism for quantum computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 445–456. DOI: 10.1145/2694344.2694357.

[112] G. Meuli, M. Soeken, M. Roetteler, N. Bjorner, and G. De Micheli, "Reversible pebbling game for quantum memory management," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 288–291. DOI: 10.23919/DATE.2019.8715092.

[113] W. Dai, T. Peng, and M. Z. Win, "Quantum queuing delay," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 3, pp. 605–618, 2020. DOI: 10.1109/JSAC.2020.2969000.

[114] C. Liu, M. Wang, S. A. Stein, Y. Ding, and A. Li, *Quantum memory: A missing piece in quantum computing units*, 2023. arXiv: 2309.14432 [quant-ph].

[115] F. Hua, Y. Jin, Y. Chen, S. Vittal, K. Krsulich, L. S. Bishop, J. Lapeyre, A. Javadi-Abhari, and E. Z. Zhang, "CaQR: A compiler-assisted approach for qubit reuse through dynamic circuit," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 3, 2023, pp. 59–71. DOI: 10.1145/3582016.3582030.

[116] M. Ying, Y. Feng, and N. Yu, "Quantum information-flow security: Noninterference and access control," in *2013 IEEE 26th Computer Security Foundations Symposium*, 2013, pp. 130–144. DOI: 10.1109/CSF.2013.16.

[117] A. Carrera Vazquez, C. Tornow, D. Riste, S. Woerner, M. Takita, and D. J. Egger, "Combining quantum processors with real-time classical communication," *Nature*, vol. 636, no. 8041, pp. 75–79, 2024. DOI: 10.1038/s41586-024-08178-2.

[118] D. Main, P. Drmota, D. P. Nadlinger, E. M. Ainley, A. Agrawal, B. C. Nichol, R. Srinivas, G. Araneda, and D. M. Lucas, "Distributed quantum computing across an optical network link," *Nature*, pp. 1–6, 2025. DOI: 10.1038/s41586-024-08404-x.

[119] S. Tani, H. Kobayashi, and K. Matsumoto, "Exact quantum algorithms for the leader election problem," *ACM Transactions on Computation Theory (TOCT)*, vol. 4, no. 1, pp. 1–24, 2012. DOI: 10.1145/2141938.2141939.

[120] D. Aharonov, M. Ganz, and L. Magnin, *Dining philosophers, leader election and ring size problems, in the quantum setting*, 2017. arXiv: 1707.01187 [quant-ph].

[121] M. Ben-Or and A. Hassidim, "Fast quantum byzantine agreement," in *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, 2005, pp. 481–485. DOI: 10.1145/1060590.1060662.

[122] L. Li, X. Sun, and J. Zhu, "Quantum Byzantine agreement against full-information adversary," in *38th International Symposium on Distributed Computing (DISC 2024)*, 2024, pp. 32–1. DOI: 10.4230/LIPIcs.DISC.2024.32.

[123] F. Le Gall and F. Magniez, "Sublinear-time quantum computation of the diameter in congest networks," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, 2018, pp. 337–346. DOI: 10.1145/3212734.3212744.

[124] K. Censor-Hillel, O. Fischer, F. Le Gall, D. Leitersdorf, and R. Oshman, "Quantum distributed algorithms for detection of cliques," in *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*, 2022, pp. 35–1. DOI: 10.4230/LIPIcs.ITCS.2022.35.

[125] T. Izumi, F. Le Gall, and F. Magniez, "Quantum distributed algorithm for triangle finding in the CONGEST model," in *37th International Symposium on Theoretical Aspects of Computer Science*, ser. STAC'20, 2020. DOI: 10.4230/LIPIcs.STACS.2020.23.

[126] T. Izumi and F. Le Gall, "Quantum distributed algorithm for the all-pairs shortest path problem in the CONGEST-CLIQUE model," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 84–93. DOI: 10.1145/3293611.3331628.

[127] P. Jorrand and M. Lalire, "Toward a quantum process algebra," in *Proceedings of the 1st Conference on Computing Frontiers*, 2004, pp. 111–119. DOI: 10.1145/977091.977108.

[128] S. J. Gay and R. Nagarajan, "Communicating quantum processes," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, 2005, pp. 145–157. DOI: 10.1145/1040305.1040318.

[129] Y. Feng, R. Duan, and M. Ying, "Bisimulation for quantum processes," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 34, no. 4, pp. 1–43, 2012. DOI: 10.1145/2400676.2400680.

[130] C. Gavoille, A. Kosowski, and M. Markiewicz, "What can be observed locally? Round-based models for quantum distributed computing," in *International Symposium on Distributed Computing*, 2009, pp. 243–257. DOI: 10.1007/978-3-642-04355-0_26.

[131] H. Arfaoui and P. Fraigniaud, "What can be computed without communications?" *ACM SIGACT News*, vol. 45, no. 3, pp. 82–104, 2014. DOI: 10.1145/2670418.2670440.

[132] F. Le Gall, H. Nishimura, and A. Rosmanis, "Quantum advantage for the LOCAL model in distributed computing," in *36th International Symposium on Theoretical Aspects of Computer Science*, 2019. DOI: 10.4230/LIPIcs.STACS.2019.49.

[133] X. Coiteux-Roy, F. d'Amore, R. Gajjala, F. Kuhn, F. Le Gall, H. Lievonen, A. Modanese, M. Renou, G. Schmid, and J. Suomela, "No distributed quantum advantage for approximate graph coloring," in *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, 2024, pp. 1901–1910. DOI: 10.1145/3618260.3649679.

[134] A. Akbari, X. Coiteux-Roy, F. d'Amore, F. Le Gall, H. Lievonen, D. Melnyk, A. Modanese, S. Pai, M. Renou, V. Rozhoň, *et al.*, "Online locality meets distributed quantum computing," in *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, 2025, pp. 1295–1306. DOI: 10.1145/3717823.3718211.

[135] A. Balliu, C. Coupette, A. Cruciani, F. d'Amore, M. Equi, H. Lievonen, A. Modanese, D. Olivetti, and J. Suomela, *New limits on distributed quantum advantage: Dequantizing linear programs*, 2025. arXiv: 2506.07574 [cs.DC].

[136] M. Elkin, H. Klauck, D. Nanongkai, and G. Pandurangan, "Can quantum communication speed up distributed computation?" In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, 2014, pp. 166–175. DOI: 10.1145/2611462.2611488.

[137] F. Magniez and A. Nayak, "Quantum distributed complexity of set disjointness on a line," *ACM Transactions on Computation Theory (TOCT)*, vol. 14, no. 1, pp. 1–22, 2022. DOI: 10.1145/3512751.

[138] J. van Apeldoorn and T. de Vos, "A framework for distributed quantum queries in the CONGEST model," in *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, 2022, pp. 109–119. DOI: 10.1145/3519270.3538413.

[139] C. Wang, X. Wu, and P. Yao, "Complexity of eccentricities and all-pairs shortest paths in the quantum CONGEST model," *SPIN*, vol. 11, no. 03, 2021. DOI: 10.1142/s2010324721400075.

[140] X. Wu and P. Yao, "Quantum complexity of weighted diameter and radius in CONGEST networks," in *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, 2022, pp. 120–130. DOI: 10.1145/3519270.3538441.

[141] P. Fraigniaud, M. Luce, F. Magniez, and I. Todinca, "Even-cycle detection in the randomized and quantum CONGEST model," in *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, 2024, pp. 209–219. DOI: 10.1145/3662158.3662767.

[142] Y. Feng, S. Li, and M. Ying, "Verification of distributed quantum programs," *ACM Transactions on Computational Logic (TOCL)*, vol. 23, no. 3, pp. 1–40, 2022. DOI: 10.1145/3517145.

[143] T. Häner, D. S. Steiger, T. Hoefler, and M. Troyer, "Distributed quantum computing with QMPI," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13. DOI: 10.1145/3458817.3476172.

[144] A. Wu, H. Zhang, G. Li, A. Shabani, Y. Xie, and Y. Ding, "AutoComm: A framework for enabling efficient communication in distributed quantum programs," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1027–1041. DOI: 10.1109/MICRO56248.2022.00074.

[145] H. J. Briegel, W. Dür, J. I. Cirac, and P. Zoller, "Quantum repeaters: The role of imperfect local operations in quantum communication," *Physical Review Letters*, vol. 81, no. 26, p. 5932, 1998. DOI: 10.1103/PhysRevLett.81.5932.

[146] W. Dür, H. J. Briegel, J. I. Cirac, and P. Zoller, "Quantum repeaters based on entanglement purification," *Physical Review A*, vol. 59, no. 1, p. 169, 1999. DOI: 10.1103/PhysRevA.59.169.

[147] L.-M. Duan, M. D. Lukin, J. I. Cirac, and P. Zoller, "Long-distance quantum communication with atomic ensembles and linear optics," *Nature*, vol. 414, no. 6862, pp. 413–418, 2001. DOI: 10.1038/35106500.

[148] L. Jiang, J. M. Taylor, K. Nemoto, W. J. Munro, R. Van Meter, and M. D. Lukin, "Quantum repeater with encoding," *Physical Review A*, vol. 79, no. 3, p. 032 325, 2009. DOI: 10.1103/PhysRevA.79.032325.

[149] N. Sangouard, C. Simon, H. de Riedmatten, and N. Gisin, "Quantum repeaters based on atomic ensembles and linear optics," *Reviews of Modern Physics*, vol. 83, no. 1, pp. 33–80, 2011. DOI: 10.1103/RevModPhys.83.33.

[150] S. Muralidharan, J. Kim, N. Lütkenhaus, M. D. Lukin, and L. Jiang, "Ultrafast and fault-tolerant quantum communication across long distances," *Physical Review Letters*, vol. 112, no. 25, p. 250 501, 2014. DOI: 10.1103/PhysRevLett.112.250501.

[151] B. Bartlett, *A distributed simulation framework for quantum networks and channels*, 2018. arXiv: 1808.07047 [quant-ph].

[152] X. Wu, A. Kolar, J. Chung, D. Jin, T. Zhong, R. Kettimuthu, and M. Suchara, "SeQUeNCe: A customizable discrete-event simulator of quantum networks," *Quantum Science and Technology*, vol. 6, no. 4, p. 045 027, 2021. DOI: 10.1088/2058-9565/ac22f6.

[153] T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J. de Oliveira Filho, M. Papendrecht, J. Rabbie, F. Rozpędek, M. Skrzypczyk, *et al.*, "Netsquid, a network simulator for quantum information using discrete events," *Communications Physics*, vol. 4, no. 1, p. 164, 2021. DOI: 10.1038/s42005-021-00647-8.

[154] S. DiAdamo, J. Nötzel, B. Zanger, and M. M. Beşe, "QuNetSim: A software framework for quantum networks," *IEEE Transactions on Quantum Engineering*, vol. 2, pp. 1–12, 2021. DOI: 10.1109/TQE.2021.3092395.

[155] R. Satoh, M. Hajdušek, N. Benchasattabuse, *et al.*, "QuISP: A quantum internet simulation package," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022, pp. 353–364. DOI: 10.1109/QCE53715.2022.00056.

[156] R. Zhou, X. Lai, Y. Gan, K. Obraczka, S. Du, and C. Qian, "A simulator of atom-atom entanglement with atomic ensembles and quantum optics," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1, 2023, pp. 1271–1277. DOI: 10.1109/QCE57702.2023.00143.

[157] R. Zhou, Y. Gan, L. Shen, Y. Liu, M. Messel, and C. Qian, "QuCloudSim: A customizable discrete event simulator for quantum cloud computing environment," in *2025 International Conference on Quantum Communications, Networking, and Computing (QCNC)*, 2025, pp. 58–65. DOI: 10.1109/QCNC64685.2025.00018.

[158] H. Lin, R. Deng, C. Z. Yao, Z. Ji, and M. Ying, "Control flow adaption: An efficient simulation method for noisy quantum networks," in *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*, 2025, pp. 1–10. DOI: 10.1109/INFOCOM55648.2025.11044741.

[159] C. Chudzicki and F. W. Strauch, "Parallel state transfer and efficient quantum routing on quantum networks," *Physical Review Letters*, vol. 105, no. 26, p. 260 501, 2010. DOI: 10.1103/PhysRevLett.105.260501.

[160] P. J. Pemberton-Ross and A. Kay, "Perfect quantum routing in regular spin networks," *Physical Review Letters*, vol. 106, no. 2, p. 020 503, 2011. DOI: 10.1103/PhysRevLett.106.020503.

[161] R. Van Meter, T. Satoh, T. D. Ladd, W. J. Munro, and K. Nemoto, "Path selection for quantum repeater networks," *Networking Science*, vol. 3, no. 1–4, pp. 82–95, 2013. DOI: 10.1007/s13119-013-0026-2.

[162] L. Zhou, L.-P. Yang, Y. Li, and C. P. Sun, "Quantum routing of single photons with a cyclic three-level system," *Physical Review Letters*, vol. 111, no. 10, p. 103 604, 2013. DOI: 10.1103/PhysRevLett.111.103604.

[163] X. Zhan, H. Qin, Z.-h. Bian, J. Li, and P. Xue, "Perfect state transfer and efficient quantum routing: A discrete-time quantum-walk approach," *Physical Review A*, vol. 90, no. 1, p. 012 331, 2014. DOI: 10.1103/PhysRevA.90.012331.

[164] E. Schoute, L. Mancinska, T. Islam, I. Kerenidis, and S. Wehner, *Shortcuts to quantum network routing*, 2016. arXiv: 1610.05238 [cs.NI].

[165] M. Caleffi, "Optimal routing for quantum networks," *IEEE Access*, vol. 5, pp. 22 299–22 312, 2017. DOI: 10.1109/ACCESS.2017.2763325.

[166] M. Pant, H. Krovi, D. Towsley, L. Tassiulas, L. Jiang, P. Basu, D. Englund, and S. Guha, "Routing entanglement in the quantum internet," *npj Quantum Information*, vol. 5, no. 1, p. 25, 2019. DOI: 10.1038/s41534-019-0139-x.

[167] S. Das, S. Khatri, and J. P. Dowling, "Robust quantum network architectures and topologies for entanglement distribution," *Physical Review A*, vol. 97, no. 1, p. 012 335, 2018. DOI: 10.1103/PhysRevA.97.012335.

[168] K. Chakraborty, F. Rozpedek, A. Dahlberg, and S. Wehner, *Distributed routing in a quantum internet*, 2019. arXiv: 1907.11630.

[169] S. Shi and C. Qian, "Concurrent entanglement routing for quantum networks: Model and designs," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 62–75. DOI: 10.1145/3387514.3405853.

[170] S. Zhang, S. Shi, C. Qian, and K. L. Yeung, "Fragmentation-aware entanglement routing for quantum networks," *Journal of Lightwave Technology*, vol. 39, no. 14, pp. 4584–4591, 2021.

[171] Y. Gan, X. Zhang, R. Zhou, Y. Liu, and C. Qian, "A routing framework for quantum entanglements with heterogeneous duration," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, IEEE, vol. 1, 2023, pp. 1132–1142. DOI: 10.1109/QCE57702.2023.00128.

[172] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, 1961. DOI: 10.1145/366622.366644.

[173] C. A. R. Hoare, "Recursive data structures," *International Journal of Computer & Information Sciences*, vol. 4, no. 2, pp. 105–132, 1975. DOI: 10.1007/BF00976239.

[174] V. V. Shende, S. S. Bullock, and I. L. Markov, "Synthesis of quantum logic circuits," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, 2005, pp. 272–275. DOI: 10.1109/TCAD.2005.855930.

[175] B. W. Lampson, "Protection," *ACM SIGOPS Operating Systems Review*, vol. 8, no. 1, pp. 18–24, 1974. DOI: 10.1145/775265.775268.

[176] N. D. Mermin, "Extreme quantum entanglement in a superposition of macroscopically distinct states," *Physical Review Letters*, vol. 65, no. 15, pp. 1838–1840, 1990. DOI: 10.1103/PhysRevLett.65.1838.

[177] L. Lamport, "Interprocess communication," SRI International, Tech. Rep., 1985.

[178] L. Lamport, "On interprocess communication: Part I: Basic formalism," *Distributed computing*, vol. 1, pp. 77–85, 1986. DOI: 10.1007/BF01786227.

[179] L. Lamport, "On interprocess communication: Part II: Algorithms," *Distributed Computing*, vol. 1, pp. 86–101, 1986. DOI: 10.1007/BF01786228.

[180] Z. Zhang and M. Ying, "Quantum register machine: Efficient implementation of quantum recursive programs," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 822–847, 2025. DOI: 10.1145/3729283.

[181] Z. Zhang and M. Ying, *Access control threatened by quantum entanglement*, 2025. arXiv: 2507.02622 [quant-ph].

[182] Z. Zhang and M. Ying, *Atomicity in distributed quantum computing*, 2024. arXiv: 2404.18592 [quant-ph].

[183] Q. Wang, Z. Zhang, K. Chen, J. Guan, W. Fang, J. Liu, and M. Ying, "Quantum algorithm for fidelity estimation," *IEEE Transactions on Information Theory*, vol. 69, no. 1, pp. 273–282, 2023. DOI: 10.1109/TIT.2022.3203985.

[184] Z. Zhang, Q. Wang, and M. Ying, "Parallel quantum algorithm for hamiltonian simulation," *Quantum*, vol. 8, p. 1228, 2024. DOI: 10.22331/q-2024-01-15-1228.

[185] Q. Wang and Z. Zhang, "Quantum lower bounds by sample-to-query lifting," *SIAM Journal on Computing*, vol. 54, no. 5, pp. 1294–1334, 2025. DOI: 10.1137/24M1638616.

[186] Q. Wang and Z. Zhang, "Fast quantum algorithms for trace distance estimation," *IEEE Transactions on Information Theory*, vol. 70, no. 4, pp. 2720–2733, 2024. DOI: 10.1109/TIT.2023.3321121.

[187] Q. Wang, J. Guan, J. Liu, Z. Zhang, and M. Ying, "New quantum algorithms for computing quantum entropies and distances," *IEEE Transactions on Information Theory*, vol. 70, no. 8, pp. 5653–5680, 2024. DOI: 10.1109/TIT.2024.3399014.

[188] Q. Wang and Z. Zhang, "Time-efficient quantum entropy estimator via samplizer," in *32nd Annual European Symposium on Algorithms (ESA 2024)*, vol. 308, 2024, 101:1–101:15. DOI: 10.4230/LIPICS.ESA.2024.101.

[189] N. Liu, Q. Wang, M. M. Wilde, and Z. Zhang, "Quantum algorithms for matrix geometric means," *npj Quantum Information*, vol. 11, no. 1, p. 101, 2025. DOI: 10.1038/s41534-025-00973-7.

[190] Q. Wang and Z. Zhang, "Tight quantum depth lower bound for solving systems of linear equations," *Physical Review A*, vol. 110, no. 1, p. 012 422, 2024. DOI: 10.1103/PhysRevA.110.012422.

[191] Q. Wang and Z. Zhang, *Sample-optimal quantum estimators for pure-state trace distance and fidelity via samplizer*, 2024. arXiv: 2410.21201 [quant-ph].

[192] K. Chen, N. Yu, and Z. Zhang, *Tight bound for quantum unitary time-reversal*, 2025. arXiv: 2507.05736 [quant-ph].

[193] K. Chen, Q. Wang, Z. Yu, and Z. Zhang, *Simultaneous estimation of nonlinear functionals of a quantum state*, 2025. arXiv: 2505.16715 [quant-ph].

[194] K. Chen, Q. Wang, and Z. Zhang, *Local test for unitarily invariant properties of bipartite quantum states*, 2024. arXiv: 2404.04599 [quant-ph].

[195] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[196] J. von Neumann, *Mathematische Grundlagen der Quantenmechanik*. Verlag von Julius Springer, 1932.

[197] P. Dirac, *The Principles of Quantum Mechanics*. Oxford University Press, 1930.

[198] J. von Neumann, "Wahrscheinlichkeitstheoretischer Aufbau der Quantenmechanik," *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, vol. 1927, pp. 245–272, 1927.

[199] L. Landau, "Das Dämpfungsproblem in der Wellenmechanik," *Zeitschrift für Physik*, vol. 45, pp. 430–441, 1927.

[200] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965. DOI: 10.2307/2003354.

[201] J. W. Backus, F. L. Bauer, J. Green, *et al.*, "Report on the algorithmic language ALGOL 60," *Communications of the ACM*, vol. 3, no. 5, pp. 299–311, 1960. DOI: 10.1145/367236.367262.

[202] J. W. Backus, F. L. Bauer, J. Green, *et al.*, "Revised report on the algorithmic language ALGOL 60," *Communications of the ACM*, vol. 6, no. 1, pp. 1–17, 1963. DOI: 10.1145/366193.366201.

[203] E. W. Dijkstra, "Recursive programming," *Numerische Mathematik*, vol. 2, no. 1, pp. 312–318, 1960. DOI: 10.1007/bf01386232.

[204] G. van den Hove, "On the origin of recursive procedures," *The Computer Journal*, vol. 58, no. 11, pp. 2892–2899, 2015. DOI: 10.1093/comjnl/bxu145.

[205] P. Selinger, "Towards a quantum programming language," *Mathematical Structures in Computer Science*, vol. 14, no. 4, pp. 527–586, 2004. DOI: 10.1017/S0960129504004256.

[206] Z. Xu, M. Ying, and B. Valiron, *Reasoning about recursive quantum programs*, 2021. arXiv: 2107.11679 [cs.LO].

[207] H. Deng, R. Tao, Y. Peng, and X. Wu, "A case for synthesis of recursive quantum unitary programs," *Proceedings of the ACM on Programming Languages*, vol. 8, pp. 1759–1788, 2024. DOI: 10.1145/3632901.

[208] C. Yuan and M. Carbin, "The T-complexity costs of error correction for control flow in quantum computation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 492–517, 2024. DOI: 10.1145/3656397.

[209] D. M. Greenberger, M. A. Horne, and A. Zeilinger, "Going beyond Bell's theorem," in *Bell's theorem, quantum theory and conceptions of the universe*, Springer, 1989, pp. 69–72. DOI: 10.1007/978-94-017-0849-4_10.

[210] R. H. Dicke, "Coherence in spontaneous radiation processes," *Physical Review*, vol. 93, no. 1, p. 99, 1954. DOI: 10.1103/PhysRev.93.99.

[211] D. W. Berry, A. M. Childs, R. Cleve, R. Kothari, and R. D. Somma, "Simulating Hamiltonian dynamics with a truncated Taylor series," *Physical Review Letters*, vol. 114, p. 090 502, 9 2015. DOI: 10.1103/PhysRevLett.114.090502.

[212] D. W. Berry, A. M. Childs, and R. Kothari, "Hamiltonian simulation with nearly optimal dependence on all parameters," in *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science*, ser. FOCS '15, 2015, pp. 792–809. DOI: 10.1109/FOCS.2015.54.

[213] R. Kothari, "Efficient algorithms in quantum query complexity," Ph.D. dissertation, University of Waterloo, 2014.

[214] S. Lloyd, M. Mohseni, and P. Rebentrost, "Quantum principal component analysis," *Nature physics*, vol. 10, no. 9, pp. 631–633, 2014. DOI: 10.1038/nphys3029.

[215] I. Kerenidis and A. Prakash, "Quantum recommendation systems," in *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*, vol. 67, 2017, 49:1–49:21. DOI: 10.4230/LIPIcs.ITCS.2017.49.

[216] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Physical Review Letters*, vol. 103, no. 15, p. 150 502, 2009. DOI: 10.1103/PhysRevLett.103.150502.

[217] A. M. Childs, R. Kothari, and R. D. Somma, "Quantum algorithm for systems of linear equations with exponentially improved dependence on precision," *SIAM Journal on Computing*, vol. 46, no. 6, pp. 1920–1950, 2017. DOI: 10.1137/16M1087072.

[218] V. Giovannetti, S. Lloyd, and L. Maccone, "Quantum random access memory," *Physical Review Letters*, vol. 100, no. 16, p. 160 501, 2008. DOI: 10.1103/PhysRevLett.100.160501.

[219] C. Bădescu and P. Panangaden, "Quantum alternation: Prospects and problems," in *Proceedings of the 12th International Workshop on Quantum Physics and Logic*, ser. EPTCS, vol. 195, 2015, pp. 33–42. DOI: 10.4204/EPTCS.195.3.

[220] M. Mints, F. Voichick, L. Lampropoulos, and R. Rand, "Compositional quantum control flow with efficient compilation in qunity," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, pp. 166–192, 2025. DOI: 10.1145/3763056.

[221] E. W. Dijkstra, "Programming considered as a human activity," in *Classics in software engineering*, 1979, pp. 1–9. [Online]. Available: https://www.cs.utexas.edu/~EWD/ewd01xx/EWD117.PDF.

[222] A. M. Childs and N. Wiebe, "Hamiltonian simulation using linear combinations of unitary operations," *Quantum Information & Computation*, vol. 12, no. 11–12, pp. 901–924, 2012. DOI: 10.26421/QIC12.11-12-1.

[223] R. Babbush, D. W. Berry, I. D. Kivlichan, A. Y. Wei, P. J. Love, and A. Aspuru-Guzik, "Exponentially more precise quantum simulation of fermions in second quantization," *New Journal of Physics*, vol. 18, p. 033 032, 2016. DOI: 10.1088/1367-2630/18/3/033032.

[224] R. Babbush, C. Gidney, D. W. Berry, N. Wiebe, J. McClean, A. Paler, A. Fowler, and H. Neven, "Encoding electronic spectra in quantum circuits with linear T complexity," *Physical Review X*, vol. 8, no. 4, p. 041 015, 2018. DOI: 10.1103/PhysRevX.8.041015.

[225] R. Babbush, N. Wiebe, J. McClean, J. McClain, H. Neven, and G. K. Chan, "Low-depth quantum simulation of materials," *Physical Review X*, vol. 8, p. 011 044, 1 2018.

[226] G. H. Low and N. Wiebe, *Hamiltonian simulation in the interaction picture*, 2019. arXiv: 1805.00675 [quant-ph].

[227] I. F. Araujo, D. K. Park, F. Petruccione, and A. J. da Silva, "A divide-and-conquer algorithm for quantum state preparation," *Scientific reports*, vol. 11, no. 1, p. 6329, 2021. DOI: 10.1038/s41598-021-85474-1.

[228] X.-M. Zhang, T. Li, and X. Yuan, "Quantum state preparation with optimal circuit depth: Implementations and applications," *Physical Review Letters*, vol. 129, no. 23, p. 230 504, 2022. DOI: 10.1103/PhysRevLett.129.230504.

[229] X.-M. Zhang and X. Yuan, "Circuit complexity of quantum access models for encoding classical data," *npj Quantum Information*, vol. 10, no. 1, p. 42, 2024. DOI: 10.1038/s41534-024-00835-8.

[230] G. H. Low, V. Kliuchnikov, and L. Schaeffer, "Trading T gates for dirty qubits in state preparation and unitary synthesis," *Quantum*, vol. 8, p. 1375, 2024. DOI: 10.22331/q-2024-06-17-1375.

[231] V. Giovannetti, S. Lloyd, and L. Maccone, "Architectures for a quantum random access memory," *Physical Review A*, vol. 78, no. 5, p. 052 310, 2008. DOI: 10.1103/PhysRevA.78.052310.

[232] C. T. Hann, C.-L. Zou, Y. Zhang, Y. Chu, R. J. Schoelkopf, S. M. Girvin, and L. Jiang, "Hardware-efficient quantum random access memory with hybrid quantum acoustic systems," *Physical Review Letters*, vol. 123, no. 25, p. 250 501, 2019. DOI: 10.1103/PhysRevLett.123.250501.

[233] C. T. Hann, G. Lee, S. M. Girvin, and L. Jiang, "Resilience of quantum random access memory to generic noise," *PRX Quantum*, vol. 2, no. 2, p. 020 311, 2021. DOI: 10.1103/PRXQuantum.2.020311.

[234] M. P. Frank, "Reversibility for efficient computing," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[235] C. J. Vieri, "Reversible computer engineering and architecture," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[236] H. B. Axelsen, R. Glück, and T. Yokoyama, "Reversible machine code and its abstract processor architecture," in *Computer Science–Theory and Applications: Second International Symposium on Computer Science in Russia (CSR 2007)*, 2007, pp. 56–69. DOI: 10.1007/978-3-540-74510-5_9.

[237] M. K. Thomsen, H. B. Axelsen, and R. Glück, "A reversible processor architecture and its reversible logic design," in *Reversible Computation: Third International Workshop, RC 2011*, 2012, pp. 30–42. DOI: 10.1007/978-3-642-29517-1_3.

[238] D. Deutsch, "Quantum theory, the Church–Turing principle and the universal quantum computer," *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 400, no. 1818, pp. 97–117, 1985. DOI: 10.1098/rspa.1985.0070.

[239] J. M. Myers, "Can a universal quantum computer be fully quantum?" *Physical Review Letters*, vol. 78, no. 9, p. 1823, 1997. DOI: 10.1103/PhysRevLett.78.1823.

[240] E. Bernstein and U. Vazirani, "Quantum complexity theory," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 11–20. DOI: 10.1145/167088.167097.

[241] M. Ozawa, "Quantum nondemolition monitoring of universal quantum computers," *Physical Review Letters*, vol. 80, no. 3, p. 631, 1998. DOI: 10.1103/PhysRevLett.80.631.

[242] N. Linden and S. Popescu, *The halting problem for quantum computers*, 1998. arXiv: quant-ph/9806054 [quant-ph].

[243] M. Ozawa, "Quantum Turing machines: Local transition, preparation, measurement, and halting," in *Quantum Communication, Computing, and Measurement 2*, 1998, pp. 241–248. DOI: 10.1007/0-306-47097-7_32.

[244] Y. Shi, "Remarks on universal quantum computer," *Physics Letters A*, vol. 293, no. 5-6, pp. 277–282, 2002. DOI: 10.1016/S0375-9601(02)00015-4.

[245] T. Miyadera and M. Ohya, "On halting process of quantum turing machine," *Open Systems & Information Dynamics*, vol. 12, no. 3, pp. 261–264, 2005. DOI: 10.1007/s11080-005-0923-2.

[246] Q. Wang and M. Ying, "Quantum random access stored-program machines," *Journal of Computer and System Sciences*, vol. 131, pp. 13–63, 2023. DOI: 10.1016/j.jcss.2022.08.002.

[247] H. B. Axelsen, "Clean translation of an imperative reversible programming language," in *International Conference on Compiler Construction*, 2011, pp. 144–163. DOI: 10.1007/978-3-642-19861-8_9.

[248] S. Jaques and A. G. Rattew, *Qram: A survey and critique*, 2023. arXiv: 2305.10310 [quant-ph].

[249] A. Ambainis, "Quantum walk algorithm for element distinctness," *SIAM Journal on Computing*, vol. 37, no. 1, pp. 210–239, 2007. DOI: 10.1137/S0097539705447311.

[250] D. J. Bernstein, S. Jeffery, T. Lange, and A. Meurer, "Quantum algorithms for the subset-sum problem," in *Post-Quantum Cryptography: 5th International Workshop, PQCrypto 2013*, 2013, pp. 16–33. DOI: 10.1007/978-3-642-38616-9_2.

[251] S. Aaronson, N.-H. Chia, H. Lin, C. Wang, and R. Zhang, "On the quantum complexity of closest pair and related problems," in *35th Computational Complexity Conference (CCC 2020)*, vol. 169, 2020, 16:1–16:43. DOI: 10.4230/LIPIcs.CCC.2020.16.

[252] S. Xu, A. Lu, and Y. Ding, "Fat-tree QRAM: A high-bandwidth shared quantum random access memory for parallel queries," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '25, vol. 2, 2025, pp. 390–406. DOI: 10.1145/3676641.3716256.

[253] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM journal of research and development*, vol. 5, no. 3, pp. 183–191, 1961. DOI: 10.1147/rd.53.0183.

[254] C. H. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973. DOI: 10.1147/rd.176.0525.

[255] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, *Open quantum assembly language*, 2017. arXiv: 1707.03429 [quant-ph].

[256] R. S. Smith, M. J. Curtis, and W. J. Zeng, *A practical quantum instruction set architecture*, 2017. arXiv: 1608.03355 [quant-ph].

[257] X. Fu, L. Riesebos, M. A. Rol, *et al.*, "eQASM: An executable quantum instruction set architecture," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 224–237. DOI: 10.1109/HPCA.2019.00040.

[258] C. Lutz and H. Derby, "Janus: A time-reversible language," *Letter to Rolf Landauer*, vol. 2, 1986.

[259] T. Yokoyama and R. Glück, "A reversible programming language and its invertible self-interpreter," in *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2007, pp. 144–153. DOI: 10.1145/1244381.1244404.

[260] T. Yokoyama, H. B. Axelsen, and R. Glück, "Principles of a reversible programming language," in *Proceedings of the 5th Conference on Computing Frontiers*, 2008, pp. 43–54. DOI: 10.1145/1366230.1366239.

[261] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Prentice Hall, 1993.

[262] A. Paradis, B. Bichsel, S. Steffen, and M. Vechev, "Unqomp: Synthesizing uncomputation in quantum circuits," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 222–236. DOI: 10.1145/3453483.3454040.

[263] A. Paradis, B. Bichsel, and M. Vechev, "Reqomp: Space-constrained uncomputation for quantum circuits," *Quantum*, vol. 8, p. 1258, 2024. DOI: 10.22331/q-2024-02-19-1258.

[264] H. Venev, T. Gehr, D. Dimitrov, and M. Vechev, "Modular synthesis of efficient quantum uncomputation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 2097–2124, 2024. DOI: 10.1145/3689785.

[265] Y. Ofman, "On the algorithmic complexity of discrete functions," in *Sov. Math. Dokl.*, vol. 7, 1963, p. 589.

[266] J. H. Reif, "Logarithmic depth circuits for algebraic functions," *SIAM Journal on Computing*, vol. 15, no. 1, pp. 231–242, 1986. DOI: 10.1137/0215017.

[267] P. W. Beame, S. A. Cook, and H. J. Hoover, "Log depth circuits for division and related problems," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 994–1003, 1986. DOI: 10.1109/SFCS.1984.715894.

[268] R. Cleve and J. Watrous, "Fast parallel circuits for the quantum Fourier transform," in *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, ser. FOCS '00, 2000, pp. 526–536. DOI: 10.1109/SFCS.2000.892140.

[269] C. Moore and M. Nilsson, "Parallel quantum computation and quantum codes," *SIAM Journal on Computing*, vol. 31, no. 2, pp. 799–815, 2002. DOI: 10.1137/S0097539799355053.

[270] F. Green, S. Homer, C. Moore, and C. Pollett, "Counting, fanout and the complexity of quantum ACC," *Quantum Information & Computation*, vol. 2, no. 1, pp. 35–65, 2002. DOI: 10.26421/QIC2.1-3.

[271] B. M. Terhal and D. P. DiVincenzo, "Adptive quantum computation, constant depth quantum circuits and arthur-merlin games," *Quantum Information & Computation*, vol. 4, no. 2, pp. 134–145, 2004. DOI: 10.26421/QIC4.2-5.

[272] P. Høyer and R. Špalek, "Quantum fan-out is powerful," *Theory of Computing*, vol. 1, no. 5, pp. 81–103, 2005. DOI: 10.4086/toc.2005.v001a005.

[273] D. Bera, F. Green, and S. Homer, "Small depth quantum circuits," *ACM SIGACT News*, vol. 38, no. 2, pp. 35–50, 2007. DOI: 10.1145/1272729.1272739.

[274] Y. Takahashi and S. Tani, "Collapse of the hierarchy of constant-depth exact quantum circuits," in *Proceedings of the 28th IEEE Conference on Computational Complexity*, 2013, pp. 168–178. DOI: 10.1109/CCC.2013.25.

[275] J. Jiang, X. Sun, S.-H. Teng, B. Wu, K. Wu, and J. Zhang, "Optimal space-depth trade-off of CNOT circuits in quantum logic synthesis," in *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '20, 2020, pp. 213–229. DOI: 10.1137/1.9781611975994.13.

[276] X.-M. Zhang, M.-H. Yung, and X. Yuan, "Low-depth quantum state preparation," *Physical Review Research*, vol. 3, no. 4, p. 043 200, 2021. DOI: 10.1103/PhysRevResearch.3.043200.

[277] G. Rosenthal, *Query and depth upper bounds for quantum unitaries via Grover search*, 2023. arXiv: 2111.07992 [quant-ph].

[278] X. Sun, G. Tian, S. Yang, P. Yuan, and S. Zhang, "Asymptotically optimal circuit depth for quantum state preparation and general unitary synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 10, pp. 3301–3314, 2023. DOI: 10.1109/TCAD.2023.3244885.

[279] P. Yuan and S. Zhang, "Optimal (controlled) quantum state preparation and improved unitary synthesis by quantum circuits with any number of ancillary qubits," *Quantum*, vol. 7, p. 956, 2023. DOI: 10.22331/q-2023-03-20-956.

[280] R. Raussendorf and H. J. Briegel, "A one-way quantum computer," *Physical Review Letters*, vol. 86, no. 22, p. 5188, 2001. DOI: 10.1103/PhysRevLett.86.5188.

[281] R. Jozsa, *An introduction to measurement based quantum computation*, 2005. arXiv: quant-ph/0508124 [quant-ph].

[282] V. Danos, E. Kashefi, and P. Panangaden, "The measurement calculus," *Journal of the ACM (JACM)*, vol. 54, no. 2, 8–es, 2007. DOI: 10.1145/1219092.1219096.

[283] A. Broadbent and E. Kashefi, "Parallelizing quantum circuits," *Theoretical Computer Science*, vol. 410, no. 26, pp. 2489–2510, 2009. DOI: 10.1016/j.tcs.2008.12.046.

[284] D. Browne, E. Kashefi, and S. Perdrix, "Computational depth complexity of measurement-based quantum computation," in *Theory of Quantum Computation, Communication, and Cryptography: 5th Conference, TQC 2010*, 2011, pp. 35–46. DOI: 10.1007/978-3-642-18073-6_4.

[285] E. Pius, "Automatic parallelisation of quantum circuits using the measurement based quantum computing model," M.S. thesis, University of Edinburgh, 2010.

[286] R. D. da Silva, E. Pius, and E. Kashefi, *Global quantum circuit optimization*, 2013. arXiv: 1301.0351 [quant-ph].

[287] M. Amy, M. Roetteler, and K. M. Svore, "Verified compilation of space-efficient reversible circuits," in *Computer Aided Verification*. 2017, pp. 3–21. DOI: 10.1007/978-3-319-63390-9_1.

[288] G. S. Graham and P. J. Denning, "Protection: Principles and practice," in *Proceedings of the May 16-18, 1972, spring joint computer conference*, 1971, pp. 417–429. DOI: 10.1145/1478873.1478928.

[289] P. J. Denning, "Third generation computer systems," *ACM Computing Surveys (CSUR)*, vol. 3, no. 4, pp. 175–216, 1971. DOI: 10.1145/356593.356595.

[290] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, no. 8, pp. 461–471, 1976. DOI: 10.1145/360303.360333.

[291] J. H. Saltzer, "Protection and the control of information sharing in Multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974. DOI: 10.1145/361011.361067.

[292] D. D. Downs, J. R. Rub, K. C. Kung, and C. S. Jordan, "Issues in discretionary access control," in *1985 IEEE symposium on security and privacy*, 1985, pp. 208–208. DOI: 10.1109/SP.1985.10014.

[293] L. J. LaPadula and D. E. Bell, "Secure computer systems: A mathematical model," The MITRE Corporation, Bedford, MA, Tech. Rep. ESD–TR–73–278–I, Mar. 1973.

[294] D. E. Bell and L. J. La Padula, "Secure computer system: Unified exposition and Multics interpretation," The MITRE Corporation, Bedford, MA, Tech. Rep. ESD-TR-75-306, Mar. 1976.

[295] K. J. Biba, "Integrity considerations for secure computer systems," The MITRE Corporation, Bedford, MA, Tech. Rep. ESD-TR-76-372, Apr. 1977.

[296] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *1987 IEEE Symposium on Security and Privacy*, 1987, pp. 184–184. DOI: 10.1109/SP.1987.10001.

[297] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976. DOI: 10.1145/360051.360056.

[298] D. F. Ferraiolo and D. R. Kuhn, "Role-based access controls," 1992, pp. 554–563.

[299] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," in *IEEE Computer*, 2, vol. 29, 1996, pp. 38–47. DOI: 10.1109/2.485845.

[300] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001. DOI: 10.1145/501978.501980.

[301] E. Bertino, P. A. Bonatti, and E. Ferrari, "Trbac: A temporal role-based access control model," in *Proceedings of the fifth ACM workshop on Role-based access control*, 2000, pp. 21–30. DOI: 10.1145/501978.501979.

[302] A. M. Childs, "Secure assisted quantum computation," *Quantum Information & Computation*, vol. 5, no. 6, pp. 456–466, 2005. DOI: 10.26421/QIC5.6-4.

[303] D. Aharonov, M. Ben-Or, and E. Eban, *Interactive proofs for quantum computations*, 2008. arXiv: 0810.5375 [quant-ph].

[304] A. Broadbent, J. F. Fitzsimons, and E. Kashefi, "Universal blind quantum computation," in *2009 50th annual IEEE symposium on foundations of computer science*, 2009, pp. 517–526. DOI: 10.1109/FOCS.2009.36.

[305] V. Dunjko, E. Kashefi, and A. Leverrier, "Blind quantum computing with weak coherent pulses," *Physical Review Letters*, vol. 108, no. 20, p. 200 502, 2012. DOI: 10.1103/PhysRevLett.108.200502.

[306] T. Morimae, "Continuous-variable blind quantum computation," *Physical Review Letters*, vol. 109, no. 23, p. 230 502, 2012. DOI: 10.1103/PhysRevLett.109.230502.

[307] T. Morimae and K. Fujii, "Blind topological measurement-based quantum computation," *Nature communications*, vol. 3, no. 1, p. 1036, 2012. DOI: 10.1038/ncomms2043.

[308] V. Giovannetti, L. Maccone, T. Morimae, and T. G. Rudolph, "Efficient universal blind quantum computation," *Physical Review Letters*, vol. 111, no. 23, p. 230 501, 2013. DOI: 10.1103/PhysRevLett.111.230501.

[309] A. Mantri, C. A. Pérez-Delgado, and J. F. Fitzsimons, "Optimal blind quantum computation," *Physical Review Letters*, vol. 111, no. 23, p. 230 502, 2013. DOI: 10.1103/PhysRevLett.111.230502.

[310] T. Morimae and K. Fujii, "Blind quantum computation protocol in which Alice only makes measurements," *Physical Review A*, vol. 87, no. 5, p. 050 301, 2013. DOI: 10.1103/PhysRevA.87.050301.

[311] T. Morimae and T. Koshiba, *Composable security of measuring-Alice blind quantum computation*, 2013. arXiv: 1306.2113 [quant-ph].

[312] T. Morimae, "Verification for measurement-only blind quantum computing," *Physical Review A*, vol. 89, no. 6, p. 060 302, 2014. DOI: 10.1103/PhysRevA.89.060302.

[313] T. Morimae, V. Dunjko, and E. Kashefi, "Ground state blind quantum computation on AKLT state," *Quantum Information & Computation*, vol. 15, no. 3–4, pp. 200–234, 2015. DOI: 10.26421/QIC15.3-4-3.

[314] J. F. Fitzsimons and E. Kashefi, "Unconditionally verifiable blind quantum computation," *Physical Review A*, vol. 96, no. 1, p. 012 303, 2017. DOI: 10.1103/PhysRevA.96.012303.

[315] J. F. Fitzsimons, "Private quantum computation: An introduction to blind quantum computing and related protocols," *npj Quantum Information*, vol. 3, no. 1, p. 23, 2017. DOI: 10.1038/s41534-017-0025-2.

[316] T. Trochatos, C. Xu, S. Deshpande, Y. Lu, Y. Ding, and J. Szefer, "A quantum computer trusted execution environment," *IEEE Computer Architecture Letters*, vol. 22, no. 2, pp. 177–180, 2023. DOI: 10.1109/LCA.2023.3325852.

[317] T. Trochatos, C. Xu, S. Deshpande, Y. Lu, Y. Ding, and J. Szefer, *Hardware architecture for a quantum computer trusted execution environment*, 2023. arXiv: 2308.03897 [cs.ET].

[318] T. Trochatos and J. Szefer, *Quantum operating system support for quantum trusted execution environments*, 2024. arXiv: 2410.08486 [quant-ph].

[319] T. Trochatos, S. Deshpande, C. Xu, Y. Lu, Y. Ding, and J. Szefer, "Dynamic pulse switching for protection of quantum computation on untrusted clouds," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2024, pp. 404–414. DOI: 10.1109/HOST55342.2024.10545385.

[320] A. Mi, S. Deng, and J. Szefer, "Securing reset operations in nisq quantum computers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2279–2293. DOI: 10.1145/3548606.3559380.

[321] C. Xu, F. Erata, and J. Szefer, "Exploration of power side-channel vulnerabilities in quantum computer controllers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 579–593. DOI: 10.1145/3576915.3623118.

[322] C. Xu, J. Chen, A. Mi, and J. Szefer, "Securing nisq quantum computer reset operations against higher energy state attacks," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 594–607. DOI: 10.1145/3576915.3623104.

[323] L. Gyongyosi and S. Imre, "Entanglement access control for the quantum internet," *Quantum Information Processing*, vol. 18, pp. 1–17, 2019. DOI: 10.1007/s11128-019-2226-5.

[324] J. S. Bell, "On the Einstein Podolsky Rosen paradox," *Physics Physique Fizika*, vol. 1, no. 3, p. 195, 1964. DOI: 10.1103/PhysicsPhysiqueFizika.1.195.

[325] J. F. Clauser, M. A. Horne, A. Shimony, and R. A. Holt, "Proposed experiment to test local hidden-variable theories," *Physical Review Letters*, vol. 23, no. 15, p. 880, 1969. DOI: 10.1103/PhysRevLett.23.880.

[326] S. J. Freedman and J. F. Clauser, "Experimental test of local hidden-variable theories," *Physical Review Letters*, vol. 28, no. 14, p. 938, 1972. DOI: 10.1103/PhysRevLett.28.938.

[327] A. Aspect, J. Dalibard, and G. Roger, "Experimental test of Bell's inequalities using time-varying analyzers," *Physical Review Letters*, vol. 49, no. 25, p. 1804, 1982. DOI: 10.1103/PhysRevLett.49.1804.

[328] D. M. Greenberger, M. A. Horne, A. Shimony, and A. Zeilinger, "Bell's theorem without inequalities," *American Journal of Physics*, vol. 58, no. 12, pp. 1131–1143, 1990. DOI: 10.1119/1.16243.

[329] R. Jozsa and N. Linden, "On the role of entanglement in quantum-computational speed-up," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 459, no. 2036, pp. 2011–2032, 2003. DOI: 10.1098/rspa.2002.1097.

[330] R. Sandhu and J. Park, "Usage control: A vision for next generation access control," in *Computer Network Security: Second International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2003*, 2003, pp. 17–31. DOI: 10.1007/978-3-540-45215-7_2.

[331] J. Park and R. Sandhu, "The ucon_{ABC} usage control model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 1, pp. 128–174, 2004. DOI: 10.1145/984334.984339.

[332] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park, "Formal model and policy specification of usage control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 4, pp. 351–387, 2005. DOI: 10.1145/1108906.1108908.

[333] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (ABAC) definition and considerations," *NIST Special Publication*, vol. 800, no. 162, pp. 1–54, 2013.

[334] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014. DOI: 10.1002/9781118625390.

[335] M. O. Rabin, "N-process synchronization by $4 \cdot \log_2 N$-valued shared variable," in *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, 1980, pp. 407–410. DOI: 10.1109/SFCS.1980.26.

[336] G. Brassard, A. Broadbent, and A. Tapp, "Recasting Mermin's multi-player game into the framework of pseudo-telepathy," *Quantum Information and Computation*, vol. 5, no. 7, pp. 538–550, 2005. DOI: 10.26421/QIC5.7-2.

[337] J. Pearl, *Causality: Models, Reasoning, and Inference*. USA: Cambridge University Press, 2000. DOI: 10.1017/CBO9780511803161.

[338] A. K. Ekert, "Quantum cryptography based on Bell's theorem," *Physical Review Letters*, vol. 67, no. 6, p. 661, 1991. DOI: 10.1103/PhysRevLett.67.661.

[339] D. Mayers and A. Yao, "Quantum cryptography with imperfect apparatus," in *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*, 1998, pp. 503–509. DOI: 10.5555/795664.796390.

[340] J. Barrett, L. Hardy, and A. Kent, "No signaling and quantum key distribution," *Physical Review Letters*, vol. 95, no. 1, p. 010 503, 2005. DOI: 10.1103/PhysRevLett.95.010503.

[341] S. Pironio, A. Acín, N. Brunner, N. Gisin, S. Massar, and V. Scarani, "Device-independent quantum key distribution secure against collective attacks," *New Journal of Physics*, vol. 11, no. 4, p. 045 021, 2009. DOI: 10.1088/1367-2630/11/4/045021.

[342] B. W. Reichardt, F. Unger, and U. Vazirani, "Classical command of quantum systems," *Nature*, vol. 496, no. 7446, pp. 456–460, 2013. DOI: 10.1038/nature12035.

[343] U. Vazirani and T. Vidick, "Fully device-independent quantum key distribution," *Physical Review Letters*, vol. 113, p. 140 501, 14 2014. DOI: 10.1103/PhysRevLett.113.140501.

[344] C. A. Miller and Y. Shi, "Robust protocols for securely expanding randomness and distributing keys using untrusted quantum devices," *Journal of the ACM (JACM)*, vol. 63, no. 4, pp. 1–63, 2016. DOI: 10.1145/2955041.

[345] R. Arnon-Friedman, R. Renner, and T. Vidick, "Simple and tight device-independent security proofs," *SIAM Journal on Computing*, vol. 48, no. 1, pp. 181–225, 2019. DOI: 10.1137/17M1144890.

[346] F. Dupuis, O. Fawzi, and R. Renner, "Entropy accumulation," *Communications in Mathematical Physics*, vol. 379, no. 3, pp. 867–913, 2020. DOI: 10.1007/s00220-018-3325-9.

[347] F. Dupuis and O. Fawzi, "Entropy accumulation with improved second-order term," *IEEE Transactions on Information Theory*, vol. 65, no. 11, pp. 7596–7612, 2019. DOI: 10.1109/TIT.2019.2914507.

[348] R. Colbeck, "Quantum and relativistic protocols for secure multi-party computation," Ph.D. dissertation, University of Cambridge, 2009.

[349] S. Pironio, A. Acín, S. Massar, A. B. de La Giroday, D. N. Matsukevich, P. Maunz, S. Olmschenk, D. Hayes, L. Luo, T. A. Manning, *et al.*, "Random numbers certified by Bell's theorem," *Nature*, vol. 464, no. 7291, pp. 1021–1024, 2010. DOI: 10.1038/nature09008.

[350] U. Vazirani and T. Vidick, "Certifiable quantum dice: Or, true random number generation secure against quantum adversaries," in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, 2012, pp. 61–76. DOI: 10.1145/2213977.2213984.

[351] M. Coudron and H. Yuen, "Infinite randomness expansion with a constant number of devices," in *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, 2014, pp. 427–436. DOI: 10.1145/2591796.2591873.

[352] C. A. Miller and Y. Shi, "Universal security for randomness expansion from the spot-checking protocol," *SIAM Journal on Computing*, vol. 46, no. 4, pp. 1304–1335, 2017. DOI: 10.1137/15M1044333.

[353] R. Colbeck and R. Renner, "Free randomness can be amplified," *Nature Physics*, vol. 8, no. 6, pp. 450–453, 2012. DOI: 10.1038/nphys2300.

[354] R. Gallego, L. Masanes, G. De La Torre, C. Dhara, L. Aolita, and A. Acín, "Full randomness from arbitrarily deterministic events," *Nature communications*, vol. 4, no. 1, p. 2654, 2013. DOI: 10.1038/ncomms3654.

[355] K. Chung, Y. Shi, and X. Wu, *Physical randomness extractors: Generating random numbers with minimal assumptions*, 2015. arXiv: 1402.4797 [quant-ph].

[356] M. Kessler and R. Arnon-Friedman, "Device-independent randomness amplification and privatization," *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 2, pp. 568–584, 2020. DOI: 10.1109/JSAIT.2020.3012498.

[357] V. C. Hu, D. Ferraiolo, and D. R. Kuhn, *Assessment of access control systems*. US Department of Commerce, National Institute of Standards and Technology, 2006.

[358] V. C. Hu and K. A. Kent, *Guidelines for access control system evaluation metrics*. US Department of Commerce, National Institute of Standards and Technology, 2012.

[359] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. DOI: 10.1109/PROC.1975.9939.

[360] R. S. Sandhu and P. Samarati, "Access control: Principle and practice," *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994. DOI: 10.1109/35.312842.

[361] I. Quantum, *IBM Quantum roadmap*. [Online]. Available: https://www.ibm.com/roadmaps/quantum.pdf (visited on 11/13/2023).

[362] M. Schlosshauer, "Decoherence, the measurement problem, and interpretations of quantum mechanics," *Reviews of Modern physics*, vol. 76, no. 4, p. 1267, 2005.

[363] E. W. Dijkstra, "An effort towards structuring of programmed processes," circulated privately, n.d. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD198.PDF.

[364] M. Y. Vardi, "Automatic verification of probabilistic concurrent finite state programs," in *26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*, 1985, pp. 327–338. DOI: 10.1109/SFCS.1985.12.

[365] R. Segala and N. Lynch, "Probabilistic simulations for probabilistic processes," in *International Conference on Concurrency Theory*, 1994, pp. 481–496. DOI: 10.1007/978-3-540-48654-1_35.

[366] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990. DOI: 10.1145/78969.78972.

[367] M. Raynal, *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[368] L. Lamport, "A new approach to proving the correctness of multiprocess programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 84–97, 1979. DOI: 10.1145/357062.357068.

[369] C. H. Bennett, D. P. DiVincenzo, C. A. Fuchs, T. Mor, E. Rains, P. W. Shor, J. A. Smolin, and W. K. Wootters, "Quantum nonlocality without entanglement," *Physical Review A*, vol. 59, no. 2, p. 1070, 1999. DOI: 10.1103/PhysRevA.59.1070.

[370] L. Lamport, "The computer science of concurrency: The early years," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 13–26. DOI: 10.1145/3335772.3335775.

[371] L. Lamport, "win and sin: Predicate transformers for concurrency," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 396–428, 1990. DOI: 10.1145/78969.78970.

[372] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977. DOI: 10.1109/TSE.1977.229904.

[373] A. Klenke, *Probability theory: a comprehensive course*. Springer Science & Business Media, 2013.

[374] G. Neiger, "Set-linearizability," in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 1994, p. 396. DOI: 10.1145/197917.198176.

[375] A. Castañeda, S. Rajsbaum, and M. Raynal, "Unifying concurrent objects and distributed tasks: Interval-linearizability," *Journal of the ACM (JACM)*, vol. 65, no. 6, pp. 1–42, 2018. DOI: 10.1145/3266457.

[376] Y.-F. Chen, K.-M. Chung, O. Lengál, J.-A. Lin, W.-L. Tsai, and D.-D. Yen, "An automata-based framework for verification and bug hunting in quantum circuits," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1218–1243, 2023. DOI: 10.1145/3591270.

[377] F. Bauer-Marquart, S. Leue, and C. Schilling, "symQV: Automated symbolic verification of quantum programs," in *International Symposium on Formal Methods*, 2023, pp. 181–198. DOI: 10.1007/978-3-031-27481-7_12.