Formal Methods for the Reliability of Non-Classical Systems

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

DISSERTATION

von

Jonas Klamroth

Tag der mündlichen Prüfung: 3. Februar 2025

Referent:	Prof. Dr. rer. nat. Bernhard Beckert, Karlsruher Institut für Technologie
Koreferent:	Prof. Dr. rer. nat. Sadegh Soudjani, Max Planck Institute for Software Systems

Acknowledgements

I would like to thank all the wonderful people who contributed to the fact that this thesis will finally see the light of day. Starting with my supervisor, Bernhard Beckert, who not only convinced me to start my PhD in the first place but also supported me over all these years, always providing invaluable feedback and new ideas. Similarly significant to my interest in the area of formal methods is Mattias Ulbrich, who not only supervised my first steps in this field but also managed to spark an enduring passion that kept me engaged for more than eight years. During this time, he also became a good friend.

I would also like to thank Jonas Schiffl, who was a friend long before we became colleagues. He not only paved the way by always being one step ahead on the path toward his PhD but also proved to be the best friend I could have asked for during this time (together with Fanny of course). My gratitude also extends to all my colleagues at KIT and the FZI. I have truly enjoyed working with all of you and am keenly aware that much of the content of this thesis could not exist without the discussions, inspiration, and feedback you provided. Thank you to all of you, and in particular, Alexander Weigl and Max Scheerer. My journey at FZI would also not have been possible without the support of my supervisors, Jörg Henß and Oliver Denninger. Thank you for always having my back. Also thank you to Sadegh Soudjani for agreeing to be my second supervisor.

Last, but by no means least, I would like to thank my family. My parents and sister have always supported me, offering both encouragement and the necessary nudges when I felt stuck or doubted myself. And above all, I want to express my deepest gratitude to my amazing wife. It is no exaggeration to say that this thesis would never have been possible without her love, guidance, and unwavering support, for which I am thankful. Endlessly.

Summary

Non-classical systems, as a broad category of computational paradigms extending beyond classical computing, hold immense potential to revolutionize the field of computing. Whether we consider quantum computing, which offers superpolynomial speedups for certain problems, or machine learning approaches, which already outperform classical techniques in various application areas, these systems demonstrate significant advantages over their classical counterparts.

However, this additional computational power comes at the cost of increased complexity. Quantum mechanics, the foundation of quantum computing's speedup, is notoriously difficult to understand. Similarly, interpreting and understanding the decision-making processes of large machine learning models is a complex challenge. This inherent complexity, combined with their deployment in resource-intensive or even safety-critical scenarios, makes non-classical systems a prime focus for reliability analysis.

In this thesis, we explore methods to classify and enhance the reliability of such systems across various levels of abstraction.

Main Contributions

In this thesis we present five main contributions: 1) We identify and characterize similarities regarding the verification of non-classical systems, 2) We present four classes of analyzability for non-classical systems, 3) We show how fault-tolerant architectural patterns can be used to increase the reliability of quantum computing systems, 4) We present an approach for the formal verification of hybrid quantum software and 5) We derive upper bounds for the rounding errors in quantum simulators. All of these contributions add to the reliability of non-classical system in some sense.

Similarities regarding Verification of Non-Classical Systems We argue that non-classical systems share several common properties that pose

significant challenges to their verification. In this context, we identify and discuss seven such key challenges and outline strategies to address them. To provide concrete insights, we focus on two representative examples: machine-learning systems and quantum systems, highlighting the similarities in their verification requirements.

Classes of Analyzability We present four classes of analyzability for non-classical systems, designed to encapsulate varying levels of assurances achievable through formal methods. These classes serve as a framework for evaluating and characterizing the reliability of a system at the architectural level, enabling the identification of potential inconsistencies and guiding the design towards improved dependability.

Fault-Tolerant Architectural Patterns for Quantum Computing The immense potential of quantum computing in the Noisy Intermediate-Scale Quantum Era (NISQ Era) is severely hampered by the inherent error-prone nature of quantum devices. Current quantum hardware is susceptible to gate and measurement errors, which can drastically reduce the reliability of quantum computations. One promising avenue to address this challenge on an architectural level is using fault-tolerant architectural patterns, which have proven effective in other domains. These patterns leverage redundancy to detect and mitigate errors. We adapted these principles to the quantum domain, where unique properties such as superposition and entanglement have to be taken into account. We investigate potential sources of redundancy in quantum components, how fault-tolerant architectural patterns can be applied to them, and evaluate the impact of these patterns on the overall reliability of quantum circuits. Our evaluation results indicate that the application of fault tolerance patterns in this domain is highly promising.

Formal Verification of Hybrid Quantum Software Quantum circuits, and consequently quantum software, are inherently complex and unintuitive. This complexity makes them an ideal candidate for formal verification, which can guarantee that a desired property holds for a given program. However, the verification process itself can become prohibitively complex, particularly when dealing with systems as intricate as quantum systems.

We propose a method for the fully automatic verification of small instances of hybrid quantum software using existing and familiar verification tools. This method relies on a property-preserving translation of quan-

tum circuits into a classical host language. The resulting program, written entirely in the classical host language, allows the application of any verification tool designed for that language. Since the translation preserves properties, verifying the translated program is equivalent to verifying the original quantum circuit.

By embedding circuits into the host language, this approach inherently supports the verification of hybrid quantum software. We demonstrate that both the translation process and the verification of the translated program can be performed fully automatically. Our approach is shown to be applicable to a range of well-known quantum algorithms.

Bounding Rounding Errors in the Simulation of Quantum Circuits Since current quantum devices are still highly error-prone, simulating quantum circuits is the de facto standard during development. Simulators offer several advantages: they are free from NISQ-era errors, more readily accessible, and provide debugging capabilities that are physically impossible to realize on real quantum devices. However, simulators introduce their own type of errors: they rely on finite-precision floating-point arithmetic, which contrasts with the real-valued arithmetic of quantum physics, leading to rounding errors that can alter simulation results.

We demonstrate that rounding errors in the simulation of quantum circuits can be bounded in terms of the number of qubits and gates in the circuit. Furthermore, we refine this bound to be independent of the number of qubits, relying instead on the size of the largest gate applied in the circuit and the total number of gates. Additionally, we present a computable variant of this bound that remains valid even when computed using floating-point arithmetic. Our evaluation of the presented bounds across a wide range of gates and qubits shows that these bounds are sufficient to rule out significant rounding errors for a substantial portion of simulated circuits.

Conclusion We present several approaches to increase the reliability of non-classical systems, with a particular focus on quantum computing systems. The proposed methods span a spectrum of analysis techniques, ranging from architectural-level approaches to formal analysis at the source code level, thereby addressing reliability challenges across multiple levels of abstraction.

Zusammenfassung

Motivation

Nicht-klassische Systeme, die als eigene Klasse von Systemen verstanden werden können, besitzen ein enormes Potenzial, die Informatik grundlegend zu revolutionieren. Egal ob Quantencomputer, die superpolynomielle Geschwindigkeitsvorteile für bestimmte Probleme versprechen, oder maschinelles Lernen, das in vielen Anwendungsbereichen bereits klassische Techniken übertrifft – diese Systeme zeigen deutliche Vorteile gegenüber klassischen Equivalenten.

Die zusätzliche Ausdrucksstärke solcher Systeme geht jedoch mit einer erhöhten Komplexität einher. Die Quantenmechanik, die Grundlage für den Geschwindigkeitsvorteil des Quantencomputings ist, gilt als extrem schwer verständlich. Ebenso ist Verständlichkeit und Interpretierbarkeit für großen Modelle beim maschinellen Lernen eine Herausforderung. Diese inhärente Komplexität, kombiniert mit dem Einsatz in ressourcenintensiven oder sogar sicherheitskritischen Szenarien, macht nicht-klassische Systeme zu einem perfekten Anwendungsgebiet für verschiedenste Arten von Zuverlässigkeitsanalysen.

In dieser Dissertation untersuchen wir Methoden, um die Zuverlässigkeit solcher Systeme auf verschiedenen Abstraktionsebenen zu klassifizieren und zu verbessern.

Hauptbeiträge

In dieser Arbeit präsentieren wir fünf wesentliche Beiträge: 1) Wir identifizieren und charakterisieren Gemeinsamkeiten hinsichtlich der Verifikation von nicht-klassischen Systemen, 2) Wir stellen vier Klassen der Analysierbarkeit für nicht-klassische Systeme vor, 3) Wir zeigen, wie fehlertolerante Architekturpatterns genutzt werden können, um die Zuverlässigkeit von Quantencomputersystemen zu erhöhen, 4) Wir präsentieren einen Ansatz

zur formalen Verifikation hybrider Quantensoftware und 5) Wir leiten obere Schranken für Rundungsfehler in Quantensimulatoren ab. Alle diese Beiträge tragen zur Zuverlässigkeit nicht-klassischer Systeme bei.

Gemeinsamkeiten bei der Verifikation nicht-klassischer Systeme Wir argumentieren, dass nicht-klassische Systeme mehrere gemeinsame Eigenschaften aufweisen, die signifikante Herausforderungen für ihre Verifikation darstellen. In diesem Zusammenhang identifizieren und diskutieren wir sieben zentrale Herausforderungen und skizzieren Strategien zu ihrer Bewältigung. Um konkrete Einblicke zu bieten, konzentrieren wir uns auf zwei repräsentative Beispiele: Maschinen-gelernte Systeme und Quantensysteme, wobei die Gemeinsamkeiten in ihren Verifikationsanforderungen hervorgehoben werden.

Klassen der Analysierbarkeit Wir stellen vier Klassen der Analysierbarkeit für nicht-klassische Systeme vor, die darauf ausgelegt sind, unterschiedliche Ebenen der durch formale Methoden erreichbaren Garantien zu erfassen. Diese Klassen dienen als Rahmenwerk zur Bewertung und Charakterisierung der Zuverlässigkeit eines Systems auf der Architekturebene. Sie ermöglichen die Identifikation potenzieller Inkonsistenzen und bieten Leitlinien für das Design hin zu höherer Verlässlichkeit.

Fehlertolerante Architektur-Muster für Quantencomputing Das enorme Potenzial des Quantencomputings wird in der NISQ-Ära (Noisy Intermediate-Scale Quantum) stark durch die Fehleranfälligkeit der aktuellen Quantenhardware eingeschränkt. Aktuelle Quantenhardware weist signifikante Gate- und Messfehler auf, welche die Genauigkeit von Berechnungen drastisch reduzieren können.

Ein möglicher Ansatz zur Bewältigung dieser Herausforderung auf architektonischer Ebene ist die Verwendung fehlertoleranter Architektur-Muster, die sich in anderen Bereichen bereits als effektiv erwiesen haben. Diese Muster nutzen Redundanz, um Fehler zu erkennen und zu reduzieren. Wir haben diese etablierten Muster für Quantensoftware-Systeme angepasst und erweitert. Hierbei müssen spezielle Eigenschaften wie Superposition und Verschränkung berücksichtigt werden. Wir untersuchen mögliche Quellen der Redundanz in Quantensystemen, wie fehlertolerante Architektur-Muster auf diese angewendet werden können, und bewerten deren Effizienz bei der Verbesserung der Zuverlässigkeit von Quantenschaltkreisen. Unsere Evaluationsergebnisse zeigen, dass die Anwendung solcher Muster in diesem Bereich vielversprechend ist.

Formale Verifikation hybrider Quantensoftware Quantenschaltkreise und damit auch Quantensoftware sind von Natur aus komplex und schwer zu durchschauen. Diese Komplexität macht sie zu idealen Kandidaten für formale Methoden, die garantieren können, dass eine gewünschte Eigenschaft für ein gegebenes Programm immer erfüllt ist.

Wir stellen eine Methode vor, die es erlaubt kleine Instanzen hybrider Quantensoftware vollautomatisiert zu verifizieren. Dabei verwenden wir existierende Verifikationswerkzeuge die sich bereits für klassische Software etabliert haben. Dieser Ansatz beruht auf einer eigenschaftserhaltenden Übersetzung von Quantenschaltkreisen in eine klassische Hostsprache. Das resultierende Programm, das vollständig in der klassischen Hostsprache geschrieben ist, ermöglicht den Einsatz von Verifikationswerkzeugen, die für diese Sprache entwickelt wurden. Da die Übersetzung eigenschaftsterhaltend ist, ist die Verifikation des übersetzten Programms gleichwertig zur Verifikation des ursprünglichen Quantenschaltkreises.

Durch die Einbettung von Schaltkreisen in die Hostsprache unterstützt dieser Ansatz die Verifikation hybrider Quantensoftware auf sehr natürlich Art und Weise. Wir zeigen, dass sowohl der Übersetzungsprozess als auch die Verifikation des übersetzten Programms vollständig automatisch durchgeführt werden können. Durch die Erfolgreiche Anwendung unseres Ansatzes an einer Reihe bekannter Quantenalgorithmen wird seine Praxistauglichkeit demonstriert.

Schranken für Rundungsfehler in der Simulation von Quantenschaltkreisen Da aktuelle Quantenhardware noch äußerst fehleranfällig ist, ist die Simulation von Quantenschaltkreisen der de-facto-Standard während der Entwicklung von Quantensoftware. Simulatoren bieten mehrere Vorteile: Sie sind frei von NISQ-Ära-Fehlern, leichter zugänglich und ermöglichen Debugging-Funktionen, die auf realen Quantencomputern physikalisch unmöglich umzusetzen sind. Simulatoren bringen jedoch ihre eigenen Fehler mit sich: Sie basieren auf endlicher Gleitkommaarithmetik, im Gegensatz zur reellen Arithmetik der Quantenphysik, was zu Rundungsfehlern führen kann, welche die Simulationsergebnisse verfälschen.

Wir zeigen, dass Rundungsfehler in der Simulation von Quantenschaltkreisen allein basierend auf der Anzahl der Qubits und Gates im Schaltkreis beschränkt werden können. Darüber hinaus verfeinern wir diese Schranke, sodass sie unabhängig von der Anzahl der Qubits ist und stattdessen von der Größe des größten im Schaltkreis angewandten Gates und der Gesamtzahl der Gates abhängt. Zusätzlich präsentieren wir eine berechenbare Variante dieser Schranke, die selbst bei der Berechnung in Gleitkommaarith-

metik gültig bleibt. Unsere Evaluierung der präsentierten Schranken zeigt, dass diese Schranken ausreichen, um signifikante Rundungsfehler für einen Großteil simulierbarer Schaltkreise auszuschließen.

Fazit Wir präsentieren mehrere Ansätze zur Erhöhung der Zuverlässigkeit nicht-klassischer Systeme mit einem besonderen Fokus auf Quantencomputingsysteme. Die vorgeschlagenen Methoden umfassen ein Spektrum von Analysetechniken, die von architektonischen Ansätzen bis hin zur formalen Analyse auf Quellcode-Ebene reichen, und erlauben somit eine erhöhte Zuverlässigkeit auf verschiedenen Abstraktionsebenen zu erreichen.

Contents

Chapter 1

Introduction

"Classical computing is an extraordinary success story. However, there is a growing appreciation that it encompasses an extremely small subset of all computational possibilities."

Susan Stepney et al. in [138]

1.1 Motivation

The advent of machine learning as a major research area has revolutionized the field of computing. Tasks that remained unsolved for years have become tractable through innovative methodologies such as deep learning. Likewise, quantum computing systems promise super-polynomial speedups for certain problems, potentially unlocking application areas previously out of reach. Both technologies share the characteristic of being non-classical systems. While non-classical systems undeniably hold immense potential, this comes at the cost of increased complexity and reduced comprehensibility. We argue that these challenges make non-classical systems prime candidates for the application of formal methods and other approaches aimed at enhancing their reliability. In this thesis, we develop methods to address this need for increased reliability in non-classical systems.

Non-classical systems The term *non-classical system* can have varying definitions depending on context. In this thesis, we adopt a broad definition that captures different types of non-classicality. We start by providing examples of what we consider to be a non-classical system and what we do not.

As a prototypical example, we consider quantum computing systems. Quantum computers derive their computational power from quantum physical effects that deviate from classical physics (further discussed in Section 2.1). Without these effects, the advantage of these specialized computers over classical machines would disappear. Thus, we classify them as non-classical systems. Similarly, any type of system built around biological models or concepts—such as neuromorphic hardware—is also considered non-classical due to the same principles.

A different branch of non-classical systems concerns data-driven systems, most prominently deep-learning systems, which fit our definition of non-classical systems. Their behavior cannot be fully described by their code but rather depends on the data they are trained on, contrasting with the hardware-based non-classicality in the prior examples. Here, the hardware remains classical, but the behavior and semantics are not.

On the other hand, we do not classify cyber-physical systems as non-classical. Although these systems operate in a physical environment, they typically run on classical hardware with classical code, interacting with a classical environment. Consequently, we categorize them as classical systems.

With these examples in mind, we present the following definition of a non-classical system:

> **Definition 1.1** (Non-classical System). *A non-classical system is a computational system that operates beyond the traditional deterministic and predictable frameworks of classical computing, leveraging principles, paradigms, or architectures that extend or deviate from conventional models of computation.*

By this definition, all examples mentioned above are categorized appropriately, allowing flexibility for a range of non-classical systems. Throughout this thesis, this definition will serve as the reference for any discussions of non-classical systems. Although some chapters broadly address the reliability of non-classical systems, we primarily focus on quantum computing systems. As quantum systems represent a quintessential example of non-classical systems, the methods we propose for them can, to some extent, be generalized to other non-classical systems—a topic we further explore in Chapter 3.

Reliability of Software Systems The reliability of software systems can be defined in various ways. Typically, the definition revolves around the system's ability to function without failure under specified environmental conditions. Under this broad understanding, various techniques to increase reliability are possible. For classical systems, the most well-known of these techniques is testing. However, for reasons detailed in Section 6.1, testing is not as efficient—if not practically infeasible—for many non-classical systems. Consequently, alternative methods must be employed to ensure their reliability.

1.2 Research Questions

As the reliability of non-classical systems is our overall concern, the main research question we are tackling in this thesis is the following:

Research Question 1. *What are suitable methods to increase the reliability of non-classical systems?*

All chapters of this thesis contribute to answering this overarching question. However, as this topic is very broad, we focus on several aspects that must be considered to successfully address it.

As a first step, we argue that, in order to apply methods to a category of systems as diverse as non-classical systems, we must understand the properties that they share. This leads to the following research question:

Research Question 2. *What reliability-related properties are characteristic of non-classical systems?*

Answering this question allows us to generalize insights gained for particular types of non-classical systems to non-classical systems as a whole. This generalization is crucial as it enables us to initially focus on specific system types, while ensuring broader applicability of our findings.

Next, we turn to approaches that directly enhance reliability for non-classical systems. We propose addressing this challenge at two distinct levels of abstraction, leading to the following two research questions:

Research Question 3. *How can the reliability of non-classical systems be increased at the architectural level?*

Research Question 4. *How can the reliability of non-classical systems be increased at the source code level?*

For these two questions, we focus specifically on quantum computing systems rather than addressing non-classical systems in general. Combined with the insights gained from answering Research Question 1, we argue that contributions targeting a single type of non-classical system represent an essential first step toward addressing the broader question. By answering these questions, we aim to develop a range of methods that collectively advance our overarching goal of increasing the reliability of non-classical systems. We will reference research questions as RQ1 to RQ4 in the remainder of this thesis.

1.3 Contributions

We address the presented research questions by providing the following contributions:

Similarities between Non-Classical Systems Regarding Verification We examine similarities between the verification of quantum computing systems and machine learning systems. Through this analysis, we identify and discuss seven shared challenges between these two research areas. This contribution represents a foundational step toward answering RQ2.

Classes of Analysability for Non-Classical Systems We propose four classes of analyzability for non-classical systems, designed to classify these systems based on the formal guarantees that can be provided for them. We demonstrate how the classification of components into these classes can guide system architects in obtaining a holistic reliability metric for the overall system. Furthermore, we analyze the relationships between the classes and discuss their practical applications. This contribution addresses both RQ2 and RQ3.

Fault-Tolerant Architectural Patterns for Hybrid Quantum Software We adapt established architectural patterns to the context of hybrid quantum computing systems. This involves tailoring these patterns from their classical counterparts, detailing their instantiation in quantum contexts, and evaluating their effectiveness. Our results demonstrate that the proposed patterns significantly enhance the reliability of targeted components when applied to typical quantum algorithms. This work contributes to answering RQ3 in the specific context of quantum systems.

Formal Verification of Hybrid Quantum Software We present a novel approach for translating quantum circuits into a classical host language. This method is formally proven to preserve properties, ensuring that the verification of the translated program is equivalent to the verification of the original quantum circuit. Building on this theoretical foundation, we implemented a Java-based tool that realizes this translation and demonstrate its applicability to several well-known quantum algorithms. This contribution marks a key step toward answering RQ4 for quantum systems.

Numerical Analysis of Quantum Circuit Simulators We derive an upper bound for rounding errors occurring during the simulation of quantum circuits. This analysis applies to a generic prototypical simulator and depends on factors such as the number of qubits and gates in the circuit. We propose multiple versions of this bound, including a computable variant valid under floating-point arithmetic. Through evaluation, we show that these bounds effectively rule out significant rounding errors for a wide range of quantum circuits. This contribution further advances our response to RQ4 for quantum systems.

1.4 New and Previously Published Material

Significant parts of this thesis have already been published. However, additional material has been written exclusively for this work, and previously published material has been significantly extended to provide a more comprehensive treatment of the topics. Below, we briefly comment on the respective chapters and their relationship to prior publications.

Chapter 2 was written entirely for this thesis and serves as a comprehensive introduction to the topic. Similarly, Chapter 3 was developed exclusively for this thesis. A condensed version of this chapter is scheduled for submission as a position paper.

Chapters 4 to 7 are based on material that has been previously published. However, each of these chapters has been significantly expanded and refined for this thesis:

- Chapter 4 builds upon the work published in [129]. While the original publication focused on machine-learning-based systems, this thesis generalizes the presented classes to encompass non-classical systems more broadly.

- Chapter 5 extends the approached discussed for a single pattern in [127] to include three fault-tolerant patterns applicable to quantum systems. Additionally, the evaluation of these patterns has been refined and significantly expanded.

- The primary approach described in Chapter 6 remains consistent with [91]. However, this thesis introduces the ability to prove the absence of rounding errors in translated circuits and extends the evaluation with additional case studies.

- Chapter 7 builds upon the bound originally presented in [90]. For this thesis, the bound has been improved to a substantially tighter version, and a computable variant of the bound is also introduced. These extensions are scheduled for future publication.

1.5 Outline

This thesis is organized as follows: In Chapter 2, we introduce the fundamental concepts and notions that are used throughout this thesis. We then begin addressing Research Question 2 in Chapter 3, where we analyze the shared properties of non-classical systems concerning reliability.

Subsequently, in Chapters 4 and 5, we present our contributions to architectural-level approaches for increasing the reliability of non-classical systems. Following this, we focus on source code-level approaches in Chapters 6 and 7, introducing methods that enhance reliability at a finer granularity.

Finally, we conclude in Chapter 8, summarizing our findings and outlining future research directions.

Chapter 2

Preliminaries

In this chapter, we establish the foundational concepts necessary for this thesis. Section 2.1 introduces the fundamental principles of quantum computing, focusing primarily on the mathematical framework underlying quantum computations. Additionally, we explore common high-level quantum programming languages and examine the deviations between theoretical quantum behavior and practical implementations due to hardware-level errors in current quantum computing devices. The second part of this chapter, Section 2.2, covers preliminaries related to software verification, particularly for Java. Here, we introduce the Java Modeling Language (JML) and the concept of bounded model checking in Section 2.2.2, both of which serve as the foundation for the verification approach developed in Chapter 6.

2.1 Quantum Computing Fundamentals

Quantum computing describes very broadly the field of research where computations are carried out on a device that exploits quantum mechanical effects. The most important of those effects are:

- Superposition: The fact that the system may be in multiple different states at once.
- Entanglement: The fact that system states may be connected in such a way that they cannot be changed independently of one another.
- Measurement: The fact that the system state is not observable directly but has to be measured. Measurements always produce a classical state and change the system to the state that was observed.

Those effects stand in stark contrast to classical systems. In this thesis, we focus exclusively on gate-based quantum computing. Specifically, we do not address quantum annealing [118] or measurement-based quantum computing [28]. To provide the necessary theoretical background, we introduce the fundamental concepts of quantum computing, following the presentation in [111], beginning with a concise history of the field.

2.1.1 A Brief History of Quantum Computing

The foundations of quantum mechanics were established in the 1920s by a group of pioneering physicists, including Max Born, Werner Heisenberg, and Wolfgang Pauli. The term "quantum mechanics" was first introduced in Max Born's seminal paper "Zur Quantenmechanik" [26]. However, it was not until the 1980s that the potential of quantum mechanics to revolutionize computation was first proposed. This concept is most notably attributed to Richard Feynman, who suggested that quantum systems could be simulated more efficiently by quantum computers than by any classical machine [60].

A few years later, David Deutsch expanded on this idea by proposing the first quantum algorithm, which demonstrated a theoretical advantage over classical algorithms [51]. This was followed by Peter Shor's work in 1994, where he introduced an algorithm capable of factoring large integers with super-polynomial speedup compared to classical methods [131]. Shor's algorithm was a pivotal moment in the field, as it highlighted the potential for quantum computers to solve *practically relevant* problems that are considered infeasible for classical computers.

Up until this point, these theoretical advancements were not accompanied by practical implementations, as no physical quantum computing devices existed. This changed in 1995 when Monroe et al. reported the first experimental realization of a quantum gate [107]. Since then, major technology companies and research institutions have committed substantial resources to the development of commercially viable quantum computers peaking in record investments in the field in the last few years.

2.1.2 Dirac notation

Dirac notation is the de facto standard for representing quantum states in the literature. It consists of bras $\langle y|$ and kets $|x\rangle$, which allow for a convenient representation of vectors in vector spaces (the terms "bra" and "ket" derive from "bracket"). In this notation, kets represent the actual vectors, while bras are transposed vectors that correspond to linear mappings from the vector space to a scalar:

$$|\phi\rangle = \begin{pmatrix}\phi_1 \ \phi_2 \ \cdots \ \phi_n\end{pmatrix}^T \qquad \langle\phi| = \begin{pmatrix}\phi_1 \ \phi_2 \ \cdots \ \phi_n\end{pmatrix}$$

This notation makes it easy to express the inner product $\langle x|y\rangle$, which is identical to the scalar product of the arguments, and the outer product $|y\rangle\langle x|$, which can be used as a short way for writing a matrix.

$$\langle\phi|\psi\rangle \ = \ \begin{pmatrix}\phi_1 \ \phi_2 \ \cdots \ \phi_n\end{pmatrix}\begin{pmatrix}\psi_1 \ \psi_2 \ \cdots \ \psi_n\end{pmatrix}^T \ = \ \phi_1\psi_1 + \phi_2\psi_2 + \cdots + \phi_n\psi_n$$

$$|\phi\rangle\langle\psi| = \begin{pmatrix}\phi_1\\ \phi_2\\ \dots\\ \phi_n\end{pmatrix}\begin{pmatrix}\psi_1 & \psi_2 & \dots & \psi_n\end{pmatrix} = \begin{pmatrix}\phi_1\psi_1 & \phi_1\psi_2 & \dots & \phi_1\psi_n\\ \phi_2\psi_1 & \phi_2\psi_2 & \dots & \phi_2\psi_n\\ \vdots & \vdots & \ddots & \vdots\\ \phi_n\psi_1 & \phi_n\psi_2 & \dots & \phi_n\psi_n\end{pmatrix}$$

In quantum computing, special kets are introduced to express quantum states: $|0\rangle$ resp. $|1\rangle$, which correspond to the classical values 0 and 1 of a classical bit. The special kets $|0\rangle$ and $|1\rangle$ are defined by

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \qquad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$$

2.1.3 Qubit

A qubit is the fundamental unit of quantum information. Its state is described by a linear combination, often called superposition, of the basic states $|0\rangle$ and $|1\rangle$:

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix} \qquad \text{where } \alpha,\beta \in \mathbb{C} \text{ with } |\alpha|^2 + |\beta|^2 = 1$$

This expression indicates that the state of a qubit is a point in a complex-valued two-dimensional Hilbert space. The vectors $|0\rangle$ and $|1\rangle$ form an orthonormal basis for this space, commonly referred to as the computational basis. The ability of quantum systems to be in a superposition of basis states is a fundamental property that distinguishes them from classical systems. A frequently encountered superposition state is the uniform superposition:

$$|\Psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\,.$$

In this case, $|\Psi\rangle$ is simultaneously in both basis states.

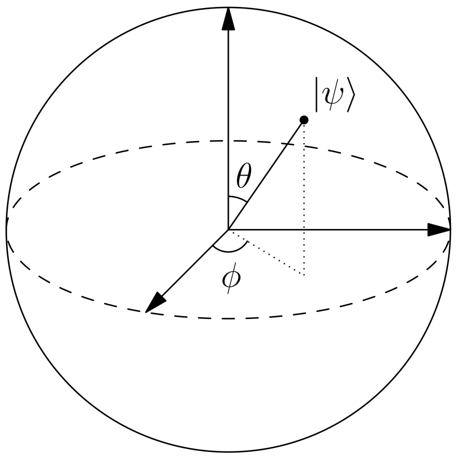


Figure 2.1: The Bloch-sphere: a point on the Bloch-sphere is a visual representation of a qubit state, source: [69]

Alternatively, the state of a qubit can be visually represented as a point on the surface of a sphere with unit radius, known as the Bloch sphere (see Figure 2.1). On the Bloch sphere, the poles correspond to the basis states $|0\rangle$ and $|1\rangle$. Any qubit state can be uniquely specified by two angles, φ and θ. Using these angles, a qubit state can be expressed as follows:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle \quad \text{where} \quad \theta, \phi \in \mathbb{R}$$

The Bloch sphere representation highlights several key properties of qubits. First, it shows that a qubit effectively has only two degrees of freedom, corresponding to the angles φ and θ. This contrasts with the initial representation using two complex numbers (four degrees of freedom), where one degree of freedom is constrained by the normalization condition and another is eliminated because the global phase of a qubit is not observable and can thus be ignored (see Section 2.1.6). Furthermore, the Bloch sphere provides an intuitive graphical understanding of quantum measurements, which will be discussed later (see Section 2.1.6). However, the Bloch sphere has a significant limitation: it cannot be generalized to represent systems of more than one qubit, a topic that will be addressed in the next section.

2.1.4 Multi-Qubit States

The concept of a qubit as a point in a 2-dimensional Hilbert space can be extended to multiple qubits, where the appropriate Hilbert space has 2^q dimensions, with q being the number of qubits. The state of a system comprising q qubits is represented in this higher-dimensional space. For a system with two qubits $|\phi\rangle$ and $|\psi\rangle$, the overall state is described by the tensor product of the individual qubit states, resulting in a 4-dimensional vector:

$$\begin{pmatrix}\phi_1 & \phi_2\end{pmatrix}^T \otimes \begin{pmatrix}\psi_1 & \psi_2\end{pmatrix}^T = \begin{pmatrix}\phi_1\psi_1 & \phi_1\psi_2 & \phi_2\psi_1 & \phi_2\psi_2\end{pmatrix}^T .$$

The normalization constraint still applies, ensuring that $|\,|\phi\rangle\,| = 1$ for any valid quantum state. However, any appropriately sized vector meeting this criterion is a valid quantum state. This notion of quantum state is what we will consider for the rest of this thesis:

Definition 2.1 (Quantum States)**.** *A quantum state is any complex valued vector $q \in \mathbb{C}^n$ such that two conditions hold:*

1. $\|q\|_2 = 1$
2. $n = 2^N$ *with* $N \in \mathbb{N}$

More specifically, we call such a state an N-qubit state.

This reveals a key property of quantum systems: they can be in more states than the sum of their subsystems' possible combinations. Consequently, not all n-qubit states can be expressed as the tensor product of n independent qubit states.

For instance, consider the two-qubit state $|\Phi^+\rangle$ (a so-called Bell state):

$$\left|\Phi^+\right\rangle = \frac{1}{\sqrt{2}}\,|00\rangle + \frac{1}{\sqrt{2}}\,|11\rangle = \frac{1}{\sqrt{2}}(|0\rangle\otimes|0\rangle) + \frac{1}{\sqrt{2}}(|1\rangle\otimes|1\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 0 & 0 & 1\end{pmatrix}^T$$

This state indicates that both qubits are either in state $|0\rangle$ or $|1\rangle$. There are no single-qubit states $|\Phi_1\rangle$ and $|\Phi_2\rangle$ such that $|\Phi^+\rangle = |\Phi_1\rangle \otimes |\Phi_2\rangle$. This is because the property that both qubits share the same value cannot be encoded in individual qubit states, making such a state inseparable, or "entangled."

Entanglement introduces the concept that changes to a quantum system are not local. In classical systems, altering the value of one bit does not affect the others. In contrast, for quantum systems with entangled states, changes to one part of the system can influence others, reflecting the coupling between qubits.

The state vector of a quantum system with q qubits grows exponentially, with the number of qubits q. This exponential growth renders even systems with a relatively small number of qubits impractical to store and simulate on classical computers. For example, a system with 500 qubits would have a state vector of size 2^{500}, exceeding the estimated number of atoms in the universe, and thus cannot be handled by any classical computer.

2.1.5 State Transitions

Quantum state transitions are described by unitary transformations, which are represented by square matrices of size $n = 2^N$, where N is the number of qubits. These transformations, often referred to as quantum gates (see Figure 2.2), play a central role in quantum computing. The application of a single gate U to a quantum state $|\phi\rangle$ is mathematically described by the multiplication of $|\phi\rangle$ with the corresponding unitary matrix U:

$$U|\phi\rangle = \begin{pmatrix} U_{1,1} & U_{1,2} & \cdots & U_{1,n} \\ U_{2,1} & U_{2,2} & \cdots & U_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ U_{n,1} & U_{n,2} & \cdots & U_{n,n} \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_n \end{pmatrix}$$

We formally define a quantum gate:

Definition 2.2 (Quantum gate). *A quantum gate is a complex valued matrix $G \in \mathbb{C}^{n \times n}$ such that:*

1. $n = 2^N$ *with* $N \in \mathbb{N}$
2. G *is unitary:* $GG^\dagger = G^\dagger G = I$

More specifically, we call G an N-qubit gate.

In this definition $G^\dagger$ is the conjugate transpose of G. Consequently, we are now able to define a state transition of quantum systems:

Definition 2.3 (Quantum state transitions). *A state transition in a quantum system in state $q \in C^n$ is described by a quantum gate $G \in \mathbb{C}^{n \times n}$ and the subsequent state q' is given by $q' = Gq$.*

Although transformations can theoretically involve any number of qubits, most practical quantum algorithms utilize compositions of a limited set of typical transformations that affect only one or two qubits at a time. This trend is likely to continue, as quantum computers are designed to efficiently perform these widely used transformations. Moreover, any quantum gate can be approximated to arbitrary precision using combinations of gates from a universal set, which can be as small as two gates [3]. Figure 2.2 illustrates some of the most commonly used quantum gates.

The X-gate, also known as the NOT-gate, performs a bit-flip operation on a qubit. It inverts the state of the qubit, transforming $|0\rangle$ to $|1\rangle$ and vice versa. Mathematically, this is represented by the Pauli-X matrix. For example, applying the X-gate to the state $|0\rangle$ yields:

$$X|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

The CX-gate, or Controlled-NOT gate, is a two-qubit gate that applies the X-gate to the target qubit if and only if the control qubit is in the state $|1\rangle$. Otherwise, the target qubit remains unchanged. This gate can create entanglement between qubits, which we established as one of the crucial resources for quantum computation.

The H-gate, or Hadamard gate, creates superposition by transforming the basis states $|0\rangle$ and $|1\rangle$ into equal superpositions of these states. Applying the H-gate to the state $|0\rangle$ results in:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle$$

The Z-gate, or phase-flip gate, performs a 180° rotation around the z-axis of the Bloch sphere. It leaves the $|0\rangle$ state unchanged and inverts the phase of the $|1\rangle$ state. Thus, applying the Z-gate to a qubit in the state $|1\rangle$ results in:

$$Z|1\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} = -|1\rangle$$

Applying an m-qubit gate to an n-qubit system (where $m < n$) consists of applying the m-qubit gate to the selected qubits and the identity operation to the remaining qubits. For instance, applying an arbitrary gate U to the second qubit in a two-qubit system is equivalent to applying the unitary operation $U' = I \otimes U$ to the system.

Controlled gates

An important subset of multiqubit gates are controlled gates. Controlled gates are characterized by the fact that they conditionally apply a unitary based on the state of one or multiple control qubits. Typically, the operation is applied only when the control qubits are in a specific state, usually $|1\rangle$. The target qubits are the qubits that the gate acts upon if the control qubits are in the required state.

One of the most common controlled gates is the Controlled-NOT (CX) gate. It flips the state of the target qubit if the control qubit is $|1\rangle$. The operation can be described by the transformation:

$$\mathrm{CX}(|c\rangle\,|t\rangle) = |c\rangle\,(|t \oplus c\rangle),$$

where c is the control qubit, t is the target qubit, and $\oplus$ denotes the XOR operation. Other common controlled gates include the Controlled-Z (CZ) gate, which applies a phase flip (Z gate) to the target qubit if the control qubit is $|1\rangle$, and the Toffoli gate (CCX), which is a three-qubit gate where two qubits are control qubits, and the third is the target qubit. The target qubit is flipped if both control qubits are $|1\rangle$.

More generally, a controlled-U gate applies an arbitrary unitary operation U to the target qubit(s) depending on the state of the control qubit(s). If the control qubit is in the state $|1\rangle$, the operation U is applied to the target qubits; otherwise, the identity operation is applied. The mathematical representation of a controlled-U gate acting on a control qubit $|c\rangle$ and a target qubit $|t\rangle$ is:

$$\text{Controlled-}U(|c\rangle\,|t\rangle) = |c\rangle\,(U^c\,|t\rangle),$$

where U^c is U if $c = 1$ and the identity operation I if $c = 0$.

Controlled gates are crucial for implementing quantum algorithms, as they enable conditional operations and entanglement between qubits, which are essential for quantum computation and quantum error correction. As such controlled gates are presented in all well-known quantum algorithms e.g. Shor, Deutsch-Jozsa, HHL.

Example

As an illustrative example, we demonstrate the creation of a Bell state, specifically $|\Phi^+\rangle$, starting from a computational basis state. This transformation can be accomplished in a two-qubit system by first applying a Hadamard gate to the first qubit, followed by a Controlled-NOT (CX) gate. The detailed steps of this process are as follows:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad P = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

$$H = \tfrac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.2: Some widely used quantum gates

$$CX(H \otimes I)\,|00\rangle =$$

$$CX \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} =$$

$$CX \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

The application of the Hadamard gate to the first qubit places it in a uniform superposition. Consequently, the control qubit of the CX-gate is in a state that is neither $|0\rangle$ nor $|1\rangle$, but rather a superposition of both. This means that the CX-gate simultaneously flips and does not flip the target qubit, resulting in a quantum state where the target qubit's value depends on the superposition state of the control qubit. Specifically, if the control qubit is $|1\rangle$, the target qubit is flipped to $|1\rangle$; if the control qubit is $|0\rangle$, the target qubit remains $|0\rangle$. Thus, the final state is a superposition where the control qubit is $|1\rangle$ and the target qubit is $|1\rangle$, or the control qubit is $|0\rangle$ and the target qubit remains $|0\rangle$.

2.1.6 Measurements

The last fundamental difference between classical and quantum computing is the need for measurements to observe the current state of the computation. In classical computing, observing a state is as easy as reading out the appropriate memory location. This is a very basic operation and does not affect the memory at all. In contrast to that, observing the state of a qubit requires conducting a measurement. Such a measurement has two effects: It yields a classical bit as a result (depending on the state of the qubit), and it changes the state of the qubit to the measured basis state. Although it is theoretically possible to conduct measurements in arbitrary bases, in this work we only consider measurements in the computational basis. That is, after a measurement, the measured qubit collapses into either state $|0\rangle$ or state $|1\rangle$. For a qubit in state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, the probability of measuring $|0\rangle$ resp. $|1\rangle$ is $|\alpha|^2$ resp. $|\beta|^2$. We already mentioned that for the case of one qubit, a measurement can be visually represented on the Bloch sphere as a projection onto the z-axis. This directly shows that there are infinitely many different states that lead to the same measurement results. As a simple example, consider the two states $|\phi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Both of those states yield 0 and 1 with equal probability and yet clearly are different states. In a multi-qubit system, the resulting state is additionally normalized. Take as an example a 2-qubit state, $|\Phi\rangle = a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle$. The probability of observing 0 when measuring the first qubit in this state is $p_0 = |a|^2 + |b|^2$ and the resulting state of the measurement is

$$|\Phi'\rangle = \frac{a|00\rangle + b|01\rangle}{\sqrt{|a|^2 + |b|^2}}$$

correspondingly, the probability of observing 1 when measuring the first qubit is $p_1 = |c|^2 + |d|^2$ and the resulting state of the measurement is

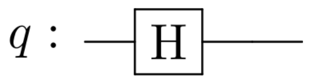
$$|\Phi'\rangle = \frac{c|10\rangle + d|11\rangle}{\sqrt{|c|^2 + |d|^2}}$$

The effect of the collapse of the state is especially interesting when measuring qubits that are entangled. Consider again the Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Measuring either qubit in this state yields $|0\rangle$ and $|1\rangle$ with 50% probability. However, measuring also collapses the state into the observed ground state such that a subsequent measurement of the other qubit always returns the same result as the previous measurement. Thus the measurement does not only change the value of the measured qubit but of the entire system and thus of the second qubit as well.

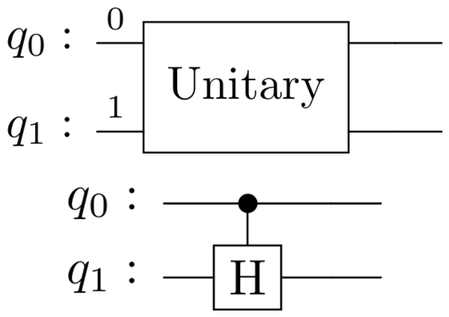
The fact that not all different states result in different measurements is best explained by the notions of relative and global phase. According to Euler, any complex number z can be rewritten as follows: $z = re^{i\theta}$ where $r = |z|$ and $\theta = arg(z)$ (the argument of z). Using this, we can represent a qubit state accordingly: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = r_1 e^{i\theta_1}|0\rangle + r_2 e^{i\theta_2} = e^{i\theta_1}(r_1|0\rangle + e^{i(\theta_2-\theta_1)}|1\rangle)$. Here, $e^{i\theta_1}$ is called global phase while $e^{i(\theta_2-\theta_1)}$ is called relative phase. The global phase is a non-observable quantity as there is no way to measure the difference between two quantum states only differing in global phase. This is explained by the fact that a quantum state can only be observed through measurement. However, as $|e^{i\theta_1}| = 1$ for any θ_1 the global phase never contributes to any measurement. Conversely, the relative phase can be observed if the system is evolving. Consider again the two states $|\phi\rangle$ and $|\psi\rangle$ which only differ in relative phase, namely $|\phi\rangle = \frac{1}{\sqrt{2}}(1\,|0\rangle + 1e^{2i\pi}\,|1\rangle)$ and $|\psi\rangle = \frac{1}{\sqrt{2}}(1\,|0\rangle + 1e^{i\pi}\,|1\rangle)$. If a Hadamard gate is applied to both of these states, then they evolve to two different basis states, despite being indistinguishable before: $H\,|\phi\rangle = |0\rangle$ and $H\,|\psi\rangle = |1\rangle$ thus a difference in relative phase is indirectly measurable and we have to treat states with different relative phase as different states.

2.1.7 Quantum circuits

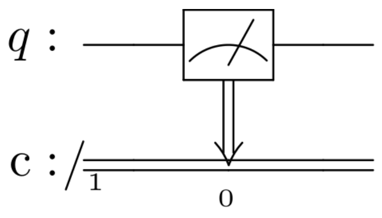
The most basic approach to programming quantum software is in the circuit model. This style of programming is inspired by classical circuits, in which logic gates (like NOT, AND or OR) are applied to wires. These two fundamental elements (gates and wires) are similarly used in quantum circuits, where for each qubit a wire is used and gates can be applied to those wires. We already introduced the most common gates in 2.2. In a circuit, those gates are typically represented by a square box with a corresponding label (e.g. H for Hadamard gate):



Multiqubit gates are either drawn over multiple wires. Or, in the special case of controlled gates, the control-qubit is marked with a special connector (while the controlled gate is depicted on the target wire).



Measurements are depicted with a meter symbol. Sometimes an additional arrow indicates to which classical wire the result of the measurement is stored:



Quantum circuits are read from left to right. The quantum gates are applied in the depicted order to qubits as indicated by their positioning on the wires. If no other state is given for a circuit we assume that the computation always starts in the state $|0...0\rangle$.

As an example, consider again the creation of the Bell state $|\Phi^+\rangle$. As shown before, this state can be obtained by first applying a Hadamard gate to a qubit and then applying a CNOT gate for which the control qubit is the same qubit. The corresponding circuit is shown in Figure 2.3. In this circuit, measurements are added at the end of the circuit which would result in measuring either $|00\rangle$ or $|11\rangle$ with 50% probability each.

In Figure 2.4 the circuit for a slightly more complicated quantum algorithm is shown. This algorithm is called quantum teleportation. The goal of this algorithm is to "send" a quantum state between two parties given that those parties share a Bell state. For this algorithm, we assume that qubit q_0 is in an arbitrary state $|\Psi\rangle$ which we would like to send to some other party. As a first step, which is often considered to be done before the actual algorithm, we create the Bell state $|\Phi^+\rangle$ with qubits q_1 and q_2. Now assume party A has control over the first qubits while party B is somewhere distant and has control over qubit q_2. In order to "transmit" the state $|\Psi\rangle$ of qubit q_0 A first applies a CNOT gate to q_0 and q_1 and then a Hadamard gate to q_0. A proceeds to measure both its qubits and then sends B the measurement results. Note that this is a transmission of 2 classical rather than quantum bits. B can then restore the state $|\Psi\rangle$ by applying two gates to q_2 conditioned on the classical bits A has sent. Specifically, B first applies an X gate to q_2 if the measurement of q_0 gave 1 and then applies a Z gate to q_2 if the measurement of q_1 gave 1. After that q_2 is now in state $|\Psi\rangle$.

2.1.8 Quantum Programming Languages

While the era of writing classical programs at the circuit level has long since passed, the same cannot yet be said for quantum programs. Currently,

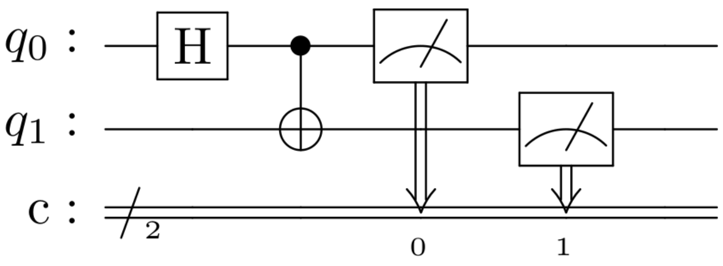


Figure 2.3: A circuit to create Bell state $|\Phi^+\rangle$

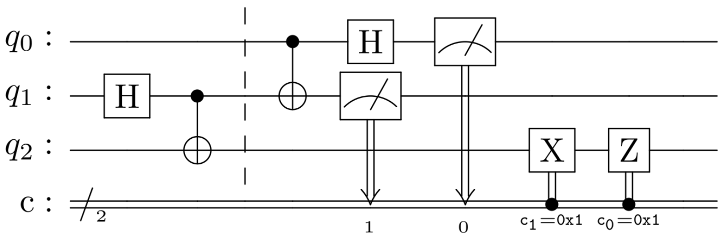


Figure 2.4: A circuit realizing the quantum teleportation algorithm

the predominant method for writing quantum programs remains circuit-level programming, where individual quantum gates are explicitly placed. Although various frameworks and libraries now offer higher-level functions for implementing common patterns—such as state preparation or quantum Fourier transformations as reusable subroutines—a substantial portion of quantum programming is still performed at the circuit level.

Typically, quantum programs are written using libraries within well-established programming languages. The most prevalent examples, such as Qiskit and Cirq, are based on Python. These frameworks operate in similar ways, centering around a circuit object to which required gates are applied via specific method calls.

For instance, consider the two implementations of the Deutsch-Jozsa algorithm shown in Listing 2.1. The algorithm is implemented once in Cirq (upper part) and once in Qiskit (lower part). Both implementations are structurally similar: the process begins with initializing a circuit object with the required number of qubits (here, n). Subsequently, gates are applied by invoking functions on the circuit object. In Cirq, gates are appended to the circuit object, while in Qiskit, the gate functions are directly called on the circuit object. Finally, measurements are applied to the qubits.

Although this similarity is demonstrated here using two frameworks, it holds for several others, including Braket and Q#. In summary, while each framework offers enhancements tailored to its specific platform, circuit-level programming remains the current standard for developing quantum software.

```python
input_qubits = [cirq.LineQubit(i) for i in range(n)]
auxiliary_qubit = cirq.LineQubit(n)
circuit = cirq.Circuit()
circuit.append(cirq.X(auxiliary_qubit))
circuit.append(cirq.H(q) for q in input_qubits + [
    auxiliary_qubit])
# Apply the oracle
circuit.append(cirq.H(q) for q in input_qubits)
circuit.append(cirq.measure(*input_qubits, key="result"))
```

```python
qc = QuantumCircuit(n + 1, n)
qc.x(n)
for qubit in range(n + 1):
    qc.h(qubit)
# Apply the oracle
for qubit in range(n):
    qc.h(qubit)
qc.measure(range(n), range(n))
```

Listing 2.1: Example implementations of the Deutsch-Jozsa algorithm in two different quantum programming frameworks: Cirq (top) and Qiskit (bottom)

2.1.9 Noise in NISQ-era Quantum Devices

The term "NISQ-era" stands for "Noisy Intermediate-Scale Quantum" era, a concept introduced by John Preskill [115] in 2018 to describe the current phase of quantum computing technology. This era is characterized by the availability of quantum processors with a sufficient number of qubits to perform non-trivial computations, but which are still significantly affected by noise and errors.

More specifically, *Intermediate-Scale* is normally considered to encompass devices with tens to a few hundred qubits. These devices have more qubits than early experimental setups, enabling more complex computations. However, they are still far from the large-scale quantum computers envisioned for the future, which will likely need millions of qubits.

The physical realization of NISQ devices can vary widely, encompassing technologies such as superconducting qubits, trapped ions, and photonic systems. While the question of which physical implementation will ultimately prove most effective remains unresolved, this thesis does not focus on that determination. Instead, a key aspect relevant to several chapters is that all NISQ devices, irrespective of their specific realization, are inherently prone to various errors that impact the execution of quantum circuits. Although the severity and nature of these errors can vary depending on the underlying technology, they can be broadly categorized into a few common types:

Decoherence Decoherence refers to the loss of quantum coherence in qubits due to their interaction with the environment (the degeneration of a state over time). It leads to the decay of superposition states into classical mixtures. Decoherence manifests in two main forms:

- **Dephasing (Phase Damping):** This type of noise causes the relative phase between the components of a superposition state to decay.
- **Amplitude Damping:** This noise process results in the loss of energy from the qubit to its surroundings, causing the state to decay from $|1\rangle$ to $|0\rangle$.

Gate Errors Gate errors occur during the implementation of quantum gates and can arise from various imperfections in the control pulses used to manipulate qubits. Types of gate errors include:

- **Overrotation and Underrotation:** These errors occur when a gate operation does not rotate the qubit state by the intended angle, leading to incorrect final states.
- **Cross-talk:** Cross-talk happens when the control signals intended for one qubit inadvertently affect neighboring qubits, causing unintended operations.
- **Systematic Errors:** These errors result from consistent inaccuracies in the calibration of quantum gates, leading to biased outcomes.

Measurement Errors Measurement errors occur when the readout process of a qubit's state produces incorrect results. This essentially means that the probability that a qubit is in the state $|0\rangle$ is incorrectly measured as $|1\rangle$ and vice versa. This error arises from imperfections in the measurement apparatus.

In the remainder of this work, we will not differentiate between those errors and simply refer to them as noise.

2.2 Java Specification and Verification

We briefly introduce the Java Modeling language and the concept of bounded model checking as preliminaries for Chapter 6.

2.2.1 Java Modeling Language

The Java Modeling Language (JML) is a specification language for Java based on first-order logic. Fundamentally, it follows the design-by-contract paradigm. JML allows one to specify, among other things, pre-, post-, and frame-conditions for Java programs. JML specifications are embedded directly within the source code as special comments, allowing specifications to be added to any codebase without altering program behavior or requiring additional files. Each JML method specification, also called contract, comprises potentially several JML clauses specifying the expected behavior of the method. Intuitively, a JML method contract has the following semantics: If a method is executed in a state where the precondition holds, then the method's execution terminates (without exception) in a state that satisfies the postcondition, with only those heap locations explicitly allowed by the frame-conditions being modified. In other words, the precondition defines the context in which the contract must hold, while the postcondition specifies the guarantees provided after the method's execution. JML offers additional clauses and constructs to specify different behaviors for execution with and without exceptions, termination conditions for recursive functions, invariants for objects and classes, among others. For this thesis, pre-, post-, and frame-conditions are sufficient to specify the case studies considered.

Moreover, JML supports auxiliary specifications, including loop invariants. In this work, we are particularly interested in specifying loop invariants, which define conditions that must hold before and after each loop execution. Some tools rely on loop invariants to be able to prove the correctness of loops in an inductive manner.

A simple example of a Java class specified with JML is provided in Listing 2.2, illustrating the most relevant JML features for this thesis. This example implements a basic list, where the list elements are stored in an array and the list's current size is tracked by a separate integer variable. The list includes two methods, one for adding elements (`add`) and another for checking if a specified value is present in the list (`contains`). Several key features of JML are employed in these methods:

First, preconditions for method contracts are used, beginning with the keyword `requires`. An example of a precondition appears on line 18 (`size < elements.length`), ensuring that the `add` method is only called when the list has not yet reached its maximum capacity, thereby preventing an index-out-of-bounds error. The postcondition, specified with the keyword `ensures`, guarantees that upon method execution, the given value is stored at the end of the current list.

Additionally, one of JML's special operators, `\old(·)`, is used in the second postcondition on line 20. This keyword references variable values from the state prior to method execution. In this case, it specifies that adding a value should increase the list's size by one compared to its size before method execution. Another postcondition, in this case for the `contains` method (`\result == (\exists int i; 0 <= i && i < size; elements[i] == value);` on line 29) guarantees that the method returns `true` if, and only if, there exists an index `i` such that `elements[i]` equals the specified `value`. This postcondition precisely captures the intended behavior of the `contains` method, i.e., checking whether the `value` is present in the array. This demonstrates the use of quantified expressions, specifically an existential quantifier, in specifications. Additionally, both methods include assignable clauses (e.g. `assignable \nothing;` on line 30), asserting that the method does only change certain heap locations.

The body of the `contains` method includes further specification of the loop using loop invariants (lines 34 and 35). The first invariant captures the valid range of the loop variable, while the second invariant resembles the postcondition of the method, a common pattern since the postcondition must be provable after the final loop iteration. Apart from the invariant and the assignable clause, the loop specification contains a decreases clause (line 36), essential for proving termination. The `decreases`-clause requires the specified expression to decrease with each loop iteration while remaining above 0. In this case, `size - i` meets these criteria, ensuring loop termination.

In addition to method specifications, the example illustrates the use of a class invariant. Class invariants are Boolean expressions that must hold in every publicly observable state of an instance of the class. In this example, the array of elements is required to never be `null` (line 6), and the size variable must reflect a valid size of the list according to the array's limits (lines 7 and 8).

JML offers many more features, but the ones presented in this example are sufficient for all the case studies considered in this work.

```
public class List {
    private /*@ spec_public*/ int[] elements;
    private /*@ spec_public*/ int size;

    //@ public invariant elements != null;
    //@ public invariant size >= 0;
    //@ public invariant size <= elements.length;

    //@ requires capacity > 0;
    //@ ensures size == 0;
    public List(int capacity) {
        elements = new int[capacity];
        size = 0;
    }

    /*@ requires size < elements.length;
      @ ensures elements[size - 1] == value;
      @ ensures size == \old(size) + 1;
      @ assignable elements[size];
      @*/
    public void add(int value) {
        elements[size] = value;
        size++;
    }

    /*@ ensures \result == (\exists int i; 0 <= i && i < size; elements[i] == value);
      @ assignable \nothing;
      @*/
    public boolean contains(int value) {
        /*@ loop_invariant 0 <= i && i <= size;
          @ loop_invariant (\forall int j; 0 <= j && j < i; elements[j] != value);
          @ decreases size - i;
          @ assignable \nothing;
          @*/
        for (int i = 0; i < size; i++) {
            if (elements[i] == value) {
                return true;
            }
        }
        return false;
    }
}
```

Listing 2.2: A simple example illustrating the specification language JML

2.2.2 Bounded Model Checking (BMC)

Bounded Model Checking (BMC) is a verification technique used to check the correctness of software programs by systematically exploring their state space up to a given bound. Instead of attempting to prove the correctness of a program for all possible inputs and execution paths, BMC limits the exploration to a finite number of steps, referred to as the bound. This approach encodes both the program and its specifications (e.g., assertions or safety properties) into a logical formula, which is then checked for satisfiability using a SAT or SMT solver. If the formula is satisfiable, the solver can provide a counterexample demonstrating a violation of the specification within the bound. If no violation is found, BMC provides the assurance that the program behaves correctly up to the specified bound, though it does not guarantee correctness beyond that limit.

Formally, the problem of BMC can be described as follows (e.g. found in [39]). The system under investigation is modeled as a Kripke structure. A Kripke structure is a triple $K =< S, R, L >$ where S is a finite set of states, L is a labeling function and $R \subset S \times S$ is a set of transitions. L assigns each state a set of atomic propositions A: $L : S \mapsto 2^A$. In other words, for each state $s \in S$, $L(s)$ describes the set of atomic propositions that are true in that state. Using this notion, a model checker can be defined as a decision procedure for $K \vDash \phi$ where K is a Kripke structure and ϕ is a temporal logic formula. If $K \vDash \phi$, then the system has property ϕ. Otherwise, a counterexample can be generated.

To practically implement this decision procedure, it is often modeled as a satisfiability problem. Bounded model checking is the idea of searching for a program run with maximum length k, that violates a given property. By restricting the maximum length of a run it becomes possible to encode this problem as a formula in propositional logic. Consider as an example the property `G p` given in LTL establishing p as a global invariant. Given propositional predicates $I(s)$, which evaluate to true if $s \in I$ and $R(s_1, s_2)$, which evaluates to true if $(s_1, s_2) \in R$, we can construct the following formula [23]:

$$\exists s_0, s_1, \ldots, s_k : I(s_0) \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \neg p(s_k)$$

This formula is satisfiable if and only if there is a run of length k which violates the property. With quantifier elimination and a Tseitin-Transformation this formula can be transformed into conjunctive normal form which is the standard input format for most SAT-solvers [23].

For the concrete application of this approach to programs, several intermediate steps are necessary. The approach outlined here is derived from CBMC [37], a bounded model checker for C. Initially, all loop constructs are transformed into equivalent `while` loops. Subsequently, functions are inlined, with recursive function calls inlined n times, where n is the bound. Loops are then unwound by copying the loop body n times, each copy guarded by an `if` statement with the same condition as the original `while` loop. All remaining `goto` statements pointing backward are unwound in a similar manner. At this stage, the program consists only of assignments and potentially forward-pointing branching `goto` statements. Finally, the program is transformed into static single assignment (SSA) form [5], enabling translation into a SAT problem as described above.

Although the described transformation was originally developed for C programs, it is also suitable for Java programs with additional preprocessing to handle exceptions and polymorphism [42]. The bounded model checker JBMC implements this approach, facilitating Java program verification. JBMC supports verifying general properties, such as the absence of null-pointer exceptions, as well as user-provided assertions.

Building on JBMC as a backend, we developed a tool capable of translating JML annotations into a pure Java program extended only by assumptions and assertions interpretable by JBMC [18]. Using this approach, we can apply JBMC to JML-annotated Java programs (see Section 6.7 for details).

Bounded model checking is inherently incomplete, as it searches for errors in program runs only up to the specified bound. Errors occurring in longer runs remain undetected, so programs with such errors may not be proven correct through bounded model checking. However, this lack of completeness is balanced by BMC's high degree of automation (the SAT translation is automated, and SAT solvers require no user interaction).

Chapter 3

Similarities Between Non-Classical Systems Regarding Verification

In the upcoming chapters, we will focus primarily on the analysis of quantum systems. However, some of our findings can be applied to other non-classical systems as well. In this chapter, we examine the similarity of non-classical systems in the context of formal verification using quantum computing systems and machine learning systems as examples. We argue that these two types of non-classical systems are particularly similar, but parallels can also be drawn with other types of non-classical systems.

While classical software systems adhere to deterministic, sequential execution models, quantum computing and ML software fundamentally deviate from these established norms, presenting unique challenges. At the same time, the inherent complexity of those two approaches and their unintuitive nature calls for rigorous quality control, thus providing a prime target for formal verification. This chapter explores the intricate interplay between the distinct programming paradigms of quantum computing, ML, and classical software, shedding light on the profound implications for formal verification practices.

Classical programming languages have long thrived on the principles of modularity, determinism, and explicit control flow, enabling developers to construct and reason about software systems through a series of well-defined steps. Verification techniques for classical software have thus traditionally relied on decomposing complex systems into smaller, more manageable components, facilitating comprehensive analysis and logical reasoning. However, the advent of quantum computing and ML disrupts these established norms, introducing a new era of computing characterized by non-classical behaviors.

In the following, we primarily consider neural networks as the most prominent representatives of ML. Certain challenges arise specifically from properties unique to neural networks, while others are generally applicable to any form of ML. For quantum computing, we do not assume any particular architecture or programming language, as all identified challenges are agnostic to these specifics.

Though fundamentally different at first glance, the challenges for the formal verification of quantum and ML software are surprisingly similar. We identified seven challenges that those two fields have in common, which we discuss in more detail now.

The remaining chapter is structured as follows: In the following section, we discuss each challenge in depth. We then elaborate on the opportunities and chances we identified through those challenges in section 3.2 before we conclude in section 3.3.

3.1 The Shared Challenges of ML and Quantum Software Verification

Fundamental Differences in the Programming Paradigm The mere fact that both quantum computing and ML software are at their core fundamentally different from classical software is a similarity in and of itself. This is especially true in terms of program semantics. While the semantics and behavior of a classical software program are directly specified by the code, we can observe a semantic shift in ML and quantum software, where code no longer is the only defining factor for the program's semantics. In ML, the semantics or behavior of a model is rather conveyed by the training data. This is even more drastic in quantum programs, where the true intention of the program is hidden behind the fundamental principles of quantum mechanics (i.e., superposition, entanglement, and measurement).

Moreover, classical programming languages and software naturally share a lot of properties from the way they are programmed over their fundamental workings up to the type of hardware they are running on. This allowed developers of verification tools to use the same backend (and theoretical foundation) for several different tools (e.g. CBMC [38], JBMC [43] and EBMC [109] as verification tools for C, Java and System Verilog respectively all based on the same backend). A similar slight adaptation of established approaches is no longer suitable for quantum computing and ML systems, leading to the fact that well-established principles on how to design a new verification tool that have worked for different types of classical software do not apply to neither quantum software nor ML-based software. Thus new methods have to be developed that deal with those new computing paradigms, or else the approaches are most likely to suffer from severe scaling issues cf. Chapter 6.

Tied to this fundamental difference in the programming paradigm is the unintuitive nature of those types of software. Quantum physics and thus quantum software is notoriously unintuitive for humans and thus hard to understand. This is unsurprising as even the most fundamental concepts of quantum physics like superposition and entanglement contradict our everyday experience of classical physics. Similarly, the calculation of a prediction of ML systems implies many internal calculation steps, especially for deep neural networks, making it unintuitive or impossible for users to understand the mapping of the input to the prediction result [106]. For a neural network that is described by a vast number of weights, no human is able to only look at these weights and tell what type of task this network is doing, let alone if it is doing this task well. In contrast, a well-written classical program can be read and understood by software engineers without prior knowledge of the code. Adding to the fundamentally unintuitive nature of such systems are effects like the tuning of hyper-parameters, the shaping of the reward function [136], or the effect of the selection of training data for the outcome.

This unintuitive nature and the fact that there are fundamental differences in the programming paradigm in relation to classical software not only constitute a similarity between quantum and ML software but also represent a convincing reason why those types of software are a prime target for formal verification. The more complex and unintuitive a software system becomes the less likely it is that errors are spotted by human developers and thus the more important it is to have formal guarantees that prove the absence of such errors. Unfortunately, for the same reasons designing verification tools for ML or quantum software is especially challenging. We mentioned that established approaches for classical software are not easily transferable to neither quantum nor ML software. Thus completely new paradigms have to be developed in order to deal with the unique challenges that come with those types of software.

Linear Algebra as the Mathematical Framework For this section, we focus specifically on neural networks (and to some extent even feedforward neural networks only). Neural networks and quantum computing have in common that their fundamental working is based on principles of linear algebra (and especially reliant on matrix multiplications). Quantum states are represented as normalized n-dimensional vectors in a complex-valued vector space. Quantum operations (often called quantum gates) are represented by unitary matrices in the same Hilbert space. Thus the application of a quantum gate to a given state is the corresponding matrix multiplication of the gate with the state. Measurements can be represented by projections on suitable subspaces (depending on the basis the measurement is conducted in). So the semantics of quantum computing is very closely tied to fundamental linear algebra.

Similarly, neural networks have a surprisingly similar structure. The output of each layer can be represented as a vector. Computing the output of the next layers can be represented as a matrix multiplication where the elements of the matrix are the appropriate weights of the neural network. This is then normally combined with an activation function which, admittedly, breaks the linearity. More formally the output of the i-th layer in a feedforward neural network can be iteratively computed as:

$$\begin{aligned} \hat{x}_{i+1} &= W_{i+1} x_i + b_{i+1}, \\ x_i &= \sigma(\hat{x}_i), \end{aligned}$$

where W_{i+1} is the weight matrix, b_{i+1} is the bias vector, and σ is the activation function.

Interestingly, the evolution of a quantum state through a series of gates can also be represented in this form. Here, W_i represents the matrix associated with the i-th quantum gate, b_{i+1} is the zero-vector, and the activation function is simply the identity. In this way, the operational semantics of quantum circuits and feedforward neural networks exhibit a striking similarity.

Consequently, verification tools tackling either neural networks or quantum circuits have to provide means to reason efficiently about linear algebra or at least matrix multiplication. Research in neural network verification has therefore focused on abstract domains that can efficiently handle matrix multiplication. Prominent examples include symbolic intervals [152, 134], zonotopes [135] or star sets [12]. A linear transformation of the values represented by these domains reduces to matrix multiplications of the abstract domain's coefficients. While the non-linear activation functions require linear relaxations and thus some precision is lost, no overapproximation is incurred for linear operations.

Since both neural networks and quantum computing heavily rely on matrix multiplications, adapting approaches that proved successful in neural network verification might also be beneficial for verifying quantum circuits. Advances in handling linear transformations and encoding non-linearities have the potential to improve the performance of verifiers in either of the two fields.

Missing Modularity Neural networks are notoriously famous for the lack of explainability, which is to a large extent based on the fact that the intermediate "states" (e.g. the outputs of hidden layers) are normally very unintuitive for humans. This is in great contrast to most classical software where the result is computed in a series of human-comprehensible steps. Quantum computing is in that regard similar to neural networks as intermediate states are very unintuitive and computations can not be easily broken down into comprehensible small steps. Note that this does not mean that the mathematical description of those intermediate states is not possible or not even necessarily complex but rather that an intuitive description of them is often hard to find.

As an example, consider three basic programs, one for each paradigm. For classical programming, consider a primitive sorting algorithm that sorts an array of integers by iterating over it and sorting it in the process. Even for such a short piece of code, an essential part of the specification is a loop invariant that may, for example, state that at the i-th iteration of the loop the array is sorted up until the i-th element. This essentially introduces a type of modularity as the algorithm is broken down into single iterations of a loop. In contrast to that consider a simple neural network that is able to classify pictures into two categories: pictures showing a "0" and pictures showing a "1". There is no straightforward way of specifying what the defining property of pictures showing a "1" is. Let alone are there easy means to obtain specifications for the hidden layers of such a network. It is thus very hard to specify the behavior of such a system. As a simple example of a quantum algorithm, consider the Deutsch-Jozsa-algorithm [52]. Similar problems arise for the specification of intermediate states. The algorithm heavily relies on the controlled application of the oracle. And the nature of the resulting state is again very unintuitive. Mathematically it is relatively straightforward to show that the resulting amplitudes either cancel out or add up depending on whether the given function was constant or balanced. However, there is no easy way to break the algorithm down into small steps that are easier to understand.

This challenge of finding specifications for intermediate states may seem like a minor problem at first sight however, for verification this can turn out to be a game-changer for two reasons: First, one of the main approaches to handle complex software is to break down the software into smaller parts (thus hopefully reducing the complexity of each part). This approach, however, relies heavily on the ability to specify how those parts work together. This is exactly what we described as very hard and sometimes impossible for neural networks and quantum circuits. It is thus oftentimes very difficult to break down the verification of such systems into smaller parts as the specification for the interfaces is hard to come by. This is supported by the fact that among the currently most successful verification tools for NNs, none uses a modular approach [29]. There is some work that is trying to tackle this issue (e.g. [71, 153, 93]), however all those approaches do not change the fact that the intermediate states and thus also their specifications are very unintuitive.

Second, the lack of modularization has a direct impact on the type of verification methods and tools that are suitable for a given system. In the described case where modularization is hard, methods and tools that tackle the entire system are naturally more prevalent. This leads to the fact that tools and methods for neural network verification and quantum circuits are similar in this regard.

Exploding State Spaces An additional similarity between quantum computing and neural networks is the fact that they have inherently large state spaces. For neural networks, this is normally due to the very large feature spaces. When considering interesting problems, the input vector has a very high dimensionality (e.g., the pixel space in image recognition, which can have in excess of 10^4 dimensions - e.g. the average resolution of the images in the popular ImageNet data set is on average approx. 400x350 [50] and the images in the HAM10000 data set for skin cancer detection have a size of 800x600 [143]). State space also increases exponentially with the number of neurons (at least in some cases [108]). The number of neurons and the possible combinations of activation values further increase the size of the state space. For quantum computing, this is driven by the fact that the state vector for the quantum system grows exponentially in the number of qubits of the system. Additionally, the state spaces of both systems are often considered continuous rather than discrete, which leads to further extending the state space.

For most automated verification tools, the size of the state space is a crucial indicator of the complexity of the underlying problem. In the case of quantum computing and neural networks, this leads to the fact that real-world problems are often very hard to tackle due to their sheer size. Conversely, methods and approaches are needed (and have been developed) to overcome this explosion of state space. This can be done by several different approaches. Typical ones include sound overapproximations, abstraction, and symbolic approaches. Alternatively, one can assume that the verification of small instances of a problem is sufficient to at least increase the confidence in the correctness of larger instances. This principle is often referred to as the small-scope hypothesis (see Section 6.8).

Use of Floating-Point Arithmetic The verification of floating-point arithmetic is notoriously difficult [92]. This is mainly due to the inherent complexities of approximating real numbers in finite representation, leading to rounding errors, precision loss, and numerical instability, which can significantly impact the correctness and reliability of computations. However, both our subjects of investigation rely heavily on exactly this type of arithmetic. Neural networks use weights that are floating-point values (and thus every matrix multiplication and subsequent operations are floating-point arithmetic). Quantum circuits do not naturally rely on floating-point arithmetic but rather real arithmetic. This can be an advantage as real arithmetic is easier to reason about in some cases (as for example no rounding errors have to be considered and no special cases like `NaN` are present). However, even in the context of quantum circuits, floats can hardly be left out of consideration at least if considering hybrid programs. Since virtually every classical language has no built-in support for reals as a data type, the classical control and execution environment of quantum circuits is relying on floats. This often translates directly to the quantum circuits as e.g. state preparation is based on classical control and thus itself has only finite precision.

Thus verification of quantum circuits and neural networks alike has to be able to deal with floating-point arithmetic. This is a challenge for a lot of verification tools (which is why some approaches ignore it completely [156, 83]). Floating points have several unintuitive and unpleasant properties. These include:

- Rounding: Due to their finite precision, all floating-point operations are potentially subject to rounding. While these errors may be very small in certain cases, they can add up over time and completely distort results in the worst case.

- Special values: Floats introduce special values like NaN (not a number) which have to be accounted for when dealing with them. While there are settings in which such special values can be ruled out, they have to be respected in general.

- Loss of real properties: Some properties that hold for real-valued arithmetic do not hold for floating-point arithmetic. This is not only unintuitive but can significantly limit the ability of solvers to apply known lemmata and thus conduct proofs (e.g. floating-points are neither associative nor distributive).

Designing verification tools that take all these properties into account is very challenging. In Chapter 7 we will discuss this challenge in more detail and present an approach that allows to prove the absence of relevant floating-point errors in certain scenarios.

System Integration Another similarity between the two types of systems lies in their integration within a classical system. As we established before, both quantum and ML software have a different programming paradigm from classical software. However, their integration into a classical software system is almost always a necessity. Quantum software will for the foreseeable future (and probably forever) only solve very specific computing-intensive mathematical problems. There is no reason to believe that tasks like a simple user interface or the storage of data will ever be done on a quantum computer. Thus a complex software system will always be at least to some part classical. Consequently, verification of quantum software has to consider the integration of quantum software into classical software in order to provide meaningful verification results.

The same argument is true of ML-based components. The machine learning component is only a small part of the entire system [130] and will rarely be used in isolation but rather for some very specific part in the software system like the control of an actuator or the recognition of some complex input. Therefore, the appropriate integration of such ML software into overall systems is also the subject of research [141, 140]. As this integration into the rest of the software is so crucial, verification tools have to support this aspect.

As a consequence, we argue that while specialized tools for either quantum or machine learning software are a necessary first step, the integration of these specialized components into the overall software system should always remain a priority. This includes ensuring that the specifications developed for quantum or machine learning components are compatible with the classical parts of the system. This integration poses a particular challenge, especially in light of the next similarity.

Probabilistic Systems One fundamental aspect of quantum software lies in its reliance on measurements to acquire classical results. These measurements introduce a form of nondeterminism, resulting in quantum algorithms producing probability distributions over all possible measurement results.

Similarly, although theoretically deterministic, the majority of ML systems exhibit probabilistic behavior. Tasks such as classification in ML often yield probability distributions over possible classes rather than a single outcome. However, the resulting softmax outputs should not be regarded as the true correctness likelihood. In order to obtain a more reliable confidence assessment, a calibration of the neural networks is usually necessary [74]. Other ML methods such as Bayesian neural networks (BNN), however, provide a direct calculation of the prediction uncertainty, but for most BNNs, the calculation of an intractable integral is required, so that approximations and other uncertainty estimation methods have been explored [67] such as MC-Dropout [66]. Thus, despite originating from different sources, both ML and quantum systems introduce forms of probabilistic behavior, inherently imbuing their application with uncertainty.

Probabilistic/nondeterministic behavior has an impact both on the specification and the verification of systems. Fundamental notions like correctness are affected by the loss of determinism. The conventional definition of correctness, where any execution of the software starting from a state satisfying a precondition Φ ends in a state satisfying a postcondition Ψ, becomes less applicable. Specifying probabilistic systems is not entirely novel, yet both ML and quantum software share a common need for this type of specification.

Moreover, verification tools and approaches must be equipped to handle these novel specifications (e.g. for ML [105] and for quantum [14]). Depending on the verification approach employed, this may necessitate adapting the underlying logic and calculus to accommodate probabilistic formulas and states. Alternatively, it may exacerbate the challenge of state explosion, as a single statement can lead to multiple subsequent states due to the probabilistic nature of the system. As a result, addressing probabilistic behavior in both specification and verification becomes crucial for ensuring the reliability and correctness of quantum and ML systems alike.

3.2 Potential Research Directions

We argue that the similarities we depicted in the last sections lead to the fact that a lot of advances made in the verification of either field may be a potential improvement in the other field. We advocate for close collaboration of the communities between those two fields as we think that this might be very beneficial for either side. In the next few paragraphs, we point out several aspects that we consider especially promising to investigate in order to advance both the verification and specification of quantum as well as ML systems.

3.2.1 Linear algebra libraries

We pointed out how both neural networks, as well as quantum computing, are fundamentally based on concepts rooted in linear algebra. It is thus crucial to have meaningful reasoning capabilities related to linear algebra in order to tackle such systems. Thus it may come as no surprise that some of the earliest verification approaches presented are based on verification frameworks that already provide libraries dedicated to such reasoning (e.g. [32] based on Why3 [61], [25] based on Isabelle [113] and [119] based on Coq [21]).

Following this line of thought, we argue that further improvements in this direction are promising. Extensions of already existing libraries in well-known frameworks like those mentioned above would be one idea. However, bringing comparable lemmata, proof rules, and so forth to highly automated verification tools is also a valuable step toward making them ready to be applied in the context of quantum and ML systems.

Additionally, this applies to the challenge of software relying on floating-point arithmetic. We outlined before why floating-point arithmetic poses a challenge for most verification tools. Thus, any general improvement for verification tools that allows easier/faster/more precise analysis of software containing floating-point numbers can be seen as a step towards the ability to apply these tools to quantum (and ML) software.

3.2.2 Robustness Verification

A huge field of research in the context of ML-based systems is the proof of robustness against adversarial examples. Essentially one would like to show that small perturbations in the input are not leading to completely different results. There are several well-known tools that are able to verify this property for neural networks (e.g. [147] and [86]). We argue that this idea is directly applicable to quantum circuits, thus showing that those circuits are robust with regard to noise. As current quantum computers are heavily influenced by noise, this type of verification is crucial when considering current real-world quantum circuits. The techniques applied for neural networks should be adaptable to quantum circuits based on the similarities we pointed out in the previous sections. It would thus be possible to show that the results of a quantum circuit are stable under the assumption that the noise of the quantum device does not exceed some given threshold.

3.2.3 Verification at System-Level

Another promising research direction relates to the verification of system-level properties. Verifying the correctness of neural networks with large model capacity is a hard problem (in fact, NP-complete [87]). For this reason, the focus of research activities has shifted to verifying properties of the ML-based system itself, rather than verifying properties of a neural network. In such settings, ML components are represented as parts of a larger (component-based) system and to some extent abstracted (e.g., [56]) or probabilistically characterized (e.g., [10, 128]). This simplifies the verification process considerably because common verification techniques (e.g., model checking), as known from classical software systems, are applicable again.

We believe that this approach is also promising for quantum software. In [126], for example, a reliability analysis approach of a hybrid quantum software system is proposed in which the quantum component is represented probabilistically (i.e., by estimating the success probability given the structure of the quantum circuit). Although the approach provides no formal guarantees, it can be seen as a first attempt to analyze system-level properties of software systems including quantum components.

3.3 Conclusion

We discussed multiple challenges that the verification of quantum and ML systems have in common and that have to be tackled to advance the verification in these two fields. By pinpointing these similarities we showed how the communities of those two research areas can profit from one another and presented several concrete research directions and existing work that we consider promising for the near future. In conclusion, we advocate for the close collaboration of the communities.

Chapter 4

Analyzability-Classes for Non-Classical Systems

Non-classical systems pose a unique set of challenges for formal verification. Their inherently complex nature and often unintuitive behavior make it not only challenging to specify the correct behavior but also increase the difficulty when conducting proofs. On the other hand, for complex and unintuitive systems the need for increased reliability measures is particularly pronounced, especially when they are applied in safety-critical scenarios. Thus the need for formal guarantees is very prevalent. In this chapter, how we can classify non-classical systems according to the formal guarantees that can be provided for them.

4.1 Four Classes of Analyzability for Non-Classical Systems

In the following, we present four classes of analyzability for non-classical systems. These are "verifiable", "fully monitorable", "partially monitorable" and "non-monitorable". Note, that the notion of monitorability in our context is different from what is often understood by that term in the literature on formal methods (being that a temporal logic formula is monitorable iff given an arbitrary trace it exists a finite extension of said trace which results in a conclusive verdict of a monitor [15]). These classes are based on classes for the analyzability of ML-systems which we first published in [129]. In this chapter, we generalize and apply these classes to non-classical systems.

Before we dive into the definitions of our classes, we would like to emphasize that these classes are somewhat subjective in the sense that different persons/organizations may categorize the same system differently. This is intended, and we will go into detail as to why that is. Nevertheless, the categorization of a system into either class enables the system architect to deduce which formal properties can be guaranteed after the system is fully developed. Thus these classes are meant to be useful for analysis at design-time. Classifying a system is therefore to be seen as a design-time decision made by the system architect. We illustrate how such a decision at design-time, in turn, influences the assurances that can be made about the system.

We define all classes w.r.t. a given system property that we denote by φ. We assume φ to be objectively observable. For example, consider a neural network supposed to classify images as to whether they contain at least one cat or not. There is no known formalization of the property "image contains a cat" but it is nevertheless clear to a human.

With this in mind, we define the first class of analyzability as the class of *verifiable systems*:

Definition 4.1 (Verifiable systems). *A system is* verifiable *w.r.t.* φ *iff it is possible to prove with* justifiable *effort that it (always) satisfies* φ*.*

In the following, when not referring to a specific property φ, we call a system verifiable if it satisfies its full specification. Additionally, we intentionally do not further specify when the verification effort is "justifiable" as that heavily depends on the context in which a system is developed. Consider for example the following list of factors:

1. **Experience:** A developer experienced in program verification is likely to prove the desired property much faster and more efficiently than someone new to the field. Experienced developers have a deeper understanding of the nuances and potential pitfalls involved in the verification process. They are familiar with common patterns and strategies, reducing the time needed to develop and apply proofs. Therefore, the effort required for formal verification of a system may be justified or not depending on the skills and expertise of a team.

2. **Computational Power:** Many methods for program verification rely on solvers that can be computationally expensive. These solvers may require substantial processing power and memory to handle complex verification tasks, such as model checking or satisfiability solving. If the available computational resources are insufficient, the verification process may be slow or infeasible. Consequently, determining

whether a verification effort is justified depends on the hardware and software available for the verification process.

3. **Field of Application:** The field in which the system is deployed dictates different safety and security requirements. For instance, developing a control unit for a spacecraft, where no physical access is possible post-launch and significant financial (and even human) resources are at stake, justifies extensive time and cost-intensive verification efforts. In such high-stakes environments, failure is not an option, and thorough verification is critical to ensure reliability and safety. Conversely, for a small system that can be easily updated and tested before deployment, it is much harder to justify a similar verification effort.

4. **Formal Method:** The time required for verification can vary greatly depending on the formal method or specific tool employed. Some methods, such as bounded model checkers, can be used fully automatically, making them suitable for the application to continuous integration and continuous deployment (CI/CD) pipelines. These tools can quickly verify certain properties with minimal human intervention. On the other hand, methods like interactive theorem provers require extensive manual effort and expert knowledge to construct and verify proofs. This aspect is also related to the type of system, as certain methods or tools may not be applicable to all kinds of systems. For example, some formal methods are better suited for verifying control systems, while others excel in analyzing software with complex data structures. Choosing the right method and tool for the specific system and verification goals is crucial for an efficient and successful verification process and thus influences the decision of whether or not the verification effort is justified or not.

The aforementioned list is not exhaustive but serves to illustrate that what is considered “justifiable” in the context of formal verification depends on various interrelated factors.

For example, consider a driver for a small device. This type of software would likely be considered “verifiable” by our definition due to its nature as an embedded system with manageable complexity and development within an environment characterized by a comparatively slow product cycle. In contrast, a video game, due to its high complexity, rapid development cycle, and the need for high performance and responsiveness, is less likely to be verifiable under the same criteria.

It is important to note, however, that despite the subjective nature of what constitutes “justifiable” verification, these factors only influence *the effort* required to verify a system. The ultimate result, that the system is indeed verified, remains unchanged once a proof is found. In other words, the property φ is guaranteed to hold if a proof is successfully established.

Therefore, systems that are not categorized as verifiable do not have an *a priori* guarantee of possessing the desired property. This could be because the system does not actually have the property (which would often be considered a bug) or because the system does have the property but the proof has not been conducted, and thus there is no formal guarantee. These two cases are indistinguishable in practice, as without proof, we cannot ascertain whether the property holds or not.

To further elucidate, consider the following scenarios:

1. **System with Proof of Property φ:** For a system where a formal proof of φ has been conducted, we know for sure that the property holds. The verification effort has been deemed justified, and the system is guaranteed to meet the specified criteria.

2. **System without Proof of Property φ:** For a system where no formal proof has been conducted, we cannot guarantee that φ holds. This could be due to the system being inherently unverifiable due to its complexity, lack of resources, or the absence of skilled personnel to perform the verification. In this case, there is no assurance of the system’s correctness concerning the property φ. However, it is still possible that the system in fact has the property without us knowing it.

3. **Erroneous System:** If a system does not possess the property φ, it is considered erroneous. In this case, a verification effort would necessarily fail, as the desired property cannot be established. Note that we assume φ represents the property genuinely intended by the developers or architects. Consequently, the absence of this property is directly regarded as an error.

In conclusion, while the criteria for justifiable verification efforts can be complex and context-dependent, the process of formal verification itself provides a definitive answer regarding the presence or absence of a desired property.

Note also that in the three cases we covered above, we explicitly omitted the scenario of a failed verification attempt. This is because, for our categorization, this case holds no significance. A system is deemed verifiable only when a verification effort succeeds. If the attempt fails, regardless of the underlying reasons, the system cannot be considered verifiable.

Motivated by the realization that not every system is verifiable, we define a second class of reliability:

Definition 4.2 (Fully monitorable systems)**.** *A system is* fully monitorable *iff it is feasible to implement a decision procedure that decides in reasonable time at runtime whether the component's current behavior satisfies φ or not.*

Intuitively, a fully monitorable system requires a monitor capable of determining *at any point in time* whether the system continues to adhere to its specified behavior. This concept differs from fully verifiable systems in two significant ways. First, it permits the system to violate the specified property, provided that such violations can be detected. Second, the performance of the decision procedure is crucial. Unlike verifiable systems, which guarantee correct behavior, a monitor simply reports deviations from the expected behavior. In certain scenarios, detecting that "something is amiss" might be sufficient.

For instance, in a production plant with a low probability of malfunction, immediate detection of an error can minimize damage and prevent costly production halts. Here, a monitor that promptly signals an issue (allowing for quick corrective actions like a restart) can be nearly as valuable as a fully verified system. Conversely, knowing that an airplane's engines are on fire without a means to safely land the plane illustrates the limitations of monitoring. Moreover, the definition does not require the monitor to explain why a state fails to meet the desired specification. Thus, while a monitor can indicate that the current behavior deviates from the specification, it does not necessarily identify the cause of the fault.

The second key difference between fully monitorable and verifiable systems is the emphasis on the performance of the decision procedure. We intentionally leave "reasonable time" unspecified, as its definition varies widely depending on the application context. In some cases, it may be acceptable to identify an issue that occurred an hour ago. However, many applications demand rapid (if not immediate) error detection to facilitate prompt responses. Therefore, "reasonable time" is context-dependent, influenced by both the application type and the hardware on which the monitor operates.

Constructing a monitor is often more feasible than achieving full verification, as verification must demonstrate a property for all potential behaviors, while a monitor only needs to assess the current behavior. Nonetheless, building efficient monitors can be highly challenging or even impossible. The requirement for the monitor to provide a real-time verdict implicitly assumes that the current system state is observable, which may not always be feasible. For example, in quantum computing, the intermediate states of quantum computations cannot be observed without destruction, making it physically impossible to construct a monitor as defined for fully monitorable systems. This limitation highlights the need for a class of systems with even weaker guarantees, which we address in the next definition.

Definition 4.3 (Partially monitorable systems). *A system is* partially monitorable *iff it is feasible to implement a decision procedure that decides in reasonable time at runtime for a non-empty subset of states whether the component's current behavior satisfies φ or not and reaches an inconclusive verdict for all other states.*

The key distinction between partial and full monitorability lies in the capability of a partial monitor to return an inconclusive verdict. This relaxation means that the monitor is not required to evaluate the property in every single state. Instead, it is sufficient for the monitor to provide a conclusive verdict in a subset of states, while returning an inconclusive verdict in others. Importantly, we do not quantify the distribution of conclusive and inconclusive verdicts, with the only stipulation being that the subset of states yielding conclusive verdicts is non-empty. In practical terms, it is reasonable to expect this subset to be meaningful, allowing the monitor to provide a conclusive verdict in some significant portion of states. The absence of a conclusive verdict can arise for several reasons:

- **Monitor Timeout:** For fully monitorable systems, a conclusive verdict is required for every state within a reasonable time. However, if a monitor can provide conclusive verdicts for all states but fails to do so for some within the required timeframe, it is classified as a partial monitor due to the time constraint not being met. Essentially, the monitor times out for some states.
- **Genuinely Partial Monitor:** Depending on its decision procedure, a monitor might be able to provide conclusive verdicts for some states but not for others. This again leads to the system being classified as partially monitorable. One could argue that such a monitor is less powerful than the one described in the previous point, but the guarantees provided by both types of monitors are identical, justifying their classification within the same category.

- **Non-Observable System:** In some otherwise fully monitorable systems, certain states may not be observable due to physical or technical limitations. For such systems, a monitor can only be partial since no conclusive verdict can be reached for the non-observable states. This situation differs from the previous case as the inability to reach a conclusive verdict stems from the properties of the system rather than the monitor.

We do not differentiate between these reasons when classifying a system, as the guarantees provided remain identical: no conclusive verdict can be reached for certain states.

The impact of classifying a system as partially monitorable heavily depends on the subset of states for which conclusive verdicts can be reached and the type of system classified. For instance, in a production plant, detecting a halt in operations with a slight delay might be acceptable, making a partially monitorable system adequate if it covers a sufficiently large (and frequently observed) subset of states. Conversely, in scenarios where even slight delays in error detection can be critical, or entire classes of errors cannot be detected in the monitorable subset of states, partial monitorability might be insufficient. Therefore, while partial monitors may suffice for some systems, the guarantees they offer are inherently weaker than those provided for fully monitorable systems.

For the sake of completeness, we introduce a last class of reliability which we call “non-monitorable”:

Definition 4.4 (Non-monitorable systems)**.** *A system is* non-monitorable *iff it is neither partially monitorable nor verifiable.*

This is the worst case where no assurances can be given for the system, neither at design-time nor at runtime. The reasons why a system might be non-monitorable can vary however they are most often a combination of the complexity of the system and the property sought to be guaranteed. Some properties like "the image shows a cat" cannot be formalized and thus there is no way to formally guarantee the absence or presence of this property for any system. Additionally some properties might be formalizable but too hard to show for sufficiently complex systems, such that they are (at least in practice) non-monitorable. This is not to say that even for non-monitorable systems there can be no quality measures like testing. However, the methods employed in this case are not able to guarantee properties but rather provide confidence in the correct behavior of the system.

4.2 The Relation of the Classes of Analyzability

As previously suggested, the order in which we presented the classes — verifiable, fully monitorable, partially monitorable, and non-monitorable — corresponds to a natural hierarchy in terms of the strength of achievable assurances. This hierarchy reflects the varying degrees of confidence and guarantees we can have about a system's behavior with respect to a property φ.

A *verifiable system* offers the strongest assurance, as it is guaranteed to behave correctly at any point in time. This means that through rigorous offline methods, we can prove with justifiable effort that the system always satisfies φ. This proof allows to assume φ after every execution of the system, which might in turn be used for any type of different analysis.

A *fully monitorable system* provides the next level of assurance. While it may not offer the comprehensive guarantees of a verifiable system, it is at least able to recognize at runtime whether or not it made a mistake with respect to φ. This implies that a decision procedure can be implemented to evaluate the system's current behavior against φ in a reasonable time frame, allowing for real-time detection of deviations from the expected behavior. The ability to detect faulty behavior can be crucial for the implementation of recovery strategies or other counter-measures. The most important feature of this class is the fact that at any point in time, it is known whether φ currently holds or not.

Partially monitorable systems offer more limited assurances. These systems can only evaluate φ in a subset of states within a reasonable time frame, returning an inconclusive verdict for the remaining states. This means that while some degree of runtime monitoring is possible, it is not comprehensive. The assurance provided by partially monitorable systems is weaker, as they cannot guarantee detection of all deviations from φ, but they still offer some level of oversight and error recognition.

At the lowest end of the hierarchy are *non-monitorable systems*, which lack any feasible method for runtime monitoring or verification of φ. These systems cannot reliably recognize or prove adherence to φ during operation, making it impossible to provide meaningful assurances about their behavior concerning the specified property.

Orthogonally to the guarantees that can be made for a system, the classes depend on a second dimension: the feasibility of achieving these guarantees. Theoretically, many properties of any system may hold, but proving them can be very difficult or practically impossible. Therefore, we deliberately introduced the dimension of feasibility into our classes of dependability. While the guarantees are hierarchical, the classes themselves are not.

It is entirely possible for a system to be verifiable but not monitorable. For instance, consider a system where it is justifiable to invest substantial effort to prove that it runs as expected, but it has very challenging real-time requirements. In such cases, the system can be proven to satisfy φ through offline methods (verifiable) but may not be capable of real-time evaluation of φ during operation (not monitorable). This distinction highlights the different demands and practical challenges associated with verification and monitoring.

Another implication of introducing the feasibility aspect to the classes is that the actual reliability of a system and the reliability assurances that can be made in practice may differ. A system might be inherently reliable and operate correctly in all scenarios, but proving this with justifiable effort could be extremely challenging. Our classification framework, however, prevents the opposite situation: if a system is classified as verifiable, it implies that strong reliability assurances can indeed be made based on the rigorous proof of φ.

4.3 Applying the Classification Scheme

As discussed previously, the proposed classes are intended for use during the design phase of systems. Each component in a complex system can be assigned to one of the four presented classes based on the outlined criteria. This systematic classification offers several advantages: (1) it enables a more accurate assessment of the overall reliability assurances based on the classification of individual components; (2) it helps to identify inconsistencies in the system design regarding reliability assurances; and (3) it serves as a useful step towards assessing overall system properties such as dependability.

Estimation of System Reliability Based on Components When a system architect classifies all components of a system under investigation into the proposed classes, assessing the overall reliability of the system becomes more accurate. The architect can employ different aggregation methods to combine the reliability classes of the components. For instance, the overall reliability might be determined by the lowest reliability class among all components (reflecting the ordering of the classes). Alternatively, using domain-specific knowledge, the architect might assign higher weights to particularly critical components while deemphasizing less significant ones in the reliability calculation.

In either case, explicitly classifying each component and systematically aggregating their classifications has been demonstrated to be more effective than directly estimating a complex property like system reliability. This approach leverages the advantages of structured decision-making, as supported by findings in [84, Part 5, Chapter 21].

Detecting Inconsistencies in System Design Given the classification of each component, the system design can be automatically checked for inconsistencies. For example, a component classified as *non-monitorable* may not be suitable as an input provider to components with higher reliability classifications. Without additional assumptions, such a design could be flawed, as the higher-reliability system cannot trust inputs from a lower-classified component. Solutions to this issue, such as incorporating runtime monitors to validate inputs, could be explored, but flagging such configurations during the design phase would offer valuable insights—particularly for large systems. This example represents just one type of potential inconsistency; more complex relationships between components could also be analyzed systematically.

Classification as One Metric Among Others The classification of components can also be treated as a single metric within a broader analysis framework. For instance, we explored this idea in the context of machine learning systems in [129], where the proposed classes were employed as one of several metrics to assess the analytic capability of components. Similarly, this classification scheme could be integrated into various analysis frameworks, not only for ML systems but for non-classical systems more generally.

4.4 Conclusion

In summary, the classification of systems into verifiable, fully monitorable, partially monitorable, and non-monitorable provides a structured approach to understanding and assessing the level of assurance that can be achieved regarding a system's behavior with respect to a property. This hierarchy not only clarifies the capabilities and limitations of different systems but also guides the development and implementation of verification and monitoring strategies tailored to the specific needs and constraints of each system. Furthermore, the application of this classification scheme has several benefits during design time.

Chapter 5

Fault-Tolerant Architectural Patterns for Quantum Software

In the previous chapter, we demonstrated the categorization of non-classical systems based on the formal guarantees that can be established for them. In contrast, there are several methods to enhance system trust without formal guarantees. One such method involves implementing fault-tolerant architectural patterns (e.g., [54]). These architectural patterns are designed generically, allowing their application across various domains.

In this chapter, we adapt fault-tolerant architectural patterns to the context of quantum computing in order to address the inherent uncertainties associated with quantum systems. To the best of our knowledge, no existing approaches specifically target gate and measurement errors at the architectural level. We investigate potential sources of redundancy in quantum components, how fault-tolerant architectural patterns can be applied to them and evaluate the impact of these fault-tolerant patterns on the overall reliability of quantum circuits. This chapter is based on [127].

5.1 Fault-tolerant Architectural Patterns

We first introduce redundancy in general and fault-tolerant architectural patterns and then show how they can be applied to the domain of quantum computing.

5.1.1 Redundancy in Software Systems

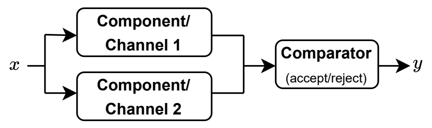
The notion of redundancy, we rely on in this thesis, is based on the work of Douglass [55]. Redundancy is employed to address two primary types of faults: *systematic* and *random* faults. Systematic faults arise from errors during design or implementation, whereas random faults occur unpredictably in otherwise functioning system components, such as hardware failures.

To mitigate these faults, redundancy is categorized into *homogeneous* and *heterogeneous* redundancy. Homogeneous redundancy targets random faults, while heterogeneous redundancy addresses both systematic and random faults. Redundancy involves deploying multiple instances of so-called *Channels* within a software system. A channel is defined as a basic unit of a redundancy pattern, encapsulating a data transformation pipeline that converts input data to output data through a sequence of components.

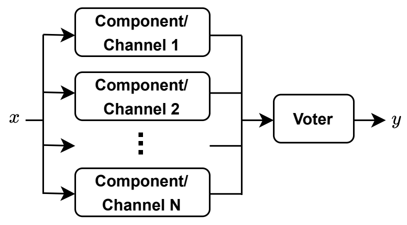
In homogeneous redundancy patterns, identical copies of a channel are deployed multiple times within the system. These channels can be queried sequentially or in parallel, with their results combined afterward in various ways to ensure operational stability in the presence of faults. Homogeneous redundancy is designed to address random faults. This is evident because systematic faults present in one channel would be replicated across all copies of that channel, rendering them undetectable by a pattern employing homogeneous redundancy. In contrast, random faults, such as hardware errors, are unlikely to occur simultaneously across all channels, thereby being effectively mitigated by homogeneous redundancy patterns.

Conversely, heterogeneous redundancy involves creating multiple distinct versions of a channel, rather than mere copies. This approach, exemplified by the *N-Version Programming Pattern* [34], entails independent implementations of each version. Typically, in the N-Version Programming Pattern, different teams of developers implement each version independently. However, a broad array of sources for channel independence may be considered, such as different hardware, algorithms, and programming languages. Heterogeneous redundancy effectively addresses both random and systematic faults. The underlying assumption is that the likelihood of identical errors across all versions is minimal, although not impossible.

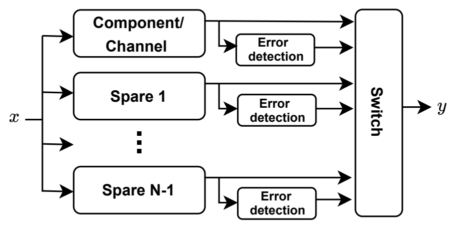
The term "channel", as defined by [55], originates from the field of real-time systems and may not be consistently applied across other domains where fault tolerance is critical. Nonetheless, this definition aligns well with the structure of quantum algorithms, which also involve multiple stages of data transformation, such as pre-processing, data preparation, unitary transformation, measurement, and post-processing [99]. Consequently, we



(a) Comparison pattern



(b) Voting pattern



(c) Sparing pattern

Figure 5.1: Overview of the fault-tolerant patterns based on [54] as depicted in [129].

introduce the term *qrChannel* (quantum redundancy channel) to denote the entire structure of a quantum algorithm, encompassing all these transformation steps.

5.1.2 Fault-tolerant Architectural Patterns

In this work, we consider three architectural patterns to improve fault tolerance of quantum software systems, namely the *Comparison Pattern*, *Voting Pattern* and *Sparing Pattern* (see Ding et al. [54] for a detailed discussion of the patterns).

Comparison pattern In systems utilizing the comparator pattern (depicted in Figure 5.1a), two channels are run in parallel. The outputs from these redundant channels are then fed into a comparator, which performs the following steps:

1. **Collection**: Gather outputs from all redundant channels.

2. **Comparison**: Compare the outputs and evaluate whether the outputs are consistent or if there is a fault based on predefined criteria.

Different strategies may be employed in order to decide whether two outputs are considered consistent. There are mainly three options to do so:

- **Exact Match**: The outputs are considered consistent if they exactly match. This strategy is suitable for systems where precision is critical but also expected from the different channels.

- **Approximate Match**: The outputs are considered consistent if they fall within a predefined tolerance range of each other. This strategy is useful for systems where minor variations are acceptable or expected from the channels.

- **Semantic Match**: The outputs are considered consistent if they satisfy some predefined predicate. This strategy can be seen as an extension of the previous one. The predicate in this case can be arbitrarily complex. A simple example would be that the returned values of the channels are known to be in a fixed interval and have a cyclic property such that very high values in that interval are close to very low values of that interval.

Voter pattern In systems utilizing the voter pattern (depicted in Figure 5.1b), N channels are run in parallel. The outputs from these redundant channels, are then fed into a voter, which performs the following steps:

1. **Collection**: Gather outputs from all redundant components.

2. **Comparison**: Compare the outputs to identify discrepancies.

3. **Decision**: Determine the correct output based on a predefined voting strategy (e.g., majority voting).

4. **Output**: Produce the final output for the system based on the voter's decision.

Different voting strategies can be employed based on the specific requirements and fault characteristics of the system:

- **Majority Voting**: The most common strategy, where the output that appears most frequently among the redundant components is selected as the correct result. This strategy is effective in systems with odd numbers of redundant components (e.g., three in Triple Modular Redundancy) to avoid ties.
- **Median Voting**: This strategy selects the median value from the outputs, which can be useful in systems where outputs are continuous values, reducing the impact of outliers.
- **Weighted Voting**: In some systems, different components may have different reliability levels. Weighted voting assigns different weights to the outputs based on the trustworthiness of each component.
- **Consensus Voting**: Used in systems where a majority consensus is required, this strategy only produces an output if a certain percentage of components agree, enhancing fault tolerance in highly critical applications.

Sparing pattern In systems utilizing the sparing pattern (depicted in Figure 5.1c), one or more standby channels are kept in reserve, ready to take over the functionality of the primary component in the event of a failure. To decide whether a channel is failing or not, each channel has a dedicated error detection unit. The sparing pattern in general follows the following steps:

1. **Monitoring**: Continuously monitor the health and performance of the primary component based on the verdict of the error detection unit.
2. **Switchover**: Automatically switch over to a standby component when a failure is detected.
3. **Recovery**: Restore the failed primary component to standby status once it has recovered.

Different types of sparing strategies can be employed based on the specific requirements and fault characteristics of the system:

- **Cold Sparing**: The standby components are not yet initiated and will only be activated when a failure occurs. This approach minimizes necessary computing resources but may result in longer recovery times.

- **Warm Sparing**: The standby components are instantiated on but not actively processing. It can take over more quickly than cold spares, offering a balance between necessary computing resources and recovery time.
- **Hot Sparing**: The standby component runs in parallel with the primary component at all times and can thus take over almost instantaneously. This approach ensures the shortest recovery time but at the cost of higher resource usage.

The use of the sparing pattern assumes that it is possible to somehow provide an error detection unit. This may not always be possible. Additionally, as Ding et al. point out in [54] these patterns can be combined to achieve more sophisticated fault-tolerant patterns. In the following sections we will, however, only consider the three basic patterns presented here.

5.2 Applying Fault-Tolerant Architecture Patterns to Hybrid Quantum Software

In this section, we discuss the application of fault-tolerant architectural patterns within the quantum computing landscape. Specifically, we examine the necessary adaptations required to apply these patterns to hybrid quantum software. We have identified three key challenges that must be addressed:

1. determining suitable sources of redundancy for quantum software
2. developing methods to combine the probabilistic results of quantum computations
3. establishing generic error detection mechanisms tailored to quantum systems

Each of these challenges is addressed in detail in the following sections.

5.2.1 Redundancy in Hybrid Quantum Software Systems

In quantum systems, as in classical systems, redundancy can be introduced at two distinct levels. The first is the *hardware level*, where different quantum devices are used to execute the same code or circuit. The second is the *software level*, where various implementations or compilations are employed to solve a given problem. These sources of redundancy can be combined in different ways to enhance system reliability.

This thesis focuses on faults in quantum devices, which are particularly prevalent in the NISQ (Noisy Intermediate-Scale Quantum, see Section 2.1.9) era. These errors exhibit a complex nature: they are neither entirely random nor fully systematic, depending on the perspective taken. For example, consider gate errors: they are random in that an error occurs with a certain probability, but this probability varies across qubits. As a result, different transpilation strategies may experience entirely different gate errors. In this way, gate errors can be systematically linked to specific qubits, making them not entirely random. Thus, whether errors are viewed as random or systematic depends on whether qubits are treated as uniform computational units or considered based on their specific hardware properties, with varying error rates. This duality blurs the distinction between homogeneous and heterogeneous redundancy in quantum computing. Clear cases of heterogeneity exist, such as two entirely different implementations of the same algorithm. However, different transpilation strategies for the same implementation are more difficult to classify, due to the nuanced nature of error distribution across qubits. Building on this insight, we explore various forms of redundancy in the following sections, evaluating which are most promising for each given context.

Note that for this thesis, we consider one run of a quantum circuit to be the execution of this circuit a given number of times, referred to as *shots*. If not otherwise specified, we adhere to the Qiskit standard of 1024 shots. The result of this execution is a set of measurement results (bitstrings) that occur with varying frequencies, referred to as counts in Qiskit terminology. These counts can be interpreted as a discrete probability distribution over the possible measurement outcomes $\{0, 1\}^N$ for N qubits.

Redundancy on Hardware Level

With redundancy on hardware level, we consider the case where N copies of a qrChannel are executed on different quantum computers or (*ii*) quantum computer-specific low level configurations of each copy are varied (e.g. adaptation for a different Topology Graph) or (*iii*) a combination of both. The measurements taken from each copy are then passed on to the next stage of the applied pattern. In the following, we examine the possible variants of hardware level redundancy that can be realized.

Each quantum computer implements a so-called *topology graph*, which defines both the number of qubits and the physical connections between them. Formally, a topology graph is represented as a graph $G := (V, E)$, where V denotes the set of qubits, and E represents the set of physical connections between qubits. Specifically, two qubits $i, j \in V$ are physically connected if and only if there exists an edge $(i, j) \in E$. The connections between qubits are crucial, as two-qubit gates can only be applied to qubits that are physically connected. Consequently, in order to execute a quantum algorithm on a quantum computer, the logical qubits of the algorithm must be mapped to the physical qubits defined by the topology graph.

Since topology graphs are typically not fully connected (or even rather sparse), two-qubit operations involving qubits without a physical connection must be swapped. This involves applying a series of CNOT gates to swap the states of the qubits onto other qubits that do have a physical connection [99]. However, this is a costly process, as it increases both the number of gates (each gate introducing the risk of additional errors, especially two-qubit gates) and the circuit depth, raising the risk of decoherence.

Additionally, the set of quantum gates executable on a quantum device can differ across devices. Quantum algorithms are typically described in a device-agnostic manner, assuming any unitary matrix can serve as a valid quantum gate. As a result, algorithms must be pre-processed to decompose gates that are not supported by a given device into equivalent operations (this is always possible if the set of supported gates is universal). This process, which encompasses both the mapping of logical qubits to physical qubits and the decomposition of unsupported gates, is known as *transpiling*. Again, this often involves the use of additional quantum gates, increasing the circuit depth and, therefore, the likelihood of errors. Consequently, the selection of the quantum computer—and its implemented topology graph—on which a quantum algorithm is executed plays a central role in the successful execution of the algorithm. For instance, Holmes et al. [80] have demonstrated how topology graphs with low connectivity can adversely affect the performance of key quantum algorithms, such as the *Quantum Fourier Transform*. In terms of architectural patterns, the question arises as to whether N copies of a qrChannel should be executed on distinct quantum computers (each implementing a different topology graph) or on quantum computers with the same—or at least similar—connectivity structures, such as linear, ladder, grid, or all-to-all [80].

In the former case, one could argue that using distinct quantum computers increases the degree of heterogeneity, potentially leading to more varied measurements, which, when combined, may produce a more refined result. However, this approach carries the risk that when combining measurements from N qrChannels, the most accurate measurement may be degraded by the less accurate $N - 1$ measurements, ultimately resulting in a less precise combined outcome than the best single measurement.

In contrast, when executing each qrChannel on a quantum computer with a similar topology graph, the combined measurement may compensate for individual gate and measurement errors of each individual measurement of a qrChannel.

All these variants arise directly from the choice of hardware as the supported set of gates and the topology graph are immutable properties of the hardware. However, the transpilation process is predominantly carried out heuristically, which allows for various implementations even on the same hardware. We explore this topic in more detail in the next paragraphs.

Redundancy on Software Level

Software redundancy refers to scenarios where redundancy is introduced at the software level. In most cases, this form of redundancy can be considered orthogonal to hardware redundancy.

The most common form of software redundancy is algorithmic redundancy. By this we understand the implementation (ideally by independent teams) of N distinct versions of a quantum software that provides the same functionality. This can be done either by implementing the same algorithm in slightly different versions or by solving the same problem with entirely different algorithms. For the Traveling-Salesperson problem, for example, one can consider either phase estimation methods [137] or *Quantum Approximate Optimization Algorithm* (QAOA) [121] as possible quantum algorithms. This type of redundancy has the advantage that it does not only cover for random faults but also systematic ones in the form of implementation errors in one of the versions. However, this form of redundancy is very closely related to classical systems for which the exact same approach can be used and has been well studied (e.g. [11]). For this reason we focus on more quantum-specific sources of redundancy for the remainder of this chapter, specifically to reduce NISQ-errors.

In the previous section, we discussed the transpilation of quantum circuits, which can also be viewed as a source of redundancy at the software level. Currently, the parameters and type of transpilation are often explicitly defined as part of the hybrid quantum software. As a result, it is both possible and common to construct different circuits for the same hardware configuration. We identify three main sources for these variations:

1. **Transpilation seed**: The heuristic nature of the transpilation process can be explicitly controlled by a given seed. Varying this seed yields different results.

2. **Optimization level**: Since transpilation is an NP-hard problem, finding the optimal solution is generally not feasible. This is one of the primary reasons why heuristics are employed. These heuristics can trade off performance for potentially better solutions. Qiskit explicitly provides optimization levels as a parameter for the transpilation process.

3. **Transpilation algorithm/library**: Given the crucial role of transpilation in contemporary quantum computing, several libraries offer different advantages and can be used interchangeably.

Overall, we consider transpilation to be one of the primary sources of redundancy in quantum computing. While certain aspects of the transpilation process are dictated by the properties of the hardware, there remains significant scope for further variations.

Having identified different types and sources of redundancy, we continue to explore how fault-tolerant architectural patterns have to be adapted in the context of quantum computing.

5.2.2 Combination of Multiple Measurements

The discussed patterns in the classical setting rely on the fact that each channel obtains classical results that can easily be compared, aggregated, or voted on. In the context of qrChannels, however, the results are a set of measurements rather than a single computed result. This represents a challenge for the application of the patterns. In this section we discuss how sets of measurements — essentially discrete probability distributions — can be combined or compared.

Aggregation of Measurements

We begin by examining the aggregation of measurements. This is essential for the voter component, which traditionally selects one outcome from the N possible outcomes of each component or qrChannel. However, the conventional view of the voter component in the context of quantum computing is overly restrictive, as it often implies pure voting implementations, such as majority voting. This approach is impractical for quantum software due to its probabilistic nature, especially when considering noisy hardware, since each result is very likely to differ.

To address this limitation, we generalize the concept of the voter by defining it as any decision procedure that combines N measurements into a single measurement. In the following discussion, we will use the term *Combiner* instead of voter to reflect this generalization.

In quantum computing, measurements refer to discrete probability distributions that must be processed to obtain a final result. However, these distributions are subject to gate and measurement errors; thus, the actual measured distribution may deviate significantly from the ideal one. The Combiner takes the measurements of multiple qrChannels and combines them in a more accurate distribution, ideally superior to the individual measurements.

In the literature, several mathematical approaches for aggregating probability distributions exist [40, 68, 81]. More specifically, these aggregation methods are often employed to combine multiple expert opinions, each represented by a probability distribution, modeling the uncertainty associated with each opinion. This uncertainty models the subjectivity or partial lack of knowledge of the expert. By aggregating these opinions into a single distribution, it is expected that a refined distribution is obtained, effectively averaging out the experts' uncertainties.

In the context of quantum computing, an opinion corresponds to a measurement. We expect that the combination or averaging over multiple measurements (distributions) will cancel out wrong results and amplify correct ones. Formally, we define a combiner $\mathcal{C}$ as follows:

$$\mathcal{C} : \mathcal{H} \to [0,1], \quad |\varphi_j\rangle \mapsto \sum_{i=1}^{N} w_i P_i(|\varphi_j\rangle), \tag{5.1}$$

where w_i are non-negative weights such that $\sum_{i=1}^{N} w_i = 1$. Moreover, $|\varphi_j\rangle \in \mathcal{H}$ represents the jth basis vector of an orthonormal basis $\{|\varphi_j\rangle\}$ of the Hilbert space $\mathcal{H}$ in which the quantum system is measured.

Equation (5.1) describes an aggregation method known as the linear opinion pool [40, 68]. In this framework, each measurement (or expert opinion) is represented by a probability measure P_i. In our case, P_i corresponds to the measurements of the i-th qrChannel (recall that the entire measurement process has already been conducted for each qrChannel). Thus, the distribution P_i is defined over $\{|\varphi_j\rangle\}$ and associates each $|\varphi_j\rangle$ with the respective probability of observing the quantum state $|\varphi_j\rangle$.

The combined distribution is the weighted sum of measures P_i, where each P_i is associated with a weight w_i (with all weights summing to 1). Other aggregation methods, such as the *Logarithmic Opinion Pool* and *Bayesian* approaches, also exist [40]. For the purposes of this discussion, we focus on the linear opinion pool, noting that alternative methods could be explored in future work.

Originally, a weight w_i represents the degree of confidence associated with expert i. In our context, the weight w_i indicates the level of trustworthiness we attribute to the measurements of a qrChannel i. If the weights are unknown a priori (e.g., when there are no preferences or prior knowledge regarding the quality of a qrChannel), a natural choice is to distribute the weights evenly. Alternatively, the weights can be calculated dynamically based on real-time data concerning the properties of the used qrChannels. The Qiskit API [104], for example, provides access to configuration and property data from the backends utilized, i.e., the quantum computers being used. This data includes gate and readout errors (i.e., measurement errors) that can serve as the basis for the on-demand calculation of weights before combining the measurements. Moreover, following the approach of Salm et al. [125], one can access the transpiler depending on the *Software Development Kit* (SDK) in use (e.g., Qiskit). These estimates provide a foundation for determining w_i.

In most settings, where there is no prior knowledge or preference of qrChannels, the weights w_i, the Hilbert space $\mathcal{H}$, and the basis vectors φ, can be chosen as follows:

- **Weights**: $w_i = \frac{1}{N}$ for $i = 1, 2, \ldots, N$ (uniform distribution).
- **Hilbert Space**: $\mathcal{H} = \mathbb{C}^N$.
- **Basis**: $\varphi = \{e_1, e_2, \ldots, e_N\}$ (the computational basis).

Under these assumptions, the combiner simplifies to the following expression:

$$\mathcal{C} : \mathbb{C}^N \to [0, 1], \quad |e_j\rangle \mapsto \frac{1}{N} \sum_{i=1}^{N} P_i(|e_j\rangle), \tag{5.2}$$

where $|e_j\rangle$ represents the j-th basis vector in the computational basis, and $P_i(|e_j\rangle)$ denotes the probability (counts) associated with observing the quantum state $|e_j\rangle$ from the i-th qrChannel.

For this thesis, we consider this choice of values for our evaluation. Note, however, that the possibilities of combining different channels are plentiful and for different application scenarios other methods might be worth exploring.

Comparison of Measurements

Similarly to the combination of measurements, we must establish a method to compare measurements. A straightforward equality check is likely to fail in most cases, as obtaining identical measurements is highly improbable, even with error-free quantum computers. Furthermore, we demonstrate how these similarity measures can be leveraged to construct a form of error detection.

The fundamental idea is the same as for combining measurements: We view measurements as discrete probability distributions. The comparison of probability distributions is a well-studied topic such that we can use established metrics. In particular, we consider the following metrics for two probability distributions P and Q:

Kullback-Leibler Divergence (KL Divergence):

$$D_{KL}(P, Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$$

Total Variation Distance:

$$D_{TV}(P, Q) = \frac{1}{2} \sum_x |P(x) - Q(x)|$$

Hellinger Distance:

$$D_H(P, Q) = \frac{1}{\sqrt{2}} \sqrt{\sum_x (\sqrt{P(x)} - \sqrt{Q(x)})^2}$$

The last two metrics, Hellinger Distance and Total Variation Distance, are particularly well-suited for our purposes, as they are normalized — yielding values between 0 and 1 — and symmetric ($D(P, Q) = D(Q, P)$). Using these similarity measures, we can define an approximate match between two measurements as their distance falling below a predefined threshold. This notion of an approximate match can then be applied, for instance, in the comparison pattern.

Expanding on this concept, these metrics can also be used to establish a form of error detection for quantum measurements. If a quantum computation produces a distribution where the correct result(s) have higher probabilities, we expect that distribution to be relatively distant from the uniform distribution. Conversely, a distribution resembling the uniform distribution likely indicates significant noise interference, as no measurement outcome dominates in probability. Thus, the proximity to the uniform distribution serves as a useful metric for assessing the likelihood of an erroneous outcome. This assessment can further be converted into a binary decision by applying a predefined threshold. Note that this type of error detection is only suitable for final results of quantum algorithms and not for the results of subroutines like the QFT. In general, this idea is only applicable if certain states are expected to be observed with comparatively high probability.

This error detection method can be integrated with the sparing pattern to determine when to switch to a spare qrChannel. Its generic nature offers the advantage of applicability across any algorithm that yields the desired output with a sufficiently high probability, requiring only an adjustment of the threshold. However, this broad applicability can result in a coarse error detection mechanism.

If we possess additional information regarding the nature of the expected output distribution, we can refine this error detection approach further. By understanding the anticipated output distribution, we can ascertain how many viable solutions are expected and their corresponding probabilities. For instance, we may know that the resulting distribution should exhibit a Dirac form—characterized by a single peak with all other values being zero. Importantly, this method does not necessitate prior knowledge of the actual result, only that it should appear with high probability.

When calculating the differences between the two distributions, we can utilize the concept of ordering by likelihood, which involves comparing the most probable outcomes first. The resulting differences between the expected and observed distributions can then serve as a more reliable error detection mechanism.

It is important to note that this method primarily addresses NISQ errors caused by hardware failures rather than algorithmic errors. If an algorithm consistently produces an incorrect output with high probability—such as in the case of a Dirac distribution—this would not be detected by our approach. In contrast, NISQ errors that occur randomly are very unlikely to produce such outcome distributions, thus enhancing the effectiveness of our error detection method.

We would like to emphasize that heuristic error detection only becomes relevant when dealing with problems outside NP. For NP problems, we can efficiently verify whether a given solution is valid, eliminating the need for heuristic error detection. For instance, in Shor's algorithm, we can easily verify whether the output is a non-trivial divisor, and if it's not, we simply repeat the query. However, this verification process is not feasible for many well-known quantum algorithms, including most optimization algorithms and algorithms like Deutsch's algorithm. In these cases, where validation of results is more complex or infeasible, heuristic error detection is often the best option available.

5.2.3 Instantiating the Patterns for Quantum Software Systems

We now turn to the instantiation of fault-tolerant patterns for quantum software systems. Specifically, we elaborate on how each of the patterns presented in Section 5.1.2 can be adapted to the context of quantum software systems. Given that we have generalized the notion of the voter to a Combiner, we use the term Combiner Pattern to emphasize the distinction from the traditional voting pattern in classical software systems.

General Considerations

When applying a redundancy pattern—regardless of the specific type—a crucial decision involves selecting the number and types of qrChannels to use. This choice significantly impacts the effectiveness of the redundancy pattern. One seemingly obvious reason is that better qrChannels tend to lead to better overall results. Additionally, the selection process can affect how heterogeneous the qrChannels are, which in turn influences the final outcome.

In Section 5.2.1, we introduced two types of redundancy: hardware-level and software-level redundancy. At this point, it is essential to evaluate which type is more appropriate for the given context. In some cases, for example, quantum computers are employed to solve problems for which there is only a single known algorithmic approach, making algorithmic redundancy difficult to achieve. Conversely, for other problems—such as graph-related problems—multiple quantum algorithms and their variants exist, making algorithmic redundancy more promising. Additionally, at the software level, variations in transpilation options always provide a form of redundancy.

In terms of hardware-level redundancy, if only a limited number of quantum computers are available to execute the algorithms, the benefits of hardware redundancy are less pronounced.

In summary, the approach to qrChannel redundancy is highly context-dependent and must be carefully considered based on available resources, including quantum computers and applicable quantum algorithms.

Combiner Pattern

For the combiner pattern, the most relevant question is how to combine the results from the different qrChannels. In Section 5.1.2, we introduced several general options for this, each of which must be adapted to the quantum computing context. For example, in the case of majority voting, there are several design choices to consider: Should each qrChannel be executed with a single shot, and then the outputs be voted on? Or should we consider only the most frequently observed outcome from each qrChannel and then vote based on those? Both approaches are theoretically valid instantiations of the voter pattern in quantum computing but may differ in quality of the resulting outputs.

In this work, we focus on a variant of the voter pattern we called the combiner pattern, which implements a form of weighted Voting. Even when limiting our discussion to this particular method, further decisions need to be made. As discussed in Section 5.2.2, different aggregation methods are possible, and for each, different weightings can be considered. In either case, determining the weights is a non-trivial task.

One approach is to compute the weights dynamically, using real-time data from quantum backends, which can be accessed through vendor-specific SDKs. Alternatively, a domain expert or quantum computing expert could assign weights to each qrChannel based on prior knowledge, such as the topology of quantum computers or the suitability of specific quantum algorithms for the problem at hand.

The complete set of decisions regarding these properties constitutes an instantiation of the Combiner pattern in the quantum context. In the evaluation, we focus on combiner patterns using linear opinion pools as the decision procedure with uniform weights.

Comparison pattern

In the Comparison pattern, the results from two qrChannels are forwarded to the Comparator component for evaluation. If the results are deemed "matching", the output is accepted; otherwise, it is rejected. The key question here is how to define "matching" results.

One approach for exact matches is to check whether the most frequently observed bitstring is identical across the qrChannels. Additionally, the comparator could verify whether this bitstring is observed with identical probabilities. However, since an exact match of probabilities is highly unlikely, an alternative approach would be to use an approximate match strategy, where the comparator checks if the difference in probabilities falls within a certain threshold for each qrChannel. Both methods focus solely on the most likely output, ignoring other measurement outcomes.

Alternatively, more sophisticated methods for comparing measurements can be employed, as discussed in Section 5.2.2. One potential candidate from probability theory is the Hellinger distance, which compares probability distributions. Since quantum measurements correspond to probability distributions, the Hellinger distance can be used to measure the divergence between the results of two qrChannels. If the Hellinger distance exceeds a certain threshold, indicating a significant deviation, the overall output is rejected and countermeasures can be taken. Given that the distributions of the measurements are discrete, simpler techniques such as calculating the average difference between individual measurements can also be considered for comparison.

Regardless of the method used, the threshold for comparison is a critical decision, as it directly affects how likely two results are deemed "not matching." While finding the ideal threshold for a given scenario can be challenging, it is important to recognize that this threshold essentially functions as a tuning parameter for balancing false positives and false negatives.

A very low threshold allows for only minimal deviations, indicating highly similar results and thus increasing the likelihood that those results are correct. However, in this case, even minor errors could cause results to be flagged as "not matching" and discarded, despite those errors not significantly affecting the correctness of the final result. Conversely, setting a higher threshold would permit greater deviations, potentially accepting results that are less accurate but still within acceptable error margins.

Sparing Pattern

Regarding the sparing pattern, the primary question is how to implement error detection. In the quantum setting, the logic of the switch component remains identical to that of other domains: if an error is detected, a properly functioning spare is selected. The choice between cold, warm, or hot sparing, in our opinion, is also unaffected by the quantum context and can be decided as in any other domain.

However, the design of the error detection component is of particular interest in the quantum setting. For NP problems, one possible approach to error detection in quantum measurements is the use of witnesses (or certificates). A witness is used to verify whether a particular solution is correct for the given problem. In Satisfiability Problems, for example, a witness is a solution that satisfies the logical expression under consideration. In this way, the witness verifies the measurement result or indicates if the result is incorrect, allowing the switch component to transfer control to one of the spares. For NP problems, verifying the validity of a witness is always efficiently possible.

In cases where witness verification is not feasible, other means of error detection must be considered. As discussed in Section 5.2.2, heuristic

Comp.	**Variant**	**%fp**	**%fn**	**%cor**
Dist	Backends	9	20	71
	Seeds	7	19	74
	Optimization	7	19	73
Max	Backends	28	3	69
	Seeds	24	4	72
	Optimization	23	6	70
Dist(R)	Backends	18	18	64
	Seeds	15	20	66
	Optimization	15	20	64
Max(R)	Backends	22	2	76
	Seeds	15	5	79
	Optimization	15	9	76

Table 5.1: Evaluation of the Comparator-pattern with two different comparing methods: comparing maximums (Max) and distributions (Dist), listing correct (cor), false positive (fp), and false negative (fn) results in percent each, for the full and restricted (R) benchmark

methods based on observed measurements (and potentially known result distributions) can be employed for error detection. Similar considerations about the choice of the threshold as for the comparison pattern apply. Note also, that a Comparison Pattern could be used as an error detection mechanism.

These are just some of the options available, and more advanced error detection mechanisms can be utilized to instantiate the Sparing pattern.

5.3 Evaluation

To assess the effectiveness of fault-tolerant architectural patterns in the context of quantum computing, we conducted a series of experiments. The primary objective of this evaluation was to analyze how the application of the proposed patterns, along with the introduced modifications, influences the reliability of quantum components. We conduct this evaluation based on the Goal-Question-Metric (GQM) approach [31]. We thus start by formulating our main goals:

Variant	**%fp**	**%fn**	**%cor**
Backends	10	16	74
Seeds	9	19	72
Optimization	11	19	70

Table 5.2: Evaluation of the Sparing-pattern (applied to the restricted benchmark): listing correct (cor), false positive (fp) and false negative (fn) results in percent each

Variant	**%fp**	**%fn**	**%cor**
Backends	22	34	46
Seeds	20	41	39
Optimization	41	25	34

Table 5.3: Evaluation of the Sparing-pattern (applied to the restricted benchmark) only considering the configurations with at least one faulty result: listing correct (cor), false positive (fp) and false negative (fn) results in percent each

Var	**P(v)**	**P(c)**	**%P(c)<P(v)**	**H(v)**	**H(c)**	**%H(c)<H(v)**
Back	28.9	11.7	88.4	0.405	0.363	100.0
Seeds	33.0	14.4	85.4	0.418	0.386	99.9
Opt	29.5	7.8	85.7	0.433	0.400	100.0
Back(R)	15.6	4.3	84.9	0.559	0.508	100.0
Seeds(R)	20.7	9.4	80.5	0.586	0.548	99.9
Opt(R)	23.0	8.4	79.7	0.587	0.547	100.0

Table 5.4: Evaluation of the Combiner-pattern (applied to the restricted (R) and full benchmark): listing position of variant (P(v)), position of Combiner (P(c)), % of configurations for which the Combiner achieved a lower position than the average variant, Hellinger distance of the correct distribution for the average variant (H(v)) and the Combiner (H(c)) as well as the % of configurations for which the Combiner achieved a better (smaller) Hellinger distance than the average variant

G1: Assess the effectiveness of the fault-tolerant patterns in increasing the reliability of quantum computing components.

G2: Identify suitable approaches for creating different variants, improving the effectiveness of fault-tolerant patterns

To achieve these goals, we answer the following questions:

Q1: Does the application of fault-tolerant architectural patterns increase the reliability of quantum software components?

Q2: For which types of variants is the performance of the examined patterns the best?

Q3: Which pattern is the most promising in the context of quantum computing?

We will answer **Q1** and **Q2** for each pattern separately and then draw conclusions regarding **Q3** by comparing the observed results for each pattern. The metrics used to answer these questions are discussed in detail in each pattern-specific section.

5.3.1 General Design of Experiments

The goal of these experiments is to evaluate the effectiveness of applying fault-tolerant architectural patterns in terms of reliability (**G1**), as proposed in the previous sections. To achieve this, we applied the three described patterns to a selection of representative quantum algorithms. The algorithms were sourced from the MQTBench benchmark suite [117], a well-known benchmark for quantum algorithms. We focused on circuits that were scalable in the number of qubits, resulting in 11 distinct algorithms. For each algorithm, we generated circuits with qubit counts ranging from three to ten, yielding a total of 88 circuits. As the patterns we examine are targeted to be applied to entire components rather than sub-processes, we also investigated how the patterns fare on a restricted benchmark where we eliminated algorithms such as state preparation and QFT which are only sensibly used as subroutines. This reduced benchmark consists of 6 algorithms resulting in a total of 48 circuits. In the evaluation we mark rows evaluated on the restricted benchmark with an (R) while rows with no additional mark concern evaluations on the full benchmark.

To generate variants of these circuits, we employed three approaches outlined in Section 5.2.1: transpilation seeds, backend selection, and optimization levels. **G2** is to compare the performance of the considered patterns on these different types of variants. All considered parameters influence the transpilation process, allowing us to derive multiple concrete implementations from a single abstract circuit described in Qiskit. We evaluated 33 compatible backends from Qiskit, excluding those that lacked required gate support or sufficient qubit capacity. Additionally, we used 50 distinct transpilation seeds and optimization levels ranging from 0 to 3 (which are all available levels in Qiskit). This process resulted in 87 variants per circuit, totaling in 7,656 unique circuits (4,176 for the restricted benchmark).

Each variant was simulated on its designated backend or, by default, on the BoeblingenV2 backend. All simulations employed the backend-specific noise model and were executed using 1024 shots, consistent with Qiskit's standard configuration. Additionally for each of the original 88 circuits, we used an error free state-vector simulator to obtain the theoretically correct result. Note, that the entire evaluation is done on simulators only, as we did not have sufficient access to real quantum devices.

5.3.2 Comparator Pattern Evaluation

For the Comparator pattern, we evaluated two different approaches. The first approach checks for equivalence of the most likely result within each distribution, comparing the bitstrings with the highest counts for each qrChannel. The second approach employs a similarity metric between the distributions of each qrChannel, accepting results only if the similarity exceeds a given threshold. In this experiment, we used the Hellinger distance as the similarity metric with a threshold of 0.4. For each type of variant, we evaluated all possible combinations of variants of that type. As we had 33 backend variants, this results in $\binom{33}{2} = 528$ possible combinations. For each of the 88 circuits, these combinations yielded a total of $46,464$ configurations for the backend variant type. Using the same calculations, there are $107,800$ configurations for the seed variant type and 528 for the optimization-level variant type.

The intuitive goal of a Comparator is to identify wrong results. We consider a result of a variant (which is a set of 1,024 measurements) to be correct if the bitstring with the highest count is one of the most likely bitstrings according to an error-free simulation. Using this definition of a correct result we can define what we consider to be a correct verdict by the Comparator. We expect the Comparator to accept the inputs of its channels if and only if both inputs are correct and reject them otherwise. As a metric to evaluate the Comparator pattern we use the percentage of correct, false positive, and false negative verdicts. A verdict is a false positive if the comparator wrongfully rejected some inputs, and a false negative if the Comparator wrongfully accepted some inputs.

M1 Percentage of correct, false positive, and false negative verdicts

False positives hinder performance by discarding valid results but do not impact system reliability. In contrast, false negatives directly affect reliability as they allow faulty results to propagate.

The experimental results for evaluating the Comparator pattern are summarized in Table 5.1. We report the percentages of false positives, false negatives, and correct results for each Comparator method and variant type (metric **M1**). Notably, even with these simple patterns and comparison methods, the correct verdict was achieved in over 70% of cases, demonstrating a clear improvement over not applying the pattern (which answers **Q1**). Furthermore, we observed a significant difference between the two comparison methods regarding the ratio of false positives to false negatives. The method based on distribution similarity predominantly produces false negatives, while the method based on maximum measurement results mostly in false positives with very few false negatives.

This discrepancy arises naturally from the different comparison mechanisms. The maximum comparison method struggles with algorithms having multiple correct solutions (e.g., Grover's algorithm), where two distinct solutions may both be valid but are rejected as different. On the other hand, the distribution similarity method fails when both results are heavily affected by errors but not entirely distinct. In such cases, the distributions appear similar, even though the likelihood of at least one result being incorrect is high.

Interestingly, the results across all variant types are consistent, indicating that either the specific variant type does not significantly contribute to Comparator errors or the evaluated variant types were too similar. This presents an opportunity for future improvement, as a broader range of variants could potentially enhance overall performance. So from this experiment a definitive answer to **Q2** can not be given.

Eventually, we evaluated the Comparator pattern on the restricted benchmark, with results presented in the lower three rows of Table 5.1 (marked with an (R)). While the overall results remain similar, certain differences are worth noting. The maximum comparison method improves slightly, with a noticeable shift from false positives to correct results. This aligns with our earlier observation that the maximum comparator struggles in scenarios with multiple correct solutions, which are less frequent in the restricted benchmark. Conversely, the distribution similarity method performs slightly worse in the restricted benchmark, likely because algorithms that produce uniform superpositions (common in e.g. state preparation) favor distribution-based comparisons over maximum-based comparisons.

Overall, the Comparator pattern demonstrates potential for significant improvements in reliability. Certain settings achieved up to 80% correct verdicts, which could lead to substantial reliability enhancements in a quantum computing component.

5.3.3 Sparing Pattern Evaluation

For the Sparing pattern, we conducted similar experiments using three qrChannels and error detection based on similarity metrics of distributions, as described in Section 5.2.3. By employing three qrChannels instead of two, the total number of possible configurations increased for each variant type. Specifically, we evaluated 260,448 configurations for the backend variant type, 940,800 configurations for the seed variant type, and 192 configurations for the optimization variant type. As the similarity metric, we used the Hellinger distance with a threshold of 0.4, compared to the uniform distribution.

The primary objective of these experiments was to assess how often the final result of a Switch corresponds to the correct result. We reused but adapted the definitions of correct, false positive, and false negative from the Comparator pattern. Specifically:

- A solution is considered a **false negative** if at least one qrChannel contains the correct solution, but the Switch fails to identify it.
- Conversely, a solution is considered a **false positive** if none of the qrChannels provide the correct result, but the Switch incorrectly accepts the result.
- Otherwise, the solution is considered **correct**.

Using this adapted definition we reuse **M1** as the metric to evaluate the Sparing pattern. The results of this evaluation are summarized in Table 5.2. The performance of the Sparing pattern is mixed. Across all variant types, the Switch produces a correct result in at most 74% of cases, achieved for the backend variant type. However, this statistic includes configurations where all qrChannels return a correct result. In the more critical scenarios where not all qrChannels are correct, the Switch identifies the correct solution in less than 50% of cases. In other words, in the most challenging cases, the

Sparing pattern does worse than a random selection strategy for identifying whether a qrChannel provides the correct result. This is shown in Table 5.3.

Despite these limitations, the Sparing pattern still functions as an effective error detection mechanism. With an overall correct identification rate of 73%, it offers a meaningful improvement in reliability compared to systems without such error-detection measures (**Q1**). Regarding the different variant types we observe similar results as for the Comparator pattern which show no significant differences between the respective variant types (**Q2**).

5.3.4 Evaluation of the Combiner Pattern

The final pattern under evaluation is the Combiner pattern. We examine a Combiner with a linear opinion pool with uniform weights as aggregation method. Again we use three qrChannels selected from either of the variant types. Since the number of qrChannels is identical to the one for the Sparing pattern so is the number of total configurations considered. This set-up of the Combiner is arguably the simplest one and further improvements can possibly be achieved by fine-tuning this set-up for a given variant generation type or specific application areas.

Unlike the previous two patterns, the Combiner pattern does not merely accept or reject the outputs of its qrChannels instead, it produces a new result that may differ from any individual output of its qrChannels. As a result, this pattern cannot be assessed based on correctly identified errors. Instead, we must consider metrics that evaluate the quality of the produced result. For this purpose, we chose the following two metrics:

M2 Position of the correct solution

M3 Hellinger distance to the true distribution

M2, *position of the correct solution*, measures the number of measurement results that need to be considered to find the first correct solution, assuming the measurements are ranked by likelihood (or equivalently, the position of the solution within the set of measurements when ordered by counts). The rationale behind this metric is that the correct solution should appear among the most likely results; thus, a lower score indicates better performance. Similarly, an ideal distribution would be identical to the theoretically optimal solution, resulting in a Hellinger distance of 0. We thus use **M2** to capture this notion. Additionally, as an additional observation we provide a direct comparison of both **M2** and **M3** values for each configuration. Specifically, we analyze how often the Combiner pattern outperforms the average of the qrChannels it is applied to, with respect to these metrics. Intuitively, if the Combiner pattern yields better results than the average qrChannel, it signifies an expected improvement over randomly selecting the output of any single qrChannel. The averages of metrics **M2** and **M3** for the Combiner and the average variant, as well as the direct comparisons are given in Table 5.4.

Overall, the findings are very promising. The Combiner pattern demonstrates superior performance according to all considered metrics. The average position of the correct result is substantially lower than the one of the average qrChannel. This is underlined by the fact that the Combiner beat the average channel in this metric in upwards of 80% of the considered configurations. While the difference in absolute values for the Hellinger distance is comparatively small in the direct comparison the Combiner produces a better result in virtually every case. This demonstrates that the Combiner pattern is very well suited to increase the reliability of obtained results (**Q1**). As for the last two experiments **Q2** could not be answered definitively based on the obtained data.

5.3.5 Discussion of Evaluation Results

The overall evaluation results are promising. **Q1** can confidently be answered affirmatively: the application of fault-tolerant patterns increases the reliability of quantum software components in all examined scenarios. Regarding **Q2**, we did not observe any significant differences between the types of variants considered. This could either suggest that the examined variants are too closely related (a plausible hypothesis given that all variants are based on transpilation options) or that the type of variant used is less significant than initially assumed. Additional experiments are required to provide a definitive answer to this question.

Concerning **Q3**, the results varied depending on the specific fault-tolerant pattern applied. Based on the experiments, we argue that the Combiner pattern is the most promising in the context of quantum computing. Furthermore, the performance of the patterns was highly algorithm-dependent. Even different types of the same pattern, such as the comparison methods used in the Comparator pattern, exhibited significant variations in performance across different algorithms. This highlights the need for further research to identify the best instantiation options for specific scenarios.

5.3.6 Threats to Validity

Several threats to validity must be considered in the context of the conducted evaluation. The most pressing concern is the exclusive use of simulators for all experiments due to limited access to real quantum devices. It is plausible that real quantum devices might behave differently from their simulated counterparts, potentially undermining the presented results. Nevertheless, a smaller evaluation reported in [127] was conducted on real devices and showed similar results, which lends credibility to the findings.

A second threat to validity is the relatively small number of algorithms in the benchmark. Evaluating the patterns on a more extensive set of algorithms would reinforce the general applicability of the approach.

Lastly, all experiments were conducted using a default instantiation. While this instantiation is valid, alternative configurations might yield significantly better or worse results. A broader evaluation encompassing a more complete set of parameters could demonstrate that the patterns are broadly applicable and not restricted to the specific instantiations used in this study.

5.4 Related Work

To the best of our knowledge, there exist no other works which applied fault-tolerant redundancy patterns on an architectural level to quantum software. In the following, we present the closest related work in two different directions: work tackling the issue of NISQ-errors (but not on the architectural level) and the application of redundancy patterns on the architectural level (but not to quantum software).

Dealing with NISQ-Errors Most closely related are approaches that tackle NISQ-errors (much like we did) on the algorithmic rather than the architectural level. The underlying idea is to employ some type of post-processing in order to reduce the amount of noise observed in the final result. These approaches can be summarized under the term quantum error mitigation. We consider the definition of quantum error mitigation given in [30] which defines it as "[...] algorithmic schemes that reduce the noise induced bias in the expectation value by post-processing outputs from an ensemble of circuit runs, using circuits at the same noise level as the original unmitigated circuit or above". A related field of research is quantum error correction which aims to eliminate errors altogether. Despite proofs that quantum error correction is possible [132] (as it is for the classical case), this requires very low gate error rates and a prohibitively high number of physical qubits [88, 95, 63] and is thus not suitable for NISQ-devices. A comprehensive survey on the topic of quantum error mitigation can be found in [30].

One of the most well known approaches for error mitigation is zero-noise-extrapolation (ZNE) [100]. The central idea of zero-noise-extrapolation is to create different variants of the original circuit in which different levels of additional noise are deliberately introduced. Depending on how the level of noise changes, one can extrapolate how the circuit would behave without any noise. Using this extrapolation one can reduce the amount of noise observed in measurements. This method has to be adjusted to every circuit individually and requires the execution of several modified circuits before being able to employ it. As ZNE is comparatively easy to implement it is one of the most used methods in practice [30]. Several different variants of ZNE exist, building on different extrapolation methods, sampling methods, and ways to modify the original circuit.

Multiple methods are concerned with the mitigation of measurement errors in particular. A common approach to measurement error mitigation is calibration-based correction. This method typically involves running calibration circuits that provide a measurement error matrix, describing the error probabilities for each possible measurement outcome [145, 85]. The inverse of this matrix is then applied to the noisy outcomes of quantum experiments, yielding estimates of the true state distribution.However, while this method is effective for low-depth circuits, its scalability to large quantum systems is limited due to the exponential growth in the size of the error matrix.

Another approach to error mitigation are learning based approaches [45, 139]. In this approach a noise-reduction function is learned from classically simulatable circuits. More specifically a number of small and easy to simulate circuits are derived that are meant to resemble the structure and error pattern of the original circuit. Since those circuits are simulatable, a perfect noise-free result can be obtained and serves as ground truth to learn how errors affect these circuits. The assumption is that an error model learned this way can then be transferred to the original circuit. Experiments have shown this approach to work in practice [144].

A very closely related approach to the Combiner Pattern is examined in [142]. Tannu et al. specifically try to find different mappings of the same algorithm to a hardware maximizing diversity. The combination of the resulting measurements of these different mappings is shown to provide an improvement over a single mapping. This aligns perfectly with the findings for our Combiner pattern. While the underlying idea of the two approaches is very similar, the Combiner pattern is more general and thus allows for a larger set of configurations compared to the approach presented in [142]. We already touched on the idea of specifically designing variants to maximize diversity earlier and this work indicates that examining this further could be a promising direction for future research.

Application of Fault-tolerant Patterns in Other Domains Fault-tolerant architecture patterns have been applied to a variety of systems. In this chapter we focus on the application to other non-classical systems.

In [103] the authors describe a fault tolerant architecture pattern for safety-critical automated driving applications. They present a new pattern which is specifically crafted to be applicable to the domain of automated driving. In this sense, this approach is similar to ours, however, our considered domain is significantly broader. Their approach is also based on the Sparing and Comparator patterns as is ours.

A similar approach for the more general case of deep neural networks (DNNs) is presented in [148] by Xu et. al. The authors propose an N-version programming for DNNs. This is closely related to the Voter/Combiner pattern we used. The paper shows how to adapt the pattern to the specific scenario of DNNs and pays special attention to separated training data and the diversity of the employed models. This is akin to our adaptations of the patterns to the quantum computing scenario, including the question of how to obtain different variants for a given problem.

5.5 Conclusion

In this chapter, we presented an approach to mitigate quantum gate and measurement errors that arise in current NISQ devices. Specifically, we addressed this challenge at the architectural level, demonstrating how well-established architectural patterns for fault-tolerant systems can be adapted to the context of quantum computing.

Our evaluation results indicate that the application of fault-tolerant patterns in this domain is highly promising. While the performance of the patterns varies, the Comparator and Combiner patterns, in particular, exhibit significant improvements compared to the selection of a random qrChannel.

Chapter 6

Translation-based Verification of Quantum Software

In this chapter, we present a translation-based verification approach for hybrid quantum software. We demonstrate that this approach is sound relative to an underlying verification tool and demonstrate its practical relevance by proving the correctness of several well-known quantum algorithms. With this work we provide answers to **RQ4** for quantum software.

6.1 Motivation

The advocacy for formally verifying software is a longstanding call within the formal methods community. However, in the context of quantum computing, there is reason to believe that this call will resonate more strongly and gain broader adoption. Beyond the general benefits of formal verification—such as the rigorous guarantees it provides regarding software correctness—the advantages of formal methods are particularly compelling in the quantum computing domain. We identify three key arguments that underscore this increased relevance that we discuss in more detail in the following.

Inherent Complexity of Quantum Algorithms The very nature of quantum computing is fundamentally unintuitive as it is based on quantum mechanical effects that are hard to grasp for humans. Richard Feynman, a pioneer in the realm of quantum computing, is often quoted with the words "I think I can safely say that nobody understands quantum mechanics". Yet it is essential to consider these effects when thinking about quantum software as only these effects enable the full potential of quantum computing. However, these principles that enable quantum computation also make understanding and developing quantum algorithms exceptionally challenging. This complexity leads to an increased likelihood of implementation errors, which are often difficult to detect and correct. In such a context, formal verification becomes crucial, providing a rigorous framework to ensure the correctness of quantum algorithms and mitigate the risks associated with their development.

Challenges in Testing Quantum Software In classical software development, testing remains the most widely used method for quality assurance. However, applying similar testing strategies to quantum software presents unique challenges [33]. The inherent probabilistic nature of quantum computing poses a significant obstacle to conventional testing methodologies. While a single test execution suffices for classical programs, quantum programs may require a prohibitively large number of executions to achieve statistically meaningful results for a single test, depending on the expected probability of the desired outcome. Moreover, even when a test fails, it may be difficult to discern whether the failure is due to an actual bug or simply an improbable outcome.

Additionally, the modular approach commonly used in unit testing — where small, isolated components of code are tested individually — often proves infeasible for quantum software. Key properties of quantum states, such as phase information, cannot be directly observed or measured without potentially altering the state itself. This limitation makes it difficult to test quantum programs in the granular, iterative manner that is typical in classical software development. Consequently, testing quantum software is not only more complex but also less effective than in classical contexts. These testing difficulties underscore the importance of formal verification methods, which can rigorously ensure correctness without suffering from similar problems.

High Cost of Quantum Computation Currently, and likely for the foreseeable future, access to quantum computers—especially the most powerful ones—is highly constrained and costly. A subscription to the H1 system of Quantinuum via Azure Quantum costs 125,000$ per month [1]) or one

[1] Azure Quantum Pricing, last visited 2024/11/27

hour of compute time on an IBM system costs 5760$ [2] compared to 197,60$ for the most expensive AWS instance (448 cores, 18GB Memory)[3]. This scarcity of quantum computing resources places a premium on making the most efficient use of available computing time. Minimizing the execution of buggy or incorrect programs is essential, as the cost of rerunning quantum computations can be prohibitive (often quantum systems have to be paid either by computing time or by the number of shots and gates executed[1,3]). Consequently, the need to ensure the correctness of quantum software is significantly higher than in classical computing environments.

This scenario creates a strong incentive for the adoption of formal verification methods. Just as in other high-stakes fields, such as spacecraft or hardware development, where the deployment of faulty software can lead to catastrophic failures, formal methods are commonly employed to guarantee correctness. In quantum computing, where both the cost and opportunity to run programs are limited, formal verification offers a robust approach to ensuring that the software is reliable and error-free before execution.

In summary, the intrinsic complexities of quantum algorithms, the formidable challenges associated with testing quantum programs, and the substantial costs linked to quantum computation all converge to make a strong case for the adoption of formal verification methods in quantum computing.

6.2 Requirements for a Formal Analysis Tool for Quantum Software

We are now going to consider what properties such a formal method should have in order to be the most useful in practice. We deliberately pronounce that our focus is on the practical application of the developed method. To this end, we acknowledge that certain trade-offs may be necessary to enhance its practical utility. With this perspective in mind, we identify the following properties as essential for a formal method in quantum computing.

High Degree of Automation Quantum algorithms are inherently complex and counterintuitive, making manual verification of their specifications both challenging and time-consuming. Therefore, we contend that a practically relevant formal method for quantum software must prioritize a high degree of automation. Such automation offers several key advantages: (a) it simplifies the verification process, allowing even non-experts to apply the

[2]AWS Pricing, last visited 2024/11/27
[3]IBM Quantum pricing, last visited 2024/11/27

method with minimal effort; and (b) it integrates seamlessly with CI/CD pipelines, thereby providing a viable alternative to traditional testing methods currently employed for classical software. However, it is important to note that high levels of automation in verification are typically associated with limitations on the complexity of properties that can be proven. We will go into more details of this trade-off later, discussing why we think, in this particular context, a higher degree of automation is advantageous despite potential limitations. When considering automatic approaches, the verification time should ideally be minimal; therefore, faster verification tools are preferable to slower ones.

Hybrid Approach Quantum algorithms are almost invariably part of hybrid systems, which combine quantum and classical components. For instance, some quantum algorithms, such as Shor's algorithm, include classical subroutines as fundamental parts of their operation, while others serve as subroutines within broader classical software systems. Consequently, a formal method for quantum software must be designed to support the verification of these hybrid quantum-classical systems. The approach should be capable of analyzing both quantum components and their interactions with classical elements, ensuring comprehensive verification across the entire system. This hybrid approach is essential for the practical application of formal methods in real-world quantum software development.

Realistic Programming Language Support To be of practical value to typical developers, the formal method should support realistic quantum programming languages. Many existing formal methods are based on simplified or toy languages that, while useful for theoretical exploration, are not employed in real-world development. Such limitations significantly reduce the utility of these methods for everyday programming tasks. Therefore, it is crucial that the formal verification tool is compatible with widely used quantum programming languages, such as Qiskit, Cirq, or Q#. This ensures that the method is relevant and applicable to the development of real-world quantum applications, rather than being confined to academic or experimental use.

Support for Debugging While proving the correctness of programs is important, the ability to identify and correct bugs is perhaps even more critical. If a verification tool merely indicates that a property does not hold, it can be extremely challenging for developers to pinpoint the source of the problem. Therefore, robust support for debugging faulty programs is essential. One effective method of providing useful feedback is through the generation of counterexamples—specific inputs and execution traces that demonstrate why a certain property is violated. In the context of quantum computing, it is especially important that these inputs and traces are presented in a clear and comprehensible manner. Quantum traces, often represented as large and complex vectors, can be difficult to interpret without sufficient explanation or summarization. Thus, the tool should offer human-friendly debugging support, making it easier for developers to understand and rectify errors in their quantum-classical programs.

Minimal Specification Specification has long been one of the major bottlenecks for formal verification in practice [120, 17]. In many tools, not only must the desired property be specified, but the tool also relies on extensive auxiliary specifications [19, 47] to guide the proof process, without which successful proofs would be infeasible. A primary source of this specification burden is the need to specify subroutines and loop invariants. For practical applications of formal methods, this requirement and the associated effort pose a considerable barrier. Consequently, an application-oriented approach is significantly enhanced by specifications that are concise and easily interpretable. Any tool capable of conducting proofs with minimal specifications written in commonly used specification languages represents a meaningful advancement towards practical applicability.

Having collected these crucial properties of a verification approach for quantum software, we present an approach that meets all the desirable properties mentioned above. We show how this can be achieved by translating quantum circuits written in a real-life programming language like Qiskit into a classical host language. Based on this translation, the verification of quantum circuits is reduced to the verification of the translated classical program for which any existing verification method can be used. We define the translation formally for a limited language and then show how to adapt this translation to real-life programming languages.

6.3 Syntax and Semantics of Quantum Circuits

In this section we present syntax and semantics for both a quantum circuit language and a simple classical language. Those two languages serve as the basis for the translation approach we present in the next section. We start by introducing a generic language *QC* for quantum circuits and define its semantics. In order to keep our definitions as readable as possible, we use `typewriter` font whenever using concrete program code and normal font for functions executed during the translation.

Definition 6.1 (Syntax of quantum circuits). *The language QC of quantum circuits is defined by the following grammar (we assume a language for complex-number expressions to be given) given a set of free variables* Σ*:*

$$\begin{aligned} C \; &::= C\texttt{;}C \mid \texttt{M}\ k \mid \texttt{G}\ g\ k \\ k \; &::= \textit{a natural number } (k \in \mathbb{N}_0) \\ g \; &::= \textit{a matrix of complex-number expressions over } \Sigma \end{aligned}$$

A quantum circuit is composed of a sequence of two distinct types of operations: gate applications ($\texttt{G}\ g\ k$) and measurements ($\texttt{M}\ k$). The application of a gate is specified by the gate's matrix g and the index of the first qubit k to which it is applied. The gate g can be any suitable matrix (more on that shortly). While we do not impose a restricted set of gates in our definition, such a restriction could be easily implemented if needed. We implicitly assume that multi-qubit gates are applied only to consecutive qubits (e.g. $k, k+1, ..., k+(m-1)$, for an m-qubit gate). This assumption does not result in a loss of generality, as qubits can always be reordered using a swap gate, which itself is an operation expressible as matrix multiplication and thus permissible within our language.

As an example, consider the application of an arbitrary 2-qubit gate g to qubits 0 and 2, which is not directly supported by our syntax. This operation can be achieved through the following equivalent *QC* program: $\texttt{G}\ S\ 1\texttt{;}\texttt{G}\ g\ 0\texttt{;}\texttt{G}\ S\ 1$. Here, S represents a swap gate that exchanges the qubit it is applied to with the adjacent qubit (in this case, qubit 1 with qubit 2). Notice that to preserve the original qubit order, it is necessary to swap the qubits back to their initial configuration after applying g.

Measurements are always performed on individual qubits (k). While many quantum algorithms involve measuring all qubits simultaneously at the end of a computation, there is no theoretical difference between parallel and sequential measurements (assuming the absence of errors such as decoherence). Therefore, we focus on the measurement of single qubits. All measurements are considered to be executed in the computational basis. Again this is without the loss of generality as measurements in any other basis can be achieved by applying an appropriate basis change before measuring.

Sequences of these two types of operations—gate applications and measurements—can be constructed using sequential composition.

Note, that the gate matrices are matrices over complex-valued *expressions*. As such, gates can be parameterized in a given set of variables. This set of variables can be considered the input or parameters to the circuit as it would otherwise have static behavior.

This definition of a circuit, however, allows for ill-defined circuits which do not fit the properties we introduced in Section 2.1.5. To account for this we define well-defined circuits:

Definition 6.2 (Well-definedness). *A circuit C is well-defined iff for each gate application* `G` g k *and measurement* `M` k *the following conditions are met:*

- g *is a unitary matrix of size* $2^m \times 2^m$ *with* $m \leq N$
- $k \leq N - m$ *for gates and* $k < N$ *for measurements*

where N is the number of qubits used in C.

A well-defined circuit fits exactly what we introduced in Section 2.1: Gates are unitary matrices applied to, possibly multiple, qubits and measurements are conducted for single qubits at a time. A gate can be applied to up to N qubits and consequently has a gate-matrix of size $2^m \times 2^m$ which limits the qubits to which it can be applied (remember that all gates are applied to consecutive qubits only).

Note that a circuit may be well-defined for certain values of variables from Σ only. In the remainder of this chapter, when we talk about well-defined circuits we only consider values for variables in Σ such that the well-definedness criteria are met.

There is a close relation of QC to the graphical representation of quantum circuits. As an example consider the circuit shown in Figure 6.1. It is a graphical representation of the following circuit in QC:

`G` H `1; G` CX `1; G` CX `0; G` H `0; M 0; M 1`

where H and CX are standard gates as introduced in Section 2.1. As you can see, the transcription of a graphically represented circuit into the

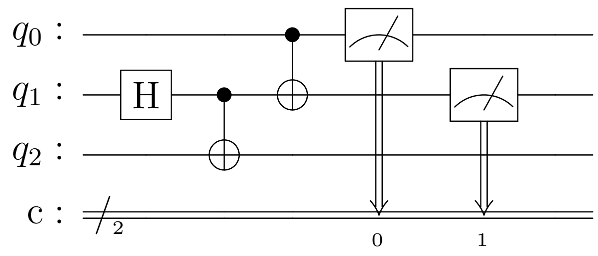


Figure 6.1: Example of the graphical representation of quantum circuits generated by Qiskit

presented textual representation is straightforward. Note that gates applied in parallel in the graphical representation can be transcribed in arbitrary order since they do not influence each other. This may lead to syntactically different circuits with identical semantics.

Before defining the semantics of circuits, we introduce two helpful notations.

Definition 6.3 (Bit sets). *We call $B_0(n,k), B_1(n,k) \subseteq \mathbb{N}_0$ the bit sets for n and k, where $B_0(n,k)$ resp. $B_1(n,k)$ consist of the natural numbers $\leq n$ (including 0) such that the k-th bit in their binary representation is 0 resp. 1. Formally:*

$$B_0(n,k) = \{j \in \mathbb{N}_0 \mid j < n \text{ and } [\![j]\!]_2[k] = 0\}$$
$$B_1(n,k) = \{j \in \mathbb{N}_0 \mid j < n \text{ and } [\![j]\!]_2[k] = 1\}$$

where $[\![j]\!]_2[k]$ is the k-th bit in the binary representation of j.

Thus, the bit-set $B_0(n,k)$ is the set of all natural numbers smaller than n such that the k-th bit in their binary representation is 0. As an example consider: $B_0(8,1) = \{0,1,4,5\}$

As a second notation, we introduce *expansion gate matrices*. Consider a (small) gate g to be applied to the i-th qubit in a larger system with several more qubits. That is equivalent to applying g to the i-th qubit while applying the identity gate I to all other qubits (see Section 2.1.5). The resulting matrix operating on the entire state is what we call the expansion matrix:

Definition 6.4 (Expansion matrix). *We call $E(g, k, N)$ the expansion matrix for applying an m-qubit gate g to qubit k of an N-qubit system:*

$$E(g, k, N) = I^k \otimes g \otimes I^{N-m-k}$$

We abuse notation to use I^0 as a scalar. This way a gate applied to the first or last qubit is only expanded at the end, respectively, the beginning. Notice again the assumption that an m-qubit gate is applying to m succinct qubits and thus the expansion only deals with the qubits before and after the gate itself. As an example consider the following case:

$$E(H, 1, 4) = I^1 \otimes H \otimes I^2 = \begin{pmatrix} H & 0 \\ 0 & H \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We proceed to define the semantics of quantum circuits. We use a transition function δ to represent the state changes that the syntactical elements of a circuit entail. The arguments of δ are a circuit and a state. The state is a triple (q, c, p) where q is the current quantum state, c a list of the measurement results observed so far, and p the probability of reaching that state. As a result, the transition relation returns a *set* of new states (a set of triples).

Definition 6.5 (Circuit semantics). *Let $n = 2^N$, where N is the number of qubits, then, the transition function*

$$\begin{aligned} \delta \ : QC \times (\mathbb{C}^n_{\|\cdot\|_2=1} \times \{0,1\}^* \times \mathbb{R}_{[0,1]}) \ \longrightarrow \\ \mathcal{P}(\mathbb{C}^n_{\|\cdot\|_2=1} \times \{0,1\}^* \times \mathbb{R}_{[0,1]}) \end{aligned}$$

is defined as follows:

$$\begin{aligned} \delta(\texttt{G}\ g\ k,\ (q,c,p)) &= \{\, (E(g,k,n)\cdot q, c, \quad p)\} \\ \delta(\texttt{M}\ k,\ (q,c,p)) &= \{\, (q^{k\to 0}, \quad c \circ \langle 0\rangle,\ p \cdot Pr(k,n,0,q)), \\ &\qquad (q^{k\to 1}, \quad c \circ \langle 1\rangle,\ p \cdot Pr(k,n,1,q))\} \\ \delta(\, C_1 ; C_2,\ (q,c,p)) &= \{\, (q', c', p') \mid \\ &\qquad \textit{there is } (q'', c'', p'') \in \delta(C_1, (q,c,p)) \\ &\qquad \textit{s.t } (q', c', p') \in \delta(C_2, (q'', c'', p''))\ \} \end{aligned}$$

where the probability of a measurement of qubit k in an N-qubit state *returning result b is*

$$Pr(m, n, b, q) = \sum_{j \in B_b(n,m)} |q[j]|^2$$

and the quantum state vectors $q^{k \to b}$ (with $b = 0$ or $b = 1$) resulting from measurement are defined by: for $1 \leq j \leq n$, the j-th element of $q^{k \to b}$ is $q^{k \to b}[j] = 0$ if $j \notin B_b(n, k)$ and is $q^{k \to b}[j] = q[j]/\sqrt{Pr(k, n, b, q)}$ otherwise.

Let's break this down: The application of a gate g solely affects the quantum state. The resulting state is obtained by applying the gate g to the qubits in question, specifically using the expansion of g to the appropriate size (see Definition 6.4). No additional measurements are recorded, and the probability of reaching this new state remains unchanged (this is to be expected as the application of a gate is deterministic).

The most interesting aspect of the definition of δ regards measurements, as all components of the state triple are affected. When a measurement is performed, the quantum state collapses, the measured classical bit is stored in c, and the probabilities for the two possible resulting states must be adjusted. To represent the quantum state collapse, all elements of the state vector that are inconsistent with the observed measurement outcome are set to zero, and the state is subsequently normalized. This normalization is done by dividing each element of the resulting vector by the probability of observing the post-measurement state. If this probability is zero, the corresponding state is unreachable, so any division by zero is inconsequential in this scenario. The measured bit is stored in a new bit, which is appended to the list of observed measurements, and the probabilities are updated accordingly.

Finally, the sequential composition of commands has the expected semantics: from a pre-state (q, c, p), a final state (q', c', p') is reachable via $C_1;C_2$ if and only if an intermediate state (q'', c'', p'') is reachable via C_1, and from there, (q', c', p') is reachable via C_2. This formalization of sequential composition accounts for the fact that any of the sequentially composed subprograms may yield a set of states as a result, thereby considering all possible execution paths. It is important to note that this is a nondeterministic semantics, as measurements inherently introduce nondeterminism. Given a circuit C and an initial quantum state q, the presented transition relation δ generates the set of all possible execution traces for the circuit C when applied to the initial state $s = (q, \epsilon, 1)$.

As a clarifying example consider the following circuit given in QC with

2 qubits:

`G` H `0; G` CX `0; M 0`

As customary, we consider the execution of the above circuit starting from the state $q = |00\rangle$. The resulting trace looks like the following:

$$\begin{aligned}
q_0 &= \{(|00\rangle, \epsilon, 1)\} \\
q_1 = \delta(\texttt{G}\ H\ 0, q_0) &= \{(1/\sqrt{2}(|00\rangle + |01\rangle), \epsilon, 1)\} \\
q_2 = \delta(\texttt{G}\ \text{CX}\ 0, q_1) &= \{(1/\sqrt{2}(|00\rangle + |11\rangle), \epsilon, 1)\} \\
q_3 = \delta(\texttt{M}\ 0, q_2) &= \{(|00\rangle, \{0\}, 0.5), \\
&\qquad (|11\rangle, \{1\}, 0.5)\}
\end{aligned}$$

The first state is the starting state where the quantum state is as set, we have no observed measurements yet, and the probability of reaching that state is 1. The next two states are gate applications which only change the quantum state. Finally, q_3 is the resulting set of states after the measurement. Notice how there are now two possible states each with probability less than one and the list of measurements got extended to include the observed value. Now imagine in this case we would add another measurement for qubit 1:

$$\begin{aligned}
q_3 = \delta(\texttt{M}\ 0, q_2) &= \{(|00\rangle, \{0\}, 0.5), \\
&\qquad (|11\rangle, \{1\}, 0.5)\} \\
q_4 = \delta(\texttt{M}\ 1, q_3) &= \{(|00\rangle, \{0,0\}, 0.5), \\
&\qquad (-, \{0,1\}, 0), \\
&\qquad (-, \{1,0\}, 0), \\
&\qquad (|11\rangle, \{1,1\}, 0.5)\}
\end{aligned}$$

Notice in this case that some states in the set of states after the second measurement have probability 0. In the same states, the quantum part of the state is no longer defined as a division by 0 would be necessary to compute this state. In the remainder of this chapter we leave out these states and thus limit the set of states to the ones with non-0 probability.

$$\begin{aligned}
q_4 = \delta(\texttt{M}\ 1, q_3) &= \{(|00\rangle, \{0,0\}, 0.5), \\
&\qquad (-, \{0,1\}, 0), \\
&\qquad (-, \{1,0\}, 0), \\
&\qquad (|11\rangle, \{1,1\}, 0.0)\} \\
&= \{(|00\rangle, \{0,0\}, 0.5), \\
&\qquad (|11\rangle, \{1,1\}, 0.5)\}
\end{aligned}$$

Here one can clearly see that the second measurement had no effect on the set of states except for adding another measurement outcome. This is to be expected as the example again resembles the creation (and destruction by measurement) of the well-known Bell-state Φ^+ that we already saw in Section 2.1.5. After the first measurement, the value of the second qubit is already determined and a measurement of it always yields the same value as the first measurement.

We have successfully defined a language for quantum circuits that is equivalent to the typically used graphical representation for quantum circuits but has some syntactic limitations to ease definitions. We now establish a classical host language in which this circuit language can be simulated. We call this classical language *QP*. The syntax of *QP* is defined as follows:

Definition 6.6 (Syntax of the target language)**.** *The syntax of the language QP is defined by the following grammar (we assume a language for complex-number expressions to be given):*

$$\begin{aligned}
S \quad &::= \mathit{var}\ \texttt{:=}\ \mathit{expr} \mid \mathit{var}\ \texttt{:=}\ \texttt{*} \mid \texttt{skip} \mid S\texttt{;}S \mid \\
&\qquad \texttt{if}\ \mathit{bexpr}\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{fi} \\
\mathit{aexpr} &::= \mathit{var} \mid \mathit{literal} \mid \mathit{aexpr}\ \texttt{+}\ \mathit{aexpr} \mid \mathit{aexpr}\ \texttt{/}\ \mathit{aexpr} \mid \\
&\qquad \mathit{aexpr}\ \texttt{*}\ \mathit{aexpr} \mid \mathit{aexpr}\ \texttt{-}\ \mathit{aexpr} \mid \sqrt{\mathit{aexpr}} \mid \mathit{abs}(\mathit{aexpr}) \\
\mathit{bexpr} &::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ \mathit{bexpr} \mid \mathit{bexpr}\ \texttt{and}\ \mathit{bexpr} \mid \\
&\qquad \mathit{bexpr}\ \texttt{or}\ \mathit{bexpr} \mid \mathit{aexpr}\ \texttt{>}\ \mathit{aexpr} \mid \mathit{aexpr}\ \texttt{=}\ \mathit{aexpr}
\end{aligned}$$

This language primarily consists of standard elements commonly found in classical programming languages, such as algebraic and Boolean operations, as well as control structures like if-then-else statements. However, it also incorporates a few distinctive features. First, the language requires support for absolute value operations (which we may denote as $|\mathit{aexpr}|$ for convenience) and square root calculations. The necessity of these operations will become clear in the subsequent definition of the translation. While these operations may not be as ubiquitous as others, they are supported by all widely used programming languages today—either as built-in functions or through user-defined implementations—so this requirement is easily met.

Second, the language includes support for complex expressions and operations. However, this can be considered primarily as syntactic sugar, since all such operations can be reduced to standard real arithmetic operations (see Section 6.5 for further details). We assume that all variables and expressions are complex-valued. In the context of Boolean expressions, any value not equal to zero ($\neq 0$) is treated as `false`.

Third, we intentionally avoid loops in this definition. While we will later lift this limitation and provide restricted support for loops, for now, we focus on a simpler language that avoids certain pitfalls associated with their usage.

Eventually, the language requires support for nondeterministic assignments, denoted as x `:= *`. Such an assignment assigns a nondeterministically chosen value to x. This feature is particularly useful for modeling the inherent nondeterminism in quantum circuits introduced by measurements. In some programming languages, this operation is also referred to as `havoc`.

We continue by defining the semantics for QP as follows:

Definition 6.7 (Semantics of the target language)**.** *Let S be the set of all states of QP, where a state is an assignment of values to variables. Then, the transition function*

$$\delta_{QP} \;:\; QP \times S \;\mapsto\; \mathcal{P}(S)$$

is defined as follows:

$$\begin{aligned}
&\delta_{QP}(\mathit{var} \mathrel{\texttt{:=}} \mathit{expr}, s) = \{s[\mathit{var} \leftarrow \mathit{val}_s(\mathit{expr})]\}\\
&\delta_{QP}(\mathit{var} \mathrel{\texttt{:=}} \texttt{*}, s) = \{s[\mathit{var} \leftarrow v] \mid v \in \mathbb{C}\}\\
&\delta_{QP}(\texttt{skip}, s) = \{s\}\\
&\delta_{QP}(S_1 \texttt{;} S_2, s) = \{s' \mid \textit{there is } s'' \in \delta_{QP}(S_1, s) \\
&\qquad\qquad\qquad\qquad \textit{such that } s' \in \delta_{QP}(S_2, s'')\}\\
&\delta_{QP}(\texttt{if}\, P\, \texttt{then}\, S_1\ \texttt{else}\, S_2\, \texttt{fi}, s) = \begin{cases} \delta_{QP}(S_1, s) \ \textit{if } \mathit{val}_s(P) = \mathit{true} \\ \delta_{QP}(S_2, s) \ \textit{if } \mathit{val}_s(P) = \mathit{false} \end{cases}
\end{aligned}$$

where $\mathit{val}_s(\mathit{expr})$ is the value of expression expr in state s (defaulting to 0*) using the natural semantics for algebraic and Boolean expressions, and the state $s[\mathit{var} \leftarrow v]$ is the result of updating the value of variable var to v in state s.*

This definition resembles the standard semantics for a classical programming language with the given syntax. The only slight deviation from typically used semantics is the nondeterministic notion of the nondeterministic assignment. In this case there is a valid successor state for each value $v \in \mathbb{C}$.

6.4 Property-preserving Translation

We now present the core of our approach: a translation of quantum circuits into a classical language. We subsequently prove that this translation preserves the semantics of the original quantum circuit.

The translation is given as a function $\sigma : QC \mapsto QP$ which operates purely on a syntactical level.

Definition 6.8 (Translation of circuits). *Given a number N of qubits and a corresponding state size of $n = 2^N$, we define the syntactic translation $\sigma : QC \mapsto QP$. We consider QP-states in which the following QP variables are defined:* `qs_`i *($1 \leq i \leq n$) of type complex number represent the quantum state,* `ps` *of type real number stores the probability of reaching the current state, and the list* `c` *of measurement outcomes.*

1. $\sigma(S_1;S_2) \rightsquigarrow \sigma(S_1);\sigma(S_2)$
2. $\sigma(\texttt{G}\ g\ k) \rightsquigarrow$

$$\begin{pmatrix} \texttt{qs\_1} \\ \vdots \\ \texttt{qs\_n} \end{pmatrix} := \begin{pmatrix} \texttt{qs\_1*u\_1\_1 + qs\_2*u\_1\_2 + ...} \\ \vdots \\ \texttt{qs\_1*u\_}n\texttt{\_1 + qs\_2*u\_}n\texttt{\_2 + ...} \end{pmatrix}$$

where the `u_`i`_`j *are the elements of the expansion matrix $E(g, k, n)$ (Definition 6.4). We abuse notation to make clear that all those assignments have to be made in parallel.*

3. $\sigma(\texttt{M}\ k) \rightsquigarrow$

```
m := *;
if (m) then $\rho(k,1)$
   else $\rho(k,0)$
fi;
c := $append$(c,m)
```

with the following subprograms $\rho(k, b)$ for $b \in \{0, 1\}$:

```
ps := ps * (|qs_$i_1$|*|qs_$i_1$| + ... + |qs_$i_s$|*|qs_$i_s$|);
if (ps > 0) then
   qs_$i_1$ := qs_$i_1$/$\sqrt{\texttt{p}}$;
   ...;
   qs_$i_s$ := qs_$i_s$/$\sqrt{\texttt{p}}$;
   qs_$j_1$ := 0;
   ...;
   qs_$j_t$ := 0;
fi;
```

where $\{i_1, \ldots, i_s\} = B_b(n, k)$ *and* $\{j_1, \ldots, j_t\} = B_{1-b}(n, k)$.

This translation is specifically crafted to preserve the semantics of quantum circuits: Gate applications are translated into the corresponding matrix multiplications. This is done by a simultaneous update of all elements of the state vector to the dot-product of the previous state and the appropriate row of the gate matrix. It is crucial that this update is done simultaneously as the previous state vector influences each element of the next state vector. Thus updating a single element and then calculating the next one would lead to errors. If a host language has no support for parallel assignments like these, an identical result can be achieved by first calculating the result of the matrix multiplication in a temporary variable and then assigning it to the actual state vector afterward. Measurements are translated into nondeterministic branchings followed by the update of the state according to the result of the measurement. This update means that some state elements have to be set to 0 while others have to be multiplied by a constant factor to normalize the state. To distinguish between the two types of elements, we use bit sets again. The exact implementation of the Bitset is skipped here as this may depend on the actual language used. We translate a measurement to a nondeterministic program. This allows a program analysis or verification to consider all possible measurement results and preserves the nondeterminism introduced by quantum measurements. Note that in the case of a measurement occurring with probability 0 the adaptation of the state is skipped to avoid divisions by 0.

The translation assumes the availability of a data type for complex and real numbers. Alternatively, constructing a program using real numbers only is straightforward as $\mathbb{C}$ is interpreted as $\mathbb{R}^2$ with the appropriate arithmetic adaptations. The handling of real numbers in a real-world programming language is discussed in Section 6.5.

We claim that the semantics of a circuit and its translation are equivalent. To formally prove this we introduce the notion of equivalence of *QP*- and *QC*-states. This definition captures the intuition that a *QP* state is equivalent to a *QC* state iff the three parts of the state, as defined for the semantics of *QC*, are captured in appropriate *QP* variables.

Definition 6.9 (Equivalence of states)**.** *We call a QP-state s_1 and a QC-state $s_2 = (qs, c, p)$ equivalent (modulo renaming of variables) iff*

- $\forall i \in \mathbb{N} : 0 \leq i < n \rightarrow \texttt{qs\_i} \in var(s_1) \land val_{s_1}(\texttt{qs\_i}) = qs[i]$

- $\forall i \in \mathbb{N} : 0 \leq i < |c| \rightarrow \texttt{c\_i} \in var(s_1) \land val_{s_1}(\texttt{c\_i}) = c[i]$
- $ps \in var(s_1) \land val_{s_1}(\texttt{ps}) = p$

$qs[i]$ and $c[i]$ is the i-th element of qs and c respectively. $var(s)$ is the set of variables defined in s. We write $s_1 \cong s_2$ if s_1 and s_2 are equivalent. We also assume a natural expansion of this equivalence definition to sets of states, the three parts of a state, and traces.

We use this definition of equivalence to show a simple lemma which states that there exist equivalent states between *QP* and *QC*.

Lemma 6.1 (Existence of equivalent states)**.** *There exist states s_1 and s_2 such that s_1 is a QP-state and s_2 is a QC-state and $s_1 \cong s_2$.*

Proof by example. Consider the simplest quantum state $|0 \dots 0\rangle$ with no measurements performed. We can construct a *QC*-state, that represents this as qs $= (1, 0, \dots, 0)$, $c = []$, $p = 1.0$. The corresponding *QP*-state is $s = [\texttt{qs\_0} \rightarrow 1, \texttt{qs\_1} \rightarrow 0, \dots, \texttt{ps} \rightarrow 1.0]$. Since there have been no measurements, no c_i exist. We can easily see that $q \cong qs$ and $p \cong ps$. As the classical measurement results c are empty, the equivalence of the state-triples follows. ■

This not only shows Lemma 6.1 but in particular shows that one of these states is the standard initial state for quantum computing as introduced in Section 2.1.5.

With this notion of equivalence in mind, we can now formally state the main theoretical insight of this chapter: The semantical equivalence of a circuit and its translation to *QP*.

Theorem 6.1 (Equivalent semantics)**.** *For any given initial quantum state $s_1 = (qs, c, p)$, and an equivalent QJ-state s_2 (meaning $s_1 \cong s_2$) for any well-defined quantum circuit $C \in QC$ holds:*

$$\delta(C, s_1) \;\cong\; \delta_{QP}(\sigma(C), s_2)$$

We now provide a proof of this theorem by induction over the structure of *QC*.

Proof by Induction.
Induction Hypothesis: Assume that for some arbitrary step, the states $s_1 \in QC$ and $s_2 \in QP$ are equivalent, i.e., $s_1 \cong s_2$.

Induction Step: We show that given the induction hypothesis $s_1 \cong s_2$ after applying a gate operation or performing a measurement, the resulting states s_1' and s_2' remain equivalent, thereby proving the equivalence for the subsequent step.
Gate Operations: We first consider the application of a gate operation in QC. Given equivalent states $s_1 = (q, c, p) \in QC$ and $s_2 \in QP$ for any appropriate gate g and qubit index k:

$$s_1 \cong s_2 \implies \delta(\mathtt{G}\ g\ k, s_1) \cong \delta_{QP}(\sigma(\mathtt{G}\ g\ k), s_2)$$

The semantics of QC define the result of applying gate g to qubit k as:

$$\delta(\mathtt{G}\ g\ k, (q, c, p)) = (E(g, k, n) \cdot q, c, p)$$

where $E(g, k, n)$ is the matrix representing gate g acting on qubit k. Here, only the quantum state q changes, while the recorded measurements c and probability p remain unchanged. In QP, the translation $\sigma(\mathtt{G}\ g\ k)$ similarly modifies the quantum state by updating the values of `qs_i`. The equivalence of the quantum states after the gate application is demonstrated by considering the individual elements q_i of q. For each element:

$$q_i' = E(g, k, n)_i \cdot q$$

where $E(g, k, n)_i$ is the i-th row of the expansion matrix. This is identical to the update applied to `qs_i` in QP. Given the induction hypothesis, $q \cong \mathtt{qs}$, the resulting states after the application of g are still equivalent.

Measurement Operations: Next, we consider the effect of a measurement on qubit k. Formally, we show:

$$\text{Given equivalent states } s_1 \in QC \text{ and } s_2 \in QP \text{ for all appropriate } k:$$

$$s_1 \cong s_2 \implies \delta(\mathtt{M}\ k, s_1) \cong \delta_{QP}(\sigma(\mathtt{M}\ k), s_2)$$

In QC, measuring qubit k splits the state into two possible outcomes, corresponding to measurement results 0 and 1. For each outcome, we obtain one new state with updated quantum state, a new measurement result and corresponding probability. In QP, the measurement operation $\sigma(\mathtt{M}\ k)$ introduces a nondeterministic Boolean variable `m`, where `m` can take values 0 or 1, corresponding to the measurement outcomes. Thus we realize that for both semantics after a measurement we have two subsequent states.

Let's consider the case where the measurement result is 0. We use primed variables to denote the values after the state change (e.g., `ps'` denotes the value of `ps` after the state change). We now show equivalence for each component of the state:
Probability:

$$\begin{aligned}\texttt{ps'} &= \texttt{ps} * \textstyle\sum_{j\in B_0(n,m)} |\texttt{qs\_j}|^2\\ &= p * \textstyle\sum_{j\in B_0(n,m)} |q[j]|^2\\ &= p'\end{aligned}$$

Given that $p \cong$ `ps` which is true by the induction hypothesis, and the fact that the indices i_x correspond exactly to $B_0(n, m)$, the probabilities remain equivalent.
Quantum State:

$$\forall i \in B_0(n,k) : \mathrm{val}_{s_2'}(\texttt{qs\_i'}) = 0 = q'[i]$$

$$\forall i \notin B_0(n,k) : \mathrm{val}_{s_2'}(\texttt{qs\_i'}) = \mathrm{val}_{s_2}(\texttt{qs\_i}/\sqrt{\texttt{ps'}}) = q[i]/\sqrt{p'} = q'[i]$$

We established that $p \cong ps$. The remaining equalities are directly using the definitions of the respective semantics. Thus, the quantum state remains equivalent.
Measurement Results: The last element of c and the last `c_i` are both 0, ensuring they are equivalent. The rest of c remains unchanged and equivalent due to induction hypothesis.

Since all three components of the state are equivalent, the overall state is equivalent after measurement.
Sequential Composition: Finally, the equivalence for the sequential composition of statements follows directly from the equivalence of the subprograms C_1 and C_2. ■

This proof establishes that whenever we can find two equivalent states, executing a circuit and its translation in these states yields equivalent states at each step including the final ones. Together with Lemma 6.1 (and its proof) this culminates in the following Corollary:

Corollary 6.1 (Equivalence of translation). *For any circuit $C \in QC$ that is executed in an initial state $q = |0\rangle$, C and its translation $\sigma(C)$ induce equivalent sets of traces.*

With this, we have established the equivalence between the semantics of a quantum circuit represented in *QC* and its translation to *QP* as defined by σ. This equivalence enables us to transfer any correctness property of a translation $\sigma(c)$ back to the original circuit c. A property of a circuit, and respectively a *QP*-program, can be expressed as a set of pairs, where each pair consists of an initial state and the set of resulting final states. We will call such pairs `TwoStates`. Let T_c be the set of all such possible `TwoStates` for a circuit c. We say c has property φ if $\varphi \subseteq T_c$. The equivalence between c and its translation $\sigma(l)$ that we established guarantees that any property φ that $\sigma(c)$ has is also a property of c. However, meta-properties that can not be expressed by `TwoStates` such as energy consumption, memory usage, or runtime are not covered by this equivalence.

As a result of Corollary 6.1 we call the presented translation $\sigma(\cdot)$ sound. Meaning any bug that is present in the original circuit will be present in its translation. Formally we express this based on `TwoStates`.

Definition 6.10 (Soundness). *Let T_x be the set of all possible* `TwoStates` *of x. Let $\zeta : QC \mapsto QP$ be a translation function from QC to QP. We call ζ sound iff:*
$\forall c \in QC, \forall \varphi : (\exists\, p \in T_c : p \notin \varphi \implies \exists\, p' \in T_{\zeta(c)} : p' \notin \varphi')$
where φ' is the QP-property equivalent to φ according to Definition 6.9.

The soundness of $\sigma(\cdot)$ follows directly from Corollary 6.1.

6.4.1 Hybrid Programs

As we commented on before, most quantum programs do not consist of a single circuit but are hybrid to varying degrees, meaning they incorporate parts of classical software. Our approach naturally allows for such hybrid programs as classical parts can be easily described in the host language, and the quantum parts fit right in. In Section 6.7, we show several examples that highlight the benefits of this tight integration between the two languages.

However, there is one limitation to the use of the host language in quantum circuits: the number of qubits and the qubit indices to which the gates are applied must be known at compile time. Theoretically, both these limitations could be lifted, but in order to achieve meaningful verification results, we consider them as given. Nevertheless, to accommodate a very typical pattern in quantum circuits, we support the application of gates based on loops, provided that those loops iterate over a continuous range of integers. The reason we allow for this is that it is straightforward to unroll loops with this special property, thereby eliminating them at compile time. More specifically, we permit loops of the form

```
for x := 0 to N do
     c.h(x);
od
```

$\rightarrow$

```
c.h(0);
c.h(1);
c.h(2);
```

Listing 6.1: An example for the unrolling of a loop with provided parameter $N = 2$

for x := I_1 to I_2 do S od

where x is an integer variable, and I_1, I_2 are compile-time integer constants. We assume the natural semantics of loops.

Unrolling these loops is a preprocessing step that occurs before the rest of the translation process. For instance, consider the code in Listing 6.1 as a demonstration of this approach.

6.4.2 Translation into Deterministic Programs

In the previous sections, we introduced a method for translating arbitrary quantum circuits into a classical host language. A key feature of this translation is its ability to capture all possible measurement outcomes. While this accurately reflects the behavior of quantum circuits, many desirable properties of quantum algorithms do not concern all possible measurement outcomes. Often, the focus is on the most likely result only. For instance, in Grover's algorithm, the correct result is obtained with the highest probability, but not with certainty. Consequently, it is not effective to specify that all possible outcomes are correct solutions — only that the most likely one is. Limiting the analysis to the single most likely outcome is not only practical for specification purposes but also reduces the number of traces that a verification tool must consider, thereby speeding up the verification process. Motivated by this insight, we present a modification to our translation that implements this tailored semantics.

The core idea is that the resulting *QP* program can be made deterministic under two conditions: (a) all qubits are measured at the end of the circuit (and not before) (b) there is a unique most likely measurement outcome. Under these conditions, all measurements can be conducted at the same time at the end of a circuit. The probability of each measurement outcome can be calculated, and the most likely one can be deterministically selected.

Definition 6.11 (Translation of circuits (deterministic)). *The translation σ_{det} is defined with reference to the nondeterministic version σ (Definition 6.8), with the following modifications:*

1. $\sigma_{\text{det}}(S_1;S_2) \rightsquigarrow \sigma_{\text{det}}(S_1);\sigma_{\text{det}}(S_2)$
2. $\sigma_{\text{det}}(\texttt{G}\ g\ k) \rightsquigarrow \sigma(\texttt{G}\ g\ k)$
3. $\sigma_{\text{det}}(\texttt{M 0 to N})$ *is the program*

```
p := 0.0;
for i := 0 to n do
  if (|qs_i| * |qs_i| > p) then
    p := |qs_i| * |qs_i|;
    res := i;
  fi
  qs_i := 0.0;
od
qs_res := 1.0
c_1, ..., c_n := res;
```

again N is the number of qubits and thus $n = 2^N$ is the size of the corresponding quantum state.

We abuse notation to assign the result of multiple measurements at the same time as `c_1, ..., c_n := res;`. This should be interpreted as assigning the bits of i to the measurement outcomes c_i. The for-loop could be unrolled into multiple if-statements as shown in the previous section. It can thus be regarded as syntactical sugar at this point. Using σ_{det} measurements can be made deterministic, obtaining the most likely outcome every time. Theoretically this can also be applied to states where there are multiple outcomes with the same highest probability and σ_{det} will always result in one of them. More specifically, it will always result in the one with the lowest numerical value.

σ_{det} is obviously not sound in the sense of Definition 6.10 as it only considers a single measurement outcome and discards all others. However, our intuition that the translation still maintains a different type of equivalence is correct. To see that, we first define the notion of a probabilistic `TwoState`:

Definition 6.12 (Probabilistic TwoState). *Let $p = (i, f)$ be a* `TwoState`. *Then we say $p^{\uparrow}$ is the corresponding probabilistic* `TwoState` *if $p^{\uparrow} = (i, \{(q, c, p) \in f \mid p = maxp\})$ where maxp is the maximum of all p's in f.*

Remember that f is a set of triples (q, c, p). Now the probabilistic `TwoState` is the pair where in the set of final states we consider only the most likely of those triples. Note that since several states can have the same probability, this might still be a set of states. With this notion we are able to define a relaxed definition of soundness:

Definition 6.13 (Probabilistic Soundness). *Let T_x be the set of all possible* `TwoState` *of x. Let $\zeta : QC \mapsto QP$ be a translation function from QC to QP. We call ζ probabilistically sound iff:*
$\forall c \in QC, \forall \varphi : (\exists\ p^{\uparrow} \in T_c : p^{\uparrow} \notin \varphi \implies \exists\ p'^{\uparrow} \in T_{\zeta(c)} : p'^{\uparrow} \notin \varphi')$
where φ' is the QP-property equivalent to φ according to Definition 6.9.

σ_{det} is probabilistically sound under the assumptions outlined above. The reason for this is that if there are no measurements then $p = p^{\uparrow}$ for any `TwoState`. Since the gate translation is identical to σ the soundness of σ_{det} follows from the soundness of σ. For the last measurements we established that the most likely outcome is chosen by σ_{det} thus the probabilistic soundness is maintained.

The intuition behind this notion of soundness is that it ensures the most likely final state(s) always possess the desired property. In a probabilistic system, it is, of course, impossible to determine from a single run whether the most likely outcome has been obtained. However, for a sufficient number of runs, the most likely outcome can be identified with very high probability, where the exact number of runs required depends on `maxp`. Consequently, for a system proven correct using a probabilistically sound method, it is always possible to achieve a correct result with arbitrarily high probability by executing the system a suitable number of times and selecting the most frequently observed outcome.

6.4.3 Size of the Resulting Program

We have demonstrated that it is possible to provide a semantics-preserving translation of quantum circuits into a classical host language. However, since we aim to apply formal verification tools to the resulting classical program—and because the performance of these tools typically depends on the size of the analyzed program — it is important to consider how large the program generated by our translation might become. We assess this by examining how the translation scales with respect to the two primary parameters of quantum circuits: the number g of gates and the number q of qubits. We consider the size of a program to be the number of arithmetic operations conducted within it. This may not always be a sensible metric for program size; however, since the programs we analyze consist solely of arithmetic operations and a few `if`-statements, we argue that, in our case, this serves as a valid abstraction.

Theorem 6.2 (Size of translated circuits). *Let $c \in QC$ with g the number of gates and q the number of qubits, then for the size of the program $|\sigma(c)|$ the following holds:*

$$|\sigma(c)| \leq g \cdot 2^q$$

Proof. Applying a single quantum gate corresponds to performing a matrix multiplication, where the matrix representing the gate is multiplied by the current state vector. Generally, for a square matrix of size $n \times n$ and a state vector with n elements (where $n = 2^q$), each element of the resulting vector is calculated as a dot product of a row of the matrix with the current state vector. Since the state vector has n elements, each dot product consists of n products and the calculation of the sum of those products resulting in $2n-1$ operations. As we calculate n dot products, one for each element of the vector, this amounts to $2n^2 - n$ operations for each gate. The final number of arithmetic operations for g gates of size $n \times n$ can thus be calculated as $g \cdot (2n^2 - n) = g \cdot (2^{q+1} - 2^q) = g \cdot 2^q$.

■

Consequently, the number of operations is linear in the number g of gates and quadratic in n. Given that $n = 2^q$, this implies that the number of operations grows exponentially with the number q of qubits. This is to be expected as we essentially simulate the quantum circuit which is exponential in the number of qubits and linear in the number of gates.

Similar calculations reveal that the scaling behavior for quantum measurements closely mirrors that of quantum gate operations. Specifically, the number of operations scales exponentially with the number of qubits and linearly with the number of measurements. It is important to note that these calculations pertain to the number of complex-valued operations. However, extending this analysis to real-valued operations is straightforward and results in a number of operations that is only larger by a constant factor.

Alternative Fully Expanding Translation As an alternative translation, instead of generating a program with temporary variables `qs_i` in which the intermediate result is stored, we could generate n closed expressions for each element of the final state vector. As there are no loops in the program (remember the loops allowed are unrolled), this can be achieved by iteratively replacing all occurrences of variables with the expressions used to compute their values. That, however, would result in expressions with a size that is not only exponential in q but also exponential in the number g of gates: The size of the expression used to compute the value of a single variable `qs_i` is $2n - 1$. After g iterative replacements of intermediate results (for each gate), this grows to $(2n - 1)^g$. Thus, the overall size of the program that has n such expressions is $n \cdot (2n - 1)^g < n \cdot 2n^g = 2^{qg+q+1}$. This is clearly exponential in the number of qubits q as well as the number of gates g. Again, corresponding calculations with identical results could be conducted for measurements.

Note that, even if we use the nondeterministic translation from Definition 6.8, a static analysis tool may internally generate expressions that are exponential in g and not only in q. We will examine this further in the evaluation (see Section 6.7).

6.5 Towards Real Programming Languages

To leverage existing formal analysis methods or tools, it is necessary to translate quantum circuits into a real-world programming language for which analysis tools exist. In this work, we selected Java as our target language due to the availability of well-established tools for analyzing Java programs. However, this choice is not restrictive; a translation into other languages, such as C, could be implemented with similar ease. A brief discussion of translations into alternative languages is provided in Section 6.7.5.

Before applying our translation to Java, we have to adapt our translation to account for the differences between Java and *QP*. Two primary differences involve the handling of nondeterminism and the appropriate data types for representing real and complex numbers. We discuss these issues in the following sections.

Handling Nondeterminism

Most classical programming languages are inherently designed to be deterministic, which poses a challenge when dealing with nondeterminism,

```
void f() {
    QuantumCircuit c = new QuantumCircuit(1);
    c.measure(0);
}
```

```
void f(measureParam) {
    qs_0 = 1.0;
    qs_1 = 0.0;
    measureVar1 = measureParam;
    if (measureVar1) {
        ...
    } else {
        ...
    }
}
```

```
void f() {
    qs_0 = 1.0;
    qs_1 = 0.0;
    measureVar1 = nondetFun();
    if (measureVar1) {
        ...
    } else {
        ...
    }
}
```

Listing 6.2: Translation of a measurement with two different sources of nondeterminism: (top) the original program, (middle) the translation with an additional parameter (bottom) the translation with a special nondet-function

a fundamental aspect of quantum computation. One potential solution is to use a random number generator (RNG), which, while not truly nondeterministic, can simulate nondeterminism for many practical purposes. However, there are more suitable solutions tailored to our goals. We propose two distinct approaches for realizing nondeterministic assignments in Java. Note that while the program itself remains a valid Java program and thus deterministic, the analysis of it becomes nondeterministic.

The first approach introduces an additional parameter to the method containing the translated circuit, which is used to resolve nondeterminism by deciding between possible measurement outcomes. Although this approach seemingly only shifts the problem to a different level, it enables analysis tools to treat the additional parameter as any other unknown input influencing the program's behavior. This approach also facilitates the creation of wrappers that consider every possible measurement outcome, essentially employing a brute-force strategy. Such wrappers are particularly useful for testing hybrid quantum programs. The main disadvantage of this approach is, however, that adding an additional parameter to a function changes its signature, which affects how it has to be called anywhere else. Accounting for these changes is not trivial, which renders this method disadvantageous, especially for methods that are called in a lot of different places in the program.

As an alternative, we support the use of underspecified methods as a source of nondeterminism. These methods can either be explicitly provided or implemented using special nondeterministic constructs offered by various analysis tools, enabling the direct incorporation of nondeterminism into the translated Java code. This approach has the advantage of preserving the original method signature, avoiding the need for modifications. However, depending on the specific implementation, the resulting program may exhibit behavior during execution that deviates from the original semantics. Consequently, this method is designed primarily for the use with analysis tools rather than for execution.

An example for both approaches for a minimalistic program is given in Listing 6.2. The original program consists only of the initialization of a quantum circuit with a single qubit and then the measurement of that qubit. This is obviously not a meaningful program but only serves to demonstrate the differences in the translation. The first translation introduces a new parameter to the function `f` and then assigns this parameter to the variable `measureVar1` which is in turn used to disambiguate which measurement result is considered. The second translation is nearly identical but differs in the assignment to `measureVar1`. The specifics of the function `nondetFun` are not relevant for this example. The only requirement is that for a verification tool the call to this function has to return a nondeterministically chosen value.

Both approaches have distinct advantages, making them suitable for different application scenarios. It is up to the user to decide which approach is more appropriate for a given situation.

Data Types for Representing Real and Complex Numbers

The translation to QP (Definition 6.8) is agnostic to the specific data type used to represent quantum states, as long as the chosen type supports the necessary arithmetic operations. However, the semantic equivalence between a quantum circuit and its translation (see Theorem 6.1) may not always hold depending on the data type employed. Additionally, the results of any analysis conducted on the translated program may vary based on the data type used. Two options for representing real numbers in a classical programming language are fixed-point and floating-point numbers. Floating-point numbers are natively supported in most classical programming languages, while fixed-point numbers can be implemented using integers with a fixed denominator, though this requires normalization after operations. Each of these representations has distinct advantages depending on the tools used for program analysis.

Regardless of whether floating-point or fixed-point representations are used, both introduce rounding errors. Since these rounding errors cause deviations from the semantics of real-valued calculations, they may lead to a loss of semantic equivalence between the translated program and the original quantum circuit. Consequently, any analysis conducted on the translated program may not be soundly applicable to the original circuit. However, we can demonstrate that such rounding errors do not affect the actual outcome of the program in almost all cases.

To achieve this, we consider a slightly modified version of the probabilistic measurement introduced in Section 6.4.2. Instead of verifying the desired property exclusively for the most likely measurement result, we extend the verification to include any result whose probability is sufficiently close to the most likely outcome, accounting for potential floating-point errors.

As an example, consider the most likely measurement outcome occurring with probability p, and assume that measurement probabilities may be perturbed by a maximum error of t due to floating-point inaccuracies. In the worst-case scenario: The probability of the most likely measurement outcome is overestimated by t, resulting in an actual probability of $p - t$. Simultaneously, the probabilities of all other measurement outcomes are underestimated by t. Even under these conditions, any measurement outcome with a probability less than $p - 2t$ cannot be the most likely result, as its adjusted probability would still fall short of $p - t$.

Leveraging this observation, we modify the translation of measurements to incorporate these bounds. Specifically, we redefine the measurement verification to accept all outcomes with probabilities in the range $[p - 2t, p]$. This ensures that even in the presence of floating-point rounding errors, we correctly identify all plausible candidates for the most likely outcome. This modified measurement translation is provided in Listing 6.3.

The main idea is as follows: we first compute the maximum probability for any result `maxp`. We then nondeterministically select any measurement outcome within the valid range $[0, N)$. If the probability of this outcome is above $p - t2$, we consider it a valid result; otherwise, we disregard it.

```
max_p := 0.0;
ps := [];
for i := 0 to N do
    ps[i] = |qs_i| * |qs_i|;
    if |qs_i| * |qs_i| >= max_p then
        max_p := |qs_i| * |qs_i|;
    fi
od
idx := rand(0, N);
if ps[idx] > max_p - 2*t then
    res := idx;
else
    assume(false);
fi
```

Listing 6.3: A sound measurement translation respecting float errors given quantum state `qs_i` and error threshold `t`

In this context, disregarding a result involves assuming `false`, effectively terminating further verification for that result.

This type of translation can still be implemented in purely classical programming languages using standard specification languages (for assuming false and nondeterministic choice). However, the resulting program does not retain the intended semantics when executed, making it suitable exclusively for formal analysis.

For the nondeterministic translation, we can guarantee that no outcome deemed possible is indeed impossible by applying a similar idea. Unfortunately, we cannot prove that all outcomes considered impossible are guaranteed to be impossible. This limitation arises because outcomes occurring with extraordinarily low probability could theoretically be deemed impossible. We cannot rule out this scenario, as it involves checking for equivalence of a single floating-point value, which is precisely the situation where reliability cannot be guaranteed in the presence of rounding errors.

Even in cases where a formal proof of the absence of relevant floating-point errors cannot be established in practice, we have never observed incorrect verification results for any of the considered examples (see Section 6.7). Given that our approach is tailored for small circuits (see Section 6.8) and that all values involved in the translation lie within the range of -1.0 to $+1.0$ (excluding rounding errors) — a range where floating-point numbers exhibit the highest precision — this outcome is unsurprising.

6.5.1 Translation to Java

We are now able to apply all the insights gained in the previous sections to provide a final translation from *QC* to Java for circuits with no mid circuit measurements: σ_J.

Definition 6.14 (Translation of circuits to Java)**.** *Let N be a number of qubits and a corresponding state size of $n = 2^N$, we define the syntactic translation σ_J : $QC \mapsto$* `Java`*. We consider Java-states in which the following Java variables are defined:* `qs[i]`i *($0 \leq i \leq n$) of type float that represent the quantum state. For any circuit $C = (S_0, ..., S_x) \in QC$ with no mid circuit measurements σ_J has the following form.*

1. $\sigma_J(S_1;S_2) \rightsquigarrow \sigma_J(S_1);\sigma_J(S_2)$
2. $\sigma_J($`G` $g\ k) \rightsquigarrow$

   ```
   qs = new float[]{qs[0]*u_1_1 + qs[1]*u_1_2 + ...,
                    ...,
                    qs[0]*u_n_1 + qs[1]*u_n_2 + ...};
   ```

   where the `u_`i`_`j *are the elements of the expansion matrix $E(g, k, n)$ (Definition 6.4).*
3. $\sigma_J($`M 0 to n`$) \rightsquigarrow$

   ```
   float max_p := 0.0;
   float[] ps = new float[]{0.0, 0.0, ..., 0.0};
   for(int i = 0; ++i; i < n) {
      ps[i] = |qs[i]| * |qs[i]|;
      if (|qs[i]| * |qs[i]| > max_p) then
         max_p = |qs[i]| * |qs[i]|;
      }
   }
   int res = nondet(0, N);
   if(ps[res] > max_p - 2*t) {
      qs = new float[]{0.0, ..., 0.0};
      qs[res] = 1.0;
   else{
      //@ assume(false);
   }
   ```

There are a few details we would like to comment on. First, we are using floats as if they were able to represent complex values. This is obviously not the case, but extending this translation to accommodate for complex values is straightforward and we avoid it here for reasons of readability. Second, we use some syntactic sugar like `|qs[i]|` for the absolute value that is not natively supported in Java, however again replacing this with the suitable pure Java code is trivial. Third, we use a `nondet` function in the code. This can be of any type discussed in Section 6.5. Forth, the result of all measurements is stored in an integer variable `res` rather than a sequence of Bools. Again this could be easily transformed to Boolean variables if necessary.

As we use floating-point variables in this translation, we lose soundness in the sense as defined earlier (both probabilistic and plain soundness). However, as argued in Section 6.5, the way we translated measurements ensures that soundness with respect to measurement outcomes is maintained. Specifically, the translation σ_J is *probabilistically sound* for any property φ that does not reference the quantum state or the explicit probabilities of the circuit. To give a formal definition of this type of soundness we first define observably equivalent states:

Definition 6.15 (Observably equivalent)**.** *Let $s_1 = (q, c, p)$ be a QC-state and $s_2 = (qs, cs, ps)$ be a* `Java`*-state. We call s_1 and s_2 observably equivalent iff $c \cong cs$ according to Definition 6.9. We write $s_1 \cong_o s_2$ for observably equivalent states.*

This gives rise to the following definition of soundness:

Definition 6.16 (Probabilistic Observable Soundness)**.** *Let T_x be the set of all possible* `TwoState` *of x. Let $\zeta : QC \mapsto$* `Java` *be a translation function from QC to* `Java`*. We call ζ probabilistically observably sound iff:*
$\forall c \in QC, \forall \varphi : (\exists\ p^{\uparrow} \in T_c : p^{\uparrow} \notin \varphi \implies \exists\ p'^{\uparrow} \in T_{\zeta(c)} : p'^{\uparrow} \notin \varphi')$ *and not* $p^{\uparrow} \cong_o p'^{\uparrow}$
where φ' is the `Java`*-property equivalent to φ according to Definition 6.9.*

This definition permits deviations in the probability of each state and the quantum state but requires that any bug related to the outcome of measurements must also be present in the translated version of the circuit.

While this may initially seem like a significant restriction, it is important to note that the quantum state and the probabilities of each state are not directly observable on real quantum devices. Intuitively, our definition captures the fact that any bug observable during the execution of the circuit on a real device with high probability will also be detectable with a method that is probabilistically observably sound.

We argue that this constitutes a reasonable and practical definition of soundness in the context of quantum computing. However, there are limitations to such a definition. Notably, it only applies to quantum software that includes measurements, and thus excludes quantum subroutines. As a result, a probabilistically observable sound method is unsuitable for verifying standalone quantum subroutines, such as the Quantum Fourier Transform (QFT). Instead, it is applicable exclusively to complete quantum algorithms that yield some classical output.

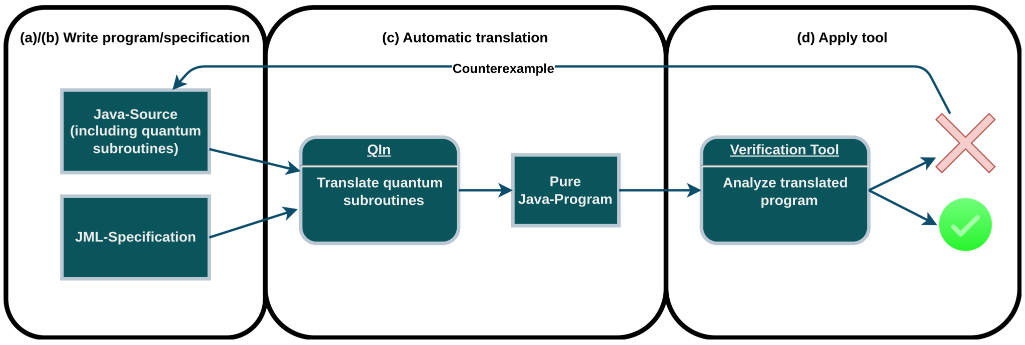

(a)/(b) Write program/specification
(c) Automatic translation
(d) Apply tool
Counterexample
Java-Source (including quantum subroutines)
JML-Specification
QIn
Translate quantum subroutines
Pure Java-Program
Verification Tool
Analyze translated program

Figure 6.2: A schematic representation of the envisioned workflow with the tool QIn

6.6 QIn: Implementation and Tool Chain

To evaluate our approach in practice, we implemented the translation presented in the previous sections in a command-line tool called QIn. Currently, QIn supports two types of translations for measurements: a translation that considers all possible measurements (i.e., measurements with a non-zero probability), and the translation for most likely results under floating-point errors shown in Section 6.5.1. We call these translations the possibilistic and the probabilistic translation respectively. The tool is openly available online[4].

[4]`https://github.com/JonasKlamroth/QIn/`

```
qr = QuantumRegister(2)
cr = ClassicalRegister(1)
djCircuit = QuantumCircuit(qr, cr)
djCircuit.h(qr[0])
djCircuit.x(qr[1])
djCircuit.h(qr[1])
# apply oracle
djCircuit.h(qr[0])
djCircuit.measure(qr[0], cr[0])
```

```
float[][] oracle = ...;      // define oracle
Circuit c = new Circuit(2); // create circuit object
c.h(0);                      // apply quantum gates
c.x(1);
c.h(1);
c.g(oracle, 0, 1);
c.h(0);
return c.measure(0);         // measurement
```

```
// setup of initial state
float[] oracle = ...;
float[] q_state_0 = new float[]{1.0F, 0.0F, ...};
float[] q_state_1 = new float[]{0.0F, 0.0F, ...};
// translation of c.h(0)
q_state_0 = new float[]{ ... };
q_state_1 = new float[]{ ... };
// translation of further gates
...
// measurement
boolean $$_tmp_measureVar1;
if ((q_state_0[2][0] * q_state_0[2][0] + ...) >
    (q_state_0[0][0] * q_state_0[0][0] + ...)) {
    q_state_0 = new float[]{0.0F, 0.0F, ...};
    q_state_1 = new float[]{0.0F, 0.0F, ...};
    $$_tmp_measureVar1 = true; //store result
} else {
    ...
    $$_tmp_measureVar1 = false;
}
return $$_tmp_measureVar1;
```

Listing 6.4: An example for a hybrid program written using Qiskit (top); our Java interface (middle); and its translation as generated with QIn (bottom).

QIn is implemented in Java and leverages OpenJML as a back-end for manipulating Java code. The tool takes as input a file containing Java code with quantum subroutines (as discussed in the following paragraph) and outputs a translated version in which the quantum circuits are converted into pure Java, based on the translation methodology presented. The resulting Java file can then be verified, tested, or validated using any tool capable of handling Java code.

The overall envisioned workflow consists of four steps: (a) write a Java program that includes quantum subroutines, (b) specify the code using a specification language of choice (e.g., JML), (c) apply QIn to translate the quantum subroutines into equivalent Java code, and (d) use any tool or technique to analyze the resulting Java code against its specification. This workflow is illustrated in Figure 6.2.

If the tool identifies no bug, the soundness of our translation guarantees that this result reflects accurately on the original program (in the sense of soundness we discussed). Consequently, the program in its current state adheres to its specification. If a bug is identified, this information can be used to debug the original quantum circuit. Depending on the verification tool used, counterexamples might be provided, which can assist in locating the precise source of the bug. Once the bug is corrected or the specification adjusted, the toolchain can be rerun to verify whether the revised version complies with the specification.

The currently prevalent way of programming quantum circuits is defined by established Python-based quantum programming frameworks like Qiskit [2] or Cirq [36]: A quantum circuit object is created, initialized with the necessary number of qubits, to which gate operations and measurements are applied by calling functions/methods. Each of those function calls takes as inputs the qubits the specific gate or measurement is applied to (see Section 2.1.8). We provide a Java facade which mimics this type of programming quantum circuits. While currently being only a facade, implementing a simulator or even an interface for real quantum devices would merely be an implementation exercise.

Let's consider a typical example and its translation: The Deutsch-Jozsa algorithm is a famous example of a quantum algorithm (see Section 6.7). The circuit for this algorithm for 1 qubit in *QC* consists of the following statements:

$$\texttt{G}\ H\ \texttt{0};\texttt{G}\ X\ \texttt{1};\texttt{G}\ H\ \texttt{1};\texttt{G}\ O\ \texttt{0};\texttt{M}\ \texttt{0}$$

We will discuss the concrete mechanics of this algorithm in Section 6.7. For now, it is sufficient to understand that the gate *O* serves as an oracle specific to this algorithm and can be considered a non-standard gate that must be explicitly provided. The gates X and H are the standard matrices introduced earlier and are applied to the first and second qubits, respectively. Listing 6.4 illustrates the Deutsch algorithm in three different implementations. The first is the Qiskit version, as presented in their community examples[5]. The only modifications made involved manually unrolling the loops and setting the circuit size to 1. The second version showcases our implementation using the Java interface, while the third is an excerpt from the generated translation (the full translation is too large to include here).

Note that in the Java implementation, we utilize syntactic sugar for known quantum gates, for instance writing `c.h(1)` instead of `c.g(H, 1)`. This notation will be adopted throughout the remainder of this chapter whenever appropriate. Apart from this minor difference, the *QC* and Qiskit implementations closely resemble the Java version. This underscores the fact that the language used to construct the quantum circuit is theoretically inconsequential; hence, choosing Java for this implementation does not preclude an equivalent implementation in other languages.

6.7 Evaluation

To evaluate our approach and demonstrate its feasibility, we first present two experiments that examine runtime performance based on the number of qubits and gates. Following this, we demonstrate the applicability of our approach to real-world quantum programs. For this second part, we study two different types of programs: first, we apply our tool to a set of case studies representing well-known quantum algorithms, and second, we use it to analyze a collection of documented bugs in quantum circuits. All translations used in this section were generated fully automatically using QIn.

For analysis and verification, we employed JJBMC [18], which was developed (with JBMC [44] as its backend) for verifying Java code specifications written in JML. JBMC in turn is a bounded model checker for Java bytecode that uses SAT solvers as backends. We selected JJBMC as the verification tool due to its fully automated nature and its support for verifying programs that include floating-point arithmetic, a feature not typically supported by verification tools. It is important to note, however, that the choice of JJBMC was made solely for the purpose of this evaluation, and it is not the only tool compatible with the Java code generated by QIn. In the

[5]`https://github.com/qiskit-community/qiskit-community-tutorials/blob/master/algorithms/deutsch_jozsa.ipynb`, last visited 2024/12/01

Case Study	**#gates**	**#qubits**	**Verification Time (s)**
BB84*	3	1	1.6
DeutschJozsaQiskit*	12	4	2.4
DeutschJozsa*	9	4	17.0
BernsteinVaziraniQiskit*	9	3	1.8
BernsteinVazirani*	9	4	10.6
Grover*	9	2	5.0
GroverQSharpTut*	19	3	3.1
Shor*	18	7	(quantum) 6.0
			(classical) 10.7
ShorQiskit	41	5	(quantum) 445.6
			(classical) 12.2
SuperdenseCoding*	6	2	1.5
GHZ*	4	3	1.5

Table 6.1: Overview of all case studies with the number of gates and qubits as well as average time needed for verification with JJBMC in seconds, the standard deviation for all case studies is below 3% of the verification time, for Shor we provide verification for the quantum (quantum) and the classical subroutines (classical) separately, case studies marked with a * include the prove of absence of floating point errors

experiments that follow, the reported runtimes reflect the performance of JJBMC, as the translation time itself was negligible for all examples. The experiments were performed on a desktop PC with an AMD Ryzen 5 3600 processor and 16GB of RAM. The JBMC version used was 6.3.1 and was run with the built-in SAT-solver. All sources necessary to reproduce the results are provided at [89].

6.7.1 Differences between Theory and Implementation

In some minor points, the implementation deviates slightly from the presented theoretical approach. This is mainly for practical reasons, either to ease implementation or to improve performance. None of these differences affect the claims made for the theoretical translations unless explicitly stated otherwise.

The main differences between the theory and the practical implementation arise from the use of real-world programming languages and have already been discussed in Section 6.5. We do not elaborate further on them here. In addition to those modifications, there is one adaptation worth mentioning: in this implementation, we introduced an optimization regarding the complexity of the translation. Specifically, as the actual probability of a result state is not needed in our approach (remember we are aiming for observable soundness), there is no requirement to normalize a state after a measurement. This is a significant simplification because it avoids multiple computationally expensive operations on floating-point numbers, including the computation of square roots. For most verification tools, bypassing these operations leads to a substantial simplification, which translates to reduced verification time.

Despite the loss of information about the exact probability of the resulting bit-string, we argue that this trade-off is justified by the benefit of a simpler program, which is easier to analyze. In particular, when updating the state after a measurement, we only set certain values to zero and leave the remaining part of the state unchanged. It is crucial to note that this optimization does not affect the probabilities of measurement outcomes with zero probability, as normalization has no impact on these probabilities. Therefore, the possibilistic translation remains unaffected by this optimization, even though it involves comparisons with a concrete probability value.

6.7.2 Scaling Experiments

To examine the performance of our approach, we conducted two experiments analyzing the scaling with the number of gates and qubits respectively. In both cases, we run JJBMC 10 times on each of the circuits and plot the average runtime. We also plot standard deviation as well as minimum and maximum values, but due to very similar runtimes these are not visible in the plots. A script to reproduce all results is included in the available online sources [89].

Scaling with the Number of Gates

To evaluate the scalability of our approach with respect to the number of gates in a circuit, we conducted the following experiment: We created circuits with 2 or 3 qubits, initializing them in a nondeterministically chosen basis state. Our goal was to verify that after applying an even number of `X` or `H` gates to the first qubit, the initial state is restored. The runtimes for JJBMC are shown in Figure 6.3 for circuits with $N = 2$ and $N = 3$ qubits, and three different gate configurations. The gates used were `X`, `H`, and `HH`/`HHH`, where `HH`/`HHH` apply Hadamard gates simultaneously to the first 2 or 3 qubits, respectively.

All of these gates are involutory, meaning that applying an even number of them restores the initial state. Notably, whether these operations are entangling or not is irrelevant for this experiment, as each gate is implemented via matrix multiplication, and we always consider the full system state. The density of the matrices representing these gates, however, is the primary factor influencing runtime performance. As shown in the figure, runtime varies significantly depending on the gate applied. This is expected, as the complexity of the expressions generated during verification correlates with the sparsity of the matrix representing the gate. For very sparse matrices, such as the `X` gate, runtime increases nearly linearly with the number of gates. In contrast, denser matrices, such as those representing `H` or `HH`/`HHH`, result in higher runtimes and poorer scaling with the number of gates. However, at least for the number of gates considered here, we did not observe the worst-case exponential blowup considered in Section 6.4. As expected, JJBMC reuses intermediate results, thus avoiding this potential issue.

The apparent decrease in runtime growth between successive data points is likely an artifact of the underlying analysis tool. This phenomenon is common when evaluating scalability with respect to specific parameters, as the runtime of tools like JJBMC (or any SAT- or SMT-based solver) is highly dependent on the specific characteristics of each problem. We additionally would like to point out that not every circuit with the same number of qubits or number of gates behaves alike in terms of runtimes.

Scaling with the Number of Qubits

We also analyze the impact of increasing the number of qubits in a circuit while fixing the number of gates. In this experiment, we constructed a circuit with a single `X` gate applied to the first qubit while varying the total number of qubits. We proved that the `X` gate consistently flipped the first qubit. The resulting runtimes are shown in Figure 6.4. The runtimes exhibit exponential growth with the number of qubits, which is expected due to the exponential increase in the state space. Notably, the most significant scaling challenges arise from the fact that JJBMC must analyze exponentially more cases with each additional qubit. While the findings are promising, they also highlight limitations. For circuits with fixed initial states—common in many simple quantum algorithms—we can analyze a non-trivial number of qubits (as demonstrated on several examples in the upcoming section). However, the exponential scaling observed with

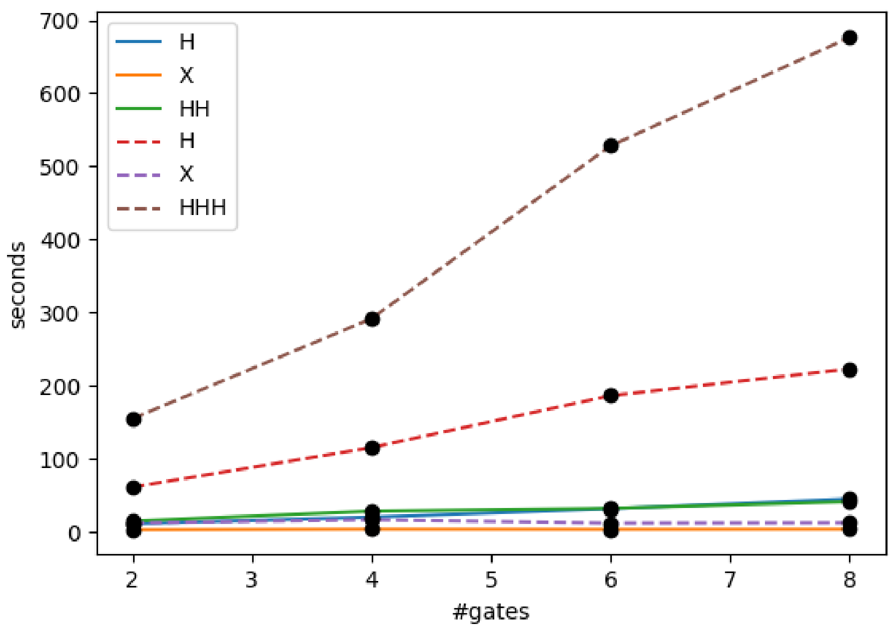


Figure 6.3: Performance benchmark for different number of gates and different gates, dashed lines indicate experiment with 3 qubits, solid lines with 2 qubits

arbitrary initial states indicates that this method, without significant improvements, is unsuitable for larger circuits. Although this analysis is based on a single tool, and the translation is not highly optimized, overcoming the fundamental exponential scaling remains a challenge.

6.7.3 Applying QIn to Well-Known Quantum Algorithms

We present several case studies to demonstrate the applicability of our approach to a range of well-known quantum algorithms. Table 6.1 provides an overview of these case studies, highlighting key characteristics and the verification times required by JJBMC (averaged over 10 runs). The translation process was completed in negligible time (< 1 second) for all examples.

As shown in the table, the verification time for most case studies is only a few seconds, with the notable exception of Shor's algorithm. Some algorithms are examined in multiple versions, and we observe significant variations in verification times for different implementations of the same algorithm. This variation highlights not only differences in implementation details but also the broad range of complexities that can arise even within

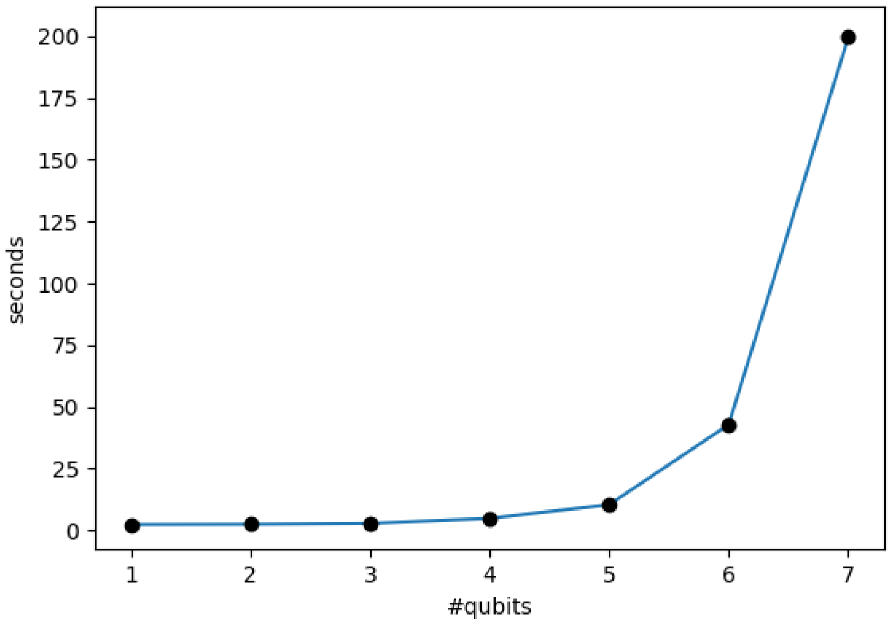


Figure 6.4: Performance benchmark for different number of qubits

Tool	**LoC**	**LoS**	**Time**	**Manual**
QBricks	11	46	79.0s	39
SQIR	10	39	-	222
Silver	9	10	7.9s	-
QIn + JJBMC	18	3	2.4s	-

Table 6.2: Comparison of statistics for verifications of the Deutsch-Jozsa algorithm with different tools. For each tool we list Lines of Code (LoC), Lines of Specification (LoS), Runtime of verification tool (Time), Manual interactions necessary (Manual)

the verification of a single algorithm. In the following sections, we discuss three of these case studies in greater detail.

Deutsch-Jozsa

The first case study focuses on the Deutsch-Jozsa algorithm [53], a foundational quantum algorithm that demonstrated quantum speedup for the first time. Given a function $f : \{0,1\}^n \mapsto \{0,1\}$ that is either constant or balanced (i.e., the number of inputs for which f returns 0 and 1 are equal), the algorithm determines whether f is balanced. We analyzed two implementations of this algorithm. The first implementation, sourced from

```java
if (oracleType == 0) {       // If oracleType is "0",
    if (oracleValue == 1) { //return oracleValue
        djCircuit.x(n);
    }
} else {  // Otherwise, return the inner product
    for (int i = 0; i < n; i++) {
        if ((a & (1 << i)) != 0) {
            djCircuit.cx(i, n);  // CNOT operation
        }
    }
}
```

```java
public float[][] getOracle(int N, boolean[] f) {
    int size = 1 << N + 1;
    float oracle[][] = new float[size][size];
    for(int i = 0; i < size; ++i) {
        for(int j = 0; j < size; ++j) {
            oracle[i][j] = 0.0f;
            float val = f[i / 2] ? 1.0f : 0.0f;
            if(i == j) {
                oracle[i][j] = 1.0f - val;
            }
            int even = (i % 2) * 2 - 1;
            if (i == j +even) {
                oracle[i][j] = val;
            }
        }
    }
    return oracle;
}
```

Listing 6.5: The implementation of the oracle for the Deutsch-Jozsa algorithm as realized in Qiskit (top) and our own implementation (bot)

the qiskit-community tutorials, was syntactically adapted to fit our Java interface, but no changes were made to the gates or the implementation itself. This version accepts three parameters: (1) `oracleType`, a Boolean-like value that indicates whether the oracle implements a balanced function (`oracleType == 1`) or a constant function (`oracleType == 0`); (2) `oracleValue`, which specifies the output for all inputs if the oracle is constant; and (3) an integer between 1 and 2^n, representing the nature of the balanced function (more specifically, the inner product of the input with this integer, interpreted as a bitstring, is returned). The oracle is constructed based on these parameters, as shown in the top half of Listing 6.5.

The second implementation constructs a single gate that realizes the oracle, using a Boolean array that represents the function f as a truth table. Based on this truth table, a function generates a gate acting on all qubits to implement the Deutsch-Jozsa oracle (see Listing 6.5 bottom half). Although this algorithm is typically viewed as purely quantum, both versions incorporate classical programming elements, making it at least partially hybrid in nature.

We evaluate the algorithm for $n \leq 5$. The resulting circuit for an n-bit function uses $n + 1$ qubits, $2n + 3$ gates (depending on the implementation), and n measurements. The algorithm's specification states that the measurement result is 0 if and only if the function is balanced, which can be determined by the parameters in both versions. In the qiskit version, oracleType directly indicates whether the function is balanced. In our implementation, a helper function counts the number of `true` values in the truth table to determine balance.

The generated program is verifiable with JJBMC in under a second for $n = 1$, but timed out when $n > 6$, with runtime exceeding six hours. This is likely due to the exponentially increasing number of possible input functions and the growing quantum state size. We encapsulate the algorithm in a method, allowing it to be used in the Java program like any other Java method. Using this approach, we were able to verify small functions, such as one that counts the number of zeros in the truth table, thus verifying a truly hybrid program. Another case study, the Bernstein-Vazirani algorithm, is closely related to the Deutsch-Jozsa algorithm and yielded similar runtime results, so we do not discuss it in detail here.

As Deutsch-Jozsa is a comparatively easy but very well-known algorithm, it has been tackled by various verification tools for quantum software. In Table 6.2 we compare several statistics between three of these approaches with our own approach. Note that this comparison is not entirely fair as it compares approaches with very different underlying concepts. The most significant of those differences is that the first two tools, QBricks and SQIR, show correctness for an arbitrary number of qubits while Silver and our approach verify only for a given number n of qubits. In contrast to Silver we have a generic description of the algorithm, but run the verification only for a certain n while Silver uses different implementations for each number of qubits. Thus, the lines of code (LoC) given for silver is for the specific case of 2 qubits (which probably means that this number would increase for a higher number of qubits). Information for Silver is taken from [97] and for the other tools from [32].

We highlight several noteworthy observations from this comparison. Firstly, the comparatively high number of LoC in our approach is evident. This is attributable to the fact that the first two tools employ a highly compact representation of the abstract algorithm, rather than a full implementation in a real-world programming language. (For Silver, this is specific to the 2-qubit case, as previously mentioned.) Secondly, our approach stands out by requiring the least amount of specification as measured by lines of specification (LoS). Given that specifying properties for verification is often a burden for developers, this represents a significant advantage of our method. Furthermore, our approach achieves the fastest verification times among the compared tools. Lastly, it is notable that both Silver and our approach operate without the need for manual interaction, in contrast to the other two tools. Taken together, these points indicate that our tool performs favorably concerning the objectives outlined in Section 6.2.

This example demonstrates that our approach can handle real-world implementations of well-known quantum algorithms. Importantly, we were able to prove the correctness of the Qiskit implementation with minimal specification and without significant changes. Unlike other approaches that focus on abstract algorithms, we proved the correctness of the actual implementation, which we consider a major strength of our approach. Also, we were able to compare our tool favorably to three established tools in the field regarding the time for verification, the amount of specification needed, and the amount of necessary manual interaction.

Grover

Our second case study focuses on a simplified version of the widely-used Grover's algorithm, which is designed to solve unstructured search problems and can also serve as a subroutine or enhance the performance of other algorithms. Grover's algorithm, given an oracle capable of distinguishing between desired and undesired elements in a database, enables the efficient identification of target elements with a quadratic speedup compared to classical approaches. Again we considered two examples: One we implemented ourselves and another one taken from a Q# tutorial. For our implementation, we demonstrate the functionality of Grover's algorithm in a minimal setting using two qubits and a circuit consisting of 10 gates. The Q#-version is specifically implementing an alternating-bit-oracle meaning that the desired states that the algorithm should identify are the ones which have an alternating bit sequence.

Our implementation expects two parameters: (1) an array of integers representing the database, which contains a permutation of the numbers 0 to n, where n is the size of the database, and (2) an integer specifying the target element to search for within the database. Given that our setup uses two qubits, n can be at most 4. To ensure the algorithm works as intended in this specific case, the database must be a permutation of distinct elements, allowing for the identification of a single target element. In cases where the element is not present, the algorithm returns -1.

We verified that the database contains the desired element at the index returned by our implementation. These functional requirements can be expressed using just five lines of JML, as shown in the following listing:

```
/*@
  @ requires database != null && database.length == n;
  @ requires (\forall int i; 0 <= i < n;
  @     database[i] >= 0 && database[i] < n);
  @ requires (\forall int i; 0 <= i < n;
  @     (\forall int j; 0 <= j < n;
  @         i == j || database[i] != database[j]));
  @ ensures ((val >= 0 && val < n) ==>
  @             val == database[\result]);
  @ ensures ((val < 0 || val >= n) ==> \result == -1);
  @*/
```

JJBMC is able to verify this specification in 5.0 seconds (average runtime over 10 runs). Notably, this simplified version of Grover's algorithm requires only a single iteration, unlike the general case where multiple iterations may be necessary to amplify the probability of finding the correct element. This simplification is one reason we were able to successfully verify this implementation. In contrast, the Qiskit version of Grover's algorithm, which involves multiple iterations and nested loops, presents significant challenges for verification. When these loops are unrolled in our approach, the resulting circuit becomes excessively large, surpassing the current capabilities of JJBMC for analysis.

The Q#-version on the other hand was even faster to verify with the alternating-bit property as the specification. The verification time for this version was 3.1 seconds even with significantly more gates and one more qubit. Which shows that the mere number of qubits or gates is not sufficient to predict verification time.

Shor

Third, we considered Shor's Algorithm, arguably the most well-known quantum algorithm, which factorizes a number n into two factors [131]. The algorithm comprises several steps, though only one — period finding — is executed on a quantum computer. We explored two different implementations: the first is the Qiskit community version, while the second replicates the approach used in a paper that demonstrated Shor's algorithm on physical hardware for the first time [146]. Both implementations handle only the minimal case where $n = 15$, but we argue that even this restricted case showcases the strength of our approach. To the best of our knowledge, this is the first time an implementation of Shor's Algorithm — including both the quantum and classical components— has been verified fully automatically.

Our implementation consists of three core components: (a) a classical routine that checks for trivial factors and invokes the order-finding subroutine, (b) the order-finding subroutine, which verifies whether the quantum routine has produced a useful period and, if so, computes a non-trivial factor from it, and (c) the quantum subroutine itself. These components are encapsulated in a while-loop that continues until a valid factor is found. Additionally, several non-trivial helper functions are required, such as those for computing the greatest common divisor (gcd), integer exponentiation, and the continued fractions algorithm. We emphasize that, while these helper functions are often treated as afterthoughts in discussions of Shor's algorithm, they are both non-trivial and critical to the algorithm's success. The ability of our approach to integrate these classical components seamlessly with the verification of the quantum circuit is essential for the verification of hybrid algorithms like Shor's.

In our case study, we also evaluated two verification strategies: modular verification, where each of the three main components is verified separately, and holistic verification, where all code is inlined (except for the main loop). Note that helper methods were always inlined and not separately specified and verified. The entire case study, using the Qiskit version, consists of 184 LoC, 79 of which pertain to the quantum circuit. With the holistic verification approach, a minimum of 8 LoS were required, with an additional 13 LoS necessary for the modular verification. The primary property we aim to demonstrate is that the algorithm returns a non-trivial divisor of the input n. In JML, this property can be expressed as follows:

```
@ ensures n % \result == 0 && \result != 1 && \result != n;
```

This ensures that the result is a valid divisor and eliminates the two trivial cases. For the holistic approach, this single line is almost sufficient to complete the entire proof. The only additional lines required address the precondition that n cannot exceed 15 (since we are considering only the minimal case) and the need to restate the specification as a loop invariant for the main loop, as well as for the method that encapsulates the three core components. Essentially, beyond specifying the key property to be proven, no further detailed specification is required. This is a stark contrast to other methods that rely heavily on detailed specifications to carry out their proofs. As an example, consider QBricks which needs a combined 1163 lines of code and specification just for the order finding subroutine of Shor's algorithm [32].

Another key point is that, although the verification is performed using a bounded model-checker, and the proof involves a potentially infinitely running loop, we avoid unrolling the loop by leveraging the loop invariant. We thus prove partial correctness of the algorithm. Note that we do not reason about how likely a successful run of the loop is but only show that if it terminates the output has the desired property. While this is a weaker guarantee than what has been proven by other tools, it nevertheless demonstrates what we consider the main property of Shor's algorithm.

6.7.4 Application of QIn to Known Errors in Quantum Circuits

We demonstrated that our approach can successfully verify the correctness of well-known quantum algorithms. However, it is arguably even more effective at identifying bugs in small quantum programs, particularly those written by developers who are new to quantum computing. In [154], Zhao et al. present a collection of bugs found in quantum programs across common platforms such as GitHub and Stack Overflow. Unfortunately, not all of the bugs in this benchmark are errors in the quantum circuits themselves; some are due to misconfigurations in Qiskit or misunderstandings of its API. Of the 52 collected bugs, 10 were directly related to errors in the quantum circuit. For 9 of these 10 circuits, the required property was either of the form "for all inputs and all possible resulting measurements, some property holds" or "for the most likely measurement, some property holds." These two types of properties align precisely with the capabilities of our approach.

We re-implemented each of these buggy programs in our Java circuit language and wrote corresponding specifications. Since no formal specifications were provided in the benchmark, we inferred the intended behavior based on the questions or bug reports. Additionally, we created a fixed version of each buggy program, as described in the original posts. We were able to verify that the fixed versions adhered to the inferred specifications. More importantly, we successfully detected all bugs in the buggy versions of the programs. This provides strong evidence that our approach is well-suited for identifying bugs in real-world quantum programs as they are currently being written by the community.

Notably, issues related to scaling and the use of floating-point numbers rather than real numbers did not pose any challenges for the programs we examined. All of the considered programs involved at most 5 qubits, and our approach was able to verify or refute the desired properties in less than 2 seconds for 8 of the 9 programs, with the remaining one taking approximately 9 seconds. These results further support the "small scope hypothesis" for quantum programs, at least for those currently being developed (see Section 6.8 for further discussion).

6.7.5 Considering Other Host Languages and Other Verification Tools

We conducted the evaluation of our approach with Java as the classical host language and a bounded model-checker as the underlying verification tool. However, as we have hinted at several times before, one of the advantages of our approach is that these choices are not forced and any other host language as well as tool for the verification is also valid. In this section, we would like to discuss why we think that the choices we made were suitable for a first evaluation and what other configurations might be interesting to examine as future work.

Other host languages We argue that different aspects should influence the choice of host language. In particular, we see the following to be the most relevant: 1) analyzability 2) popularity 3) additional features. We briefly comment on all of these aspects.

Analyzability: The goal of our approach is to be able to analyze quantum circuits with existing tools. Thus, the choice of the host language should depend on the availability of tools for that language. Due to their nature, some languages are fundamentally easier to analyze than others. This is obviously also dependent on what type of analysis one wants to conduct. While unit tests are easily implemented in nearly every language, tools for rigorous formal verification are not that readily available for all languages. As our focus was on formal verification, we considered languages for which multiple formal verification tools were available. The two most analyzed languages in that regard are arguably C and Java. Thus we chose one of them. We conducted preliminary experiments in both languages and found that small examples are verifiable in both languages. As such, we consider both suitable for our proposed approach and the choice remains one of personal preference.

Popularity: To provide an approach that can be easily used by most developers, it is crucial to base it on a language that is well established. For programming quantum circuits, the obvious choice here would be Python as nearly all quantum programming approaches are either Python libraries or special purpose languages. The latter do not fulfill the property of being known by a wide range of developers. However, since we consider hybrid programs, we also have to consider the classical part of the programs which could be written in any other language like C/C++, Java, JavaScript, C#, and many more. We consider choosing any of these a valid choice regarding the popularity aspect.

Additional features: Our approach, while being in general applicable to any language, may profit from some special feature of a programming language. The most relevant in our opinion is the native support for a data type for reals. If present, this feature allows the analysis without caring for rounding errors due to the nature of floating-points. Thus any language with reals as a native data type is worth considering. As an example, we examined our approach for Dafny [96]. Dafny offers the built-in data type "real". We were able to show minimal case studies like the BB84 protocol using Dafny. However, for all examples examined, the verification time for Dafny was slower than the one compared to JJBMC.

Other verification tools Once settled on Java as the host language, there are still multiple possible tools for the formal verification. As we used JJBMC, which shows the correctness regarding a given JML specification, we considered other tools with the same input format. The most prominent tools in our opinion in that regard are KeY [4] and OpenJML [41]. Both tools support the functional verification of Java code specified with JML. The main advantage of using such similar tools is that in theory, we can reuse the generated translated programs as we used them for JJBMC. For OpenJML this was working as expected. We were able to show several of the smaller case studies with OpenJML without any modifications to the code or the specification. OpenJML was however slower compared to JJBMC in all considered examples. Additionally, OpenJML lacks some features that come in handy for the verification of such small programs. Most notably, OpenJML and KeY both expect fully specified code including function contracts for all subroutines and loop-invariants/variants for all loops as their ability to unroll loops and inline methods is limited. These features, however, proved very useful in the verification, especially for functions implementing oracles.

For KeY we tried to apply the automatic SMT-strategy to a simple case study. Unfortunately, this did not work out of the box as KeY seems to have issues with the array assignments which we used for every state transition. However, with minor adaptations to the generated program KeY was able to verify the given program only relying on its own SMT-translation. Note, that even though the SMT-strategy itself is fully automatic in order to successfully apply it some manual steps are necessary and even after that JJBMC is faster with the verification. This manifested our intuition that while KeY is theoretically able to be applied to the programs generated by our approach (with minor adjustments), other tools are more suitable.

Overall we were still able to show what we proclaimed earlier: The verification tool used is not predetermined, and different tools may be used for the verification of the same translated circuit. It is also evident that not all tools perform equally well on all considered examples, thus it is worth experimenting and finding the most suitable tool for each language/example in order to achieve optimal performance. In our opinion, after considering all options, JJBMC is the best fit for our needs as it enables us to conduct fully automatic verification even with only partial specifications.

6.8 On the Small Scope Hypothesis

We previously argued that although our approach is primarily suited for small circuits, it remains highly valuable. This is largely due to the small-scope hypothesis, which suggests that the majority of bugs in a program can be found within a relatively small scope, often defined by the input size. The intuition behind this hypothesis is that since programs are generally parameterized by input size, any flaw in the algorithm or implementation is likely to manifest with small inputs. This hypothesis has been tested in multiple studies (e.g., [7]) and is a foundational principle behind techniques such as bounded model checking. These methods rely on the assumption that analyzing programs with small input sizes and a limited number of loop iterations can provide sufficient confidence in their correctness. Remember that, despite only analyzing circuits for a fixed number of qubits, our language and entire tool chain can handle generic descriptions of quantum algorithms (parameterized in the number of qubits). Thus we argue that the small scope hypothesis applies to our approach.

We believe this hypothesis is especially pertinent to current quantum algorithms for several reasons:

1. Limitations of current quantum hardware
2. Absence of common counterexamples to the hypothesis
3. The typical scale of present-day quantum algorithms
4. The mathematical nature of problems addressed by quantum computing

We examine each of these points in detail. Notably, the underlying structure of quantum computing problems and their current applications sidestep many of the weaknesses typically associated with the small-scope hypothesis.

Capabilities of Current Hardware Our approach assumes that the program is executed on logical qubits. However, real quantum computers currently face numerous challenges, such as decoherence, readout errors, and gate errors. Virtually every operation, whether applying gates, conducting measurements, or even remaining idle, carries a risk of introducing computational errors. We discussed these errors in detail in Section 2.1.9.

It has been demonstrated that qubits with sufficiently low error rates can be used to construct logical qubits. However, depending on the error rate of the physical qubits and the error correction scheme, creating a single logical qubit requires between 10^3 and 10^4 physical qubits [64]. Given that the largest quantum devices today offer around 100 physical qubits — with error rates well above the error threshold for reliable error correction — it is currently impossible to create a single logical qubit. Therefore, although our approach targets small quantum circuits, it remains capable of verifying any quantum program that can be executed on present-day hardware, assuming the algorithm relies on logical qubits. This situation is expected to persist for several years, given the current pace of quantum hardware development.

Absence of Typical Counterexamples to the Small-Scope Hypothesis In classical computing, there are well-known counterexamples where the small-scope hypothesis fails. These often involve edge cases, arbitrarily set bounds, or configuration-specific issues. However, such scenarios are less prevalent in the domain of quantum computing as these examples stand in contrast to the mathematical nature of quantum algorithms (see next paragraph).

Mathematical Nature of Quantum Computing Problems Quantum computing is typically applied to mathematical problems, such as optimization, cryptography, and linear algebra. It is unlikely that quantum computers will be used for tasks like building user interfaces or simple database applications, which are efficiently handled by classical systems. Instead, quantum computers will likely remain focused on highly specialized problems. These problems, being inherently mathematical, are well-suited to the small-scope hypothesis, as they often apply uniformly to a range of problem sizes and avoid special cases.

Typical Size of Current Quantum Algorithms We have already noted the constraints on the number of qubits available in contemporary quantum devices, limiting the size of quantum algorithms that can be executed. However, even beyond hardware constraints, quantum algorithms themselves are typically concise. For example, well-known algorithms such as Shor's algorithm [131] and the HHL algorithm [77] can be expressed in just a few lines of pseudocode. This conciseness seems characteristic of quantum algorithms, whose complexity lies in their underlying mathematical theory rather than in the code itself. This contrasts with many classical algorithms, which can be both large and intricate. While future quantum algorithms may grow in complexity, for now, the small-scope hypothesis appears to align well with the size and structure of quantum algorithms.

Motivation for Using the Small-Scope Hypothesis Assuming the Small-scope hypothesis holds, there are clear benefits to its use. Specific to the quantum setting for small instances, simulation is feasible. And relying on simulation a number of analysis techniques (including our own approach) become available. In particular, the direct observation of state vectors is very beneficial for debugging. Additionally, for developers it is often easier to analyze small examples than very large and complex ones. Thus limiting analysis to the easiest to understand instances increases the chances of resulting in accessible counter-examples. Last but not least, the analysis itself gets easier and thus is faster more often than not. This faster feedback is especially crucial in early development to spot simple mistakes. Overall, we argue that analyzing small instances of a given problem (first) can significantly benefit the developer.

Summary We have outlined four key reasons why the small-scope hypothesis is particularly relevant to quantum computing. While these points may evolve as quantum computing and algorithms advance, we believe the overall reasoning will remain valid. The nature of quantum computing lends itself to the small-scope hypothesis, making methods that rigorously examine quantum programs with small input sizes a valuable contribution to quantum software development.

6.9 Related Work

The verification of quantum software as a research area has been extensively covered by two surveys: one by Lewis et al. [97] and another by Chareton et al. [33]. Both surveys overlap significantly, particularly in the area most relevant to this thesis—the presentation and evaluation of tools for formally verifying quantum software. Additionally, Ying and Feng have provided a survey focusing on challenges and current approaches to model-checking quantum systems [150], which is closely aligned with the approach proposed in this thesis.

This work concentrates on tools and frameworks that enable the verification of quantum software rather than delving into the underlying logics or calculi that form the basis of these tools.

Fundamentally, tools for verifying quantum software can be categorized along two primary dimensions. The first dimension concerns the underlying verification technique employed to conduct proofs. Broadly, tools can be divided into those that are based on theorem provers and those that leverage automated verification techniques, such as SAT- and SMT-solvers. This distinction has a significant impact on the degree of automation achievable.

The second dimension relates to the level of abstraction for both code and specifications. This ranges from abstract representations of quantum circuits — often embedded in simplified while-languages — to fully-fledged quantum programming languages in practical use today. Notably, the choice of abstraction often correlates with the verification technique: higher automation is frequently associated with lower levels of abstraction, while theorem prover-based approaches typically operate on more detailed and customizable representations.

In the following sections, we review existing approaches to quantum software verification along these two dimensions, starting with tools based on theorem provers.

6.9.1 Proof Assistant-Based Approaches

The formal verification of quantum programs has seen significant progress through the use of well-established proof assistants and theorem provers. These tools provide rigorous frameworks to ensure the correctness of quantum circuits and algorithms, leveraging formal semantics and interactive theorem proving.

QWire, SQIR, and CoqQ: Proving Quantum Circuits in Coq

QWire [114] and SQIR [78] are quantum circuit description languages deeply embedded within the Coq proof assistant [21], facilitating the formal verification of quantum algorithms and circuits. QWire was among the first quantum programming languages to allow the construction of quantum circuits embedded in a classical host language. Its semantics were formalized in Coq, enabling verification of simple quantum programs [119].

SQIR builds on QWire, particularly its matrix library, addressing some of QWire's limitations in verifying complex algorithms. The introduction of "unitary SQIR" in SQIR simplified the verification of purely unitary circuits (those without measurements) and restricted the language to a predefined number of qubits, unlike QWire, which permits dynamic allocation and deallocation of qubits. SQIR has been used to formally verify several well-known algorithms, including Grover's search, the Quantum Fourier Transform (QFT), and Quantum Phase Estimation (QPE).

Both QWire and SQIR benefit from Coq's interactive theorem-proving environment, offering a verification framework with features such as a proof language, graphical user interface, and extensive libraries. However, this approach demands substantial manual effort, as proofs often involve numerous detailed steps.

More recently, CoqQ [155] introduced a new method for verifying quantum while programs in Coq. This work formalized the programming language itself along with the underlying Hoare logic. Specifications are expressed using labeled Dirac notation, and CoqQ supports the verification of quantum algorithms for arbitrary numbers of qubits. Because of this generality, significant manual interaction is still required. For example, verifying Grover's algorithm involved approximately 180 lines of code for the algorithm and 140 lines for the proof. To support verification, the MathComp library was utilized.

QHLProver

QHLProver [101] is a verification tool implemented in the Isabelle theorem prover, based on the quantum Hoare logic introduced by Ying [149]. This approach enables the deductive verification of quantum circuits. Similar to SQIR, QHLProver benefits from its integration with an established theorem-proving framework, inheriting both its strengths and limitations.

QHLProver targets a quantum-while language, omitting a classical host language and dynamic qubit allocation. The tool's significant achievement lies in proving the soundness and completeness of the QHL deduction system (for partial correctness). As a practical demonstration, Grover's algorithm was successfully verified.

QBricks

QBricks [32] represents an alternative approach to quantum program verification, built upon the Why3 verification platform [62]. In contrast to Coq and Isabelle, Why3 provides a higher degree of automation by relying on SMT solvers to discharge proof obligations.

QBricks departs from the standard matrix-vector representation of quantum states, instead adopting parameterized path-sums introduced in [6]. Quantum circuits and their specifications are expressed using two domain-specific languages: QBricks-DSL for circuit descriptions and QBricks-Spec for formal specifications. Using this framework, several prominent quantum algorithms, including Grover's search, QFT, QPE, and the phase estimation subroutine of Shor's algorithm, were successfully verified.

However, QBricks has certain limitations compared to other approaches. It lacks built-in support for measurements and does not consider integration with a classical host language. These restrictions may limit its applicability for hybrid quantum-classical programs.

Isabelle Maries Dirac (IMD)

Isabelle Marries Dirac (IMD) leverages the Isabelle theorem prover to formalize the matrix-vector representation of quantum states and gates. IMD allows for the verification of fundamental quantum properties, such as the no-cloning theorem, by reasoning directly on the mathematical representation of quantum systems [97].

IMD has been described as "a verifiable mathematical library, rather than a verifiable programming language,"[97] emphasizing its focus on foundational quantum properties rather than algorithm-level verification. The approach has been applied to small quantum algorithms, such as Deutsch-Jozsa, and to analyze the quantum version of the Prisoner's Dilemma.

In summary, proof assistant-based approaches offer rigorous frameworks for quantum program verification, enabling precise reasoning about complex quantum algorithms. However, their reliance on interactive theorem proving often requires substantial manual effort.

6.9.2 Fully Automated Verification Approaches

In contrast to proof assistant-based verification, fully automated verification approaches rely entirely on automated solvers to perform proofs. These methods aim to reduce human involvement while achieving rigorous verification of quantum programs.

SilVer

SilVer [98] is a verification tool designed for the quantum programming language Silq [22]. It introduces a specification language, `SilSpeq`, which allows developers to define preconditions and postconditions for Silq programs using purely classical expressions. Given a program and its specification, SilVer generates proof obligations that are discharged by an SMT-solver. SilVer models the classical and quantum components of programs separately, supporting Silq's hybrid capabilities. Verification has been demonstrated on algorithms like Deutsch-Jozsa and Bernstein-Vazirani, emphasizing its ability to handle hybrid quantum-classical programs. Notably, SilVer is closely related to our work as it focuses on the verification of hybrid quantum programs through classical input/output specifications. The tool exhibits similar characteristics, such as verification times scaling exponentially and applicability to fixed qubit counts only.

QPMC and `Entang`λe

QPMC [59] is a model checker for quantum circuits based on IscasMC [76]. It represents quantum programs and protocols as quantum Markov chains (QMCs) and extends the PRISM language [94] to capture these QMCs. State and gate matrices are represented using IEEE 754 floating-point values, although the potential for floating-point errors affecting verification results is not addressed. Classical host languages are not considered, and the approach has been applied to small quantum programs such as superdense coding and quantum key distribution, with verification times under a second. However, the circuit sizes used in these examples are not detailed.

QPMC was later extended to `Entang`λe[9, 8], which supports a sublanguage (Quip-E) of the real-world quantum programming language Quipper [73]. `Entang`λe translates Quipper programs into QPMC-compatible QMCs and includes a graphical user interface (GUI) for program modeling and verification result analysis. Examples demonstrate support for tail recursion and verification of fixed-size quantum circuits, though parameterized algorithm descriptions are not supported.

symQV

symQV [16] is a symbolic verification approach capable of verifying quantum circuits against first-order logical specifications. The method employs an SMT-solver to verify circuits with up to 24 qubits, achieving scalability through a sound abstraction technique. Unlike some other tools, symQV avoids rounding errors by using real numbers to represent quantum states during SMT translation.

The approach has been evaluated on examples such as the Quantum Fourier Transform (QFT) and Quantum Phase Estimation (QPE). However, it does not cover standard algorithms like Deutsch-Jozsa, Grover's search, or Shor's algorithm, limiting its comparative applicability. Moreover, symQV does not support integration with a classical host language.

NQPV

Feng and Xu [58] introduce a verification approach for nondeterministic quantum while-programs, extending the standard quantum-while language with a nondeterministic choice statement ($S_1 \square S_2$), which selects either S_1 or S_2 for execution. The authors present formal semantics and a logic for this nondeterministic language, alongside a prototypical implementation, NQPV, which implements a weakest precondition calculus for the proposed logic.

NQPV supports fully automated verification but requires user-provided loop invariants. It has been demonstrated on examples like Grover's algorithm and nondeterministic quantum walks. Similar to other tools, NQPV verifies properties for a fixed number of qubits, limiting its scalability for parametric quantum programs.

Fully automated verification approaches significantly reduce the need for manual proof construction by leveraging model checkers, SMT-solvers, and symbolic reasoning techniques. These tools demonstrate promising results for verifying small-scale quantum programs or circuits with fixed qubit counts. However, they often face challenges in scalability, particularly for parameterized quantum algorithms or hybrid quantum-classical systems.

6.10 Conclusion

We have demonstrated that hybrid software incorporating real-world quantum circuits can be specified and verified by translating quantum algorithms into a classical programming language. This translation approach allows the use of familiar languages and established practices without the need for modifications. We introduced our approach for a theoretical minimal language and proved the translation to have equivalent semantics to the translated circuit. We then discussed how this theoretical work can be instantiated for a real-world programming language and verification tool on the example of Java and JJBMC.

However, there are two notable limitations to our approach. First, by using floating-point numbers instead of real numbers, we introduce rounding errors as a natural source of inaccuracy. We showed that these errors do not affect the verification result, however, we relied on a theoretical upper limit for rounding errors occurring during simulation of quantum circuits. This bound will be derived in the next chapter. The second limitation is related to scalability. While the translation scales linearly with the number of gates in the circuit, it scales exponentially with the number of qubits. Although we identify several opportunities to optimize and improve the scalability of our approach, this exponential scaling imposes a fundamental limit on handling large qubit counts. Furthermore, the overall scalability depends not only on the translation but also on the efficiency of the analysis tool used.

Despite these limitations, the proposed approach fulfills the majority of the requirements outlined in Section 6.2. Specifically, our approach is fully automatic, achieving the highest degree of automation. For all considered examples, including several well-known quantum algorithms, the method successfully established the desired properties within a matter of seconds. Although scaling was not a primary concern for these small examples, the behavior for larger qubit counts demonstrated the expected trends.

While the language supported by our tool is not a fully-fledged quantum programming language, it closely aligns with prominent programming frameworks such as Qiskit and Cirq, as evidenced by the examples provided. Importantly, by grounding our verification process in a tool designed for the classical host language, we achieve comprehensive support for this language, making our approach truly hybrid. This capability stands in contrast to other approaches, which either lack support for classical code or provide only limited integration within quantum algorithms.

Moreover, this reliance on the classical host language enables additional debugging capabilities. When code fails to adhere to its specification, our tool can generate counterexamples, including the input parameters leading to the failure and a complete execution trace.

Finally, the restriction to specifications expressed within the classical host language, (combined with the fully automated nature of the approach that facilitates inlining of subroutines), results in highly concise specifications, as demonstrated through comparisons with other tools.

Despite the scalability challenges, we argue that the flexibility of our approach makes it highly practical. According to the small scope hypothesis, which posits that a significant proportion of errors manifest in small inputs, we argue that errors in quantum algorithms predominantly already appear with a small number of qubits. This allows us to verify hybrid quantum programs efficiently and automatically, increasing their reliability. Our experiments and case studies support this claim, demonstrating that the approach successfully verifies real-world quantum circuits. We were able to prove several well-known quantum algorithms, including a minimal version of Shor's algorithm, and detect known bugs in multiple publicly available quantum programs. These results confirm the applicability of our approach to real-world quantum and hybrid algorithms.

Chapter 7

Floating-Point Errors in Quantum Circuit Simulations

In the previous chapter, we demonstrated how hybrid quantum software can be formally verified by translating quantum circuits into equivalent classical software. To ensure sound verification in classical languages using floating-point arithmetic, it is, however, necessary to account for potential rounding errors introduced in this translation. In this context, we examine a broader question: how do floating-point errors affect quantum circuit simulators? As our translation essentially constitutes a specialized simulator for a given circuit, results from this broader analysis can be directly used to establish the soundness of using floating-point arithmetic in our verification approach.

We show that an upper bound on the error introduced by floating-point arithmetic in quantum circuit simulation can indeed be derived. This bound is sufficient to rule out significant errors in most circuits that are practically simulatable on standard desktop computers. Specifically, we provide two types of bounds: one that applies generally, parameterized by the number of qubits and gates, and another that applies to circuits with limited gate sizes, independent of the qubit count. This chapter contributes to answer **RQ4** for quantum software systems.

7.1 Motivation

As discussed earlier, quantum computers today are affected by numerous types of errors, which significantly limit their usability for most practical applications (Section 2.1.9). Consequently, developers of quantum software frequently turn to simulators to run their programs. Simulators offer several advantages, such as avoiding long queue times, providing enhanced capabilities for testing and debugging, and, most critically, enabling developers to examine quantum software without the interference of NISQ-related errors found in physical devices. However, using classical simulators introduces a distinct form of error: rounding errors arising from floating-point arithmetic, as opposed to exact real-number arithmetic. These inaccuracies may diverge from the behavior of ideal qubits, introducing a risk of costly deviations.

While rounding errors in floating-point computations—particularly in matrix multiplications—are well studied, we are unaware of any work that applies these findings specifically to quantum simulators. In this chapter, we establish theoretical upper bounds for the rounding errors that may occur during quantum circuit simulation. To do this, we describe the specific simulation approach considered, then demonstrate that the rounding errors introduced at each computational step are constrained by theoretical upper bounds. These bounds are formally proven, providing formal guarantees on rounding error behavior in quantum circuit simulation. Our analysis considers a generic circuit defined by its qubit and gate count alone.

Following this, we evaluate the derived bounds in practical scenarios by applying them to three distinct quantum simulation cases. We demonstrate that the bounds are effective for ensuring accuracy in a large set of circuits, confirming that rounding errors have a minimal impact on these simulations.

7.2 Foundations and Notation

We now introduce foundational concepts related to the floating-point format and the arithmetic operations defined over such numbers. This preliminary discussion establishes the necessary background to understand the numerical behaviors that affect floating-point operations. Following this, we define key notation that is used throughout this chapter.

7.2.1 Floating-Point Format and IEEE 754

Floating-point numbers are widely used to represent real numbers with finite precision. The most common format for floats is a triple (s, m, e), where s is a single bit denoting the sign, m is the mantissa, and e is the exponent (which effectively shifts the "point," hence the term floating-point). The value of this triple is given by $s \times (1.m) \times b^e$, where b is a predefined base, typically 2 or 10. In this work, we focus on the IEEE 754 floating-point standard, the most widely used. It defines two primary formats, differing in precision: 23 and 52 bits for the mantissa, and 8 and 11 bits for the exponent, both with a base of 2 and a 1-bit sign. These formats correspond to the float and double types in programming languages like Java and C.

The IEEE standard imposes several requirements on implementations. The key property for this work is that certain operations must be rounded to the nearest representable floating-point number [70]. These operations include addition, subtraction, multiplication, division, and square root calculations, among others.

The standard also defines special values, such as NaNs ("not a number") and positive/negative infinity. However, these special values can be disregarded in this context, as they do not arise in the relevant calculations.

7.2.2 Notation

We make use of some standard notions and notation in the context of floating-point arithmetic. The maximal relative rounding error, called *unit roundoff*, is $u = 2^{-p}$ where p is the precision (bits of the mantissa). We write $\mathit{fl}(x)$ to denote the result of rounding x, i.e., the nearest number to x that is floating-point-representable. That is:

$$x \in \mathbb{R} : |\mathit{fl}(x) - x| = \min\{|f - x| : f \in \mathbb{F}\}$$

where $\mathbb{F}$ is the set of floating-point values. Elementwise application of $\mathit{fl}(\cdot)$ to vectors and matrices follows naturally. We also use this notation to indicate that operations are conducted in floating-point arithmetic. That is, $\mathit{fl}(a \circ b)$ indicates that a and b are rounded according to $\mathit{fl}(\cdot)$, and then the result of the operation $\circ$ is again rounded.

In this chapter the absolute value when applied to vectors or matrices is always considered to be applied elementwise unless explicitly stated otherwise.

We borrow the concept of *unit in the first place* $\mathit{ufp}(r)$ from [124], which is the value of the most significant non-zero bit in the binary representation of r, i.e.,

$$0 \neq r \in \mathbb{R} \Rightarrow \mathit{ufp}(r) := 2^{\lfloor log_2|r| \rfloor}$$

The unit in the first place has the following properties for all $x \in \mathbb{R}$ and $f = \mathit{fl}(()x) \in \mathbb{F}$:

$$|x - f| \ \leq \ u \cdot \mathit{ufp}(x) \ \leq \ u \cdot \mathit{ufp}(f) \tag{7.1}$$

$$0 \neq x \in \mathbb{R} : \quad ufp(x) \leq |x| < 2\, ufp(x) \tag{7.2}$$

The ufp of a floating-point number can be easily computed in floating-point arithmetic with the following operations [123]:

```
    $\phi := (2u)^{-1} + 1$
    q := $\phi$ * p
    ufp := |q - (1 - u) * q|
```

We repeat standard definitions of vector as well as matrix norms for convenience. For $x \in \mathbb{C}^n$:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

$$\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{\frac{1}{2}}$$

$$\|x\|_\infty = \underset{1 \leq i \leq n}{max} |x_i|$$

and for $A \in \mathbb{C}^{m \times n}$

$$\|A\|_1 = \underset{1 \leq j \leq n}{max} \sum_{i=1}^{m} |a_{ij}|$$

$$\|A\|_2 = \underset{\|x\|_2 = 1}{max} \|Ax\|_2$$

$$\|A\|_\infty = \underset{1 \leq i \leq m}{max} \sum_{j=1}^{n} |a_{ij}|$$

Since we focus on the simulation of quantum circuits, we primarily consider unitary matrices. Remember that a matrix U is unitary if and only if $U^\dagger U = UU^\dagger = I$, where $U^\dagger$ is the complex-conjugate transpose of U. Key properties of unitary matrices relevant to our analysis include:
- $\|U\|_2 = 1$,
- Each row and column vector u_i of U satisfies $\|u_i\|_2 = 1$,
- For any element u_{ij} of U, we have $|u_{ij}| \leq 1$.

These properties are fundamental in bounding errors within quantum circuit simulations.

Last but not least we make use of typical Wilkinson-style bounds and thus use the γ_k-notation:

$$\gamma_k = \frac{ku}{1 - ku}$$

7.3 Simulation of Quantum Circuits

Quantum circuit simulation can be approached using various state representations, but the most straightforward method represents states as complex-valued vectors and gates as appropriately sized matrices. To avoid confusion and because floating-point analysis can be highly sensitive to even subtle variations, we present the exact simulation algorithm we examine in Listing 7.1.

The simulation we consider is subject to a few assumptions. First, as shown in line 8, we assume the simulation begins in a specific state, particularly $|0\rangle$. This assumption is common and can be made without loss of generality, as any state can be prepared by applying suitable gates. Second, we assume all gates to be of size $n \times n$, where n corresponds to the system's dimension. Any smaller gate can be expanded to this size without introducing rounding errors, so this assumption also holds without loss of generality. Finally, we assume that all qubits are measured at the end of the circuit. While this limits the simulation to algorithms without mid-circuit measurements, or those requiring only partial qubit measurements, it simplifies the analysis in the following sections. However, the algorithm can easily be adapted to accommodate these cases.

All inputs and outputs in the simulation are considered complex-valued. The resulting state vector is thus also complex-valued. It is worth noting that each complex value could be represented by two real or floating-point numbers with suitable adaptations to the arithmetic operations. For clarity and readability, however, we maintain the use of complex numbers throughout. In this context, real-valued literals should be interpreted as complex numbers (e.g., 1 is equivalent to $1 + 0i$).

The algorithm begins by initializing the state vector to the starting state $|0\rangle$. This is both a common and practical choice, as it ensures that the initial state can be represented without any rounding errors, which might not be the case for an arbitrary starting state. The algorithm then applies the gates in sequence, with each gate application corresponding to a matrix multiplication between the current state vector and the gate matrix. This matrix multiplication is performed in the classical manner, where each element of the resulting vector is computed as the dot product of the current state vector and the corresponding row of the gate matrix. It is important to note the order of operations, especially that recursive summation is used in the dot product. Finally, all qubits are measured, which involves two phases: (1) calculating the probability of each measurement outcome, and (2) adapting the state according to the observed result.

```
Inputs:
    - $A_1, ..., A_i$ (list of n x n matrices)
    - x (vector of length n)
Output:
    - y (final vector after simulation)
    - $b_1, ..., b_n$ (measurement results)

y := (1, 0, ... , 0)

for A : gates:
    y' := (0, ...., 0)
    for i = 0 .. n:
        y'[j] := 0
        for j = 0 .. n:
            y'[i] := y[i] + (A[i][j] * x[j])
    y := y'

for j = 0 ... n:
    p := 0
    for i = 0..n:
        if (i // $2^j$) % 2 == 0:
            p := p + $|y[i]|^2$
    p := sqrt(p)
    r := random([0, 1))
    if r < p:
        for i = 0..n:
            if (i // $2^j$) % 2 == 1:
                y[i] := 0
        y := y / p
        $b_j$ := 0
    else:
        for i = 0..n:
            if (i // $2^j$) % 2 == 0:
                y[i] := 0
        y := y / (1 - p)
        $b_j$ := 1
```

Listing 7.1: Algorithm to simulate a quantum circuit as considered in this chapter

The probability computation is done for one of the two possible outcomes, with the other obtained as its complement. The probability of a given outcome is calculated by summing the squared absolute values of the corresponding state elements. Listing 7.1 checks each element to see if the j-th bit of its index is zero (as seen in line 21). The operator // denotes integer division, as is standard in most programming languages. The squared absolute value of a complex number, as computed in line 22, is simply the sum of the squares of its real and imaginary parts.

The second phase, state adaptation, consists of two steps. First, the elements of the state vector that did not contribute to the chosen outcome are set to zero. Then, the state is normalized by dividing the remaining vector by the probability, ensuring the final state vector has a unit length. The same condition used to determine which elements to zero out can be reused here, with the roles of ones and zeros swapped.

The final part of the algorithm involves determining which measurement result is observed. Since this is a probabilistic process, we use a random value from the interval $[0, 1)$ to compare with the calculated probability. The comparison in line 25 determines the observed result. There are two corner cases to consider: when $p = 1$, any random value leads to the selection of the if-branch, and when $p = 0$, the else-branch is always selected. These cases justify the use of the half-open interval $[0, 1)$ for the random value rather than a closed interval.

This completes our formulation of the quantum circuit simulation algorithm, which we examine in the remainder of this chapter. Importantly, this simulation method is identical to the one employed in our translation method described in the previous chapter. Consequently, any insights derived from our analysis of rounding errors in this chapter directly inform the understanding of rounding errors in the programs translated by QIn.

7.4 Static Error Bounds for Quantum Circuit Simulators

In this section, we give bounds on the error that possibly occurs due to rounding during the simulation of a quantum circuit based on the number of gates and the number of qubits alone. This error is mainly characterized by the error that each application of a gate introduces. Special care must be taken to account for the consequences that rounding errors may have on subsequent operations.

7.4.1 Extended Standard Error Bounds

We can derive an upper bound on the error for quantum simulations by recursively applying the standard error bounds for matrix multiplication as given in [79]. For $A \in \mathbb{R}^{m\times n}$ and $x \in \mathbb{R}^n$:

$$|Ax - fl(Ax)| \le \gamma_n |A||x|$$

where $|\cdot|$ is applied componentwise to vectors and matrices. Higham [79] shows that this induces similar bounds based on norms and can readily be extended to include complex values. For $A \in \mathbb{C}^{m\times n}$ and $y \in \mathbb{C}^n$:

$$\|Ax - fl(Ax)\|_\infty \le \gamma_{n+2}\|A\|_\infty\|x\|_\infty \tag{7.3}$$

Thus, we can get an error estimate solely knowing the ∞-norms of A and x. Since the matrices are gates applied to a state and are typically directly given rather than the result of complex computations, we assume that rounding errors only occur during the conversion from reals to floats. Since any row of a unitary matrix is a unit vector, we can use the following theorem to bound the rounding errors resulting from that conversion:

Theorem 7.1 (Rounding error unit vector)**.** *Let $x = (x_1, \dots, x_n) \in \mathbb{C}^n$ with $\|x\|_2 = 1$. Then*

$$\|x - fl(x)\|_1 \le 4u$$

where $fl(x) = (fl(x_1), \dots, fl(x_n))$, provided that no underflow occurs.

Proof. We first prove a stronger bound for $\|x - fl(x)\|_1$ with $x \in \mathbb{R}^n$ and then argue that this is sufficient for the complex case. Using (7.1), we know that

$$\begin{aligned}\|x - fl(x)\|_1 &= \|x - (x+\delta)\|_1 \le \|\delta\|_1 \\ &= \sum_{i=0}^{n}|\delta_i| \le \sum_{i=0}^{n}(u \cdot ufp(x_i)) \\ &= u\sum_{i=0}^{n} ufp(x_i)\end{aligned}$$

Also, since $\|x\|_2 = \sqrt{\sum_{i=0}^{n}|x_i|^2} = 1$, we know that $\|x\|_2 = \sum_{i=0}^{n}|x_i|^2 = 1$. And from (7.2) we get $\sum_{i=0}^{n} ufp(x_i^2) \le \sum_{i=0}^{n}|x_i|^2 = 1$. Using the fact that $\sqrt{|x_i|} \le 2|x_i|$ for all $x_i \in [-1, 1]$, this is implies $\sum_{i=0}^{n} ufp(|x_i|) \le 2$. Then, using our previous reasoning, we get $\|x - fl(x)\|_1 \le u\sum_{i=0}^{n} ufp(x_i) \le$

$2u$, which concludes the proof for reals. A simple rewriting of the initial equation gives the desired result for the complex case (now $x \in \mathbb{C}^n$):

$$\begin{aligned}\|x - fl(x)\|_1 &= \|x - (re(x) + \delta_1 + i(img(x) + \delta_2))\|_1 \\ &= \|\delta_1 + \delta_2\|_1 \leq \|\delta_1\|_1 + \|\delta_2\|_1 \\ &\leq 2u + 2u = 4u\end{aligned}$$

Where $re(\cdot)$ and $img(\cdot)$ are the real and imaginary part of the complex value x. Note that the δ_i are real-valued vectors, which allows us to reuse our previous result. This concludes the proof. ■

As mentioned above, any row in a unitary matrix is a unit vector, which implies $\|A\|_\infty \leq \sqrt{n}$. Theorem 7.1 gives a bound on the rounding errors that can affect this value such that:

$$\|fl(A)\|_\infty \leq \sqrt{n} + 4u$$

This bounds the first component we were looking for in eq. 7.3. However, for quantum states in a simulation, the assumption that only the conversion to floats can lead to rounding errors is wrong. The states are the result of repeated matrix multiplications, which may amplify rounding errors. Therefore, it is crucial to examine these errors more thoroughly. In the following, we consider a circuit consisting of g gates represented by matrices A_i ($1 \leq i \leq g$). Let x_i be the state after applying i matrices/gates with a starting state $x_0 = (1, 0, \dots, 0)^T$. We show the following theorem:

Theorem 7.2 (Rounding error gates)**.** *Let x_g be the resulting vector of a quantum circuit simulation with g gates and q qubits.*

Then, if $(\sqrt{n} + 4u)\gamma_{n+2} \leq 1$, the following holds:

$$\|x_g\|_\infty \leq 1 + (\sqrt{n} + 4u) \cdot \gamma_{n+2} \cdot 2^g$$

If, moreover, $(g-1)\gamma_{n+2}(\sqrt{n} + 4u) \leq 1$ (or equivalent $g \leq \frac{2^{p-q-1}}{\sqrt{n}}$), then

$$\|x_g\|_\infty \leq 1 + (\sqrt{n} + 4u) \cdot \gamma_{n+2} \cdot \frac{(g-1) \cdot g}{2}$$

Remark 7.3. *The assumption $(\sqrt{n}+4u)\gamma_{n+2} \leq 1$ implies the bounds $q \leq 15$ and $q \leq 35$ on the number of qubits for single and double precision floating-points respectively.*

Remark 7.4. *In Table 7.1, the maximum number of gates such that the stronger assumption in Theorem 7.2 is still met is given for the corresponding amount of qubits. One can easily see that for each additional qubit, the* *number of gates is less than half the previous one. For double precision, these amounts of gates are very high, which shows that for most realistic simulations, the tighter bounds are applicable (e.g. for 10000 gates, circuits up to 26 qubits would satisfy the assumption). For single precision, the situation is very different. Even for comparatively small circuits the number of gates to fulfill the assumption becomes very restricted.*

Remark 7.5. *The theorem holds equivalently for $\|\cdot\|_1$. The proof is analogous, which is why we do not show it here.*

#Qubits	**#Gates (double)**	**#Gates (float)**
1	$\approx 3 \cdot 10^{14}$	$\approx 3 \cdot 10^{6}$
2	$\approx 1 \cdot 10^{14}$	$\approx 1 \cdot 10^{6}$
3	$\approx 3 \cdot 10^{13}$	$\approx 4 \cdot 10^{5}$
4	$\approx 1 \cdot 10^{13}$	$\approx 1 \cdot 10^{5}$
5	$\approx 5 \cdot 10^{12}$	$\approx 5 \cdot 10^{4}$
6	$\approx 2 \cdot 10^{12}$	$\approx 1{,}6 \cdot 10^{4}$
7	$\approx 6 \cdot 10^{11}$	≈ 5800
8	$\approx 2 \cdot 10^{11}$	≈ 2000
9	$\approx 8 \cdot 10^{10}$	≈ 700

Table 7.1: Maximum number of gates to meet the stronger assumption in Theorem 7.2 for double and single precision

Proof. Essentially, the proof is a recursive application of (7.3). However, to be able to apply (7.3), we have to bound the ∞-norm of each state vector accounting for accumulating rounding errors. We know that each state is the result of the previous state being multiplied with a matrix plus a rounding error. So we can write each x_i as:

$$x_i = A_i x_{i-1} + (\epsilon_i, \dots, \epsilon_i)^T$$

and thus

$$\|x_i\|_\infty \leq \|A_i x_{i-1}\|_\infty + \epsilon_i \tag{7.4}$$

Knowing that $\|x_0\|_\infty = 1$, we can repeatedly apply (7.4) and get

$$\|x_i\|_\infty \leq 1 + \sum_{j=1}^{i} \epsilon_j \tag{7.5}$$

Let $\theta = (\sqrt{n} + 4u) \cdot \gamma_{n+2}$. Then, using (7.3) and (7.5), we can estimate ϵ_i:

$$\begin{aligned}\epsilon_i &\leq \gamma_{n+2} \|A_i\|_\infty \|x_{i-1}\|_\infty \\ &\leq \theta \cdot \left(1 + \sum_{j=1}^{i-1} \epsilon_j\right) \\ &= \theta + \theta \cdot \sum_{j=1}^{i-1} \epsilon_j\end{aligned}$$

We can recursively apply this estimation of e_i to see that this error is a polynomial of the form:

$$\epsilon_i = \sum_{k=1}^{i} c_k \cdot \theta^k$$

where the c_k have the following values for different i:

$$\begin{aligned}\text{if } i = 1 \text{ then } c_1 &= 1 \\ \text{if } i = 2 \text{ then } c_1, c_2 &= 1, 1 \\ \text{if } i = 3 \text{ then } c_1, c_2, c_3 &= 1, 2, 1 \\ \text{if } i = 4 \text{ then } c_1, c_2, c_3, c_4 &= 1, 3, 3, 1 \\ \text{if } i = 5 \text{ then } c_1, c_2, c_3, c_4, c_5 &= 1, 4, 6, 4, 1 \\ &\vdots\end{aligned}$$

Notice three properties of the coefficient c_k: (a) The sum of all c_k is always 2^{i-1}. (b) For any two consecutive c_k, c_{k+1}, the maximum relative difference is between c_1 and c_2, i.e., $\max \frac{c_{k+1}}{c_k} = \frac{c_2}{c_1}$ for all i and c) For all $i \geq 2$, we have $c_1 = 1$ and $c_2 = i - 1$.

These properties give rise to the assumptions in Theorem 7.2. The first assumption is that $\theta = (\sqrt{n} + 4u)\gamma_{n+2} \leq 1$. Making this assumption here, we know that $\theta^k \leq \theta$ for any $k \geq 1$. Thus, we can approximate ϵ_i as follows:

$$\epsilon_i = \sum_{k=1}^{i} c_k \cdot \theta^k \leq \theta \sum_{k=1}^{i} c_k \leq \theta \cdot 2^{i-1}$$

Using this in eq. (7.5), gives us:

$$\|x_i\|_\infty \leq 1 + \sum_{j=1}^{i} \epsilon_j \leq 1 + \sum_{j=1}^{i} (\theta \cdot 2^{j-1}) \leq 1 + \theta \cdot 2^i$$

which is exactly what Theorem 7.2 states. This concludes the proof of the first part of the theorem.

The idea behind the additional assumption is that, under certain circumstances, we can show that $f_k \cdot \theta^k \leq \theta$. This is trivially true for $k = 1$, as $c_1 \cdot \theta^1 = 1 \cdot \theta \leq \theta$. For all other k, consider the relation to the previous iteration: $\frac{c_{k+1} \cdot \theta^{k+1}}{c_k \cdot \theta^k} = \theta \cdot \frac{c_{k+1}}{c_k} \cdot \frac{\theta^k}{\theta^k} = \theta \cdot \frac{c_{k+1}}{c_k} \leq \theta \cdot \frac{c_2}{c_1} = \theta \cdot (i-1)$ (the last two equations follow directly from properties (b) and (c) we noted about the coefficients c_k). This relation between the coefficients shows that, whenever $\theta \cdot (i-1) \leq 1$, we know that $c_{k+1} \cdot \theta^{k+1} \leq c_k \cdot \theta^k$ and thus $c_k \cdot \theta^k \leq \theta$ for all k. The assumption $\theta \cdot (i-1) \leq 1$ is exactly our second assumption in Theorem 7.2. As hinted before, we can use this again in eq. (7.5) with the following effect:

$$\|x_i\|_\infty \leq 1 + \sum_{j=1}^{i} \epsilon_j \leq 1 + \sum_{j=1}^{i} \theta \cdot (j-1) \leq 1 + \theta \cdot \frac{(i-1) \cdot i}{2}$$

■

This result is useful in two different ways:

- It provides an upper bound on the rounding error introduced by the application of gates to a quantum state.
- It provides an upper bound on the largest value that can occur during the simulation.

In order to fully bound the error of the simulation of quantum circuits we also have to provide bounds for measurements. This is what we tackle next.

We consider measurements to be conducted as described in Listing 7.1, which consists of two parts: (a) the computation of the measurement probability and (b) the adaptation of the state.

The computation of the probability conducted here is basically the computation of a dot product. And we can thus again use our familiar error estimates. This results in the following theorem:

Theorem 7.6 (Rounding error measurement)**.** *For a measurement conducted in the computational basis in state x_i results in a computed probability p such that:*

$$|p - fl(p)| \leq \gamma_{n+2} \cdot (1 + \sqrt{n} \cdot \gamma_{n+2} 2^{i+1})^2$$

and, for the resulting state $x_i' = x_i/p$, it holds that:

$$\|x_i/p - fl(x_i/p)\|_\infty \leq u$$

Remark 7.7. *If the additional assumption in Theorem 7.2 is met, we can use its sharper error estimates here as well.*

Proof. The first bound essentially follows directly from (7.3). Calculating the probability is equivalent to calculating the dot product of the state with itself (and drawing the square root). As the dot product is a special case of matrix multiplication, we can use (7.3), and we have that $A = x^T$ and, thus, $\|A\|_\infty = \|x\|_1$. Using that together with the bounds from Theorem 7.2, we get the following:

$$\begin{aligned}|p - fl(p)| &\leq \gamma_{n+2} \cdot \|x\|_1 \cdot \|x\|_\infty \\ &= \gamma_{n+2} \cdot (1 + \sqrt{n} \cdot \gamma_{n+2} 2^{i+1})^2\end{aligned}$$

The normalization of the state is effectively a componentwise division, where the divisor is the probability we just calculated. This is, however, not done on the entire state but only on the part that is actually observed (the other part is set to zero beforehand). We use a finding by Boldo who showed that, for any $a, b \in \mathbb{F}$, it holds that $\frac{a}{\sqrt{a^2+b^2}} \leq 1$ even when calculated in floating-point arithmetic [24]. We can easily extend that to:

$$a_0, \dots, a_n \in \mathbb{F} : \frac{a_i}{\sqrt{\sum_{i=0}^n a_i^2}} \leq 1$$

which is also true for the complex case. The normalization step is done by this exact computation so we know that for any element the rounding error for the normalization can be at most u and thus:

$$\|x_i/p - fl(x_i/p)\|_\infty \leq u$$

which concludes the proof. ■

As we have characterized the errors for both gate applications and measurements, we have fully characterized the possible rounding errors in a quantum circuit.

7.4.2 Improved Error Bounds

The bounds just presented heavily rely on the underlying bound for a single matrix multiplication. Thus it is natural that an improvement of that bound translates to an improvement in the resulting overall bound. In this section we are going to use this intuition and rely on an improved bound provided by Rump in [122]:

$$|Ax - float(Ax)| \leq n \cdot u \cdot |A| \cdot |x|$$

$$|Ax - fl(Ax)| \leq n \cdot u \cdot |A| \cdot |x|$$

Here, the inequality is applied element-wise. This shows that the error in a matrix-vector multiplication between a matrix A and a vector x depends on the absolute values of both the matrix and the vector, as well as their dimensions. This bound can be extended directly to matrix norms. We mainly use the following form:

$$\|Ax - fl(Ax)\|_2 \leq n \cdot u \cdot \||A| \cdot |x|\|_2$$

This form is derived by applying the norm to both sides of the original inequality. We now demonstrate how this bound can be used to estimate errors in the simulation of quantum circuits. Consider the following notation: Let A_k represent the k-th matrix applied in the quantum circuit, and let x_k denote the quantum state resulting from multiplying this matrix with the previous state vector, i.e., $x_k = A_k x_{k-1}$.

As a first step, we derive a bound for real-valued matrices and vectors, assuming that all A_k matrices are free of rounding errors, i.e., $fl(A_k) = A_k$. Additionally, we assume an upper bound b on the norm of each A_k, specifically that $\||A_k|\|_2 \leq b$ for all k. Although we will later show that such bounds are always achievable, for now we assume this condition holds. Using these assumptions, we can derive a bound on the error introduced during the simulation of a quantum circuit.

Theorem 7.8 (Improve rounding error gates). *Given that $A_i \in \mathbb{R}^{n\times n}$ and $fl(A_i) = A_i$:*

$$\|fl(A_i x_{i-1}) - A_i x_{i-1}\|_2 \leq (1 + n \cdot u \cdot b)^i - 1$$

or equivalently

$$\|fl(x_i) - x_i\|_2 \leq (1 + n \cdot u \cdot b)^i - 1$$

Remark 7.9. *Note that this bound exhibits exponential scaling in the number of gates; however, we will demonstrate that for a substantial class of circuits, the term $n \cdot u \cdot b$ can be limited to a very small value. Consequently, this exponential growth is less problematic than it may initially appear.*

Remark 7.10. *Observe that the number of qubits affects only the matrix size, n. Later, we will establish a bound that is entirely independent of the number of qubits.*

Remark 7.11. *Since this bound resembles a typical Wilkinson-type bound, it can be upper-bounded by the more computationally manageable expression* $\frac{c \cdot k}{1 - c \cdot k}$*, provided* $c \cdot k \leq 1$ *with* $c = n \cdot u \cdot b$.

We continue by proving Theorem 7.8.

Proof. We prove Theorem 7.8 by induction.

Base Case: $(i = 1)$

$$\begin{aligned}
&\|fl(A_1 x_0) - A_1 x_0\|_2 && (7.6)\\
&\leq n \cdot u \cdot \||A_1||x_0|\|_2 && (7.7)\\
&\leq n \cdot u \cdot \||A_1|\|_2 \cdot \||x_0|\|_2 && (7.8)\\
&= n \cdot u \cdot b \cdot 1 && (7.9)\\
&= (1 + n \cdot u \cdot b)^1 - 1 && (7.10)
\end{aligned}$$

This derivation primarily relies on definitions introduced above. We use Rump's original bound to get to (7.7). Notably, since we assume the initial state $x_0 = |1\rangle$, we know $\||x_0|\|_2 = \|x_0\|_2 = 1$, applied in (7.9). Ultimately, in (7.10), we confirm that the base case $i = 1$ has the expected form.

Induction Step: Assuming Theorem 7.8 holds for a k, we now show that this implies that it holds for $k + 1$ as well. For brevity, we define $c = n \cdot b \cdot u$.

$$\begin{aligned}
&\|fl(x_{k+1}) - x_{k+1}\|_2 && (7.11)\\
=&\|fl(A_{k+1} \cdot x_k) - A_{k+1} x_k\|_2 && (7.12)\\
\leq&\|A_{k+1} \cdot fl(x_k) - A_{k+1} \cdot x_k\|_2 + \|n \cdot u \cdot |A_{k+1}| \cdot |fl(x_k)|\|_2 && (7.13)\\
\leq&\|A_{k+1}\|_2 \cdot \|fl(x_k) - x_k\|_2 + n \cdot u \cdot b \cdot \|fl(x_k)\|_2 && (7.14)\\
=&\|fl(x_k) - x_k\|_2 + c \cdot \|fl(x_k) - x_k + x_k\|_2 && (7.15)\\
\leq&\|fl(x_k) - x_k\|_2 + c \cdot \|fl(x_k) - x_k\|_2 + c \cdot \|x_k\|_2 && (7.16)\\
=&\|fl(x_k) - x_k\|_2 \cdot (1 + c) + c && (7.17)\\
\leq&\Big((1 + c)^k - 1\Big) \cdot (1 + c) + c && (7.18)\\
=&(1 + c)^{k+1} - 1 && (7.19)
\end{aligned}$$

We discuss these steps in more detail now. We use the original bound to get to (7.13). Note, however, that the original bound only accounts for the error that is introduced due to the dot-product operation. The errors to compute the two parameters A_{k+1} and x_k are not covered by that. This is why we have to use $fl(x_k)$. Since we assumed $fl(A_k) = A_k$ we can avoid this for the matrix A_k. In (7.15) we use the fact that $\|A_{k+1}\|_2 = 1$. Note, the difference between $\||A_k|\|_2 \leq b$ and $\|A_k\|_2 = 1$. The same reason allows us to neglect the factor of $\|x_k\|_2 = 1$ in (7.17). Last but not least we use the induction hypothesis in (7.18) to replace $\|fl(x_k) - x_k\|_2$ with $(1 + n \cdot u \cdot b)^k - 1 = (1 + c)^k - 1$. Eventually (7.19) again shows that the error has the expected form and thus concludes the proof. ■

Remember that this upper bound applies to a simplified scenario where we consider only real-valued matrices and vectors, with the additional assumption that matrices are exactly representable as floating-point numbers without rounding errors. However, these assumptions are not realistic for quantum computing. Quantum computations typically require complex-valued arithmetic and matrices with elements that are transcendental numbers. Such elements cannot be represented precisely as floating-point values, and any bound derived under these assumptions does not fully account for the intricacies of practical quantum simulations.

We extend our bounds to incorporate rounding errors arising within gate matrices. Specifically, we consider only the rounding errors introduced when each matrix element is approximated to the nearest representable floating-point value. Errors related to the actual computation of each matrix element, however, remain outside the scope of this analysis.

Since each A_i is unitary its elements fall within $[-1, 1]$. Consequently u serves as an upper bound for rounding each element. Even more specifically we can use:

$$|fl(A_i) - A_i| \leq u \cdot |A_i|.$$

Using this idea, we can bound the perturbed matrix norm $fl(A_i)$ as follows:

$$\|fl(A_i)\|_2 \leq \|A_i + u \cdot |A_i|\|_2 \leq 1 + b \cdot u,$$

and similarly,

$$\|fl(|A_i|)\|_2 \leq b + b \cdot u.$$

Using these bounds on $\|fl(A_i)\|_2$ and $\|fl(|A_i|)\|_2$, we obtain:

$$\|fl(x_k) - x_k\|_2 \leq (1 + n \cdot u^2 \cdot b + u \cdot b + n \cdot u \cdot b)^k - 1.$$

This bound is derived from Theorem 7.8, substituting matrix norms with the just derived upper bounds. For brevity, the full induction-based proof is omitted.

This provides an upper bound on the error in quantum circuit simulation given a valid b such that $b \geq \||A_i|\|_2$. The optimal choice under this condition is $b = \sqrt{n}$, justified by the inequality

$$\|A\|_2 \leq \sqrt{\|A\|_1 \cdot \|A\|_\infty}.$$

Given a unitary matrix A and the matrix A' obtained by taking element-wise absolute values of A, each row (or column) v of A' has a 1-norm of 1 due to unitarity. Applying Cauchy-Schwarz to this 1-norm we find:

$$\|v\|_1 = \sum_{i=1}^{n} |v_i| \leq \sqrt{n}\|v\|_2 = \sqrt{n}.$$

It follows that $\|A'\|_1 = \|A'\|_\infty \leq \sqrt{n}$, and thus due to the aforementioned inequality of the norms $\|A'\|_2 \leq \sqrt{n}$, providing the required upper bound for $\||A_i|\|_2$.

7.4.3 Extending to Complex-Valued Arithmetic

To extend our bounds to complex-valued matrices and vectors, we represent complex arithmetic using real arithmetic by mapping complex numbers to pairs of real numbers. Specifically, any complex vector in $\mathbb{C}^n$ can be transformed into a corresponding real-valued vector in $\mathbb{R}^{2n}$.

Building on this intuition, matrix multiplication in $\mathbb{C}^n$ can also be represented as an equivalent operation in $\mathbb{R}^{2n}$. Given a matrix $A_k \in \mathbb{C}^{n\times n}$ and a vector $x_k \in \mathbb{C}^n$, we define the following equivalent real-valued representations:

$$A'_k = \begin{pmatrix} \mathrm{Re}(A_k) & -\mathrm{Im}(A_k) \\ \mathrm{Im}(A_k) & \mathrm{Re}(A_k) \end{pmatrix} \in \mathbb{R}^{2n\times 2n}$$
$$x'_k = \begin{pmatrix} \mathrm{Re}(x_k) \\ \mathrm{Im}(x_k) \end{pmatrix} \in \mathbb{R}^{2n}$$

Note that if A_k is unitary, then A'_k is also unitary, and similarly, if x_k is a unit vector, then x'_k is also a unit vector. This rephrasing of complex matrix multiplication as a real-valued matrix multiplication with doubled dimensions enables us to apply the previously derived bounds for the real-valued case, simply adjusting for the dimension by using $2n$ in place of n throughout. As we set $b = \sqrt{n}$ in the real case this is also affected, and we now have to use $b = \sqrt{2n}$.

Consequently, we state our main result:

Theorem 7.12 (Rounding error circuits). *For a circuit consisting of k gates $A_i \in \mathbb{C}^{n\times n}$ and initial state $x_0 = |1\rangle$, a simulation as outlined in Listing 7.1 (without measurements) yields a final state x_k for which*

$$\begin{aligned}&\|fl(x_k) - x_k\|_2 \\ \leq&(1 + 2 \cdot n \cdot u^2 \cdot b + u \cdot b + 2 \cdot n \cdot u \cdot b)^k - 1\end{aligned}$$

holds.

This eventually gives us the desired bound for quantum circuit simulations based only on the number of applied gates and qubits. Note, that this bound is not accounting for underflow errors. This could be fixed by adding a small additive term. For the remainder of this chapter we will not consider this option but all further extensions could be be adapted accordingly.

7.4.4 Computable Bound

The derived bound is valid but not directly computable without rounding errors. In most cases where a rough estimate suffices, this may not be a significant issue. However, for formal guarantees, it is necessary to find a bound that can be calculated without incurring rounding errors. A trivially computable bound derived from Theorem 7.12 is:

$$\|fl(x_k) - x_k\|_2 \leq fl(\frac{8 \cdot n^2 \cdot u \cdot 2^{\lceil \log_2(k)\rceil}}{1 - 8 \cdot n^2 \cdot u \cdot 2^{\lceil \log_2(k)\rceil}} + u)$$

given that $\frac{8 \cdot n^2 \cdot u \cdot 2^{\lceil \log_2(k)\rceil}}{1 - 8 \cdot n^2 \cdot u \cdot 2^{\lceil \log_2(k)\rceil}} < 1$.

We use the fact that the bound in Theorem 7.12 follows a Wilkinson-style form for which the upper bound $\frac{c \cdot k}{1 - c \cdot k}$ is known. In our case we have $c = 2 \cdot n \cdot u^2 \cdot b + u \cdot b + 2 \cdot n \cdot u \cdot b$. To make the bound computable, we exploit the error-free nature of multiplication by powers of two in floating-point arithmetic. Since both n and u are powers of two, no adjustments are needed for them. Overapproximating b as $2n$ also yields a power of two and allows us to use $n \cdot u \cdot b$ as an upper bound for both $b \cdot u$ and $n \cdot u^2 \cdot b$ which leaves us with $c = 8 \cdot c \cdot n^2 \cdot u$. Last but not least we overapproximate k by simply calculating the next biggest power of two.

The only rounding error remaining that we have to consider is the division, which is known to yield results in the range $[0, 1]$ (otherwise the condition explicitly given wouldn't hold). Hence, the maximum rounding error is bounded by u, and adding u sufficiently accounts for this error. While this results in a bound that overestimates the error significantly, it remains practical and sufficient for many scenarios (see Section 7.5).

7.4.5 Considering Gate Sizes

For now, we have considered the most general case of gates acting on the entire quantum state. While theoretically possible, this is uncommon in practice. In fact, most gates that can be applied on actual quantum devices operate on a maximum of two qubits at a time. Naturally, one would expect that a gate acting on only a small subset of qubits introduces a proportionally smaller error. This expectation holds, and we can use it to tighten our error bound.

Since our bound depends explicitly on the size n of the quantum state, which corresponds to the matrix dimensions, we can adjust it straightforwardly. When applying a gate acting on only p qubits, this operation corresponds to a matrix multiplication of size $2^p \times 2^p$. Thus, for any simulation where the largest gate acts on p qubits, we can replace n in our bound with 2^p, yielding a more accurate estimate.

7.4.6 Bounds for measurement probabilities

We have derived bounds for the application of an arbitrary number of gates. Next, we extend the improved bounds to measurements. To achieve this, we observe that computing the probability of a measurement outcome can be interpreted as a dot product of a vector composed of the relevant elements of the original quantum state.

Let q represent the quantum state of a system with 3 qubits. Suppose we aim to compute the probability of measuring the first qubit and observing 0 as the outcome. This probability is given by $\sum_i |q_i|^2$, where the sum runs over the appropriate indices i. Notably, this calculation can be viewed as the dot product of a vector $\mathbf{q}'$, whose elements are identical to the relevant q_i. Consequently, the rounding errors that occur during measurement can be bounded using established bounds for dot products.

Using this insight, we arrive at the following theorem:

Theorem 7.13 (Improved bound for measurement)**.** *Let q be the state of an N qubit system. Conducting a measurement of k qubits in that state will result in probability p_i for each possible outcome i such that:*

$$|p_i - fl(p_i)| \leq 2^{N-k} \cdot u \cdot (1 + 2 \cdot \|\varepsilon\|_2 + \|\varepsilon\|_2^2)$$

where ε is the maximum error introduced by gates before the measurement.

Remark 7.14. *A computable version of this bound can be derived easily with the following result:*

$$|p_i - fl(p_i)| \leq ((2^{N-k} \cdot u) + (4 \cdot \|\varepsilon\|_2 \cdot 2^{N-k} \cdot u)) + u$$

Remark 7.15. *We will use the computable version of this bound to deduce the maximum possible rounding error in QIn translation as discussed in Section 6.5.*

Proof.

$$\begin{aligned}
|p_i - fl(p_i)| &= |fl(x) \cdot fl(x) - fl(x \cdot x)| \\
&\leq n \cdot u \cdot |fl(x)| \cdot |fl(x)| \\
&= n \cdot u \cdot (|x| + \varepsilon) \cdot (|x| + \varepsilon) \\
&= n \cdot u \cdot (|x|^2 + 2 \cdot |x| \cdot |\varepsilon| + |\varepsilon|^2) \\
&\leq n \cdot u \cdot (\|x\|_2^2 + 2\|x\|_2\|\varepsilon\|_2 + \|\varepsilon\|_2^2) \\
&\leq n \cdot u \cdot (1 + 2\|\varepsilon\|_2 + \|\varepsilon\|_2^2)
\end{aligned}$$

where **x** denotes the vector containing all elements relevant to the measurement outcome i, for which we compute the probability p_i. If all qubits are measured, **x** reduces to a single element. However, for each qubit that is not measured, the number of elements required to compute a single outcome doubles. Consequently, the total number of elements needed to compute a measurement outcome for k measured qubits in an N-qubit system is 2^{N-k}. ■

With this we have established the final bounds for simulating quantum circuits.

7.5 Evaluation

In the previous section, we derived theoretical bounds for rounding errors in quantum simulations. We now discuss the implications of these theoretical findings for realistic scenarios and compare the different bounds presented. In particular, we examine the following three scenarios:

- **Simulation of test circuits**: In this scenario, we consider circuits with only very few qubits and gate operations. This type of circuit often occurs as minimal working examples.

#qb	**#gates**	**naive(f)**	**imp(f)**	**comp(f)**	**naive(d)**	**imp(d)**	**comp(d)**
3	10	1.52e-04	8.11e-05	1.96e-03	1.41e-13	7.55e-14	1.82e-12
3	100	1.67e-02	8.11e-04	1.59e-02	1.55e-11	7.55e-13	1.46e-11
3	10000	1.69e+02	8.44e-02	-	1.57e-07	7.55e-11	1.86e-09
3	100000	1.69e+04	1.25e+00	-	1.57e-05	7.55e-10	1.49e-08
5	10	1.03e-03	6.20e-04	3.23e-02	9.61e-13	5.77e-13	2.91e-11
5	100	1.13e-01	6.22e-03	3.33e-01	1.06e-10	5.77e-12	2.33e-10
5	1000	1.15e+01	6.39e-02	-	1.07e-08	5.77e-11	1.86e-09
5	10000	1.15e+03	8.59e-01	-	1.07e-06	5.77e-10	2.98e-08
10	100	1.94e+01	2.00e+00	-	1.80e-08	1.03e-09	2.38e-07
10	10000	-	5.54e+47	-	1.82e-04	1.03e-07	3.05e-05
10	1000000	-	-	-	1.82e+00	1.03e-05	1.96e-03
15	10000	-	-	-	3.29e-02	1.86e-05	3.23e-02
20	10000	-	-	-	5.96e+00	3.38e-03	-
25	10000	-	-	-	1.08e+03	8.41e-01	-
30	10000	-	-	-	-	5.25e+47	-

Table 7.2: Results for the general case of n-qubit gates comparing the naive bound, the improved bound and the computable bound for single (f) and double (d) precision

#gates	**naive(f)**	**imp(f)**	**comp(f)**	**naive(d)**	**imp(d)**	**comp(d)**
10	6.44e-05	3.03e-05	4.89e-04	6.00e-14	2.66e-14	4.55e-13
1000	7.15e-01	3.04e-03	3.23e-02	6.65e-10	2.66e-12	2.91e-11
100000	7.15e+03	3.55e-01	-	6.66e-06	2.66e-10	3.73e-09
10000000	-	1.51e+13	-	6.66e-02	2.66e-08	4.77e-07
1000000000	-	-	-	6.66e+02	2.66e-06	3.05e-05

Table 7.3: Results for the case of only applying 1- and 2-qubit gates: comparing the naive bound, the improved bound and the computable version of the new bound for single (f) and double (d) precision

- **Simulation on PCs**: This scenario comprises all circuit sizes that are still realistically simulated on standard PCs. We consider this to be the case for circuits with 10-20 qubits, depending on the actual hardware used (as a reference Qiskit defines the maximum number of qubits simulatable with their basic simulator to be 24 qubits [116] however, this already leads to very long simulation times).

- **HPC simulation**: For this scenario, we consider circuits that exceed the capabilities of PCs and are only realistically tackled on high-performance-computers (HPC). We consider this to be the case for circuits with upwards of 20 qubits.

The results are summarized in Tables 7.2 and 7.3. Table 7.2 presents the bounds for the general case, where all gates are assumed to be applied to all qubits. Table 7.3, on the other hand, lists the bounds under the assumption that only 1- and 2-qubit gates are applied. In each case we compare three different types of bounds: the first presented bound (naive), the improved bounds based on Rump's improved matrix multiplication error (improved), and the computable version of this later (comp). All calculations for the results in these tables were performed using floating-point arithmetic with double precision ($p = 53$). The bounds are computed based on the theoretical insights provided in the previous section, separately for single and double precision (denoted by the (f) and (d) columns). A dash ($-$) indicates that the bound could not be computed for a particular combination of qubits and gates, either because the conditions required for the application of the bound were not met or because an overflow occurred during the computation.

Building on these values, we discuss the three quantum circuit simulation scenarios mentioned: the simulation of test circuits, simulations on personal computers (PCs), and high-performance computing (HPC) simulations.

Simulation of Test Circuits For very small quantum circuits, it is clear that simulation-induced errors remain minimal (on the order of $\leq 10^{-11}$ for double precision). This level of precision should be sufficient to exclude any significant errors arising from rounding in almost all practical applications. This conclusion holds, to some extent, for single precision as well. For instance, a circuit with 3 qubits and 100 gates exhibits a maximum error on the order of 10^{-2}.

It may be surprising, however, that even for relatively small circuits, such as those with 5 qubits and 100 gates, the error in single-precision simulations starts to become prohibitively large. Despite this, for circuits of this size, rounding errors can be considered negligible when double precision is used, assuming no exceptional precision requirements.

When considering circuits composed solely of 1- and 2-qubit gates, no significant advantage is observed for very low qubit numbers. However, the error scaling independent of qubit count, makes circuits with high qubit numbers but limited gate depth manageable. Notably, for very small circuits, the naive bounds outperform the computable bounds, albeit only slightly. This discrepancy arises from over-approximations made to ensure the computability of the derived bounds.

Simulation on PCs Now consider circuits with a qubit count in the range between 10 and 15. This should be a manageable task for current PCs. As Table 7.2 shows for single precision even here no guarantees can be provided unless the restriction to 2-qubit gates is taken into account. Which as a first takeaway leads to the recommendation that double precision should be used from this point on (or rather in general as a precaution). Double precision, however, is still quite accurate as even for 15 qubits and 10000 gates the maximum error even for the general naive bound is below 0.04. For most algorithms, this precision should still be enough to acquire reasonably accurate results. However, for even higher numbers of qubits the general naive bound starts to provide meaningful results. Similarly the computable bound stops being applicable altogether. Notably, the improved bound provides significantly lower results allowing simulations of circuits with up to 20 qubits comparatively high accuracy. For the limited case of only 1- and 2-qubit gates, all bounds provide meaningful bounds even for gate counts as high as 100000 (using double precision).

HPC Simulation Circuits with a large number of qubits are only feasible to simulate within a reasonable timeframe on HPC systems. As discussed, in such cases, the general bounds we have derived are too coarse to offer meaningful guarantees for these larger circuits. However, the state-size independent bounds become highly relevant, as they continue to provide reasonable error guarantees even for high-qubit circuits, with error growth dependent only on the number of gates.

Remarkably, even the computable bound in double precision yields significant results—errors on the order of 10^{-6}—for circuits comprising up to 10 million gates. The improved bounds demonstrate particularly favorable scaling in this context. While the naive bound fails to provide reliable error guarantees as gate count increases, the computable bound remains in the range of 10^{-5}, underscoring its robustness in high-qubit, high-gate scenarios.

Further Considerations Note, how all values in Tables 7.2 and 7.3 for which we were able to provide a bound meet the stronger assumption of Theorem 7.2. This shows that the tighter bound is essentially the only relevant one. In most cases, if the stronger assumption is not met, the bound on the rounding errors is so wide that it is of no practical use. The tighter bound, however, is able to provide meaningful bounds for a wide range of circuits.

For higher amounts of qubits the error bounds are increasingly coarse. This stems from the fact that the presented considerations are worst-case calculations. However, one has to keep in mind that the bounds we presented are overapproximations of the actually occurring errors in two ways: (1) We mathematically overapproximated the errors during the proofs we made. And (2) even exact bounds are bounds for worst-case calculations which in reality rarely occur. The presented values are therefore to be seen as absolute guarantees for worst-case scenarios rather than estimations of the average rounding error that occurs during the simulation of quantum circuits. Thus the main takeaway should be that for the simulation of small to medium-size circuits, rounding errors can essentially be guaranteed to have no impact on the results. For bigger circuits, these guarantees can not (yet) be given, which however, does not indicate that the results are inherently flawed.

7.6 Related Work

Floating-point arithmetic has been extensively studied by researchers from various perspectives. Numerous general investigations have addressed its challenges comprehensively (e.g., [70, 110]). In contrast, our approach focuses on a restricted subset of floating-point numbers (e.g., within a certain interval) and operations, leveraging the specific properties associated with these constraints.

In numerical analysis, significant work has been conducted on error bounds for (fast) matrix multiplications, such as in [49, 13, 48]. These studies typically examine matrix multiplication in a general context rather than within a specific application domain. As a result, the derived bounds are broadly applicable but tend to be more coarse-grained, as they cannot exploit the unique properties of specialized applications. In contrast, the bounds we have presented are tailored specifically to quantum simulations. These works lay a solid foundation and could inspire future research for our approach. Our current analysis is restricted to a single simulation approach, limiting its general applicability. However, [49] demonstrates that an entire class of matrix multiplication algorithms shares stability properties. Building on this result, it may be possible to extend the bounds introduced in this work to encompass a wider range of simulation techniques.

Another approach to analyzing floating-point errors involves focusing on concrete programs rather than conducting numerical error analysis upfront.

For Java, tools like JBMC [44], which we employed, and the KeY verification tool [1] provide examples of this methodology. The latter demonstrates how to verify the absence of special values and functional properties involving floating-point arithmetic. Beyond Java, a variety of tools and approaches tackle the verification of floating-point programs across different programming languages and abstraction levels. For instance, Why3 [65] includes support for floating-point arithmetic, and several interactive theorem provers formalize floating-point arithmetic to enable the verification of complex properties [24, 82, 151]. However, these approaches often require significant user interaction to complete proofs. More automated techniques frequently rely on abstract interpretation [20, 35, 46, 72] or SAT-/SMT-solvers [133, 27].

In the context of quantum circuit verification or simulation, floating-point arithmetic has received relatively little attention. Some studies, such as [75], explore how floating-point computations might be performed on quantum devices, but this does not pertain to simulators. Fatima and Markov [57] recognize the potential errors introduced by floating-point arithmetic and propose methods to reduce the number of operations required for quantum circuit simulations. Combining their techniques with our approach could be a promising avenue for future research. Liu et al. [102] present a simulator employing a mixed-precision strategy, where double precision is used selectively for critical parts of the simulation while defaulting to single precision elsewhere. However, their approach lacks formal analysis and is primarily based on empirical observations and experimental results.

Additionally, Niemann et al. [112] examine how floating-point errors affect the compactness of quantum multiple-valued decision diagrams. They investigate trade-offs between accuracy and performance by determining which values can be considered identical under floating-point arithmetic. Their proposed arithmetic approach eliminates inaccuracies by representing all values in a ring $\mathbb{D}[\omega]$. While this could potentially be an interesting extension of our approach, the increased computational overhead that comes with it might pose significant challenges for automated verification.

7.7 Conclusion

We have presented theoretical considerations on how rounding errors due to the use of floating-points affect the simulation of quantum circuits. We derived upper bounds on the error that can affect the final state of a simulation only based on the number of qubits and the number of gate operations. All upper bounds are formally proven, thus providing guarantees on the nature of the errors that might occur. Additionally, we used these bounds to examine the effects of rounding errors for three different scenarios of quantum simulation. We were able to show that the presented bounds are sufficient to essentially guarantee the absence of relevant floating-point errors for a large set of quantum circuits.

Chapter 8

Conclusion and Future Work

We have presented three approaches to enhance the reliability of non-classical systems, addressing different levels of abstraction.

On the architectural level, we introduced a classification framework that categorizes components of non-classical systems into four distinct classes of reliability based on the guarantees they provide. While this approach offers only a coarse-grained analysis, it allows for an approximate evaluation of the overall reliability of a system when applied systematically.

Subsequently, we demonstrated how fault-tolerant architectural patterns can be adapted and applied to quantum components to improve their reliability. This approach operates at a relatively high level of abstraction, as it does not rely on assumptions about the internal structure or properties of individual components. Nonetheless, our evaluation revealed that these patterns significantly enhance the reliability of the considered components.

While these methods increase reliability, they are not able to provide guarantees about system behavior. Consequently, we proposed an approach for verifying hybrid quantum software at the source code level via translation. This method enables the verification of quantum algorithms by translating quantum circuits into a classical host language while preserving relevant properties. Using this approach, we successfully proved the correctness of implementations of several well-known quantum algorithms and identified known bugs in others. Importantly, this method is fully automated and, due to its reliance on a classical host language, accessible even to non-expert users.

Finally, we analyzed rounding errors that arise during the simulation of quantum circuits due to the use of floating-point arithmetic. We derived a formally proven bound that quantifies the size of these errors based on the number of qubits and gates in a circuit. This bound, refined to depend on the largest gate applied, was shown to effectively rule out significant rounding errors for a wide range of circuits. Additionally, we introduced a computable variant of this bound, making it practical for real-world applications.

Together, these approaches provide a comprehensive framework for increasing the reliability of non-classical systems across multiple levels of abstraction. From coarse-grained architectural categorizations to fine-grained guarantees at the source code and numerical levels, these methods contribute significantly to the reliability of non-classical systems.

Future Work

Despite the progress made in improving the reliability of non-classical systems, there remains ample room for further research. Several promising directions can be pursued to build on the results presented in this thesis.

For the proposed fault-tolerant architectural patterns, an immediate extension would involve exploring additional patterns that have demonstrated success in other domains and adapting them to quantum computing or other non-classical systems. Beyond introducing new patterns, further experiments with alternative aggregation and error detection methods, as well as exploring additional strategies for generating variants of components, could yield valuable insights. Combining the architecture-level patterns with other established noise reduction techniques is another promising avenue. Identifying the most effective combinations of methods could lead to further reliability improvements.

In the context of formal verification of hybrid software, various enhancements can improve scalability and efficiency. Optimizing the translation process, for instance by reducing the number of classical state updates to only the affected portions of the state, could lead to significant performance gains. Supporting loops without unrolling and handling a variable number of qubits would further extend the applicability of this approach. Additionally, exploring alternative programming languages and verification tools may reveal configurations that are better suited to specific tasks. Applying the proposed method to Python-based frameworks, such as Qiskit and Cirq, would be particularly impactful, though it requires robust Python verification tools, which currently present a challenge.

For the analysis of rounding errors in quantum circuit simulation, there is substantial potential to refine the presented bounds. While the derived bounds are effective, they are not tight, leaving room for improvement. Identifying the optimal, tightest bound would be a rewarding achievement. Moreover, instead of relying solely on the number of qubits and gates, runtime analysis of the simulation algorithm could dynamically compute bounds for observed values. This approach could yield tighter results in most cases, although it would come at the cost of increased computational effort. This trade-off may be acceptable in scenarios where tighter bounds are critical.

Finally, while this thesis focuses primarily on quantum computing systems, the methods and insights presented here are not limited to this domain. As discussed in Chapter 3, significant parallels exist between quantum computing and other non-classical systems, such as machine learning. Investigating how the presented approaches can be adapted and extended to other non-classical systems represents a valuable direction for future research and could amplify the impact of this work.

List of Figures

List of Listings

List of Tables

List of Theorems and Definitions

Bibliography

[1] Rosa Abbasi et al. “Deductive Verification of Floating-Point Java Programs in KeY”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Springer International Publishing, 2021, pp. 242–261. ISBN: 978-3-030-72013-1. DOI: `10.1007/978-3-030-72013-1_13`.

[2] Héctor Abraham et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2021. DOI: `10.5281/zenodo.2573505`.

[3] Dorit Aharonov. “A simple proof that Toffoli and Hadamard are quantum universal”. In: *arXiv preprint quant-ph/0301040* (2003). DOI: `10.48550/arXiv.quant-ph/0301040`.

[4] Wolfgang Ahrendt et al., eds. *Deductive Software Verification – The KeY Book*. Vol. 10001. Lecture Notes in Computer Science. Springer International Publishing, 2016. ISBN: 978-3-319-49811-9. DOI: `10.1007/978-3-319-49812-6`.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. “Detecting Equality of Variables in Programs”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. New York, NY, USA: ACM, 1988, pp. 1–11. ISBN: 978-0-89791-252-5. DOI: `10.1145/73560.73561`.

[6] Matthew Amy. “Towards Large-scale Functional Verification of Universal Quantum Circuits”. In: *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 2019), pp. 1–21. ISSN: 2075-2180. DOI: `10.4204/eptcs.287.1`.

[7] Alexandr Andoni et al. “Evaluating the “small scope hypothesis””. In: *In Popl*. Vol. 2. 2003.

[8] Linda Anticoli et al. “Entangle: A Translation Framework from Quipper Programs to Quantum Markov Chains”. In: *New Frontiers in Quantitative Methods in Informatics*. Ed. by Simonetta Balsamo,

Andrea Marin, and Enrico Vicario. Springer International Publishing, 2018, pp. 113–126. ISBN: 978-3-319-91632-3. DOI: 10.1007/978-3-319-91632-3_9.

[9] Linda Anticoli et al. “Towards Quantum Programs Verification: From Quipper Circuits to QPMC”. In: *Reversible Computation*. Ed. by Simon Devitt and Ivan Lanese. Springer International Publishing, 2016, pp. 213–219. ISBN: 978-3-319-40578-0. DOI: 10.1007/978-3-319-40578-0_16.

[10] Erfan Asaadi, Ewen Denney, and Ganesh Pai. “Quantifying assurance in learning-enabled systems”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2020, pp. 270–286.

[11] Algirdas Avizienis. “The methodology of n-version programming”. In: *Software fault tolerance* 3 (1995), pp. 23–46.

[12] Stanley Bak et al. “Improved Geometric Path Enumeration for Verifying ReLU Neural Networks”. In: *Computer Aided Verification*. Vol. 12224. Lecture Notes in Computer Science. 2020, pp. 66–96.

[13] Grey Ballard et al. “Improving the Numerical Stability of Fast Matrix Multiplication”. In: *SIAM Journal on Matrix Analysis and Applications* 37.4 (Jan. 2016), pp. 1382–1418. DOI: 10.1137/15M1032168.

[14] Pedro Baltazar, Rohit Chadha, and Paulo Mateus. “Quantum computation tree logic—model checking and complete calculus”. In: *International Journal of Quantum Information* 6.02 (2008), pp. 219–236.

[15] Ezio Bartocci et al. “Introduction to runtime verification”. In: *Lectures on Runtime Verification*. Springer, 2018, pp. 1–33.

[16] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. “symQV: Automated Symbolic Verification of Quantum Programs”. In: *Formal Methods*. Ed. by Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker. Springer International Publishing, 2023, pp. 181–198. ISBN: 978-3-031-27481-7. DOI: 10.1007/978-3-031-27481-7_12.

[17] Christoph Baumann et al. “Lessons Learned From Microkernel Verification — Specification Is the New Bottleneck”. In: *EPTCS* 102 (), pp. 18–32.

[18] Bernhard Beckert et al. “Modular Verification of JML Contracts Using Bounded Model Checking”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles.* Vol. 12476. Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 60–80.

[19] Bernhard Beckert et al. “Proving JDK’s Dual Pivot Quicksort Correct”. In: *Verified Software. Theories, Tools, and Experiments.* Ed. by Andrei Paskevich and Thomas Wies. Springer International Publishing, 2017, pp. 35–48. ISBN: 978-3-319-72308-2. DOI: `10.1007/978-3-319-72308-2_3`.

[20] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. “A Dynamic Program Analysis to Find Floating-Point Accuracy Problems”. In: *SIGPLAN Not.* 47.6 (June 2012), pp. 453–462. ISSN: 0362-1340. DOI: `10.1145/2345156.2254118`.

[21] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions.* Springer Science & Business Media, 2004.

[22] Benjamin Bichsel et al. “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.

[23] Armin Biere and Daniel Kröning. “SAT-Based Model Checking”. en. In: *Handbook of Model Checking.* Ed. by Edmund M. Clarke et al. Springer International Publishing, 2018, pp. 277–303. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_10`.

[24] Sylvie Boldo and Guillaume Melquiond. “Flocq: A unified library for proving floating-point algorithms in Coq”. In: *2011 IEEE 20th Symposium on Computer Arithmetic.* IEEE. 2011, pp. 243–252.

[25] Anthony Bordg, Hanna Lachnitt, and Yijun He. “Certified Quantum Computation in Isabelle/HOL”. In: *Journal of Automated Reasoning* 65.5 (June 2021), pp. 691–709. ISSN: 1573-0670.

[26] Max Born and Pascual Jordan. “Zur quantenmechanik”. In: *Zeitschrift für Physik* 34.1 (1925), pp. 858–888.

[27] Martin Brain, Florian Schanda, and Youcheng Sun. “Building Better Bit-Blasting for Floating-Point Problems”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Tomáš Vojnar and Lijun Zhang. Springer International Publishing, 2019, pp. 79–98. ISBN: 978-3-030-17462-0. DOI: `10.1007/978-3-030-17462-0_5`.

[28] Hans J Briegel et al. “Measurement-based quantum computation”. In: *Nature Physics* 5.1 (2009), pp. 19–26.

[29] Christopher Brix et al. “First three years of the international verification of neural networks competition (VNN-COMP)”. In: *Software Tools for Technology Transfer* 25.3 (2023), pp. 329–339.

[30] Zhenyu Cai et al. “Quantum Error Mitigation”. In: *Reviews of Modern Physics* 95.4 (Dec. 2023), p. 045005. DOI: `10.1103/RevModPhys.95.045005`.

[31] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[32] Christophe Chareton et al. “An Automated Deductive Verification Framework for Circuit-building Quantum Programs”. In: *Programming Languages and Systems*. Ed. by Nobuko Yoshida. Vol. 12648. 2021, pp. 148–177.

[33] Christophe Chareton et al. “Formal Methods for Quantum Algorithms”. In: *Handbook of Formal Analysis and Verification in Cryptography*. 1st ed. Boca Raton: CRC Press, Aug. 2023, pp. 319–422. ISBN: 978-1-00-309005-2. DOI: `10.1201/9781003090052-7`.

[34] Liming Chen and Algirdas Avizienis. “N-version programming: A fault-tolerance approach to reliability of software operation”. In: *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*. Vol. 1. 1978, pp. 3–9.

[35] Wei-Fan Chiang et al. “Efficient Search for Inputs Causing High Floating-Point Errors”. In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’14. Association for Computing Machinery, Feb. 2014, pp. 43–52. ISBN: 978-1-4503-2656-8. DOI: `10.1145/2555243.2555265`.

[36] Cirq Developers. *Cirq*. Version v0.11.0. May 2021. URL: `https://quantumai.google/cirq`.

[37] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 168–176. ISBN: 978-3-540-24730-2.

[38] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.

[39] “Introduction to Model Checking”. en. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Springer International Publishing, 2018. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8`.

[40] Robert T Clemen and Robert L Winkler. “Combining probability distributions from experts in risk analysis”. In: *Risk analysis* 19.2 (1999), pp. 187–203.

[41] David R. Cok. “JML and OpenJML for Java 16”. In: *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*. FTfJP ’21. Association for Computing Machinery, July 2021, pp. 65–67. ISBN: 978-1-4503-8543-5. DOI: `10.1145/3464971.3468417`.

[42] Lucas Cordeiro et al. “JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 183–190. ISBN: 978-3-319-96145-3.

[43] Lucas Cordeiro et al. “JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode”. In: *Computer Aided Verification (CAV)*. Vol. 10981. LNCS. 2018, pp. 183–190.

[44] Lucas Cordeiro et al. “JBMC: A bounded model checking tool for verifying Java bytecode”. In: *Computer Aided Verification*. International Conference on Computer Aided Verification (CAV 2018). Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 183–190.

[45] Piotr Czarnik et al. “Error Mitigation with Clifford Quantum-Circuit Data”. In: *Quantum* 5 (Nov. 2021), p. 592. DOI: `10.22331/q-2021-11-26-592`.

[46] Eva Darulova et al. “Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Springer International Publishing, 2018, pp. 270–

287. ISBN: 978-3-319-89960-2. DOI: `10.1007/978-3-319-89960-2_15`.

[47] Stijn de Gouw et al. "Verifying OpenJDK's Sort Method for Generic Collections". In: *Journal of Automated Reasoning* 62.1 (Jan. 2019), pp. 93–126. ISSN: 1573-0670. DOI: `10.1007/s10817-017-9426-4`.

[48] James Demmel, Ioana Dumitriu, and Olga Holtz. "Fast Linear Algebra Is Stable". In: *Numerische Mathematik* 108.1 (Nov. 2007), pp. 59–91. DOI: `10.1007/s00211-007-0114-x`.

[49] James Demmel et al. "Fast Matrix Multiplication Is Stable". In: *Numerische Mathematik* 106.2 (Mar. 2007), pp. 199–224. DOI: `10.1007/s00211-007-0061-6`.

[50] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255.

[51] David Deutsch and Richard Jozsa. "Rapid Solution of Problems by Quantum Computation". In: *Proceedings: Mathematical and Physical Sciences* 439.1907 (1992), pp. 553–558.

[52] David Deutsch and Richard Jozsa. "Rapid Solution of Problems by Quantum Computation". In: *Proceedings: Mathematical and Physical Sciences* 439.1907 (1992), pp. 553–558.

[53] David Deutsch and Richard Jozsa. "Rapid solution of problems by quantum computation". In: *Proceedings of the Royal Society of London. Series A* 439.1907 (1992), pp. 553–558.

[54] Kai Ding, Andrey Morozov, and Klaus Janschek. "Classification of hierarchical fault-tolerant design patterns". In: *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE. 2017, pp. 612–619.

[55] Bruce Powel Douglass. *Real-time design patterns: robust scalable architecture for real-time systems*. Addison-Wesley Professional, 2003.

[56] Tommaso Dreossi, Alexandre Donzé, and Sanjit A Seshia. "Compositional falsification of cyber-physical systems with machine learning components". In: *Journal of Automated Reasoning* 63.4 (2019), pp. 1031–1053.

[57] Aneeqa Fatima and Igor L Markov. "Faster schrödinger-style simulation of quantum circuits". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 194–207.

[58] Yuan Feng and Yingte Xu. "Verification of Nondeterministic Quantum Programs". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. Association for Computing Machinery, Mar. 2023, pp. 789–805. ISBN: 978-1-4503-9918-0. DOI: `10.1145/3582016.3582039`.

[59] Yuan Feng et al. "QPMC: A Model Checker for Quantum Programs and Protocols". In: *FM 2015: Formal Methods*. Ed. by Nikolaj Bjørner and Frank de Boer. Springer International Publishing, 2015, pp. 265–272. ISBN: 978-3-319-19249-9. DOI: `10.1007/978-3-319-19249-9_17`.

[60] Richard Feynman. "Simulating physics with computers". In: *International Journal of Theoretical Physics* 21 (1982), pp. 553–555.

[61] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. 2013, pp. 125–128.

[62] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". en. In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 125–128. ISBN: 978-3-642-37036-6.

[63] Austin G. Fowler et al. "Surface Codes: Towards Practical Large-Scale Quantum Computation". In: *Physical Review A* 86.3 (Sept. 2012), p. 032324. ISSN: 1050-2947, 1094-1622. DOI: `10.1103/PhysRevA.86.032324`.

[64] Austin G. Fowler et al. "Surface codes: Towards practical large-scale quantum computation". In: *Physical Review A* 86.3 (Sept. 2012). ISSN: 1094-1622.

[65] Clément Fumex, Claude Marché, and Yannick Moy. "Automating the verification of floating-point programs". In: *Verified Software. Theories, Tools, and Experiments: 9th International Conference, VSTTE 2017*. Springer. 2017, pp. 102–119.

[66] Yarin Gal and Zoubin Ghahramani. “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Vol. 48. Proceedings of Machine Learning Research. PMLR, 2016, pp. 1050–1059.

[67] Jakob Gawlikowski et al. “A survey of uncertainty in deep neural networks”. In: *Artificial Intelligence Review* 56 (S1), pp. 1513–1589.

[68] Christian Genest and James V Zidek. “Combining probability distributions: A critique and an annotated bibliography”. In: *Statistical Science* 1.1 (1986), pp. 114–135.

[69] Glosser.ca. *Bloch Sphere*. 2012. URL: `https://commons.wikimedia.org/wiki/Category:Bloch_spheres#/media/File:Bloch_Sphere.svg` (visited on 12/06/2024).

[70] David Goldberg. “What every computer scientist should know about floating-point arithmetic”. In: *ACM Computing Surveys* 23.1 (1991), pp. 5–48.

[71] Divya Gopinath et al. “Property Inference for Deep Neural Networks”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 797–809.

[72] Eric Goubault and Sylvie Putot. “Robustness Analysis of Finite Precision Implementations”. In: *Programming Languages and Systems*. Ed. by Chung-chieh Shan. Springer International Publishing, 2013, pp. 50–57. ISBN: 978-3-319-03542-0. DOI: `10.1007/978-3-319-03542-0_4`.

[73] Alexander S. Green et al. “Quipper: A Scalable Quantum Programming Language”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Association for Computing Machinery, June 2013, pp. 333–342. ISBN: 978-1-4503-2014-6. DOI: `10.1145/2491956.2462177`.

[74] Chuan Guo et al. “On Calibration of Modern Neural Networks”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. 2017, pp. 1321–1330.

[75] Thomas Haener et al. “Quantum Circuits for Floating-Point Arithmetic”. In: *Reversible Computation*. Ed. by Jarkko Kari and Irek Ulidowski. Springer International Publishing, 2018, pp. 162–174. ISBN: 978-3-319-99498-7. DOI: `10.1007/978-3-319-99498-7_11`.

[76] Ernst Moritz Hahn et al. “iscas M c: a web-based probabilistic model checker”. In: *FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings 19*. Springer. 2014, pp. 312–317.

[77] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. “Quantum Algorithm for Linear Systems of Equations”. In: *Physical Review Letters* 103.15 (Oct. 2009), p. 150502. DOI: `10.1103/PhysRevLett.103.150502`.

[78] Kesha Hietala et al. “Proving Quantum Programs Correct”. In: *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2021.21*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. DOI: `10.4230/LIPIcs.ITP.2021.21`.

[79] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2002. DOI: `10.1137/1.9780898718027`.

[80] Adam Holmes et al. “Impact of qubit connectivity on quantum algorithm performance”. In: *Quantum Science and Technology* 5.2 (2020), p. 025009.

[81] Robert A Jacobs. “Methods for combining experts’ probability assessments”. In: *Neural computation* 7.5 (1995), pp. 867–888.

[82] Charles Jacobsen, Alexey Solovyev, and Ganesh Gopalakrishnan. “A parameterized floating-point formalizaton in HOL Light”. In: *Electronic Notes in Theoretical Computer Science* 317 (2015), pp. 101–107.

[83] Kai Jia and Martin Rinard. “Exploiting verified neural networks via floating point numerical error”. In: *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings 28*. Springer. 2021, pp. 191–205.

[84] Daniel Kahneman, Olivier Sibony, and Cass R Sunstein. *Noise: A flaw in human judgment*. Hachette UK, 2021.

[85] Peter J. Karalekas et al. “A Quantum-Classical Cloud Platform Optimized for Variational Hybrid Algorithms”. In: *Quantum Science and Technology* 5.2 (Mar. 2020), p. 024003. ISSN: 2058-9565. DOI: `10.1088/2058-9565/ab7559`.

[86] Guy Katz et al. “Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks”. In: *Computer Aided Verification: 29th International Conference, CAV*. 2017, pp. 97–117.

[87] Guy Katz et al. "Reluplex: An efficient SMT solver for verifying deep neural networks". In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 97–117.

[88] Ian D. Kivlichan et al. "Improved Fault-Tolerant Quantum Simulation of Condensed-Phase Correlated Electrons via Trotterization". In: *Quantum* 4 (July 2020), p. 296. DOI: `10.22331/q-2020-07-16-296`.

[89] Jonas Klamroth. *QIn Source Code and Examples*. Zenodo. Dec. 2024. DOI: `10.5281/zenodo.14267251`.

[90] Jonas Klamroth and Bernhard Beckert. "Bounding Rounding Errors in the Simulation of Quantum Circuits". In: *2024 IEEE International Conference on Quantum Software (QSW)*. July 2024, pp. 99–106. DOI: `10.1109/QSW62656.2024.00024`.

[91] Jonas Klamroth et al. "QIn: Enabling Formal Methods to Deal with Quantum Circuits". In: *2023 IEEE International Conference on Quantum Software (QSW)*. IEEE. 2023, pp. 175–185.

[92] Ronald T Kneusel and Ronald T Kneusel. "Pitfalls of Floating-Point Numbers (and How to Avoid Them)". In: *Numbers and Computers* (2017), pp. 117–135.

[93] Suhas Kotha et al. "Provably bounding neural network preimages". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 80270–80290.

[94] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Springer, 2011, pp. 585–591. ISBN: 978-3-642-22110-1. DOI: `10.1007/978-3-642-22110-1_47`.

[95] Joonho Lee et al. "Even More Efficient Quantum Computations of Chemistry Through Tensor Hypercontraction". In: *PRX Quantum* 2.3 (July 2021), p. 030305. DOI: `10.1103/PRXQuantum.2.030305`.

[96] K Rustan M Leino. "Dafny: An automatic program verifier for functional correctness". In: *International conference on logic for programming artificial intelligence and reasoning*. Springer. 2010, pp. 348–370.

[97] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. "Formal Verification of Quantum Programs: Theory, Tools, and Challenges". In: *ACM Transactions on Quantum Computing* 5.1 (Mar. 2024), pp. 1–35. ISSN: 2643-6809, 2643-6817. DOI: `10.1145/3624483`.

[98] Marco Lewis, Paolo Zuliani, and Sadegh Soudjani. “Automated Verification of Silq Quantum Programs Using SMT Solvers”. In: *2024 IEEE International Conference on Quantum Software (QSW)*. July 2024, pp. 125–134. DOI: 10.1109/QSW62656.2024.00027. eprint: 2406.03119.

[99] Frank Leymann and Johanna Barzen. “The bitter truth about gate-based quantum algorithms in the NISQ era”. In: *Quantum Science and Technology* 5.4 (2020), p. 044007.

[100] Ying Li and Simon C. Benjamin. “Efficient Variational Quantum Simulator Incorporating Active Error Minimization”. In: *Physical Review X* 7.2 (June 2017), p. 021050. DOI: 10.1103/PhysRevX.7.021050.

[101] Junyi Liu et al. “Formal Verification of Quantum Algorithms Using Quantum Hoare Logic”. In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 187–207. ISBN: 978-3-030-25543-5. DOI: 10.1007/978-3-030-25543-5_12.

[102] Yong (Alexander) Liu et al. “Closing the "Quantum Supremacy" Gap: Achieving Real-Time Simulation of a Random Quantum Circuit Using a New Sunway Supercomputer”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. Association for Computing Machinery, Nov. 2021, pp. 1–12. ISBN: 978-1-4503-8442-1. DOI: 10.1145/3458817.3487399.

[103] Yaping Luo et al. “An architecture pattern for safety critical automated driving applications: Design and analysis”. In: *2017 Annual IEEE International Systems Conference (SysCon)*. IEEE. 2017, pp. 1–7.

[104] David C McKay et al. “Qiskit backend specifications for openqasm and openpulse experiments”. In: *arXiv preprint arXiv:1809.03452* (2018).

[105] Matthew Mirman et al. “Robustness certification with generative models”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 1141–1154.

[106] Christoph Molnar. *Interpretable machine learning: a guide for making black box models explainable*. Christoph Molnar, 2022. 318 pp.

[107] Chris Monroe et al. “Demonstration of a fundamental quantum logic gate”. In: *Physical review letters* 75.25 (1995), p. 4714.

[108] Guido Montúfar et al. “On the Number of Linear Regions of Deep Neural Networks”. In: *Neural Information Processing Systems*. 2014.

[109] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. “Hardware Verification using Software Analyzers”. In: *IEEE Computer Society Annual Symposium on VLSI*. 2015, pp. 7–12.

[110] Jean-Michel Muller et al. *Handbook of floating-point arithmetic*. Springer, 2018.

[111] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. en. 10th anniversary. Cambridge University Press, 2010.

[112] Philipp Niemann et al. “Overcoming the Tradeoff Between Accuracy and Compactness in Decision Diagrams for Quantum Computation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (2020), pp. 4657–4668. DOI: `10.1109/TCAD.2020.2977603`.

[113] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. 2002.

[114] Jennifer Paykin, Robert Rand, and Steve Zdancewic. “QWIRE: A Core Language for Quantum Circuits”. In: *Proceedings, 44th ACM SIGPLAN Symp. on Principles of Programming Languages (POPL 2017)*. ACM, 2017, pp. 846–858.

[115] John Preskill. “Quantum computing in the NISQ era and beyond”. In: *Quantum* 2 (2018), p. 79.

[116] IBM Quantum. *IBM Qiskit Documentation: BasicSimulator*. URL: `https://docs.quantum.ibm.com/api/qiskit/qiskit.providers.basic_provider.BasicSimulator` (visited on 03/28/2024).

[117] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. “MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing”. In: *Quantum* (2023). MQT Bench is available at `https://www.cda.cit.tum.de/mqtbench/`.

[118] Atanu Rajak et al. “Quantum annealing: An overview”. In: *Philosophical Transactions of the Royal Society A* 381.2241 (2023), p. 20210417.

[119] Robert Rand, Jennifer Paykin, and Steve Zdancewic. “QWIRE Practice: Formal Verification of Quantum Circuits in Coq”. In: *Electronic Proceedings in Theoretical Computer Science* 266 (Feb. 2018), pp. 119–132. ISSN: 2075-2180. DOI: `10.4204/EPTCS.266.8`.

[120] Kristin Yvonne Rozier. “Specification: The Biggest Bottleneck in Formal Methods and Autonomy”. In: *Verified Software. Theories, Tools, and Experiments.* Ed. by Sandrine Blazy and Marsha Chechik. Springer International Publishing, 2016, pp. 8–26. ISBN: 978-3-319-48869-1. DOI: `10.1007/978-3-319-48869-1_2`.

[121] Yue Ruan et al. “The quantum approximate algorithm for solving traveling salesman problem”. In: *Computers, Materials and Continua* 63.3 (2020), pp. 1237–1247.

[122] S. M. Rump. “Error Bounds for Computer Arithmetics”. In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH).* Los Alamitos, CA, USA: IEEE Computer Society, June 2019, pp. 1–14. DOI: `10.1109/ARITH.2019.00011`.

[123] Siegfried M. Rump. “Error Estimation of Floating-Point Summation and Dot Product”. In: *BIT Numerical Mathematics* 52.1 (Mar. 2012), pp. 201–220. ISSN: 0006-3835, 1572-9125. DOI: `10.1007/s10543-011-0342-4`.

[124] Siegfried M. Rump, Takeshi Ogita, and Shin’ichi Oishi. “Accurate Floating-Point Summation Part I: Faithful Rounding”. In: *SIAM Journal on Scientific Computing* 31.1 (Jan. 2008), pp. 189–224. ISSN: 1064-8275, 1095-7197. DOI: `10.1137/050645671`.

[125] Marie Salm et al. “The NISQ analyzer: Automating the selection of quantum computers for quantum algorithms”. In: *Symposium and Summer School on Service-Oriented Computing.* Springer. 2020, pp. 66–85.

[126] Max Scheerer, Jonas Klamroth, and Oliver Denninger. “Engineering Reliable Hybrid Quantum Software: An Architectural-driven Approach.” In: *Q-SET@ QCE.* 2021, pp. 29–37.

[127] Max Scheerer, Jonas Klamroth, and Oliver Denninger. “Fault-Tolerant Hybrid Quantum Software Systems”. In: *2022 IEEE International Conference on Quantum Software (QSW).* July 2022, pp. 52–57. DOI: `10.1109/QSW55613.2022.00023`.

[128] Max Scheerer and Ralf Reussner. “Reliability Analysis of Architectural Safeguards for AI-enabled Systems”. In: *2023 27th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE. 2023, pp. 61–70.

[129] Max Scheerer et al. “Towards classes of architectural dependability assurance for machine-learning-based systems”. In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2020, pp. 31–37.

[130] D. Sculley et al. “Hidden technical debt in Machine learning systems”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. 2015, pp. 2503–2511.

[131] Peter W Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.

[132] Peter W. Shor. “Scheme for Reducing Decoherence in Quantum Computer Memory”. In: *Physical Review A* 52.4 (Oct. 1995), R2493–R2496. DOI: 10.1103/PhysRevA.52.R2493.

[133] Stephen F. Siegel et al. “Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs”. In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA ’06. Association for Computing Machinery, July 2006, pp. 157–168. ISBN: 978-1-59593-263-1. DOI: 10.1145/1146238.1146256.

[134] Gagandeep Singh et al. “An abstract domain for certifying neural networks”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 41:1–41:30.

[135] Gagandeep Singh et al. “Fast and Effective Robustness Certification”. In: *Advances in Neural Information Processing Systems 31*. 2018, pp. 10825–10836.

[136] Joar Skalse et al. “Defining and characterizing reward gaming”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 9460–9471.

[137] Karthik Srinivasan et al. “Efficient quantum algorithm for solving travelling salesman problem: An IBM quantum experience”. In: *arXiv preprint arXiv:1805.10928* (2018).

[138] Susan Stepney et al. “Journeys in Non-Classical Computation I: A Grand Challenge for Computing Research”. In: *International Journal of Parallel, Emergent and Distributed Systems* (Mar. 2005). DOI: 10.1080/17445760500033291.

[139] Armands Strikis et al. “Learning-Based Quantum Error Mitigation”. In: *PRX Quantum* 2.4 (Nov. 2021), p. 040330. DOI: `10.1103/PRXQuantum.2.040330`.

[140] Marius Take et al. “Modeling the Integration of Machine Learning into Business Processes with BPMN”. In: *Proceedings of Eighth International Congress on Information and Communication Technology*. Springer Nature Singapore, 2024, pp. 943–957.

[141] Marius Take et al. “Software Design Patterns for AI-Systems.” In: *EMISA*. 2021, pp. 30–35.

[142] Swamit S. Tannu and Moinuddin Qureshi. “Ensemble of Diverse Mappings: Improving Reliability of Quantum Computers by Orchestrating Dissimilar Mistakes”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Association for Computing Machinery, Oct. 2019, pp. 253–265. ISBN: 978-1-4503-6938-1. DOI: `10.1145/3352460.3358257`.

[143] Philipp Tschandl, Cliff Rosendahl, and Harald Kittler. “The HAM 10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions”. In: *Scientific Data* 5 (2018).

[144] Miroslav Urbanek et al. “Mitigating Depolarizing Noise on Quantum Computers with Noise-Estimation Circuits”. In: *Physical Review Letters* 127.27 (Dec. 2021), p. 270502. DOI: `10.1103/PhysRevLett.127.270502`.

[145] Ewout van den Berg, Zlatko K. Minev, and Kristan Temme. “Model-Free Readout-Error Mitigation for Quantum Expectation Values”. In: *Physical Review A* 105.3 (Mar. 2022), p. 032620. DOI: `10.1103/PhysRevA.105.032620`.

[146] Lieven MK Vandersypen et al. “Experimental realization of Shor’s quantum factoring algorithm using nuclear magnetic resonance”. In: *Nature* 414.6866 (2001), pp. 883–887.

[147] Shiqi Wang et al. “Efficient Formal Safety Analysis of Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 31. 2018.

[148] Hui Xu et al. “NV-DNN: towards fault-tolerant DNN systems with N-version programming”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. 2019, pp. 44–47.

[149] Mingsheng Ying. "Floyd–Hoare Logic for Quantum Programs". In: *ACM Transactions on Programming Languages and Systems* 33.6 (Jan. 2012), 19:1–19:49. ISSN: 0164-0925. DOI: `10.1145/2049706.2049708`.

[150] Mingsheng Ying and Yuan Feng. "Model-Checking Quantum Systems". In: *National Science Review* 6.1 (Jan. 2019), pp. 28–31. ISSN: 2095-5138. DOI: `10.1093/nsr/nwy106`.

[151] Lei Yu. "A formal model of IEEE floating point arithmetic". In: *Archive of Formal Proofs* (2013), pp. 91–104.

[152] Huan Zhang et al. "Efficient Neural Network Robustness Certification with General Activation Functions". In: *Advances in Neural Information Processing Systems 31*. 2018, pp. 4944–4953.

[153] Xiyue Zhang, Benjie Wang, and Marta Kwiatkowska. "Provable Preimage Under-Approximation for Neural Networks". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernd Finkbeiner and Laura Kovács. 2024, pp. 3–23.

[154] Pengzhan Zhao et al. "Bugs4q: A benchmark of real bugs for quantum programs". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 1373–1376.

[155] Li Zhou et al. "CoqQ: Foundational Verification of Quantum Programs". In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023), pp. 833–865. ISSN: 2475-1421. DOI: `10.1145/3571222`.

[156] Dániel Zombori et al. "Fooling a Complete Neural Network Verifier". In: *International Conference on Learning Representations*. 2021.