Automated Verification Techniques for Quantum Computers

Marco John Lewis

Thesis submitted for the degree of
Doctor of Philosophy
Supervisors: Paolo Zuliani, Sadegh Soudjani

School of Computing
Newcastle University
Newcastle upon Tyne
United Kingdom

October 17, 2024

To all the teachers in my life, for pushing me that little bit further.

Acknowledgements

Firstly, I would like to thank my supervisors, Dr. Paolo Zuliani and Dr. Sadegh Soudjani, for their continual support throughout my course. During the last few years, they have taught me so much and have given me excellent advice in how to be an effective communicator, as well as commenting on countless drafts of my writings. I could not have asked for a better pair of mentors.

I would like to thank the friends and colleagues in the AMBER group, the lunch breaks and seminars allowed me to expand my thinking further. Additionally, I would like to thank Prof. Francesco Mezzadri and Dr. Miranda Mowbray for when I was at the University of Bristol. Without their recommendation, this thesis would not exist.

I send out my appreciation to the various shihans, senseis and martial arts instructors in my life. They have taught me to persevere in the face of adversity. Last but not least, I would like to thank my family and friends for their support and encouragement through the years.

Abstract

Quantum computing has become a field with high interest in both academia and industry due to recent advances in the development of quantum computers. Whilst these devices may provide speedups to solving certain problems in the future, there are still several issues to address. This thesis explores the automated verification of different aspects of quantum computers. Quantum computers are more complex than classical computers and so the verification techniques that are commonly used classically need to be adapted for the quantum domain. In this thesis, two techniques are adapted to handle the verification of quantum computers.

Firstly, `SilVer` is presented as an automated tool for checking quantum programs written in the high-level programming language Silq. `SilVer` converts Silq programs into a model that is based on quantum RAM (QRAM) devices, allowing for representation of both quantum and classical operations. User-defined specification are used to encode the desired behaviour of a program. This model and specification are automatically converted into proof obligations that are checked using a Satisfiability Modulo Theory (SMT) solver. Several case studies are provided with their setup and verification run times.

Secondly, the usage of barrier certificates as a verification technique for quantum computers is explored. Barrier certificates are a technique used in control theory to check if a dynamical system enters an unsafe region or not. Quantum computers are based on dynamical systems and so barrier certificates can be used to reason about their safety. Barrier certificates are adapted to handle complex variables and different quantum systems. Computational methods are provided for generating barrier certificates given a quantum system and its specification. Case studies are provided to explore the usage of barrier certificates for quantum computing.

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Introduction

Chapter 1

Introduction

Much work has gone into the development and creation of quantum computers in the last three decades. Their original conception was due to Benioff [19] and Manin [107, 108] in 1980, but brought to the forefront by Feynman [73] who discussed the use of the behaviours of quantum physics to perform simulations of quantum systems faster than on a classical computer. Interest in research into the development of quantum computers did not pick up until algorithms created by Deutsch and Jozsa [62], Grover [85], and Shor [139] further demonstrated the computational advantage that such devices could provide. Since then, quantum computing has become a major research field with interest from academia, industry, and government.

Similar to classical computers, quantum computers face numerous sources of error. There are three major sources of error for a quantum computer: the probabilistic nature of quantum algorithms, error in hardware and error in software. The first source of error (the probabilistic nature of quantum algorithms) is due to the nature of quantum physics, as most quantum algorithms only return a value with high probability. Due to the randomness of measuring quantum states, an algorithm that uses a quantum computer has a distribution of states that are measured, and thus a probability of returning the correct result. Some algorithms, such as the Bernstein-Vazirani algorithm [22], will always return the correct answer, *i.e.*, have the correct state with a probability of 1. However, it is more common that a quantum algorithm will return the correct result with a certain probability, *e.g.*, in the case of Grover's [85] and Shor's algorithm [139]. This error is mitigated by running a quantum algorithm multiple times to see which result is most likely, or by changing the quantum state to increase the likelihood of the correct result being returned.

The second source of error (error in hardware) occurs when the physical system and qubits themselves are interfered with [116]. This interference can occur through a variety of different ways. One is through noise of having the quantum system be open; as the quantum state evolves, it faces noise from outside the system and decoherence, which changes the quantum state. Another error is readout error, which occurs during measurement, where a quantum state may be measured incorrectly, *i.e.*, the "measured" quantum state differs from the state that the user reads. The current structure of quantum computers means two qubits need to be next to each other to be entangled. Gate/qubit connectivity is the error that occurs between two qubits, usually the noise that occurs when a two qubit operation is applied to the qubits, and can affect how entangled two qubits are. Additionally, moving the qubits next to each other can introduce errors into the system.

The final source of error (error in software) can occur at various stages [117,165]. Even before considering a quantum program, an error can be in the theory of an algorithm. The errors can occur when the programmer codes a quantum program using a quantum programming language. The program itself could be incorrect or the language itself could contain bugs. These bugs could be an error in the compilation process or in the syntax of the language (*e.g.*, allowing measurement to be controlled in some way). Additionally, errors can occur not only when compiling a program into a quantum circuit, but also when compiling a quantum circuit into another circuit that works on a specific quantum device. Further errors can occur when considering the actual software that controls the quantum computer, since quantum gates need to be applied precisely and quickly to avoid decoherence of a quantum state. Thus, the mapping of logical qubits in the program to physical qubits on the device needs to be checked carefully, as well as the signals that need to be sent to the quantum chip.

It is important that these errors are prevented at all levels of the quantum stack. Table 1.1 shows the quantum stack [77] with different sorts of behaviours that can be checked for errors.

Techniques need to be developed to prevent or mitigate the errors that occur in the quantum stack within both hardware and software. For example, error-correcting codes can be used to prevent some errors within hardware [131]. Investigation of techniques to mitigate error in hardware have already taken place [116]. Formal methods provide a way to verify the behaviour of both hardware and software components of quantum computers. It is only in the last 15 years that formal meth-

Table 1.1: The quantum computing system stack, based on Figure 3 in [77], with behaviour that can be verified depending on the stack level. The top half of the stack is where software errors are most likely and the bottom half of the stack is where hardware errors are most likely. Further note that hardware errors are architecture dependent.

The Quantum Stack	**Behaviour to Verify**
Quantum Algorithm	Algorithm correct
Programming Language	Program behaviour
Quantum Arithmetic/Gates | Runtime | Compiler	Conversion correct
Quantum Instruction Set	Quantum circuit verification
Quantum Execution | Quantum Error Correction	Mapping (Logical to Physical)
Quantum-Classical Interface	Signals to chip
Quantum Chip	Hamiltonians and controllers

ods have been applied to verify behaviours in quantum computing [14, 38, 94, 158].

Formal methods includes two broad categories: deductive verification and model checking. When applying formal methods, one of the major factors to consider is whether to use an automated technique or a manual one. These provide different advantages as an automated technique requires very little effort from the user but understanding where errors occur can be difficult, a user may be given a program trace but this does not necessarily show how to fix the error (especially with quantum programs). On the other hand, manual techniques provide a better understanding of the program or system but the user requires very technical knowledge and a lot of effort to show a a behaviour is followed. Model checking includes automated techniques but deductive verification contains a mixture of automated and manual techniques.

There exists a variety of these verification techniques applied to the verification of quantum computers and programs. Many of the tools currently available involve using manual techniques such as theorem provers [88, 104, 105, 166]. Other approaches have investigated using automated techniques such as Satisfiability Modulo Theory (SMT) solvers [18, 39], automata based verification [41] and abstract interpretation [162].

1.1 Thesis Structure

This thesis looks at the usage of two automated techniques to verify behaviours of different aspects of quantum computers. It is divided into four major parts, with

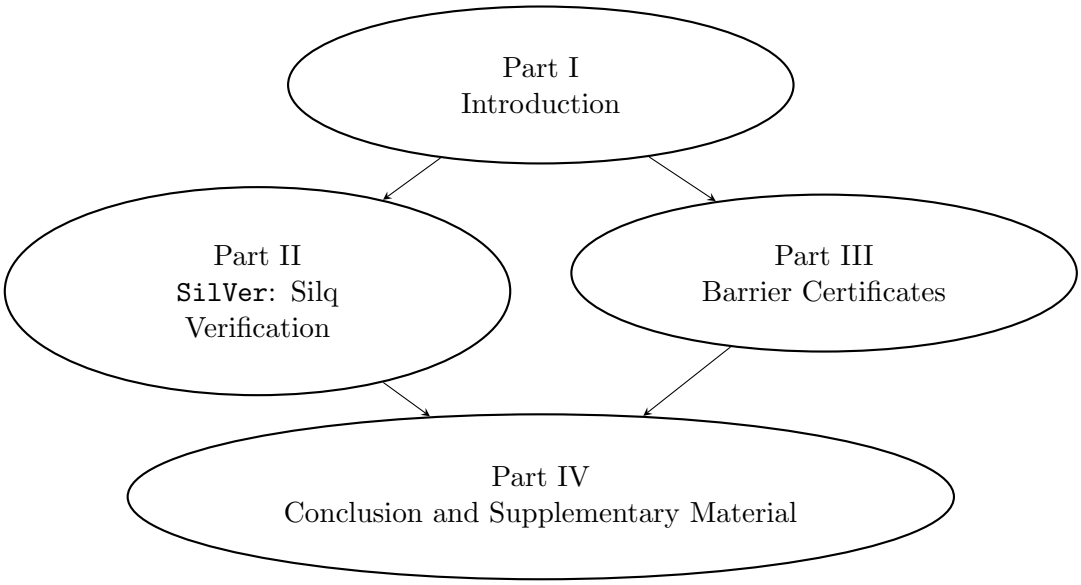


Figure 1.1: Thesis dependency graph

a schematic overview of the thesis structure given in Figure 1.1. Parts II and III depend on Part I, and Part IV is dependent on all other parts (although it can be read independently from other parts).

In all Chapters, the work presented in this thesis is based on various publications and manuscripts written in collaboration with my supervisors (Paolo Zuliani and Sadegh Soudjani). The writing contained within this thesis is my own writing with comments from my supervisors, and the various tools were developed by myself.

Part I provides an introduction to the research field of formal verification of quantum computers. Chapter 2 provides a basic introduction to quantum computing and formal verification (Section 2.1) and a look at the most well known developed tools used to verify quantum programs and circuits (Section 2.2). The contents of Chapter 2 provide a background to quantum computing, formal verification, and recent tools for verifying quantum computers and are based on a survey paper published in the ACM Transactions on Quantum Computing under the title *"Formal Verification of Quantum Programs: Theory, Tools, and Challenges"* [101]. Further excerpts and a more detailed background are provided in Appendix A.

In Part II, the first automated technique is discussed, where SMT solvers are used to verify programs written in Silq [25], a high-level quantum programming language. The developed tool, `SilVer`, is one of the first to use a quantum programming language and the software verification framework to allow fully automated verifica-

tion of programs. The contents of Chapter 3, which covers the details behind the implementation of `SilVer`, are part of an accepted manuscript to the 2024 IEEE International Conference on Quantum Software (QSW) titled *"Automated Verification of Silq Quantum Programs using SMT Solvers"*.

In Part III, the second automated technique analysed is a technique known as barrier certificates, which is applied to the gate and circuit level of the quantum stack. The extension of barrier certificates into the complex domain is shown and then the generation of barrier certificates is also extended. The contents of Chapter 4, which looks at applying barrier certificates to quantum systems and using a linear programming approach to generate barriers, is from work presented at the International Conference on Quantitative Evaluation of SysTems (QEST) 2023 under the title *"Verification of Quantum Systems using Barrier Certificates"* [102]. Chapter 5 contains an investigation into verifying quantum circuits and uses Hermitian Sum of Square (HSOS), a complex extension of Sum of Squares, to generate valid barriers. The contents of Chapter 5 are from an unpublished manuscript titled *"Verification of Quantum Circuits through Discrete-Time Barrier Certificates"*, which follows up on the previous work [102].

Finally, in Part IV, the results of the various Chapters are brought together. Additionally, the bibliography and various appendices can be found in this part.

1.2 List of Outputs

1.2.1 Thesis Related Outputs

Papers published

- Marco Lewis, Paolo Zuliani, and Sadegh Soudjani, Automated Verification of Silq Quantum Programs using SMT Solvers. *IEEE International Conference on Quantum Software (QSW)*, Shenzhen, China, 2024, pages 125-134.
- [101] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. Formal verification of quantum programs: Theory, tools, and challenges. *ACM Transactions on Quantum Computing*, 5(1), December 2023
- [102] Marco Lewis, Paolo Zuliani, and Sadegh Soudjani. Verification of quantum systems using barrier certificates. In Nils Jansen and Mirco Tribastone, editors, *Quantitative Evaluation of Systems*, pages 346–362, Cham, 2023. Springer Nature Switzerland.

Manuscripts under Submission

- Marco Lewis, Paolo Zuliani, and Sadegh Soudjani. Verification of Quantum Circuits through Discrete-Time Barrier Certificates. Submitted to *ACM Transactions on Quantum Computing*.

Software

- Discrete-time barrier certificates for quantum circuits (discussed in *"Verification of Quantum Circuits through Discrete-Time Barrier Certificates"*)
  GitHub: `https://github.com/marco-lewis/discrete-quantum-bc`
- `SilVer` Framework (discussed in *"Automated Verification of Silq Quantum Programs using SMT Solvers"*)
  GitHub: `https://github.com/marco-lewis/silver`
  Zenodo: `https://doi.org/10.5281/zenodo.8343751`
- Continuous-time barrier certificates for quantum systems (discussed in *"Verification of quantum systems using barrier certificates"*)
  GitHub: `https://github.com/marco-lewis/quantum-barrier-certificates`

Posters

- Verifying Silq Programs using SMT Solvers/Automatic Verification of Silq Programs using SMT Solvers. Presented at Quantum Information Processing (QIP) 2022, Quantum Computing Theory In Practice (QCTIP) 2022, Programming Languages and Quantum Computing (PLanQC) 2022.
- Automatic Verification of Quantum Systems using Barrier Certificates. Presented at PLanQC 2022.

1.2.2 Non-Thesis Related Outputs

Papers published

- Andrew Wright, Marco Lewis, Paolo Zuliani, and Sadegh Soudjani. T-Count Optimizing Genetic Algorithm for Quantum State Preparation. Accepted at *IEEE International Conference on Quantum Software (QSW)*, Shenzhen, China, 2024, pp. 58-68.

Manuscripts under Submission

- Viktorija Bezganovic, Marco Lewis, Paolo Zuliani, and Sadegh Soudjani. High-level quantum algorithm using Silq.

Chapter 2

Background

In this chapter, the fields of quantum computing and formal verification are introduced. Further related works are introduced on how formal verification has been used in quantum computing in the last decade. This gives the reader an introduction into the research topic and ideas covered in this thesis.

2.1 Quantum Computing and Formal Verification

This section introduces the standard notation used in quantum computing, and the field of formal verification. Whilst there are many techniques for formal verification, this section focuses on the two most popular ones: model checking and deductive verification.

2.1.1 Quantum Computing Notation

Nielsen and Chuang's volume [114] is the standard textbook for quantum computing and a full introduction can be found therein. This section will briefly cover some notation used throughout the thesis, however new notation is introduced where appropriate.

Throughout, the Dirac/bra-ket notation is used to describe quantum states and operations. Quantum states are written using kets, $|.\rangle$. The states $|0\rangle = [1,0]^\intercal$ and $|1\rangle = [0,1]^\intercal$ describe the computational basis states. In general, a quantum state is described as $|\phi\rangle = \sum_j \alpha_j |\phi_j\rangle$ where ϕ_j is typically a bitstring. The dual of a quantum state is denoted by a bra, $|\phi\rangle^\dagger = \langle\phi|$.

Unitary operations, denoted by U, are operations from quantum states to quantum states and their inverse is their adjoint, *i.e.*, $U^{-1} = U^\dagger = \overline{U}^\intercal$. The application of U to $|\phi\rangle$ is represented by $U|\phi\rangle$. Some common unitary operations include, for example, the Hadamard (H), Phase (Z), NOT (X), and Controlled-NOT ($CNOT$) gates:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \tag{2.1}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{2.2}$$

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \tag{2.3}$$

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}; \tag{2.4}$$

which behave as

$$\begin{aligned} H|j\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + (-1)^j|1\rangle), \\ Z|j\rangle &= (-1)^j|j\rangle, \\ X|j\rangle &= |\neg j\rangle, \\ CNOT|j,k\rangle &= |j, j \oplus k\rangle, \end{aligned}$$

where $j, k \in \{0, 1\}$ and $\oplus$ is the XOR Boolean operation.

Unitary operations can be combined using either matrix multiplication (applying operations one after another) or using the Kronecker product (for applying operations to multiple qubits). Operations from either product are still unitary and can be applied to quantum states. There is a notion of completeness for quantum operations known as universality [61]. Essentially, a set of unitary operations, $\mathcal{U} = \{U_1, \ldots, U_k\}$, is a universal set if for any unitary operation, U, can be represented by operations from $\mathcal{U}$, using the matrix and Kronecker product, up to arbitrary accuracy.

The other type of operation that can occur is quantum measurement, which

causes the quantum state to "collapse" into a basis quantum state. Measurement is encapsulated by a collection of measurement operators $\{M_m\}$, where m denotes the measurement outcome and $\sum_m M_m^\dagger M_m = I$. For a quantum state $|\phi\rangle$, the probability of measuring m is given by $P(m) = \langle\phi| M_m^\dagger M_m |\phi\rangle$ and the collapsed state after measurement is $\frac{M_m|\phi\rangle}{\sqrt{P(m)}}$.

One important concept in quantum computing, which is discussed in this section, is the principle of no-cloning. The no-cloning theorem [156] states that there exists no unitary operation that can copy a quantum state, *i.e.*, there is no unitary U such that for all quantum state $|\phi\rangle$, $U|\phi\rangle|0\rangle = |\phi\rangle|\phi\rangle$. If a quantum state is to be copied, then the original quantum state must collapse in some way.

The density matrix formalism is also discussed and used instead to represent quantum states. Hermitian operators, denoted by H, are operators that are self-adjoint, so $H = H^\dagger$. Further, Hermitian operators have real eigenvalues. In the density matrix setting, states are described by Hermitian matrices and often are written as $\rho = \sum_j \alpha_j |\phi_j\rangle\langle\phi_j|$. This representation is used in this chapter (and related appendix) only, notably in Section 2.2.2 and Appendix A.2.2.

2.1.2 Model Checking and Verification

Verifying software with model checking involves modelling the software through a formal representation. Then the desired behaviour is specified through an appropriate logic. Once these two components are created, model checking algorithms can be used to check whether the model follows the specified behaviour. Typically, models are created using Kripke structures (a type of transition system) and behaviours are specified using a temporal logic, such as Computation Tree Logic (CTL). More details can be found in [47] and a brief study of Kripke structures and CTL is given in Appendix A.1.2.

A reason for choosing model checking as a verification technique is that it efficiently searches over all possible states of a Kripke structure in a completely automated way. However, the main issue of model checking is the state explosion problem [48]. Due to the design of the Kripke structure, an increase in the size of the system can increase the number of states in the structure exponentially. This can make it very difficult for the system to be verified quickly. Over the last few decades a number of methods have been developed to address the state explosion problem. For example, bounded model checking (BMC) [26] only considers finite computation

trees, effectively meaning that the system is only checked up to a certain point in its temporal evolution.

Another technique is CounterExample-Guided Abstraction Refinement (CEGAR) [44, 46], which starts by creating an abstract model that *over-approximates* the behaviour of the original (concrete) model. The abstract system is then model checked against a given universally path-quantified temporal logic property. If the property is satisfied then the system follows the behaviour, since the abstract model encompasses all the possible behaviours of the concrete model. If instead a counterexample to the property is returned, then this counterexample is compared in the concrete model: if it is an actual counterexample, then the model checking fails since this is a "real" bug of the concrete model. Otherwise, the counterexample is spurious and the abstraction is refined so that the counterexample no longer fails inside the abstraction. The newly-obtained abstraction is then model checked again – the process is repeated until the property is either verified or a concrete counterexample is found [44].

2.1.3 Deductive Verification

For a full review of deductive verification the reader is referred to, *e.g.*, [86]. Unlike model checking, which explores the possible states of software, deductive verification formally verifies programs through logical inference. Further model checking can ensure certain properties about software automatically, whereas deductive verification can be used to verify complex properties about programs in a way that is understandable by humans. For example, a subroutine can be shown to always return a certain result no matter the input by using deductive verification. The Floyd-Hoare logic [89] is studied below as an example.

A program is given some preconditions, the assumptions held at the start of a program; and postconditions, which are goals or requirements to meet after the program has run with the given preconditions. A program is valid if the postconditions can be inferred from the given preconditions using inference rules. This is often written in the form of a Hoare triple denoted by $\{P\}S\{Q\}$, where P is a precondition, Q is a postcondition and S is a program statement. A Hoare triple is considered valid if a sequence of inference rules can be used to derive it. The basic inference rules are given in Equation (2.5).

The proof of a program or system can be created from the Hoare triple and the

$$
\begin{array}{ll}
\dfrac{}{\{P\}\ \textbf{skip}\ \{P\}} & \text{(Skip)} \\[2ex]
\dfrac{}{\{P[a/x]\}x := a\{P\}} & \text{(Assign)} \\[2ex]
\dfrac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}} & \text{(Composition)} \\[2ex]
\dfrac{\{b \wedge P\}S_1\{Q\}, \{\neg b \wedge P\}S_2\{Q\}}{\{P\}\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2\{Q\}} & \text{(Conditional)} \\[2ex]
\dfrac{\{P \wedge b\}S\{P\}}{\{P\}\textbf{while } b \textbf{ do } S\{\neg b \wedge P\}} & \text{(While)} \\[2ex]
\dfrac{P_2 \to P_1, \{P_2\}S\{Q_2\}, Q_2 \to Q_1}{\{P_1\}S\{Q_1\}} & \text{(Consequence)}
\end{array}
\tag{2.5}
$$

use of inference rules. These proofs are converted into proof obligations, which are mathematical formulae that are checked using one of a variety of software tools. The most common tools for software verification are theorem provers and Satisfiability Modulo Theory (SMT) solvers. Theorem provers [153] allow programmers to write the obligations that are to be met in a completely formal environment. Then lemmas and theorems about these obligations can be derived from definitions created within the tool. Normally, the process of proving an obligation is interactive and so the programmer will write the proof with assistance from the tool. Theorem provers are used in a number of the tools discussed in Section 2.2.

In comparison, SMT solvers [59] convert obligations into logical formulae over a theory, such as, *e.g.*, the natural numbers, rationals and bit vectors. Alongside a statement the user wishes to assert, the solver can automatically check if the formulae are valid. If not, the solver can provide a counterexample. In particular, at least one tool, `QBricks` [39], has made use of SMT solvers for verifying quantum programs; which is discussed in Section 2.2.4.

Deductive verification still suffers from scalability issues, albeit different from the state explosion problem model checking faces [48]. The scalability problem faced in deductive verification is that as the system to verify grows and becomes more complex, it becomes harder to verify the increasingly complex behaviours (either

manually or automatically). Unlike model checking, deductive verification requires programmers to have a deeper understanding of why the obligations are correct. This is both an advantage and a setback, since it can take a long time to prove complex obligations that could be solved automatically using model checking. However, it is possible to create human-readable proofs as to why a program is correct.

For examples of some of the most used theorem provers, the reader is referred to [153]; an introduction to SMT solvers is given in [59] and a deeper study is given in [16].

2.2 Related Works

Here various tools that can verify quantum programs and circuits are discussed, highlighting their trade-offs, where they excel and their limitations. At the end of this section, other quantum verification tools, whose focus is not on the formal verification of programs, are briefly discussed.

2.2.1 SQIR (and QWIRE)

The languages QWIRE [120] and SQIR (Small Quantum Intermediate Representation) [88] are domain specific languages built in the Coq interactive theorem prover [23]. QWIRE was one of the first quantum programming languages to be released with verifiable programs, while SQIR is a more recent language that has various improvements over QWIRE. These improvements include shorter code, better handling of ill-typed programs and the separation of semantics for unitary and non-unitary (*e.g.*, measurement) operations.

SQIR uses the various functionalities of Coq to act as a proof assistant for writing proofs about quantum programs. To verify a program, firstly the program is defined using a dedicated type, with qubit indices being referred to by a numerical value (*e.g.*, `X 3;` refers to applying the X operation to the qubit in index 3). Programs can be defined by one of two types: **base_ucom**, which only contains unitary operations; or **com**, which includes the unitary operations and measurement (no other non-unitary operations are possible). Classical subroutines cannot be performed within a SQIR program, but it is possible to generate circuits using classical parameters.

Theorems can then be conjectured about the program. Unitary SQIR uses state vectors as part of the semantics, whereas full SQIR extends this to density matrices.

```
Definition Program : base_ucom 2 := H 0; X 0; CNOT 0 1; X 0.

Local Open Scope R_scope.

Definition xor_state : Matrix (2^2) (1^2) : =
1/\sqrt{2} .* (|0,1> .+ |1, 0>).

Theorem Prog_Correct: uc_eval Program x (|0,0>) = xor_state.
Proof.
  intros
  unfold Program; simpl.
  unfold xor_state; simpl.
  autorewrite with eval_db; simpl; try lia.
  solve_matrix.
Qed.
```

Figure 2.1: A simple program and proof written in SQIR, which transforms the state $|00\rangle$ into $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$.

The user proves the theorem with the assistance of the Coq framework. This format separates programs from how they are specified or proved. No-cloning is satisfied through the use of unitary gates and measurement being the only operations allowed in the language.

An example program and proof can be seen in Figure 2.1. Whilst the example given is quite trivial, it is possible to make generalisations about what can be proven. The creators of SQIR have already proven some properties about Grover's algorithm, notably they proved that the algorithm measures a marked element with probability $\sin^2((2T+1)\arcsin\left(\sqrt{k/2^n}\right))$ after T steps (where n is the number of qubits and k is the number of marked elements).

Coq requires interaction from the user for theorem proving, but there is some automation when reducing matrices as seen in the *solve_matrix* function given in Figure 2.1 (*solve_matrix* is a tactic that attempts to simplify a matrix equality using a variety of sub-tactics). Further, SQIR benefits from a number of gates already implemented and verified within the language. With its capabilities, SQIR has been able to prove properties about most textbook algorithms, including Grover's, the Quantum Phase Estimation and Shor's algorithms.

SQIR is designed such that circuits use a predefined number of qubits. Should ancillary qubits be required in the circuit, the user must define these at the start,

theorem *prog-partial-correct*:
$\models_p$
$\{proj\text{-}10\}$
cnot-circ
$\{proj\text{-}11\}$
using
prog-partial-deduct
well-com-cnot qp-pre qp-post
hoare-partial-sound **by** *auto*

Figure 2.2: Documentation of a simple program that implements a single CNOT gate and checks that the state $|10\rangle$ is transformed to $|11\rangle$. The state $|10\rangle$ is represented by the predicate $proj\text{-}10 = |10\rangle\langle 10|$ and a similar predicate represents $|11\rangle$.

rather than introducing them when required. This design choice reflects quantum hardware that cannot perform on-the-fly ancillary allocation but could be restrictive should allocation be possible after beginning the computation.

Overall, SQIR is very useful for showing that quantum programs are correct. SQIR is capable of showing the correctness of programs for an arbitrary number of qubits and has verified several example algorithms already. However, being a theorem prover based tool means that SQIR requires expertise in Coq, which developers of quantum programs do not necessarily have. This is where an automated tool could be more beneficial. Additionally SQIR cannot initialise additional qubits after the start of the program.

2.2.2 QHLProver

The Quantum Hoare Logic [158] is a quantum extension of Hoare logic formally described in Appendix A.2.2. It has been implemented into the Isabelle/HOL proof tool [104]. This implementation uses a slightly simpler version of the quantum-while language (introduced in Appendix A.2.2), omitting the initialisation term $q := 0$.

A full documentation of the implementation is available in the Isabelle Archive of Formal Proofs (AFP) [105]. This implementation is referred to as QHLProver. By using Isabelle, verification is not fully automated, but some automation is used to make manual proving easier when handling complex matrices, which are used to define gates and oracles. An example can be seen in Figure 2.2, showing how a program is validated within the Isabelle framework.

QHLProver is similar to SQIR in a few ways. The way programs are proved is akin to that of Coq, where programs, states and density matrices are defined

and then theorems can be conjectured about them. Similar to SQIR, programs in QHLProver require a predefined number of qubits and so ancillary qubits need to be defined at the start of computation.

Programs are written using a specific type (com), which encodes the terms of quantum-while, and the predicates used within the pre- and post-conditions are density matrices. These all need to be verified for a specific triple to be correct as seen in Figure 2.2.

Unlike SQIR, which defines programs as quantum circuits, QHLProver uses an extension of the quantum-while language to define programs. Whilst there is no theory written for no-cloning within the documentation, no-cloning is adhered to due to the simple grammar that only allows for unitary operations and measurement.

As mentioned previously, there is no classical functionality in the quantum-while language and this is reflected within the Isabelle implementation. Classical parameters are used to extend gates to operate on a subset of qubits. Constructing oracles for different algorithms is done by writing functions defined in terms of natural numbers to Booleans, which are then used to create a complex matrix from the matrix indexes.

QHLProver suffers from a lack of implemented logic gates with verified properties. Within QHLProver, a user is required to prove properties of a logic gate to use it within the implemented QHL system (*e.g.*, one needs to prove a gate is unitary). Currently, the language implements the Pauli gates, Hadamard gates and some gates used within Grover's algorithm. Out of the common quantum gates, only the Hadamard gate is verified. This makes it difficult to create new programs as the user must verify properties of simple gates (such as CNOT).

Similarly to SQIR, QHLProver benefits from using a theorem prover to verify arbitrary size programs but, unlike SQIR, is missing advanced examples. Although, QHLProver has an already developed theory behind it, Quantum Hoare Logic [158], that is being extended with additional features [71, 103]. Once again though, it requires the developer to have expertise in using theorem provers to prove properties about quantum programs, where an automated tool will be easier and quicker for a developer to use. Additionally, while QHL reasons about the quantum-while language, this is not a language used in practice or that features high-level features.

CoqQ A recent tool called CoqQ [166] builds on the work from QHLProver. It shares a number of similarities, such as using QHL and the quantum-while language.

definition *circ*:: *complex Matrix.mat* **where**
circ ≡ *CNOT* ∗ ((*X-on-ctrl*) ∗ (|*zero*⟩ ⨂ |*zero*⟩))

lemma *circ-result* [*simp*]:
shows *circ* = ψ_{11}
using *circ-def* ψ_{00}-*is-zero-zero* ψ_{00}-*to*-ψ_{10} ψ_{10}-*to*-ψ_{11}
by *simp*

Figure 2.3: Documentation of a simple CNOT program written in IMD, similar to that of QHLProver. The circuit performs a NOT-gate on the first qubit and then the CNOT gate with the first qubit as control. The lemma *circ-result* uses the circuit definition and how the state evolves through each transition to show the resulting state. The full Isabelle theory file for this example is available in this repository: `https://github.com/marco-lewis/IMD_CNOT`.

As in the name, CoqQ uses the Coq theorem prover instead of Isabelle/HOL.

CoqQ enhances the semantics of QHL by using improved inference rules and allowing dynamic initialisation of qubits. This allows the framework to verify non-textbook algorithms such as algorithms for solving the hidden subgroup problem [114] and the hidden linear function problem [33]; and the HHL algorithm (see Appendix A.5.1).

2.2.3 Isabelle Marries Dirac (IMD)

Another work [30] provides another instance of verified quantum computing using the Isabelle theorem prover. Unlike QHLProver's use of the quantum-while language and Hoare logic, Isabelle Marries Dirac (IMD) uses the standard matrix formalisation approach of quantum computing to prove properties about algorithms and protocols. Because of this, IMD is closer to a verifiable mathematical library, rather than a verifiable programming language. An example can be seen in Figure 2.3.

Firstly, matrices are defined to have a fixed size, whether through a variable n or a value. Ancillaries are needed to be defined by the user and taken into account when defining matrices. Fortunately, programs in IMD are defined using the dot and Kronecker product of matrices. This makes it easier to add in ancillary definitions.

Since IMD is a mathematical library, classical functionality is possible so long as proofs are developed in the Isabelle theorem prover. Oracles are constructed in a similar way to QHLProver, where the matrix value at an index is determined by a function with indexes as input. Various properties about measurement are implemented within the library such as the probability of a given outcome. It is possible

to prove properties not just about standard algorithms (*e.g.*, the Deutsch-Jozsa algorithm) but also different quantum information theoretic results such as quantum teleportation [21]. One result to mention in particular is that IMD explicitly proves the no-cloning theorem within Isabelle. The other tools within this section do not provide such a proof in the framework they use, but the definitions of states and operations that the tools use allow for no-cloning to be followed. More results are discussed in [30].

Further, Echenim and Mhalla [66] have used IMD and density matrices from QHLProver to prove properties about projective measurements (measuring qubits in a basis different from the computational basis $\{|0\rangle, |1\rangle\}$). They continued to develop more information theoretic results by proving the CHSH inequality [49]. As can be seen, the featured libraries have more proofs relating to mathematical concepts of quantum computing that cannot be implemented in a programming language. The documentation for IMD and the extension [66] are available in the Archive of Formal Proofs [31, 65].

Once again, IMD uses a theorem prover, which gives the same benefits and drawbacks as SQIR and QHLProver. Unlike other works, IMD takes a mathematical focus on quantum computing, this being beneficial since quantum properties can be investigated, *e.g.*, the CHSH inequality. However, taking this mathematical perspective of quantum computing makes it harder to integrate with actual programs to be verified.

2.2.4 `QBricks`

`QBricks` [39] is a circuit-based verifiable quantum programming language built in the Why3 framework [74]. The language is purposefully built so that the program syntax is separated from the specification to be proved about the program. Programs are written using `QBricks-DSL`, which is a domain specific language, and the specifications are written in `QBricks-SPEC`. Figure 2.4 gives the specification and definition for the oracle used in the Deutsch-Jozsa algorithm.[1]

The verification process of programs uses the ideas of the weakest precondition, path sums and quantum Hoare logic to generate proof obligations, building on previous works to suit the requirements of the language. Programs written in `QBricks-DSL` are converted into a path sum representation, which can then be used

[1] At the time of writing, a tutorial article is being written by the team.

```
val function deutsch_oracle (f: bitvec -> int)(n:int): circuit
  requires{1<=n}
  requires{(not (constant_bin f n)) -> balanced_bin f n}
  ensures{width result = n+1}
  ensures{forall x: bitvec. forall  y: matrix complex.
    is_a_ket_l y 1 ->
        path_sem result (kronecker (bv_to_ket x) y) =
            kronecker (bv_to_ket x)  (xor_qbits (ket 1 (f x)) y)
    }
```

Figure 2.4: The definition of a Deutsch-Jozsa oracle within the `QBricks` language [39]. The *require* statements note the preconditions of the oracle (there is at least 1 qubit and if f is not constant then it is balanced). The first *ensures* statement maintains the width of the oracle from input to output. The second *ensures* statement gives the usual definition of a quantum oracle ($O|x\rangle|y\rangle = |x\rangle|f(x) \oplus y\rangle$) from necessary preconditions. This function can be used later to prove properties about the Deutsch-Jozsa algorithm. Full documentation is available at `https://github.com/qbricks/qbricks.github.io`.

in the specification of their program in `QBricks-SPEC` to verify various properties. Several Hybrid Quantum Hoare Logic (HQHL) rules are given and can be used to generate the proof obligations from the specification and program written in path sums. These obligations are then proved using automatic SMT solvers such as Alt-Ergo [52] and Z3 [58]. `QBricks` still requires some interactivity from the user when writing the specification, but as mentioned these are mostly proved automatically.

The language features a vast array of functionalities, including the capability of introducing ancillary qubits in the middle of code, unlike SQIR and QHLProver. The framework adheres to the no-cloning theorem as it only allows certain unitary operations within its DSL.

`QBricks` still has some limitations though. Currently, there are no built-in capabilities to measure qubits within `QBricks-DSL`. Further, `QBricks` is not designed to interact with classical data, but classical parameters can be used in the generation of circuits. Despite these limitations, so far `QBricks` has been able to verify properties for the Phase Estimation and Shor's algorithms, which are the most complex algorithms verified by languages so far.

Overall, `QBricks` avoids the manual drawbacks that theorem prover based approaches suffer from by making use of the Why3 framework to allow for usage of SMT solvers for some automation. However, the tool is not fully automated and

still requires some interaction from the user to prove properties. Additionally, whilst `QBricks-DSL` is a good language for representing quantum circuits, the language is specific to `QBricks`. It does not represent quantum programs and cannot be used to simulate or run the quantum circuits it represents.

2.2.5 Further Related Works

This section introduces other notable works on using formal verification to verify quantum programs.

Quantum Model Checking The Quantum Program/Protocol Model Checker (QPMC) [70] is a tool developed to check a quantum extension of probabilistic CTL, known as qCTL, against programs that are modelled by quantum Markov chains. This allows programmers to use model checking techniques against quantum programs. However, QPMC is limited in that one needs to write their program as a Markov chain for it to be checked. This can be circumvented by a tool known as Entangλe [9, 10], which allows quantum programmers to convert Quipper [83], a functional quantum programming language, programs into Markov chains that can then be checked against qCTL properties.

`symQV` Making use of SMT solvers, `symQV` [18] is a tool that automatically checks the quantum state after going through a quantum circuit. Notably, `symQV` checks the quantum state only and has no functionality with quantum circuits that include measurement. However, `symQV` gets some speedups over a default SMT solver and default Bloch sphere representation by making use of dReal (a δ-satisfiability SMT solver) [78] and using an over-approximation of the Bloch sphere to represent quantum states [11, 28].

Feynman - Path Sums Using the path sum technique (details in Appendix A.2.4), a Haskell library known as Feynman was produced to perform simulation, verification and equivalence checking of quantum circuits [6, 7]. Equivalence checking involves checking if two circuits (*e.g.*, a circuit and its compilation to a specific quantum computer) are functionally equivalent, even if they use different gates.

Equivalence Checking using Binary Decision Diagrams (BDDs) Another form of equivalence checking has been developed through the use of BDDs and exten-

sions. There are two notable tools that take this direction. The first approach [34, 154] makes use of an extension of binary decision diagrams called quantum multiple-valued decision diagrams (QMDDs) [115]. The second approach SliQEC [40, 151] uses a bit-slicing technique to represent complex numbers efficiently in standard BDDs [147].

AutoQ Another quantum circuit verifier is AutoQ [41], which makes use of tree automata and the complex number representation introduced in [147] to verify properties of quantum circuits such as Grover's algorithm.

PyZX This tool is a module in the Python language that implements the ZX-calculus [95], a graphical language for reasoning about quantum circuits. Still currently under development, PyZX is able to convert back and forth between circuits and ZX graphs. This allows circuits to be simplified and optimised using the rules of the ZX-calculus. PyZX is not designed for reasoning about programs, but is useful for checking the equivalence of circuits.

CertiQ CertiQ is a verification framework developed for Qiskit [129] that verifies if a compiled circuit is the same as the circuit that is programmed by the user [138]. Interestingly, it makes use of Z3 and other SMT solvers to perform this verification automatically, requiring the user to only input a few lines of specification. While this uses automatic verification, again it should be highlighted that this is used for verifying circuit equivalence, similarly to PyZX, and not for reasoning about programs.

QSharpCheck This tool extends Q# with a means of testing programs [90]. Users can initialise qubits, notably their phase, and a number of different postconditions to be met by the resulting qubits. With this, the user can then define parameters that are used to run the tests. When run, test cases are randomly generated, executed on the program and checked they meet the postconditions given. This makes it very easy to quickly test a few properties of a program. However, this tool is designed for testing purposes and not for formal verification of programs. Despite this, it may find use for simpler programs that do not need to be verified extensively[2]. There have

[2]In early 2021, Microsoft added some testing and debugging functionality to Q#: `https://docs.microsoft.com/en-us/azure/quantum/user-guide/testing-debugging?tabs=tabid-vs2019`

been further works looking into testing and debugging of quantum programs [123, 146, 150].

Quantum Process Calculi Whilst the tools discussed focus on verifying programs and circuits, there are few tools available that verify properties of quantum processes or communication protocols [57, 68, 79]. Normally, the properties verified are related to bisimulation, showing that two processes or protocols are equivalent through some relation [99, 160]. This equivalence is shown by using a calculus to describe the specification, translating it to a transition system, and then comparing it against the process (described through a transition system). A common calculus for quantum processes is qCCS [68], a quantum extension of CCS [112]. The tool from [127] looks at verifying various quantum communication protocols for ground bisimulation using qCCS, but most studies focus on the theory rather than implementation of the bisimulation verification technique. Note that these works would require programs to be translated into the suitable transition system to be useful.

2.2.6 Comparison to Thesis Contributions

With the introduction of all the different tools available for verifying quantum computers, this Section discusses the comparison between them and the contributions of the thesis.

Tools such as SQIR, QHLProver, and Isabelle Marries Dirac (IMD) face the same benefits and issues that theorem provers face (understanding of proofs, person-hours to prove properties, ...). Beyond that each tools faces different issues, some of which have been discussed. For example, SQIR and QHLProver do not allow qubit allocation during a program, and IMD proves mathematical properties about quantum computing (rather than proving properties about software). `QBricks` whilst being very adaptable by making use of different theorem provers and SMT solvers, is not fully automated. Some of the theorems and lemmas require the user to prove that they are correct. The main downside to all of these approaches is that a developer for quantum software will not necessarily have the expertise to use theorem provers.

`SilVer`, discussed in Part II, takes a fully automated approach to verifying quantum programs, which is more useful for developers. Previous automated tools focus on verifying quantum circuits, whereas previous theorem prover-based approaches can verify quantum programs at the consequence of having to manually prove them.

Further, unlike many of the other tools mentioned, `SilVer` works on Silq [25], a high-level quantum programming language.

The work developed in Part III on barrier certificates is unique compared to the previous tools. All the tools discussed focus on the software levels of the quantum stack, whereas the barrier certificate approach focuses on the hardware levels. Whilst both the barrier certificate technique and many previous tools share a similarity in verifying quantum circuits, the barrier certificate approach developed in this thesis also investigates at a lower level and considers the verification of the quantum systems that work under-the-hood of quantum gates. This approach is therefore novel in that it verifies at the lowest level of the quantum stack, which has not been considered before. Additionally, the methods developed in Chapter 5 can be easily adapted to other types of quantum systems by extending the appropriate type of barrier certificate.

Comparison on Intermediate Representations

In Part II (specifically Section 3.4), an intermediate representation is introduced to represent Silq [25] programs. There are other intermediate representations available that have been used for verification. The Small Quantum Intermediate Representation (SQIR) [88] is used both for program correctness and optimisation purposes. Giallar [145], used to to verify the Qiskit compiler; and the quantum program model for `symQV` [18], for verifying aspects of quantum algorithms, feature a symbolic representation for quantum circuits. `QBricks` [39] includes its own domain specific language, `QBricks-DSL`, that is used to represent programs, but this is a programming language rather than a representation of programs.

The QRAM program model, the representation introduced in Section 3.4, acts as a model for a QRAM style quantum computer [96], where a classical computer sends instructions to a quantum chip to execute. Most other models, apart from `QBricks`, focus purely on the representation of quantum circuits rather than instructions for QRAM devices, which are more complex and allow for classical operations to occur.

In comparison to other representations, the controls that affect a system are separated from the operations to be performed. This allows the representation to represent not only have quantum controls, but classical controls as well; meaning only uncontrolled unitary operations are considered in the quantum instructions.

Whilst QASM/OpenQASM [55] bears some similarities to the QRAM program model introduced, it is not suitable for representing Silq programs. The reason for

this is because Silq can have can have very arbitrary controlled operations through its conditional statements that use quantum variables. For instance, one can write **if** $x >= 6$ **then** $y := X(y)$ in Silq to refer to a controlled operation on any state in the quantum register `x` with a value greater than 6. This has two components: a controlled operation on a register (x), and an operation controlled on multiple states ($|6\rangle, |7\rangle, \dots$). The QRAM program model can handle these features through the list of CTRL instructions. In OpenQASM, controlled operations can only be done at the qubit level. For a quantum register to act as a controller, a function needs to be defined to handle this. Further, an operation being controlled on multiple states would also need to be defined in a function where the explicit states that meet the operation must be given. This low-level handling of controlled operations makes OpenQASM unsuitable to model conditional statements and, therefore, Silq programs.

The symbolic QRAM program model presented has no defined semantics as it is meant to represent the operations that occur for a QRAM-based device. The benefit of this approach is that one can translate a quantum programming language into a general representation that can then be translated into different formats for different purposes of verification. Whilst this modularity is achievable with other models, they focus on quantum circuits rather than quantum programs. Additionally, building the infrastructure to translate programs into an intermediate representation and then into the appropriate verification format is also important. The QRAM program model is implemented in `SilVer` and so one can either change the programming language or the verification technique used, as long as the user implements the necessary translations from language to representation or from representation to verification technique.

2.3 Conclusion

With the basic concepts in both fields introduced and an exploration of related works, the reader should now be familiar with the kinds of problems that are explored in this thesis (*i.e.*, specifying and verifying behaviour in quantum computers). The following parts explore different automated approaches to solving this problem.

Part II

SilVer: Silq Verification

Chapter 3

SilVer: Automated Verification of Silq Programs

Having looked at a basic introduction to quantum computing, formal verification, and related works in Part I, the first automated technique is investigated here. In this Part, a standard software verification framework using SMT solvers, described in Section 2.1.3, is applied to the verification of a quantum programs, one of the higher levels of the quantum stack.

3.1 Introduction

Writing quantum programs from algorithms is hard and ensuring their correctness is even tougher. Several quantum programming languages have been released to program current or future quantum computers. Some are libraries of classical languages, whereas others are new languages specifically designed for writing quantum programs. Such languages include Cirq [43], Q# [143], Qiskit [129] and Quipper [83]. Silq [25] stands out as a higher-level programming language in comparison to others, along with a typing system and semantics (some of the other languages listed have typing or semantics developed later [83, 140]). There has been some effort put into formally specifying languages that have no specification currently. An example of this is with Q# being formalised in [140]. Whilst some of these languages may have testing functionality, most of them cannot be formally verified yet. This makes implementations of quantum algorithms at risk of being incorrect. As quantum computers and programming languages will become larger and more complex, there

will be a need for quantum programs to be verified.

Several approaches have been taken to formally verify quantum programs. Tools such as SQIR [88], QHLProver [104] and CoqQ [166] make use of theorem provers to reason about the programs they are verifying. Whilst theorem provers are powerful tools, they require a user to spend many person-hours proving that their programs are correct. Thus, it is important to consider automatic techniques that offload to a computer the burden of proving a program correct to a computer.

One way of making the proving of programs automated is to use model checking techniques. Tools such as QPMC [70] and Entangλe [10] allow users to convert programs into models (specifically quantum Markov chains) that are then checked against a specification written as a temporal logic formula. Another automated approach is taken by `QBricks` [39], which makes use of the Why3 framework to prove properties about programs or provide a counterexample to the property specified. However, `QBricks` requires the user to learn a domain-specific language and to write their program within the Why3 framework. Giallar [145] is another automated tool that uses solvers of satisfiability modulo theories (SMT) to perform verification on the Qiskit compiler (rather than programs).

In this Chapter, `SilVer`, an SMT-based verification tool for reasoning about programs written in the Silq quantum programming language, is presented. Silq programs are transformed from their syntax tree into a symbolic structure, which models a quantum RAM (QRAM)-style processor [96] that allows for both quantum and classical processes with a separated control channel. The intermediary representation, which is referred to as the *QRAM program model*, is unique as it separates on quantum and classical memories, allowing the instructions affecting the memory to be specified precisely. Additionally, the model keeps track of controls affecting the program, which encapsulate classical conditional statements and quantum controlled operations. `SilVer` performs this conversion automatically and further converts the generated model into proof obligations.

Additionally, a specification language, `SilSpeq`, is provided for reasoning about programs written in Silq. `SilSpeq` allows users to make pre-conditions about a programs inputs before it is run and post-conditions for expected results. Importantly, pre-conditions allow for the specification of oracles for oracle-based algorithms. A unique feature of `SilSpeq` is that the user can specify properties of a measurement of a quantum state that should be obeyed using measurement *flags*. This allows a user to easily specify what properties of the measurement should be met, *e.g.*, the

outcome of measuring a qubit in the $|0\rangle$ state should be at least 60%. It should be noted that conditions written in `SilSpeq` are purely classical, the only interaction with quantum pre-conditions is through *flags* and measured variables.

In this Chapter, `SilVer` and `SilSpeq` are presented, the QRAM program model that is used to represent quantum programs, how the model is converted and verified using an SMT solver, notably Z3 [58]. Further, case studies are provided on programs written in Silq and verified using `SilVer`.

3.2 Preliminaries

An introduction to Silq is given in this section. The required notation for quantum computing that is used (standard Dirac bra-ket notation) is provided in Section 2.1.1.

3.2.1 Silq

Silq [25] is a high-level, imperative quantum programming language that features safe, automatic uncomputation of qubits. In contrast, Qiskit [129] and Cirq [43] are instead modules of a classical programming language. Further, Silq features a formally defined specification; although recent work has given Q# a formal specification [140].

The Silq-Hybrid Fragment

In [25], the authors describe a fragment of Silq (Silq-Core) for the purposes of showing its properties. Since Silq is a vast language, the programs to verified are restricted to a fragment called Silq-Hybrid. The idea is to take a fragment of Silq that is expressive enough to handle several textbook quantum algorithms while capturing the core ideas required for verification of a program. The core algorithms targetted are those that make use of an oracle, *e.g.*, the Deutsch-Jozsa algorithm.

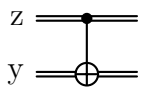
Definition 3.1. The expressions for *Silq-Hybrid* are defined as follows:

$$\begin{aligned} e ::= {} & c \mid x \mid \textbf{measure} \mid \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \\ & \mid \lambda(x_1, \ldots, x_n).e \mid x := e \\ & \mid e'(e) \mid e'(e_1, e_2) \mid \textbf{return } x_c \end{aligned} \tag{3.1}$$

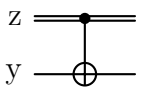
and are

- constants and built-in functions (c);
- variables (x);
- assignment ($x := e$);
- quantum measurement (**measure**);
- conditional statements (**if** $e \dots$);
- lambda abstraction ($\lambda(x_1, \dots, x_n).e$), which describes a function;
- function calls ($e'(e), e'(e_1, e_2)$), which are restricted to up to two inputs as Silq's in-built functions only take up to two inputs;
- and return (**return** x_c) is included with the restriction that only classical variables can be returned.

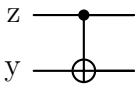
Importantly, conditional statements can use either quantum or classical variables within their conditions. This allows a quantum operation to either be added or removed for a classical condition, or for a controlled quantum operation to take place if a quantum condition is used, *i.e.*, a conditional statement **if** z **then** $y := X(y)$ could be equivalent to any of the three circuits in Figure 3.1 based on the types of z and y.



(a) Classical variable controlling a classical variable.



(b) Classical variable controlling a quantum variable.



(c) Quantum variable controlling a quantum variable.

Figure 3.1: Different ways a conditional expression in Silq be interpreted.

Remark 3.1. The Silq language itself has several syntax differences to Silq-Hybrid or Silq-Core. This is because the syntax of the Silq fragments are designed for developing theory, whereas the Silq language is designed for the programmer's convenience. For example, conditional statements in the Silq fragments

if e **then** f **else** g

are written in a Silq program as

```
if e { f; } else { g; }.
```

Examples

Here examples of the Silq programming language syntax are provided to build up to the programs used in this thesis.

```
def coin_flip(){
    x := 0:B;
    x := H(x);
    y := measure(x);
    return y;
}
```

Figure 3.2: A simple quantum coin-flip program

Firstly, a simple coin flip program is provided in Figure 3.2. The first line of the program initialises a qubit as a quantum boolean (`B` with basis states $|0\rangle, |1\rangle$) starting in the state $|0\rangle$. The second line applies the Hadamard operation to the qubit putting it into superposition. The qubit is then measured and stored in a new variable y, which is a classical boolean (`!B`).[1] Finally, the value of y is returned.

```
def f(const x : uint[2]):qfree B{
    return x >= 2;
}
```

Figure 3.3: An example oracle function.

Next we consider the oracle function given in Figure 3.3. The input of the function is a quantum variable (`x`) that represents an unsigned integer with 2 qubits (`uint[n]` for n qubits with basis states $|0\rangle, |1\rangle, \ldots, |2^n - 1\rangle$). The input is annotated as being constant (`const`), which means that the variable will not change value in the function. The function is also annotated as being `qfree`, which indicates that no superpositions are introduced or destroyed, *i.e.*, , no Hadamard (or similar) operations are performed. The oracle itself states that it returns whether the individual computational states of x are greater than 2, *i.e.*, , if the quantum state is $\sum_j a_j |j\rangle$, then the function modifies the quantum state to be $\sum_j a_j |j\rangle |j \geq 2\rangle$. This function

[1] An exclamation mark is used to denote classical types in Silq.

is what can be commonly used to create an oracle and is useful for controlling on multiple qubits (as we will see).

```
def control(x : uint[3], y:B){
    if x >= 6 {
        y : = H(y);
    }
    return x, y;
}
```

Figure 3.4: An example controlled function.

A controlled operation is introduced in Figure 3.4. The function takes a quantum register, x, with 3 qubits; a single qubit, y; and returns them. The controlled operation works by looking for basis states that have a value greater than 6, then applies the Hadamard operation to y controlled on those basis states, entangling the two variables together (if they aren't already). As a quantum operation,

$$U\left|x\right\rangle\left|y\right\rangle = \begin{cases} \left|x\right\rangle H\left|y\right\rangle & \text{if } x >= 6 \\ \left|x\right\rangle\left|y\right\rangle & \text{otherwise} \end{cases},$$

is equivalent to `control` in Figure 3.4.

```
def fixed_dj(f: const uint[2]!->qfree B){
    x := 0:uint[2];

    x[0] := H(x[0]);
    x[1] := H(x[1]);

    if f(x){ phase(pi); }

    x[0] := H(x[0]);
    x[1] := H(x[1]);

    x := measure(x);
    return x;
}
```

Figure 3.5: Deutsch-Jozsa Silq Program

A 2-qubit version of the Deutsch-Jozsa algorithm [62] written in Silq is given in Figure 3.5. The function has an input which is of the type described previously

```
x = 2
x = x + 2
y = 3 - x
```
(a) A classical program

```
x0 = 2
x1 = x0 + 2
y0 = 3 - x1
```
(b) Proof obligations

```
(= x0 2)
(= x1 (+ x0 2))
(= y0 (- 3 x1))
```
(c) SMT-LIB2 format

Figure 3.6: Converting a program for verification using Single Static Assignment (SSA).

(taking in a quantum register as a constant, `const uint[2]!->`, and returning a boolean `qfree B`). Initialisation, quantum operations, measurement, and return statements are present within the program as described before. An important feature introduced is the conditional statement and the phase operation in the fourth line. The phase operation, `phase(r)`, multiplies the phase of the quantum state by e^{ir}, but it is only observable through conditional statements. The conditional statement narrows the quantum states that the operation should be applied on. Essentially, the conditional statement in Figure 3.5 transforms the quantum state, as follows

$$\sum_j |j\rangle \to \Big(\sum_j |j\rangle\,|f(j)\rangle \to \sum_j (-1)^{f(j)}\,|j\rangle\,|f(j)\rangle \Big) \to \sum_j (-1)^{f(j)}\,|j\rangle\,,$$

where the transformation in brackets are the individual steps taken by the conditional and phase statements. A further discussion on the behaviour of conditionals is discussed in Section 3.6.3.

3.2.2 Classical Program Verification using SMT Solvers

The standard technique for checking a program using SMT solvers is to convert variables into symbols that keep track of when a variable is updated. This approach is also referred to as Static Single Assignment (SSA) [5, 17, 56, 132]. The converted program expressions are referred to as proof obligations, and an SMT solver is called to find a model for the proof obligations. Figure 3.6 shows an example of a generic classical program being converted into proof obligations (with the respective SMT-LIBv2 format [15]).

3.3 SilVer Architecture

SilVer (**Sil**q **Ver**ification) is a framework for verifying Silq programs by allowing the programmer to specify behaviour using a simple specification language called

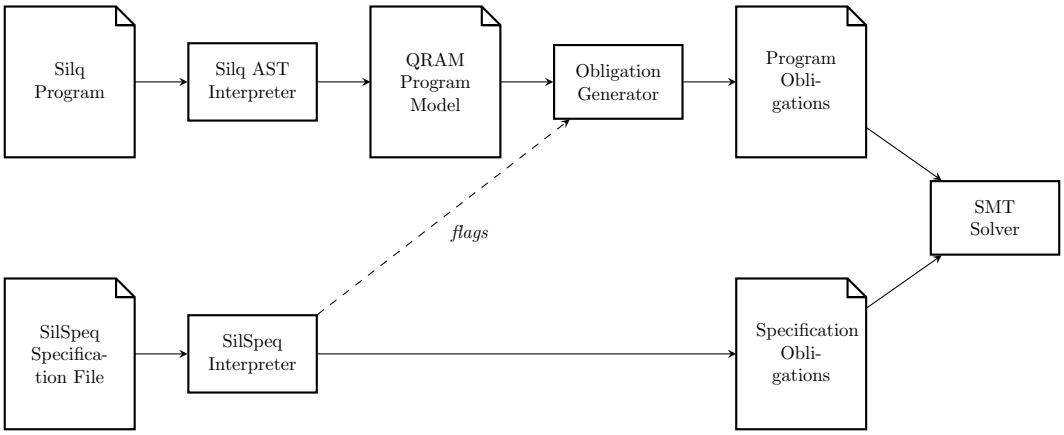


Figure 3.7: SilVer Design

SilSpeq. The framework combines these together by converting programs and SilSpeq desired behaviours into proof obligations to be proven in a SMT solver.

The design of SilVer is given in Figure 3.7. SilVer converts Silq programs into an intermediate representation (discussed in Section 3.4), which can then be further converted into proof obligations. These obligations are generated automatically and may change depending on the behaviours given in SilSpeq. The behaviours that can change the generated obligations that are are referred to as *flags* and are discussed in Section 3.5.3. Otherwise, desired behaviours in SilSpeq can also be converted into proof obligations very easily.

The benefit of the designed SilVer architecture is that the components of SilVer are modular. One for instance can replace Silq with another quantum programming language (such as Q#) and specify the behaviour using SilSpeq, which is then verified using SMT solvers. Alternatively one can replace the form or method of verification. As an example, one could instead translate Silq programs into a theorem prover based tool (such as QBricks or SQIR), where the theorems and lemmas that are to be proved by the user are automatically generated.

The implementation details and examples of SilVer verification are discussed in Section 3.7.

3.4 The QRAM Program Model for Verification

As part of the conversion process, Silq programs are represented using an intermediate representation. To handle both classical and quantum operations, a model

based on quantum RAM (QRAM) [96] is used to represent programs.

At a high level, the QRAM program model represents the classical and quantum memory independently. The model receives an instruction that only affects the respective memory, *i.e.*, a quantum instruction only affects the quantum memory and a classical instruction only affects the classical memory. The last element of the model is a list of controls, which models classical or quantum conditions in `if` statements, that restrict the instruction being run only if the control conditions are met in the classical case (*i.e.*, standard classical conditional statement behaviour), or create a controlled version of the quantum operation being performed in the quantum case.

Throughout, assume a set of variable names, $\mathcal{X}$, and the size function, $size(x)$, which gives the number of bits/qubits needed to represent x is defined. The size of variable x can be found based on the type of $x \in \mathcal{X}$. Note that types are removed when converted into the model, although future research can investigate implementing types into the model.

3.4.1 Memory

For the program, variables are kept track through objects known as Registers, with a collection of Registers known as a Memory.

Definition 3.2 (Register)**.** A register is a tuple $\mathcal{R} = (s, ver)$ where $s \in \mathbb{N}$ is the number of (qu)bits required and $ver \in \mathbb{N}$ is the version number.

Definition 3.3 (Memory)**.** A memory $\mathcal{M}$ is a mapping between variables and registers, where if $x \in \mathcal{X}$ is a variable and $\mathcal{R}$ is its associated register, then $\mathcal{M}(x) = \mathcal{R}$.

The notation $\mathcal{M}' = \mathcal{M}[x \to \mathcal{R}]$ is used to denote $\mathcal{M}'(x') = \mathcal{R}$ if $x' = x$ and $\mathcal{M}'(x') = \mathcal{M}(x')$ otherwise. Some operations that can be performed on a memory are defined in Figure 3.8.

A register or memory by itself simply contains information about variables, regardless if the variables are quantum or classical. It is only by designating a memory or register that the type of the data is known. For the intermediate representation, two different memories are created, which are a classical memory and a quantum memory, denoted $\mathcal{M}_c$ and $\mathcal{M}_q$ respectively. This allows a clear separation between variables used to represent classical and quantum data.

The version of variables are tracked throughout the program since when converting into SMT format, obligations are made about a variable at certain time steps

$$
\begin{aligned}
add_{\mathcal{M}}(x) &= \mathcal{M}[x \to (size(x), 0)] \\
iter_{\mathcal{M}}(x) &= \mathcal{M}[x \to (size(x), ver + 1)] \\
amend_{\mathcal{M}}(x) &= \begin{cases} iter_{\mathcal{M}}(x) & \text{if } \mathcal{M}(x) \neq () \\ add_{\mathcal{M}}(x) & \text{otherwise} \end{cases} \\
del_{\mathcal{M}}(x) &= \mathcal{M}[x \to ()]
\end{aligned}
$$

Figure 3.8: Some operations that can be performed on a memory $\mathcal{M}$: *add* adds a new variable to $\mathcal{M}$; *iter* updates the version of a variable in $\mathcal{M}$; *amend* updates a variable if it is already in $\mathcal{M}$, or adds it in if it isn't; and *del* removes a variable from $\mathcal{M}$.

as the program advances. The obligations made about a variable at the start of a program may differ from those at the end of the program. Changes are tracked at the register level, through *ver*, rather than the memory level as it is more efficient to track variables and only update the values of the variable when they are changed, leading to fewer proof obligations being generated. This gives a quantum version of Static Single Assignment (SSA) [5, 17, 56, 132], where the quantum state is represented as a complex vector at each point of the program where the quantum state is modified through initialisation, evolution, or measurement. This is further discussed with examples in Section 3.6.2.

Example 3.1. To give an example of memories and applying operations to them, let

$$
\begin{aligned}
&\mathcal{M}_q(a) = (5, 2) \quad \mathcal{M}_q(b) = (8, 0) \\
&\mathcal{M}_c(u) = (3, 3)
\end{aligned}
$$

where $\mathcal{M}_q$ is a quantum memory and $\mathcal{M}_c$ is a classical memory. It can be seen that $\mathcal{M}_q$ has two registers: one contains a register of 5-qubits for variable a that has a version number of 2, the other is a register of 8 qubits for variable b that has a version number of 0; whereas $\mathcal{M}_c$ only has one register of 3 bits for variable u, which has a version number of 3.

If $del_{\mathcal{M}_q}(a)$ is applied, then $\mathcal{M}_q$ is now

$$\mathcal{M}_q(b) = (8, 0).$$

If $add_{\mathcal{M}_c}(a)$ is applied, then $\mathcal{M}_c$ becomes

$$\mathcal{M}_c(u) = (3, 3) \quad \mathcal{M}_c(a) = (5, 0).$$

3.4.2 Operations

Operations are unary or binary operators that occur between variables and/or constants. Examples include negation ($-a$), addition ($a+b$), inequality checking ($a \leq b$) and square root ($\text{sqrt}(a)$).

Definition 3.4 (OP)**.** An operation instruction, OP, is defined as

$$\text{OP} ::= \text{UNARY}(\diamond, a) \mid \text{BINARY}(l, \star, r),$$

where $\diamond$ is some unary operation, $\star$ is a binary operation and a, l, r are arguments.

Arguments can be variables $x \in \mathcal{X}$ that have been defined previously or numbers. The OP instructions can be interpreted as performing $\diamond a$ for unary operations or $l \star r$ for binary operations. For some operation, b, that is unary or binary, $\text{OP}(b)$ represents its appropriate symbolic instruction representation.

Operations are used to represent function application and standard built-in Silq operations ($+$, $\leq$, sqrt, ...) applied to values or variables. However, they do not consider how the result is used; they are given meaning through instructions and controls.

Example 3.2. In Table 3.1 a few Silq expressions and their associated Operations are provided.

Silq Expression	Operation
`sqrt(2)`	$\text{UNARY}(\sqrt{\ }, 2)$
`-x`	$\text{UNARY}(-, x)$
`3 + 5`	$\text{BINARY}(3, +, 5)$
`2 * x`	$\text{BINARY}(2, *, x)$
`a < b`	$\text{BINARY}(a, <, b)$

Table 3.1: Examples of Operations

3.4.3 Instructions

Silq expressions are mostly modelled by instructions. Similar to how memory is separated, instructions are separated into classical and quantum instructions because it is convenient to distinguish when an instruction affects the quantum state from when it affects classical variables.

Definition 3.5. The *quantum instructions*, QINST, are

- QINIT(x, n, c), initialisation of variable x with n qubits and initial state c (*e.g.*, QINIT$(x, 3, 0)$ models `x := 0:uint[3]`);
- QOP(U, x), unitary evolution of a variable x using U (*e.g.*, QOP(H, x) models `x := H(x)`);
- QMEAS(x), measure a quantum variable x (*e.g.*, QMEAS(x) partly models `y := measure(x)`).

Definition 3.6. The classical instructions, CINST, are

- CSET(x, s, c), setting a variable x (with s bits) to a value c (*e.g.*, CSET$(x, 3, 2)$ models `x := 2:!uint[3]`);
- CSET(x, s, OP), setting a variable x (with s bits) as the result of some operation OP (*e.g.*, CSET$(x, 5, \text{OP}(3, +, y))$ models `x := 3 + y`, where `x,y:!uint[5]`);
- CMEAS(x'), capture the result of a measurement in a variable x' (*e.g.*, CMEAS(y) partly models `y := measure(x)`);
- RETURN(x), returning a classical value x (*i.e.*, `return x`).

Only one of the quantum or classical instructions can be performed at a time; to accommodate this both instruction sets have access to a SKIP instruction that does nothing. The only time a quantum and classical instruction can be performed at the same time is when a measurement is performed on a quantum variable, which converts the quantum variable into a classical one. Since the quantum and classical memories are separated from one another and instructions can only be performed on the respective memories (*i.e.*, QINST on quantum and CINST on classical), then one quantum instruction is required to remove the variable from the quantum memory and a classical instruction is required to store the result into

classical memory. Whilst it is possible to perform a measurement that results in a quantum state by measuring on a different basis, this is not the case for Silq. Silq performs measurement on the computational basis state only and any measured variable is converted to an appropriate classical type.[2]

The design of the instructions is to mimic how operations are written, *e.g.*, CSET follows the structure of $c := a*b$ whereas QOP follows how one would apply quantum operations, $U|x\rangle$. Additionally, the size of a quantum variable cannot change (its type can, but not its size), whereas classical values can easily change types and sizes as the program progresses, hence why CSET requires the size of a variable whereas QOP does not.

3.4.4 Controls

Conditional expressions are exempt from being represented as an Instruction. The condition of a conditional expression (**if** $e\ldots$) is converted into a special operation called a Control, which are a restricted set of OP instructions.

Definition 3.7 (CTRL)**.** A control instruction, CTRL, is an OP instruction that can use either classical or quantum variables in their argument and uses logical or comparison operators for their operation ($\neg$, $\leq$, $==$, $\ldots...$).

For example, a conditional statement **if** $x \leq 2$ **then** e will have a control instruction BINARY$(x, \leq, 2)$, where x is a previously defined variable.

A generated control operation is added to a vector of controls, Γ, which keeps track of all conditionals that currently affect the program. This allows us to separate the conditions that are affecting a program from the actual instructions that are performed. This has the additional benefit of having both quantum and classical controls inside. This novel approach of separating controls from instructions is used to arbitrarily create controlled unitaries based on the controls used.

3.4.5 Processes and Programs

The previously defined instructions and memory representation are combined to give a representation of a single expression.

[2]It is possible to change the basis using quantum operations pre-measurement and to generate an appropriate state post-measurement, but the measurement itself would still be in the computational basis.

Definition 3.8 (QRAM Process)**.** A QRAM process is a quintuple

$$p = (Q, \mathcal{M}_q, C, \mathcal{M}_c, \Gamma),$$

where Q is a quantum instruction; C is a classical instruction; $\mathcal{M}_q, \mathcal{M}_c$ are quantum and classical memory respectively; and $\Gamma = (c_1, \ldots, c_n)$ is a list of CTRL instructions.

For simplicity, only one instruction occurs at a time, *i.e.*, at least one of Q or C is the SKIP instruction. The motivation behind this design is to separate out what operations are affecting the classical and quantum memory. The only exception is when performing measurement, which uses QMEAS and CMEAS for QINST and CINST, respectively. This is to separate the quantum and classical components of measurement.

Now that expressions can be represented through processes, programs are represented as a list of processes.

Definition 3.9 (QRAM Program Model)**.** A QRAM program model is a sequentially composed list of processes $P = (p_1, p_2, \ldots, p_n)$ whose memories differ by up to one memory operation, *i.e.*, for all $i \in \{1, \ldots, n-1\}$, $\mathcal{M}_q^{i+1} = \mathcal{M}_q^i$ or $\mathcal{M}_q^{i+1} = f(\mathcal{M}_q^i)$ where f is one of the operations given in Figure 3.8 and $\mathcal{M}_q^j$ is the quantum memory for p_j (similarly for classical memory M_c^j).

Remark 3.2. The symbol, $\oplus$ is used to denote the concatenation of tuples (such as processes and controls) onto the end of a tuple. For example,

$$p_1 \oplus p_2 = (p_1, p_2),$$

and

$$(p_1, p_2) \oplus (p_1', p_2', p_3') = (p_1, p_2, p_1', p_2', p_3').$$

This operation is used to build QRAM programs from individual operations. This is mainly useful in Section 3.6.

Example 3.3. A 2-qubit version of the Deutsch-Jozsa algorithm, provided previ-

ously in Figure 3.5, is represented by the following QRAM program model:

$$\begin{aligned}\big(&(\text{QINIT}(x, 2, |0\rangle), \mathcal{M}_e[x \to (2,0)], \text{SKIP}, \mathcal{M}_e, \Gamma_e),\\ &(\text{QOP}(H \otimes H, x), \mathcal{M}_e[x \to (2,1)], \text{SKIP}, \mathcal{M}_e, \Gamma_e),\\ &(\text{QOP}(\pi I \otimes I, x), \mathcal{M}_e[x \to (2,2)], \text{SKIP}, \mathcal{M}_e, \Gamma_1),\\ &(\text{QOP}(H \otimes H, x), \mathcal{M}_e[x \to (2,3)], \text{SKIP}, \mathcal{M}_e, \Gamma_e),\\ &(\text{QMEAS}(x), \mathcal{M}_e, \text{CMEAS}(x), \mathcal{M}_e[x \to (2,4)], \Gamma_e),\\ &(\text{SKIP}, \mathcal{M}_e, \text{RETURN}(x), \mathcal{M}_e[x \to (2,4)], \Gamma_e)\big)\end{aligned}$$

where $\Gamma_e = ()$, $\Gamma_1 = (\text{BINARY}(\text{UNARY}(f, x), ==, 1)$, and $\mathcal{M}_e$ is an empty memory (for all x', $\mathcal{M}_e(x') = ()$).

3.5 SilSpeq: Specifying Behaviour of Silq programs

In this section, the language that is used for specifying behaviour for programs written in Silq is introduced. SilSpeq is designed in a way such that it is easy for the user to write specification that is easily translatable into SMT obligations.

3.5.1 Defining Variables

In SilSpeq, the user defines variables before they are used. Definitions require the user to provide a variable name and a type. The types that variables can take are limited to integers of fixed bit size and functions of these types. These are respectively represented by the following syntax

$$\langle type \rangle ::= \{0,1\}^n \mid \langle type \rangle \to \langle type \rangle,$$

where $n \in \mathbb{N}$. A definition is then usually written as

$$\text{define } \langle name \rangle : \langle type \rangle.$$

When converted into SMT format, integers are represented as integers with obligations on the values they can take. Functions are converted into functions on integers, with restrictions on their inputs and outputs.

3.5.2 Expressions

SilSpeq uses a combination of arithmetic and logical expressions. Its syntax is:

$$\alpha ::= m \in \mathbb{Z} \mid var \mid f(\alpha_1) \mid \alpha_1 * \alpha_2 \mid \alpha_1 + \alpha_2 \mid \alpha_1^{\alpha_2} \mid (\alpha_1)$$
$$l ::= ff \mid \alpha_1 = \alpha_2 \mid \alpha_1 < \alpha_2 \mid \neg l_1 \mid l_1 \& l_2 \mid @var.l$$

where var is a defined variable, f is a defined function, ff denotes falsity ($\bot$), & represents logical and ($\wedge$), and @ represents for all ($\forall$). Other arithmetic and logical expressions can be derived by combining the above (e.g. $-, /$ for arithmetic expressions and $\leq, >, \vee, \implies, \exists, \ldots$ for logical expressions).

For convenience, the implementation includes several useful expressions as shorthand for certain expressions. For example, $a.b$ is used to denote the bitwise dot product of two integers with the same number of bits. That is $a.b = \sum_{i=0}^{n-1} a_i b_i$ where a_i, b_i are the bits of a, b respectively.[3]

In SilSpeq, these expressions are wrapped in an assert statement:

$$\text{assert}(l).$$

Expressions can easily be converted into SMT format by using the appropriate SMT expression and referring to variables when needed. Shorthand expressions, such as $a.b$, can also be converted using appropriate SMT expressions as well.

3.5.3 Flags

Many quantum verification frameworks perform measurement based on a conditional statement structure; *i.e.*, if a qubit is measured and the result is $|1\rangle$, perform one command, otherwise perform another. In SilVer, the properties of measurement to verify are considered instead. For instance, some quantum programs may require certainty in any result that might be measured, whereas others may simply require the most likely result.

To achieve this, *flags* are included in the SilSpeq function definition that tell us what property of measurement to verify in a given function. This changes the measurement outcome that can occur. Additionally this allows us to either restrict the cases considered. For instance, if only measurement to be done with certainty is allowed, then the solver only needs to consider measurement values that satisfy

[3] This summation can also be written as $\sum_{i=0}^{n-1}[(a/2^i)(b/2^i) \mod 2]$.

measurement with certainty. On the other hand, the flexibility of flags mean that the solver can consider all possible measurement values and take these into account as well.

Flags allow the user to specify certain approaches that should be taken when verifying Silq programs, changing the obligations that are generated from the intermediate representation. For SilVer, flags mainly affect measurement. For numerous quantum algorithms, there are certain properties about measurement that are verifiable.

The types of flags available are described:

- The *rand* flag puts no obligations on measurement and so any measurement result is possible so long as it has a non-zero probability.
- The *cert* flag specifies that for any run of the program, there must be a state that has a measurement probability of 1, *i.e.*, with *cert*ainty.
- The $whp(x)$ (with high probability) flag states that for any run of the program, measurement results that occur with probability greater than x are only considered. For ease, $whp = whp(0.5)$ and note that $cert = whp(1)$.

Flags interact with the obligations generated from measurement instructions (QMEAS and CMEAS). How the obligations generated from measurement instructions are affected by flags is shown in Section 3.6.

3.5.4 Function Structure

A SilSpeq file is automatically generated by SilVer if one does not exist. This file contains each function written within the Silq file, along with its input arguments defined as a SilSpeq type and an additional return argument that encapsulates the output of the function. This return argument is denoted by $\langle function\ name \rangle\_ret$. Additionally, within a SilSpeq file, each function contains *pre*- and *post*-condition blocks.

3.6 Conversion of Silq-Hybrid to Proof Obligations

Silq programs and SilSpeq specifications are converted each into an SMT encoding. A Silq program is converted into the QRAM program model first, and then into the SMT-LIBv2 format. Write *prog* to denote a Silq program's SMT encoding. SilSpeq specifications can be directly converted into an SMT encoding, as mentioned in Section 3.5.2. Additionally, *pre* and *post* are used to represent a SilSpeq specification's pre- and post-conditions SMT encoding respectively.

When a program is checked against its specification, the logical statement

$$prog \wedge pre \wedge \neg post \tag{3.2}$$

is checked by an SMT solver, which looks for an execution (a model from *prog*), with pre-conditions given by *pre*, that does not satisfy the post-conditions, *post*, given by the user. If conjunction (3.2) is satisfiable, then there is a possible program execution that does not meet the user defined specification. However, if the conjunction is unsatisfiable and $prog \wedge pre$ is true for all states, then $\neg post$ must be false and therefore $prog \wedge pre \wedge post$ must be true always. Checking if $prog \wedge pre$ is true is fast since checking *prog* entails looking for a working program trace and checking *pre* just means ensuring there is no falsity generated in the pre-condition by having no conflicting statements, *e.g.*, $a = 0 \wedge a = 1$. Thus, checking the encodings together is fast on a SMT solver.

The conversion of Silq programs into proof obligations is described, with particular focus on how conditional (**if** e **then** e_1 **else** e_2) and measurement (**measure**) statements, as well as flags, affect generation of proof obligations.

Throughout, $[i, j)$ is used to denote the set $\{i, i+1, \ldots, j-2, j-1\}$ for $i < j$.

3.6.1 Silq-Hybrid to QRAM Program Model

The conversion of Silq-Hybrid, with the addition of sequences of statements, to the QRAM program model is given in Figure 3.9. The subscript c, q is used to denote classical and quantum variables or conditions respectively, and denote $\mathcal{M}_q^1, \mathcal{M}_c^1$ as the quantum and classical memory of $[\![e_1]\!]_{\mathcal{M}_q, \mathcal{M}_c, \Gamma}$ respectively.

$$
\begin{aligned}
[\![e_1; e_2]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& [\![e_1]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} \oplus [\![e_2]\!]_{\mathcal{M}_q^1,\mathcal{M}_c^1,\Gamma} \\
[\![x_q := c]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& (\mathrm{QINIT}(x_q, size(x_q), c), add_{\mathcal{M}_q}(x_q), \mathrm{SKIP}, \mathcal{M}_c, \Gamma) \\
[\![x_q := U(x_q)]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& (\mathrm{QOP}(U, x_q), iter_{\mathcal{M}_q}(x_q), \mathrm{SKIP}, \mathcal{M}_c, \Gamma) \\
[\![x_c := f(x_1)]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& (\mathrm{SKIP}, \mathcal{M}_q, \mathrm{CSET}(x, size(x_c), \mathrm{UNARY}(f, x_1)), \\
& amend_{\mathcal{M}_c}(x), \Gamma) \\
[\![x_c := g(x_1, x_2)]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& (\mathrm{SKIP}, \mathcal{M}_q, \mathrm{CSET}(x, size(x_c), \mathrm{BINARY}(x_1, g, x_2)), \\
& amend_{\mathcal{M}_c}(x), \Gamma) \\
[\![x_c := \textbf{measure}(x_q)]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& (\mathrm{QMEAS}(x_q), del_{\mathcal{M}_q}(x_q), \mathrm{CMEAS}(x_c), \\
& amend_{\mathcal{M}_c}(x_c), \Gamma) \\
[\![\textbf{if } b_c \textbf{ then } e_1 \textbf{ else } e_2]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& [\![e_1]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma\oplus\mathrm{OP}(b_c)} \oplus [\![e_2]\!]_{\mathcal{M}_q^1,\mathcal{M}_c^1,\Gamma\oplus\mathrm{OP}(\neg b_c)} \\
[\![\textbf{if } b_q \textbf{ then } e_1 \textbf{ else } e_2]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& [\![e_1]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma\oplus\mathrm{OP}(b_q)} \oplus [\![e_2]\!]_{\mathcal{M}_q^1,\mathcal{M}_c,\Gamma\oplus\mathrm{OP}(\neg b_q)} \\
[\![\textbf{return } x]\!]_{\mathcal{M}_q,\mathcal{M}_c,\Gamma} :=& (\mathrm{SKIP}, \mathcal{M}_q, \mathrm{RETURN}(x), \mathcal{M}_c, \Gamma)
\end{aligned}
$$

Figure 3.9: Conversion of Silq-Hybrid to QRAM program model.

3.6.2 Conversion of QRAM Program Model into Proof Obligations

Purely classical instructions, CINST, are converted in a similar way as described in Section 3.2.2, where classical variables are represented by symbols at certain time steps of the computation. The values of variables at later time steps are determined by the values of the same variable at an earlier time step.

The quantum state throughout the computation is represented by a vector of symbols whose (exponential) size is based on the sizes of the quantum variables throughout the program. While there are possible reductions or other representations available, such as the one used by `symQV` [18], this is the simplest representation that works without needing to make major changes to the specification. In a similar way to the classical obligations, there is a vector at each time step within the program. Later quantum symbols have values that are based on obligations from older symbols (akin to SSA as discussed in Section 3.2.2). Whilst costly, this representation is unavoidable as the full quantum state is needed for model checking. For example, a quantum memory consisting of

$$\mathcal{M}_q(p) = (p, size(p), V_p), \quad \mathcal{M}_q(r) = (r, size(r), V_r)$$

is represented by the symbol

$$\mathrm{p}P\mathrm{v}V_p|\mathrm{r}R\mathrm{v}V_r,$$

where $P \in [0, 2^{size(p)})$ and $R \in [0, 2^{size(r)})$.

To provide a further example, the evolution of a quantum operation,

$$U = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix},$$

applied to a single newly initialised qubit, $\mathcal{M}_q(p) = (p, 1, 0)$ is shown. This provides two initial symbols p0v0 for the state $|0\rangle$ and p1v0 for $|1\rangle$ with new symbols after the quantum operation being p0v1 and p1v1 for the respective states. The obligations

$$\mathrm{p0v1} == u_{00} * \mathrm{p0v0} + u_{01} * \mathrm{p1v1}$$
$$\mathrm{p1v1} == u_{10} * \mathrm{p0v0} + u_{11} * \mathrm{p1v1}$$

capture the evolution of a quantum state.

As the program progresses, the versions of the quantum variables are updated each time they are operated upon. The symbol vector representing the states of the quantum variables is referred to as the *quantum state* throughout.

Initialising a new quantum variable, QINIT, expands the quantum state and quantum unitary operations, QOP, affect the quantum state using their matrix representation (with the example demonstrated above can be expanded for more quantum variables).[4] Measurement statements are converted into proof obligations that encapsulate the probabilities of measuring a variable in the quantum state and puts additional constraints on those probabilities depending on the flag used. Additionally the quantum state is modified depending on what value is measured.

Conditional statements affect the program depending on whether the variable affected is classical or quantum. A classical conditional acts in a similar way to how a classical program would be verified, whereas a quantum case requires the generation of a quantum gate to be applied to the quantum state.

[4]In the implementation, an Instruction that allows parallel unitary operations to occur in a single matrix is included, reducing the number of obligations generated.

3.6.3 Conversion of Conditional and Measurement Statements

Throughout, write $\tau(p)$ as the proof obligations generated for process p.

Proof Obligations from Conditionals

There are two types of controls to consider, classical and quantum. Let Γ_c and Γ_q respectively denote the classical and quantum controls of a list of controls, Γ.

Classical The classical controls, Γ_c, determine if a block of code should be ran at all. Consider the following Silq code:

```
if b {
    x = x + 2;
    q := X(q);
}
```

where b is a boolean (0 or 1), x is a classical integer, and q is a variable referring to a qubit. This is represented by the Program (p_1, p_2), where

$$\begin{aligned} p_1 =&(\text{SKIP}, \mathcal{M}_q, \text{CSET}(x, size(x), \text{BINARY}(x, +, 2)),\\ &\mathcal{M}_c[x \to (size(x), 1)], (\text{BINARY}(b, ==, 1))), \text{ and}\\ p_2 =&(\text{QOP}(X, q), \mathcal{M}_q[x \to (q, size(q), 1)], \text{SKIP},\\ &\mathcal{M}_c[x \to (size(x), 1)], (\text{BINARY}(b, ==, 1))). \end{aligned}$$

If b is true, then the code in the **if** block is ran changing the value of x and the quantum state. However, if b is false, then the code is not ran and the values remain the same.

This means that when converting Programs to proof obligations, two types of obligations are generated. The first type state that when b is true, then the new variable symbols are updated in the way described by the operation. The second type state that when b is false, then the new symbols are set to be the same as the previous symbols. Note that for quantum variables, the next quantum state needs to be set to be the same as the previous quantum state.

If $\Gamma_c = (b_1, \ldots b_n)$, then the individual boolean conditions are included in a logical and, $\bigwedge$, statement. Therefore, proof obligations for classical conditionals work in a

similar as to how one would do it classically

$$\begin{aligned}&\tau((\text{QINST}, \mathcal{M}_q, \text{CINST}, \mathcal{M}_c, \Gamma)) = \\ &\text{if} \bigwedge_{b_i \in \Gamma_c} b_i : \tau((\text{QINST}, \mathcal{M}_q, \text{CINST}, \mathcal{M}_c, \Gamma_q) \\ &\text{if} \neg(\bigwedge_{b_i \in \Gamma_c} b_i) : id((\text{QINST}, \mathcal{M}_q, \text{CINST}, \mathcal{M}_c, \Gamma_q)),\end{aligned}$$

where $id(p)$ sets any variables modified by process p to be the same as their previous version.

Quantum Quantum controls, Γ_q, only affect the quantum state of a program and should not affect classical variables. This behaviour is already captured in Silq, therefore the change of classical variables does not need to be discussed. Additionally, measurement cannot be controlled by a quantum variable either (this causes a syntax error in Silq). Thus, only Programs with quantum controls that only use QOP operations need to be considered. These are generated by Silq statements such as

```
if r == 1{
    s := H(s);
}
```

and only Processes of the form

$$(\text{QOP}(U, s), amend_{\mathcal{M}_q}(s), \text{SKIP}, \mathcal{M}_c, (\text{OP}(r), \dots)),$$

need to be considered. For the example given $U = H$ and $\text{OP}(r) = \text{BINARY}(r, ==, 1)$.

The quantum control acts as a control operator on a unitary gate for another quantum variable. The unitary operation (within the conditional block) performed on the quantum state is modified depending on the quantum variable that is being controlled.

This modification to the unitary operation is computed in the following way. Let U' be the unitary operation that modifies the quantum variable s. Write U for the quantum operation that applies U' to s and the identity operation I on every other quantum variable. Consider $\text{OP}(b(r))$ (in the list of controls Γ_q) which applies some boolean operation b on a quantum variable r. Let S be the size of the quantum

state. For $v \in [0, 2^S)$, let $b_v \in \{0, 1\}$ denote the result of applying b to the value of r in the quantum state with value v. Create the matrix $B = \text{diag}((b_v)_{v \in [0,2^S)})$. Then the controlled unitary operation is

$$CU = I_q + (U - I_q).B \tag{3.3}$$

where I_q is the identity operation on the quantum state.

If there are multiple quantum controls, $\Gamma_q = (\text{OP}(b^1(r_1)), \ldots \text{OP}(b^n(r_n)))$, then one finds the vectors (b_v^i) for each $1 \leq i \leq n$ and calculates $(b_v) = (\bigwedge_i b_v^i)$. This new vector b_v is then used in B to calculate CU as given in Equation (3.3). Additionally, even though the example of U provided only affects one quantum variable, U can be replaced with a unitary operation that affects the entire quantum state apart from any control variables.

The proof obligations generated are

$$\begin{aligned}&\tau((\text{QOP}(r), \mathcal{M}'_q = iter_{\mathcal{M}_q}(r), \text{SKIP}, \mathcal{M}_c, \Gamma_q)) = \\ &(\mathcal{M}'_q)_i == \sum_j CU_{ij}(\mathcal{M}_q)_j \text{ for } i \in [0, 2^S),\end{aligned}$$

where $(\mathcal{M})$ denotes the quantum state (vector of symbols) for memory $\mathcal{M}$.

Proof Obligations from Measurement

When generating the proof obligations for measurement, we have

$$\begin{aligned}&\tau((\text{QMEAS}(x_q), del_{\mathcal{M}_q}(x_q), \text{CMEAS}(x_c), amend_{\mathcal{M}_c}(x_c), \Gamma)) = \\ &\quad obs_{prob} \wedge obs_{meas} \wedge obs_{post},\end{aligned}$$

where obs_{prob}, obs_{meas} and obs_{post} are *probability*, *measurement* and *post-state* obligations respectively. Within these obligations are restrictions on the measurement probabilities of values that a quantum variable can take. Measurement probabilities are represented by symbols as Pr_q_vw_i where q is a quantum register/variable, w is the current version and i is the measured state. Additionally the measured value of a register q is represented by the symbol meas_q.

Probability Obligations The obligations generated by obs_{prob} are obligations on the probability of measuring different values. They simply state that the probability

of any measurement value of the quantum state is between 0 and 1, calculate the values of the probabilities (based on the state measured) and that the sum of these probabilities is 1. These are expressions from some of the formula in Section 2.1.1. For example, measurement of a single qubit q generates the obligations

```
0 <= Pr_q_v0_0
Pr_q_v0_0 <= 1
0 <= Pr_q_v0_1
Pr_q_v0_1 <= 1
Pr_q_v0_0 + Pr_q_v0_1 = 1
Pr_q_v0_0 = |q_v0_0|^2
Pr_q_v0_1 = |q_v0_1|^2
```

where, again, Pr_q_vw_i is the symbol representing the probability of measuring the quantum register version w of the quantum register q results in the value i.

Measurement Obligations Obligations generated by obs_{meas} put restrictions on how and what can be measured. These obligations are influenced by the flags from `SilSpeq` (as described in Section 3.5.3): a *rand* flag puts no restrictions on measurement, *i.e.*, any value can be picked; a *cert* flag restricts the measured value such that values must have a measurement probability equal to 1, *i.e.*,

```
meas_q = i => Pr_q_vw_i = 1;
```

and *whp*(a) flags say that a measured value, i, can only be picked if its measurement probability is higher than the set value, *i.e.*,

```
meas_q = i => Pr_q_vw_i >= a.
```

Beyond flags, a requirement also needs to be met that the measured value's probability must be non-zero, *i.e.*,

```
meas_q = i => not(Pr_q_vw_i = 0).
```

Post-state Obligations Post-state obligations determine the state after measurement which is calculated for each possible measurement based on the standard normalisation formula for the quantum state after measurement given in Section 2.1.1.

For instance, consider a program with two registers, q and p, each with a qubit. If q is measured, then obligations for the quantum state post-measurement are

```
meas_q == 0 => sqrt(Pr_q_v0_0) * p_v1_0 == q_v0_0|p_v0_0
meas_q == 0 => sqrt(Pr_q_v0_0) * p_v1_1 == q_v0_0|p_v0_1
meas_q == 1 => sqrt(Pr_q_v0_1) * p_v1_0 == q_v0_1|p_v0_0
meas_q == 1 => sqrt(Pr_q_v0_1) * p_v1_1 == q_v0_1|p_v0_1
```

where `q_v0_i|p_v0_j` is the amplitude of the quantum state $|ij\rangle$ before measurement; `p_v1_j` is the state after measurement ($|j\rangle$); and `sqrt(Pr_q_v0_i)` is the normalisation based on the probability of measuring q in state $|i\rangle$.[5]

3.7 Implementation and Case Studies

In this section, details are given about the implementation of `SilVer` and benchmarks on program instances are provided.

3.7.1 Implementation

`SilVer` is implemented in Python with over 2000 lines of code, 300 of which are used for implementing `SilSpeq`. The Python interface for Z3 [58], z3py, is used to make obligations. Users can provide Silq programs, restricted to the Silq-Hybrid format, and `SilSpeq` specification. `SilVer` is available at `https://github.com/marco-lewis/silver`.

3.7.2 Case Studies

Here details are provided of three algorithms that are tested for verification. The benchmarks are tested on specified qubit instances of the following programs: GHZ state generation, the Deutsch-Jozsa algorithm, and the Bernstein-Vazirani algorithm. The provided Silq programs are based on test files found in the Silq repository.[6]

The programs given and tested use a specific number of qubits. This is because SMT solvers cannot solve for arbitrary size programs (which theorem provers are

[5] The normalisation constant is multiplied with the post-state amplitude to avoid potential errors from division by 0.

[6] `https://github.com/eth-sri/silq/tree/master/test` (accessed 22/03/2024)

```
def ghz(){
    x := 0:B;
    y := 0:uint[2];
    x := H(x);
    if x{
        y[0] := X(y[0]);
        y[1] := X(y[1]);
    }
    y := measure(y);
    x := measure(x);
    return y;
}
```

(a) GHZ Silq Program

```
ghz[whp(0.5)]()->
    (define ghz_ret:{0, 1}^2)
pre{}
post{
    assert(ghz_ret = 0 | ghz_ret = 3)
}
```

(b) GHZ SilSpeq specification

Figure 3.10: Generation of GHZ states. The specification states that the only values to be read with at least 50% probability should be the $|0\rangle$ or the $|3\rangle$ state (the all 1 state for 2 qubits).

capable of). Examples using 2 qubits are provided throughout this section. To increase the program or specification for different qubit sizes, the number of gates applied, some constants, and the size of data types needs to be changed, which can easily be implemented easily.

GHZ State Generation The Greenberger–Horne–Zeilinger (GHZ) state [84] for an n-qubit system is

$$\frac{1}{\sqrt{2}}|0\rangle^{\otimes n} + \frac{1}{\sqrt{2}}|1\rangle^{\otimes n},$$

i.e., there is a 50/50 chance of measuring n 0's or n 1's. The state is prepared using multiple controlled-NOT gates and a single qubit in superposition. The goal is to check that the only quantum states returned with 50% probability $|0\rangle^{\otimes n}$ and $|1\rangle^{\otimes n}$. Note that while this does not describe the GHZ state fully, this is due to restricting the specification to be classical in nature. For example, the state $\frac{1}{\sqrt{2}}|0\rangle^{\otimes n} + (\frac{1}{\sqrt{2}} - \epsilon)|1\rangle^{\otimes n} + \epsilon|0\rangle^{\otimes n-1}|1\rangle$, where $0 < \epsilon \in \mathbb{R}$, would also follow this specification.

Figure 3.10 contains the program and specification for setting up the 2-qubit GHZ state (also the Bell state) in the y register using x as an additional qubit. The program and specification can be extended to any qubit number. Note that the *whp*() flag is used to specify that states with at least 50% probability matter and that no preconditions are required.

```
def fixed_dj(
f: const uint[2]!->qfree B
){
    x := 0:uint[2];

    x[0] := H(x[0]);
    x[1] := H(x[1]);

    if f(x){ phase(pi); }

    x[0] := H(x[0]);
    x[1] := H(x[1]);

    x := measure(x);
    return x;
}
```

(a) Deutsch-Jozsa Silq Program

```
fixed_dj[rand]
    (define f:{0, 1}^2->{0, 1})->
    (define fixed_dj_ret : {0, 1}^2)
pre{
    define y : N
    define x : {0,1}^2
    define bal : {0,1}
    assert(SUM[x](f) = y)
    assert((bal = 0 & (y = 0 | y = 4))
        | (bal = 1 & y = 2))
}
post{
    assert(bal = 0 -> fixed_dj_ret = 0)
    assert(bal = 1 -> ¬fixed_dj_ret = 0)
}
```

(b) Deutsch-Jozsa SilSpeq specification

Figure 3.11: Deutsch-Jozsa algorithm. Note that $\text{SUM}[x](f) \equiv \sum_{x=0}^{3} f(x)$ and the second line in the post-conditions is commented. The specification for the pre-condition says a function is constant (not balanced) if and only if $\sum_x f(x) = 0$ or 2^n (as $f(x) = 0$ or 1 for all x); and the function is balanced if and only if $\sum_x f(x) = \frac{1}{2}2^n$ (since half the inputs return 1 and the other half return 0).

Deutsch-Jozsa Algorithm The Deutsch-Jozsa algorithm [62] was one of the first oracle-based quantum algorithms to be developed. The algorithm can distinguish if a function has one of the following two properties. A function $f : X \to \{0, 1\}$ is said to be *constant* if all inputs return the same value (0 or 1). Further, f is *balanced* if half the inputs of f return 0 and the other half return 1.

Given $f : \{0, 1\}^n \to \{0, 1\}$ such that f is either constant or balanced, the Deutsch-Jozsa algorithm will discover which property f has, using only a single evaluation of f. The algorithm will return 0 if the function is constant, otherwise it will return a non-zero value if the function is balanced.

The Silq program is restated and the SilSpeq specification for a two qubit version of the Deutsch-Jozsa algorithm is given in Figure 3.11. A SilSpeq file is provided for each of the postconditions provided in the specification, the function definition and preconditions remains the same across the files.

Bernstein-Vazirani Algorithm The Bernstein-Vazirani algorithm [22] is similar to the Deutsch-Jozsa algorithm. The problem the Bernstein-Vazirani algorithm

```
fixed_bernvaz[cert](define f:{0, 1}^2->{0, 1})->
    (define fixed_bernvaz_ret : {0, 1}^2)
pre{
    define s : {0,1}^2
    define x : {0,1}^2
    assert(@x. f(x) = (s.x) mod 2)
}
post{
    assert(fixed_bernvaz_ret = s)
}
```

Figure 3.12: SilSpeq specification for the Bernstein-Vazirani algorithm for 2 qubits. Note that $@x.\phi(x)$ denotes $\forall x.\phi(x)$ and $s.x$ is the dot product as described in Section 3.5.2.

solves is: given an oracle $f : \{0,1\}^n \to \{0,1\}$ such that $f(x) = x.s$ for some $s \in \{0,1\}^n$, find s. The dot product $x.s = x_0 s_0 \oplus \ldots \oplus x_{n-1} s_{n-1}$ where $\oplus$ denotes addition modulo 2. The quantum algorithm is the same as the Deutsch-Jozsa: using n qubits in the state $|0^n\rangle$, put the qubits into superposition, call the oracle, undo the superposition and finally perform a measurement. This process returns the string s with certainty, using a single evaluation of f.

The 2-qubit Silq program for the Bernstein-Vazirani algorithm is the same program used for the Deutsch-Jozsa algorithm, given in Figure 3.11a. Specification for the behaviour of the Bernstein-Vazirani algorithm on two qubits is given in Figure 3.12.

3.7.3 Benchmarks

The benchmarks are tested on specified qubit instances of the following programs: GHZ state generation, the Deutsch-Jozsa algorithm, and the Bernstein-Vazirani algorithm. The experiments were performed on a laptop with an Intel(R) Core(TM) i5-10310U CPU @ 1.70GHz x 8 cores processor and 16GB of RAM. The device uses Ubuntu 20.04.3 LTS and the results are the average of 10 runs. The benchmarks for the programs are given in Table 3.2.

For the benchmarks, SilVer is not compared against the theorem proving tools mentioned [88, 104, 166], since these are manual approaches to verifying programs. Discussion is provided on two other tools SilVer is not compared against.

SilVer is not compared with symQV [18] since the programs and type of specification checked between each tool is different. symQV works purely on quantum

Table 3.2: Benchmarks of running `SilVer` on different programs: Qubits = number of qubits used; Setup Time = CPU time for `SilVer` to translate a Silq program and `SilSpeq` obligations into the SMT solver; Verification Time = CPU time for the solver to check the obligations are unsatisfiable; Memory = RAM used by the SMT solver. Times are average and standard deviation over 10 runs. The timeout is set to 3 hours.

Benchmark	Qubits	Setup Time (s)	Verification Time (s)	Memory (MiB)
GHZ	2	0.14±0.02	$0.003\pm5\times10^{-4}$	17
	5	3.4±0.03	$0.029\pm9\times10^{-4}$	18
	7	53±1.8	0.4±0.02	95
	8	210±3.9	1.7±0.07	350
Deutsch-Jozsa	2	0.066±0.019	0.009±0.002	17
	3	0.15±0.02	0.041±0.007	17
	4	0.41±0.05	7.9±1	19
	5	1.4±0.1	`timeout`	23
Bernstein-Vazirani	2	0.06±0.01	0.01±0.001	17
	3	0.14±0.01	0.07±0.01	17
	4	0.46±0.06	0.5±0.03	18
	5	3.8±2.2	3.7±0.36	19
	6	13±1.3	41±7.1	39
	7	30±0.46	1000±260	67

programs (no classical commands) and its specification is restricted to quantum states, making it useful to very programs such as quantum teleportation [21] and the diffusion operation of Grover's algorithm [85] but unable to reason about programs with oracle behaviour. To compare these approaches, `symQV` would require the specification of oracles or `SilSpeq` would need to be expanded to include quantum specification.

Despite `QBricks` [39] predominately working on quantum circuits, the way it specifies behaviour is very similar to `SilVer` and includes additional specification in relation to quantum circuits (*e.g.*, circuit width and depth). However, as `QBricks` is based in the Why3 framework [74], it uses both manual and automatic approaches to prove properties about circuits, which allows it to prove properties about arbitrary size circuits. This makes it difficult to compare against `SilVer`, but taking a specific qubit size program may provide `QBricks` a fully automatic benchmark to compare against.

3.7.4 Discussion

The speed at which SilVer verifies a program depends mainly on the complexity of the program, where this complexity can depend on the number of oracle functions to consider. In some cases verification setup takes a much longer time than actual verification (e.g. GHZ) whereas verification is slower in other cases past a certain point (e.g. Bernstein-Vazirani).

With regards to algorithms, checking the Bernstein-Vazirani algorithm becomes faster than checking Deutsch-Jozsa once the number of qubits gets larger. This is because the model checking approach needs to consider all possible functions used for the oracle. Specifically, for n qubit instances of the problems, there are only 2^n possible functions for Bernstein-Vazirani (since the secret string $s \in \{0,1\}^n$), whereas the Deutsch-Jozsa algorithm has $\binom{2^n}{2^{n-1}} + 2$ possible functions (due to the nature of balanced functions). This is where manual theorem provers may be a better tool for verification of certain quantum algorithms.

Note that there are some restrictions to SilVer. Notably, if running on a laptop or a device with a small amount RAM, SilVer will be unlikely to verify anything larger than 10 qubits. Additionally, the SMT-LIBv2 files for the obligations become large very quickly; a 3-qubit instance of the Bernstein-Vazirani algorithm only requires about 20kB of space whereas the 10-qubit version takes up over 250MB of memory.

3.8 Conclusion

SilVer was presented as a tool for automatically verifying programs written in the Silq programming language. The automatic translation of Silq programs into a model that can be converted into an SMT suitable format was shown. SilSpeq was also presented as a method for writing behaviours of programs using simple expressions. SilSpeq introduces the concept of *flags* for ensuring certain properties from measurement. The focus of design for SilVer is usability for developers, making the verification of programs as simple as possible.

The modular design of SilVer is influenced by Vellvm [164] (verified LLVM), which reasons about programs written within the LLVM intermediate representation. SilVer is capable of converting Silq into a QRAM Program Model and then converting that into suitable verification formats. The quantum intermediate repre-

sentation (QIR) is a representation for quantum programming languages based on the LLVM intermediate representation under development by the QIR Alliance [128]. Work has been done to formalise QIR such for code safety already [106]. In the future, adapting the Vellvm approach for the QIR could be valuable for verifying quantum programs.

This concludes the investigation of the first automated technique in the thesis, which demonstrates how the standard software verification framework can be adapted for quantum programs. Further discussion on the consequences of this technique can be found in Part IV; whereas the next Part, III, looks at the second automated technique, which looks at verifying behaviour at lower levels of the quantum stack.

Part III

Barrier Certificates

Chapter 4

Verification of Quantum Systems using Barrier Certificates

In this Part, the second automated technique, barrier certificates, for verifying behaviours in quantum systems is investigated. This technique is used for verifying properties at a lower level of the quantum stack, either the gates on the quantum chip or a quantum circuit to be executed. This chapter gives an introduction to the theory that needs to be adapted to handle quantum systems and an initial automated process to generate and verify safety properties for quantum gates.

4.1 Introduction

Quantum systems evolve according to the Schrödinger equation from some initial state. However, the initial state may not be known completely in advance. One can prepare a quantum system by making observations on the quantum objects, leaving the quantum system in a basis state, but this omits the global phase which is not necessarily known after measurement. The resultant phase may have an affect the later evolution of a system. Further, the system could be disturbed through some external influence before it begins evolving. This can slightly change the quantum state from the basis state to a state in superposition or possibly an entangled state.

By taking into account these uncertain factors, a set of possible initial states, from which the system evolves, can be constructed. These possible initial states can be determined based on the setup of the system, *e.g.*, the initial states could be the most likely ones based on the noise the system faces before measurement (for

example, an area where the initial state has a 90% chance of occurring). In practice, a noise model can be used to determine the initial region. A noise model [80] creates a model based on the unique sources of noise a quantum computer faces and the models take into account how much the sources affect the system. These can be used to approximate the initial setup of the quantum state for the evolution of the system.

From the initial set, it can be investigated if the system evolves according to some specified behaviour such as reaching or avoiding a particular set of states. As an example, consider a single qubit system that evolves according to a Hamiltonian $\hat{H}$ implementing the controlled-NOT operation. Through measurement and factoring in for noise, assume the system starts close to $|10\rangle$. The controlled-NOT operation keeps the first qubit value the same and, as the system evolves via $\hat{H}$, the property that the quantum state does not evolve close to $|00\rangle$ or $|01\rangle$ needs to be verified.

The main purpose of this chapter is to study the application of a technique called *barrier certificates*, used for verifying properties of classical dynamical systems, to check properties of quantum systems similar to the one mentioned above. The concept of barrier certificates has been developed and used in Control Theory to study the safety of dynamical systems from a given set of initial states on real domains [125]. This technique can ensure that given a set of initial states from which the system can start and a set of unsafe states, the system will not enter the unsafe set. This is achieved through separating the unsafe set from the initial set by finding a *barrier*.

Barrier certificates can be defined for both deterministic and stochastic systems in discrete and continuous time [4, 100]. The concept has also been used for verification and synthesis against complicated logical requirements beyond safety and reachability [93]. The conditions under which a function is a barrier certificate can be automatically and efficiently checked using SMT solvers [13]. Such functions can also be found automatically using learning techniques even for non-trivial dynamical systems [121].

Dynamical systems are naturally defined on real domains ($\mathbb{R}^n$). To handle dynamical systems in complex domains ($\mathbb{C}^n$), one would need to decompose the system into its real and imaginary parts and use the techniques available for real systems. This has two disadvantages, the first being that this doubles the number of variables being used for the analysis. The second disadvantage is that the analysis may be easier to perform directly with complex variables than their real components. As

quantum systems use complex values, it is desirable to have a technique to perform the reachability analysis using complex variables; since for an n-qubit systems, 2^{n+1} variables are required to represent the quantum system as a real dynamical system, rather than 2^n variables for a complex dynamical system.

In this Chapter, the problem of safety verification in quantum systems is explored by extending barrier certificates from real to complex domains. The extension is inspired by a technique developed by Fang and Sun [67], who studied the stability of complex dynamical systems using Lyapunov functions (where the goal is to check if a system eventually stops moving). Further, an algorithm to generate barrier certificates for quantum systems is provided and used to generate barriers for several examples.

4.2 Background

4.2.1 Notation

Throughout this part, the following notation is used:

- the imaginary unit, $\mathrm{i} = \sqrt{-1}$ (i is not used as an iterator or variable);
- for a complex number $z = a + b\mathrm{i}$, its complex conjugate is $\overline{z} = a - b\mathrm{i}$;
- for a $n \times m$ matrix, z, write
  - $(z)_{jk}$ as an element of z where $0 \leq j \leq n-1, 0 \leq k \leq m-1$ (for vectors, simply write $(z)_j$),
  - $\overline{z}$ as the conjugate of z ($(\overline{z})_{jk} = \overline{(z)_{jk}}$),
  - $z^\intercal$ as the transpose of z ($(z^\intercal)_{jk} = (z)_{kj}$),
  - and $z^* = \overline{z}^\intercal$ as the conjugate transpose of z;
- I_n for the $n \times n$ identity operation.

4.2.2 Safety Analysis

The problem of safety for dynamical systems with real state variables, $x \in \mathbb{R}^n$, is introduced first. More details can be found in [125]. A continuous dynamical system is described by

$$\dot{x} = \frac{\mathrm{d}x}{\mathrm{d}t} = f(x), \quad f : \mathbb{R}^n \to \mathbb{R}^n,$$

where the evolution of the system is restricted to $X \subseteq \mathbb{R}^n$ and f is usually Lipschitz continuous to ensure existence and uniqueness of the differential equation solution. The set $X_0 \subseteq X$ is the set of initial states and the unsafe set $X_u \subseteq X$ is the set of values that the dynamics $x(t)$ should avoid. These sets lead to the idea of safety for real continuous dynamical systems:

Definition 4.1 (Safety)**.** A system, $\dot{x} = f(x)$, evolving over $X \subseteq \mathbb{R}^n$ is considered safe if the system cannot reach the unsafe set, $X_u \subseteq X$, from the initial set, $X_0 \subseteq X$. That is for all $t \in \mathbb{R}_+$ and $x(0) \in X_0$, then $x(t) \notin X_u$.

The safety problem is to determine if a given system is safe or not. Numerous techniques have been developed to solve this problem [75]. Barrier certificates are discussed in Section 4.2.3. Here, two other common techniques are described.

Abstract Interpretation One way to perform reachability analysis of a system is to give an abstraction [53,54] of the system's evolution. An initial abstraction that over-approximates the evolution of the system can be refined using approaches such as CEGAR [44] (as discussed in Section 2.1.2, where counterexamples are used to guide and refine the abstraction such that the abstraction captures the behaviour of the system as accurately as possible. Using this method, an abstraction can be made and shown that the system does not reach the unsafe region through the behaviour of the abstraction. If a counterexample in the abstraction is found, the abstraction is updated to reflect the behaviour if it is a spurious/false counterexample (*i.e.*, it does not enter the unsafe region in the actual system), or return that the system is unsafe if it is a true counterexample (*i.e.*, the counterexample does enter the unsafe region in the actual system). This method has been investigated for quantum programs in [162], where the authors can verify programs using up to 300 qubits.

Backward and Forward Reachability A second approach is to start from the unsafe region and reverse the evolution of the system from there. A system is considered unsafe if the reversed evolution enters the initial region. This is backward reachability. Conversely, forward reachability starts from the initial region and is considered safe if the reachable region does not enter the unsafe region. Both backward and forward reachability are discussed in [113, 141, 142].

4.2.3 Barrier Certificates

Barrier certificates [125] are another technique used for safety analysis. This technique attempts to divide the reachable region from the unsafe region by putting constraints on the initial and unsafe set, and on how the system evolves. The benefit of barrier certificates over other techniques is that one does not need to compute the system's dynamics at all to guarantee safety, unlike in abstract interpretation and backward (or forward) reachability.

A barrier certificate is a differentiable function, $B : \mathbb{R}^n \to \mathbb{R}$, that determines safety through the properties that B has. Generally, a barrier certificate needs to meet the following conditions:

$$B(x_0) \leq 0, \forall x_0 \in X_0 \tag{4.1}$$

$$B(x_u) > 0, \forall x_u \in X_u \tag{4.2}$$

$$x(0) \in X_0 \implies B(x(t)) \leq 0, \forall t \in \mathbb{R}_+. \tag{4.3}$$

Essentially, these conditions split the evolution space into an over-approximation of the reachable region and an unsafe region, encapsulated by Conditions (4.1) and (4.2) respectively. These regions are separated by a "barrier", which is the contour along $B(x) = 0$. One may view barrier certificates as an abstraction of the dynamical system. However, it would be more appropriate to view barrier certificates as a kind of measurement or analysis on the system, *i.e.*, the barrier certificate takes "measurements" on the state space whereas an abstraction simplifies the dynamics of a system.

Condition (4.3) prevents the system evolving into the unreachable region and needs to be satisfied for the system to be safe. However, Condition (4.3) can be replaced with stronger conditions that are easier to check. For example, the definition of one simple type of barrier certificate is given.

Definition 4.2 (Convex Barrier Certificate)**.** For a system $\dot{x} = f(x)$, $X \subseteq \mathbb{R}^n$, $X_0 \subseteq X$ and $X_u \subseteq X$, a function $B : \mathbb{R}^n \to \mathbb{R}$ that obeys the following conditions:

$$B(x) \leq 0, \forall x \in X_0 \tag{4.4}$$

$$B(x) > 0, \forall x \in X_u \tag{4.5}$$

$$\frac{\mathrm{d}B}{\mathrm{d}x} f(x) \leq 0, \forall x \in X, \tag{4.6}$$

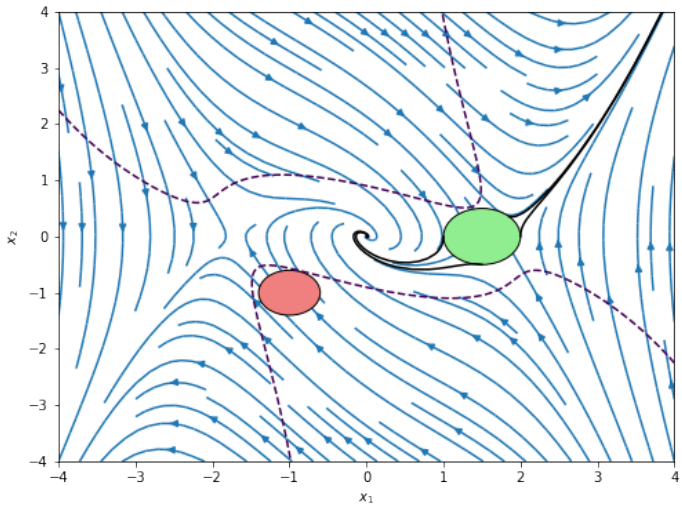


Figure 4.1: Example adapted from Section V-A in [125]. The initial region is the green circle centred at $(1.5, 0)$ and the system evolves according to the dynamical system given by differential equations $\dot{x} = [(x)_2, -(x)_1 + \frac{1}{3}{(x)_1}^3 - (x)_2]$. The unsafe region is the red circle centred at $(-1, -1)$ and is separated from the initial region by a barrier, the dashed purple line defined by $B(x) = 0$ where $B(x) = -13 + 7{(x)_1}^2 + 16{(x)_2}^2 - 6{(x)_1}^2{(x)_2}^2 - \frac{7}{6}{(x)_1}^4 - 3(x)_1{(x)_2}^3 + 12(x)_1(x)_2 - \frac{12}{3}{(x)_1}^3(x)_2$.

is a convex barrier certificate.

Note that in Condition (4.6): $\frac{\mathrm{d}B}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{\mathrm{d}B}{\mathrm{d}t}$. This condition can be viewed as a constraint on the evolution of the barrier as the system evolves over time.

Now, if a system has a barrier certificate, then the system is safe. The safety theorem for convex barrier certificates is shown.

Theorem 4.1. *If a system, $\dot{x} = f(x)$, has a convex barrier certificate, $B : \mathbb{R}^n \to \mathbb{R}$, then the system is safe [125].*

Proofs of Theorem 4.1 are standard and can be found in, *e.g.*, [125]. The intuition behind the proof is that since the system starts in the negative region and the barrier can never increase, then the barrier can never enter the positive region. Since the unsafe set is within the positive region of the barrier, this set can therefore never be reached. Thus, the system cannot evolve into the unsafe set and so the system is safe. Figure 4.1 shows an example of a dynamical system with a barrier based on the convex condition.

Remark 4.1. The term "convex" is used for these barriers as the set of barrier certificates satisfying the conditions in Definition 4.2 is convex. In other words, if B_1 and B_2 are barrier certificates for a system, the function $\lambda B_1 + (1-\lambda)B_2$ is also a barrier certificate for any $\lambda \in [0,1]$. See [125] or the proof of Proposition 4.3 in Appendix B.1.2 for (similar) details.

There are a variety of different barrier certificates to choose from with different benefits, *e.g.*, the convex condition given is simple but may not work for complicated or nonlinear systems. In comparison, the non-convex condition given in [125] changes Condition (4.6) such that $\frac{\mathrm{d}B}{\mathrm{d}x} f(x) \leq 0; \forall x \in X, B(x) = 0$ (instead of $\forall x \in X$). This is a weaker condition allowing for more functions to be a suitable barrier certificate. However, a different computational method is required because the set of such barrier certificates is non-convex. Each barrier certificate requires a different proof that if the system has a satisfying barrier certificate, then the system is safe. It should be noted that Theorem 4.1 only has a one way implication, a system does not necessarily have a barrier certificate even if it is safe. There are works that show the converse and this necessity [130, 155]. For example, in [155], the authors showed the converse holds for systems defined on a compact manifold and using convex barrier certificates, *i.e.*, for a specific family of systems, if a system is safe, then it has an associated barrier certificate.

4.3 Complex Continuous-Time Barrier Certificates

Now the use of barrier certificates is extended into a complex space ($\mathbb{C}^n$). The complex dynamical systems considered are of the form

$$\dot{z} = \frac{\mathrm{d}z}{\mathrm{d}t} = f(z), \quad f : \mathbb{C}^n \to \mathbb{C}^n,$$

which evolves in $\mathcal{Z} \subseteq \mathbb{C}^n$ and f remains Lipschitz continuous. The initial and unsafe sets are defined in the usual way except now $\mathcal{Z}_0 \subseteq \mathcal{Z}$ and $\mathcal{Z}_u \subseteq \mathcal{Z}$, respectively. The notion of safety for this system is similar to Definition 4.1.

Definition 4.3 (Safety)**.** A complex system, $\dot{z} = f(z)$, with $\mathcal{Z} \subseteq \mathbb{C}^n$, $\mathcal{Z}_0 \subseteq \mathcal{Z}$ and $\mathcal{Z}_u \subseteq \mathcal{Z}$, is considered safe if for any $z(0) \in \mathcal{Z}_0$, then $\forall t \in \mathbb{R}^+, z(t) \notin \mathcal{Z}_u$.

Whilst it is easy to extend the safety problem and required definitions into the complex plane, extending the notion of barrier certificates requires particular atten-

tion. Conditions (4.1), (4.2) and (4.3) are changed respectively to

$$B(z_0) \leq 0, \forall z_0 \in \mathcal{Z}_0; \tag{4.7}$$

$$B(z_u) > 0, \forall z_u \in \mathcal{Z}_u; \tag{4.8}$$

$$z(0) \in \mathcal{Z}_0 \implies B(z(t)) \leq 0, \forall t \in \mathbb{R}_+. \tag{4.9}$$

Many barrier certificates use differential equations to achieve Condition (4.9), which restricts the class of functions that can be used. This is because differentiable complex functions must satisfy the Cauchy-Riemann equations.

A holomorphic function, $g(z) : \mathbb{C}^n \to \mathbb{C}$, is considered to be a function whose partial derivatives, $\frac{\partial g(z)}{\partial (z)_j}$, are holomorphic on $\mathbb{C}$, *i.e.*, they satisfy the Cauchy-Riemann equations (for several variables). That is for $(z)_j = (x)_j + \mathrm{i}(y)_j$ and $g(z) = g(x, y) = u(x, y) + \mathrm{i}v(x, y)$, then

$$\frac{\partial u}{\partial (x)_j} = \frac{\partial v}{\partial (y)_j} \qquad \frac{\partial u}{\partial (y)_j} = -\frac{\partial v}{\partial (x)_j}.$$

An adapted technique developed by Fang and Sun [67] is used to reason about barrier certificates in the complex plane. First, a family of complex functions that are key to technique are introduced.

Remark 4.2. Note that the Cauchy-Riemann equations are not necessary for the dynamical system; since the differential equations for the dynamics describe a change over time, which has a real variable rather than a complex variable.

Definition 4.4 (Conjugate-flattening function)**.** A function, $b : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n$, is conjugate-flattening if $\forall z \in \mathbb{C}^n, b(z, \overline{z}) \in \mathbb{R}$.

Definition 4.5 (Complex-valued barrier function)**.** A function, $B : \mathbb{C}^n \to \mathbb{R}$, is a complex-valued barrier function if $B(z) = b(z, \overline{z})$ where $b : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n$ is a conjugate-flattening, holomorphic function.

Remark 4.3. These functions have a Hermitian-like form to them, in that if $b(z, w) = w^\intercal H z$, for some complex matrix H, then $b(z, \overline{z}) = z^\dagger H z \in \mathbb{R}$, which is a property of Hermitian matrices.

Suppose now that $z(t)$ is a system that evolves over time. To use the complex-valued barrier function, $B(z(t))$, for barrier certificates the differential of B with

respect to t is required. Calculating this differential reveals that

$$\begin{aligned}\frac{\mathrm{d}B(z(t))}{\mathrm{d}t} = \frac{\mathrm{d}b(z(t),\overline{z(t)})}{\mathrm{d}t} &= \frac{\mathrm{d}b(z,u)}{\mathrm{d}z}\bigg|_{u=\overline{z}} \frac{\mathrm{d}z}{\mathrm{d}t} + \frac{\mathrm{d}b(z,u)}{\mathrm{d}u}\bigg|_{u=\overline{z}} \overline{\frac{\mathrm{d}z}{\mathrm{d}t}} \\ &= \frac{\mathrm{d}b(z,u)}{\mathrm{d}z}\bigg|_{u=\overline{z}} f(z) + \frac{\mathrm{d}b(z,u)}{\mathrm{d}u}\bigg|_{u=\overline{z}} \overline{f(z)},\end{aligned} \tag{4.10}$$

where $\frac{\mathrm{d}b(z,u)}{\mathrm{d}z} = \left[\frac{\partial b(z,u)}{\partial(z)_1}, \frac{\partial b(z,u)}{\partial(z)_2}, \ldots, \frac{\partial b(z,u)}{\partial(z)_n}\right]$ is the gradient of $b(z,u)$ with respect to z and the gradient is defined with respect to u in a similar way. Given Equation (4.10), barrier certificates that include a differential condition can be extended into the complex domain quite naturally. For example, the convex barrier certificate is extended to the complex domain.

Definition 4.6 (Complex-valued Convex Barrier Certificate)**.** For a system $\dot{z} = f(z)$, $\mathcal{Z} \subseteq \mathbb{C}^n$, $\mathcal{Z}_0 \subseteq \mathcal{Z}$ and $\mathcal{Z}_u \subseteq \mathcal{Z}$; a complex-valued barrier function $B : \mathbb{C}^n \to \mathbb{R}$, $B(z) = b(z,\overline{z})$, that obeys the following conditions,

$$b(z,\overline{z}) \leq 0, \forall z \in \mathcal{Z}_0 \tag{4.11}$$

$$b(z,\overline{z}) > 0, \forall z \in \mathcal{Z}_u \tag{4.12}$$

$$\frac{\mathrm{d}b(z,u)}{\mathrm{d}z}\bigg|_{u=\overline{z}} f(z) + \frac{\mathrm{d}b(z,u)}{\mathrm{d}u}\bigg|_{u=\overline{z}} \overline{f(z)} \leq 0, \forall z \in \mathcal{Z}, \tag{4.13}$$

is a complex-valued convex barrier certificate.

With this definition, the safety of complex dynamical systems can be ensured:

Theorem 4.2. *If a complex system, $\dot{z} = f(z)$, has a complex-valued convex barrier certificate, $B : \mathbb{C}^n \to \mathbb{R}$, then the system is safe.*

Proposition 4.3. *The set of complex-valued barrier certificates satisfying the conditions of Definition 4.2 is convex.*

The proofs of these results are given in Appendix B.1.1 and B.1.2 respectively. Whilst these definitions and proofs are simple extensions and adaptions of the ones for real dynamical system, this is the first time that they have been applied to barrier certificates associated with complex dynamical systems.

4.4 Generating Satisfiable Barrier Certificates for Quantum Systems

The computation of a complex-valued barrier function is described in this section. Throughout, let $\dot{z} = f(z)$, $\mathcal{Z} \subseteq \mathbb{C}^n$, $\mathcal{Z}_0 \subseteq \mathcal{Z}$ and $\mathcal{Z}_u \subseteq \mathcal{Z}$ be defined as before. The general family of functions that will be used is introduced as "templates" for complex barrier certificates.

Definition 4.7. A k-degree polynomial function is a complex function, $b : \mathbb{C}^n \to \mathbb{C}$, such that

$$b(z_1, \ldots, z_n) = \sum_{\boldsymbol{\alpha} \in A_{n,k}} a_{\boldsymbol{\alpha}} z^{\boldsymbol{\alpha}} \tag{4.14}$$

where $A_{n,k} := \{\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \subseteq \mathbb{N}^n : \sum_{j=1}^{n} \alpha_j \leq k\}$, $a_{\boldsymbol{\alpha}} \in \mathbb{C}$, and $z^{\boldsymbol{\alpha}} = \prod_{j=1}^{n} (z)_j{}^{\alpha_j}$.

The family of k-degree polynomials are polynomial functions where no individual term of the polynomial can have a degree higher than k. Note that k-degree polynomial functions are holomorphic. Further, some k-degree polynomials are conjugate-flattening. For example, the 2-degree polynomial $b(z_1, u_1) = z_1 u_1$ is conjugate-flattening since $z\overline{z} = |z|^2$, whereas the 1-degree polynomial $b(z_1, u_1) = z_1$ is not. Thus, a subset of this family of functions are suitable to be used for barrier certificates as complex-valued barrier functions.

The partial derivative of the polynomials in Equation (4.14) is required for ensuring the function meets Condition (4.13). The partial derivative of the function is

$$\frac{\partial b}{\partial (z)_j} = \sum_{\boldsymbol{\alpha} \in A_{n,k}} a_{\boldsymbol{\alpha}} (\alpha)_j (z)_j{}^{-1} z^{\boldsymbol{\alpha}}. \tag{4.15}$$

Denote the barrier certificates with coefficients as

$$B(a, z) := b(a, z, \overline{z}) := \sum_{\substack{(\boldsymbol{\alpha}, \boldsymbol{\beta}) \in A_{2n,k} \\ \boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \\ \boldsymbol{\beta} = (\alpha_{n+1}, \ldots \alpha_{2n})}} a_{\boldsymbol{\alpha}, \boldsymbol{\beta}} z^{\boldsymbol{\alpha}} \overline{z}^{\boldsymbol{\beta}},$$

where $a = (a_{\boldsymbol{\alpha}, \boldsymbol{\beta}}) \in \mathbb{R}^{|A_{2n,k}|}$ is a vector of real coefficients to be found and $\overline{z}^{\boldsymbol{\beta}} = \prod_{j=1}^{n} \overline{(z)_j}^{\alpha_{n+j}}$.

The following (polynomial) inequalities find the coefficient vector:

$$\begin{aligned} \textbf{find } & a^T \\ \textbf{subject to } & B(a,z) \leq 0, \forall z \in \mathcal{Z}_0 \\ & B(a,z) > 0, \forall z \in \mathcal{Z}_u \\ & \frac{\mathrm{d}B(a,z)}{\mathrm{d}t} \leq 0, \forall z \in \mathcal{Z} \\ & B(a,z) \in \mathbb{R} \\ & -1 \leq a_{\boldsymbol{\alpha},\boldsymbol{\beta}} \leq 1. \end{aligned} \tag{4.16}$$

The coefficients, $a_{\boldsymbol{\alpha},\boldsymbol{\beta}} \in \mathbb{R}$, are restricted to the range $[-1, 1]$ since any barrier certificate $B(a, z)$, can be normalised by dividing B by the coefficient of greatest weight, $m = \max |a_{\boldsymbol{\alpha},\boldsymbol{\beta}}|$. The resulting function $\frac{1}{m}B(a, z)$ is still a barrier certificate. A barrier certificate generated from these polynomial inequalities can then freely be scaled up by multiplying it by a constant.

4.4.1 An Algorithmic Solution

One approach of solving the inequalities in (4.16) is to convert the system to real numbers and solve using sum of squares (SOS) optimisation [125]; another method is to use SMT solvers to find a satisfiable set of coefficients; or it is possible to use neural network based approaches to find possible barriers [1, 121]. Here, a special case can be considered, where $\frac{\mathrm{d}B(a,z)}{\mathrm{d}t} = 0$ rather than $\frac{\mathrm{d}B(a,z)}{\mathrm{d}t} \leq 0$, which allows the problem to be turned into a linear program. This restriction allows a subset of barrier certificates that still ensures the safety of the system to be considered. This is motivated by the fact that simple quantum systems of interest exhibit periodic behaviour; that is for all $t \in \mathbb{R}^+$, $z(t) = z(t + T)$ for some T. The barrier must also exhibit periodic behaviour,[1] and this can be achieved by setting $\frac{\mathrm{d}B(a,z)}{\mathrm{d}t} = 0$. Whilst there are other properties that ensure a function is periodic, these would involve non-polynomial terms such as trigonometric functions. Further, linear programs tend to be solved faster than SOS methods. This is because SOS programs are solved through semidefinite programming techniques, which are extensions of linear programs and therefore harder to solve.

To begin with, the differential constraint, $\frac{\mathrm{d}B(a,z)}{\mathrm{d}t} = 0$, is transformed. To obey

[1]The barrier being periodic can be seen by interpreting the barrier as a function over time: $B(t) = B(z(t)) = B(z(t + T)) = B(t + T), \forall t \in \mathbb{R}^+$

the third condition for the complex-valued convex barrier certificate, terms in Equation (4.10) can be substituted with the partial derivatives from Equation (4.15). Essentially, the equation is of the form

$$(\mathbf{A}a)^\top \zeta = 0,$$

where ζ is a vector of all possible polynomial terms of $(z)_j, \overline{(z)_j}$ with degree less than k,[2] and $\mathbf{A}$ is a matrix of constant values. By setting $\mathbf{A}a = \vec{0}$ the constraint is satisfied. Therefore, each row of the resultant vector, $(\mathbf{A}a)_j = 0$, is added as a constraint to a linear program.

To transform the real constraint ($B(a,z) \in \mathbb{R}$) note that if $x \in \mathbb{C}$, then $x \in \mathbb{R}$ if and only if $x = \overline{x}$. Therefore, $B(a,z) - \overline{B(a,z)} = 0$ and

$$\begin{aligned} B(a,z) - \overline{B(a,z)} &= \sum_{\substack{(\alpha_j)\in A_{2n,k} \\ \boldsymbol{\alpha}=\{\alpha_1,\ldots,\alpha_n\} \\ \boldsymbol{\beta}=\{\alpha_{n+1},\ldots\alpha_{2n}\}}} a_{\boldsymbol{\alpha},\boldsymbol{\beta}} z^{\boldsymbol{\alpha}} \overline{z}^{\boldsymbol{\beta}} - \sum_{\substack{(\alpha_j)\in A_{2n,k} \\ \boldsymbol{\alpha}'=\{\alpha_1,\ldots,\alpha_n\} \\ \boldsymbol{\beta}'=\{\alpha_{n+1},\ldots\alpha_{2n}\}}} \overline{a}_{\boldsymbol{\alpha}',\boldsymbol{\beta}'} z^{\boldsymbol{\beta}'} \overline{z}^{\boldsymbol{\alpha}'} \\ &= \sum_{\substack{(\alpha_j)\in A_{2n,k} \\ \boldsymbol{\alpha}=\{\alpha_1,\ldots,\alpha_n\} \\ \boldsymbol{\beta}=\{\alpha_{n+1},\ldots\alpha_{2n}\}}} (a_{\boldsymbol{\alpha},\boldsymbol{\beta}} - \overline{a}_{\boldsymbol{\beta},\boldsymbol{\alpha}}) z^{\boldsymbol{\alpha}} \overline{z}^{\boldsymbol{\beta}}. \end{aligned}$$

The whole polynomial is equal to 0 if all coefficients are 0. Thus, taking the coefficients and noting that a_j are real gives the transformed constraints $a_{\boldsymbol{\alpha},\boldsymbol{\beta}} = a_{\boldsymbol{\beta},\boldsymbol{\alpha}}$ for $\boldsymbol{\alpha} = (\alpha_j)_{j=1}^{n}, \boldsymbol{\beta} = (\alpha_j)_{j=n+1}^{2n}, (\alpha_j) \in A_{2n,k}$. These constraints to the coefficients are then also added to the linear program.

The final constraints we need to transform are the constraints on the initial and unsafe set: $B(a,z) \leq 0$ for $z \in \mathcal{Z}_0$ and $B(a,z) > 0$ for $z \in \mathcal{Z}_u$, respectively. Begin by noting that $B(a,z) = c + b(a,z,\overline{z})$ where $b(a,z,\overline{z})$ is a k-degree polynomial (with coefficients a) and $c \in \mathbb{R}$ is a constant. When considering the differential and real constraint steps, c is not involved in these equations since c does not appear in the differential term and c is cancelled out in the real constraint ($c - \overline{c} = c - c = 0$).

Considering the initial and unsafe constraints, it is required that

$$\begin{aligned} &\forall z_0 \in \mathcal{Z}_0,\ c + b(a, z_0, \overline{z_0}) \leq 0, \text{ and} \\ &\forall z_u \in \mathcal{Z}_u,\ c + b(a, z_u, \overline{z_u}) > 0. \end{aligned}$$

[2] *e.g.*, for $k = 2$ acceptable terms include $(z)_j{}^a, (z)_j(z)_l, (z)_j\overline{(z)_l}, \overline{(z)_j}^a, \overline{(z)_j(z)_l}$ for $0 \leq a \leq 2$.

Algorithm 1: Computing the barrier certificate using linear programming

1: Solve the linear program

$$\begin{aligned} &\textbf{find } a^T \\ &\textbf{subject to } \mathbf{A}a = \vec{0} \\ &\qquad a_{\boldsymbol{\alpha},\boldsymbol{\beta}} = a_{\boldsymbol{\beta},\boldsymbol{\alpha}} \qquad \text{for } \boldsymbol{\alpha} = \{\alpha_j\}_{j=1}^{n}, \boldsymbol{\beta} = \{\alpha_j\}_{j=n+1}^{2n}, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{and } \{\alpha_j\}_{j=1}^{2n} \in A_{2n,k} \\ &\qquad -1 \leq a_j \leq 1. \end{aligned}$$

2: $c \leftarrow \min_{z \in \mathcal{Z}_0} -b(a, z, \overline{z})$
3: **if** $c > \max_{z \in \mathcal{Z}_u} -b(a, z, \overline{z})$ **then return** $B(a, z) = c + b(a, z, \overline{z})$
4: **else fail**

Therefore, c is bounded by

$$\max_{z \in \mathcal{Z}_u} -b(a, z, \overline{z}) < c \leq \min_{z \in \mathcal{Z}_0} -b(a, z, \overline{z}).$$

Finding $c = \min_{z \in \mathcal{Z}_0} -b(a, z, \overline{z})$ and then checking $\max_{z \in \mathcal{Z}_u} -b(a, z, \overline{z}) < c$ will ensure the initial and unsafe constraints are met for the barrier. The final computation is given in Algorithm 1.

Note that the algorithm can fail since the function b may divide the state space in such a way that a section of $\mathcal{Z}_0$ may lie on the same contour as a section of $\mathcal{Z}_u$. This means that either the function b is unsuitable or the system is inherently unsafe.

4.5 Application to Quantum Systems

Quantum systems that evolve within Hilbert spaces, $\mathcal{H}^n = \mathbb{C}^{2^n}$ for $n \in \mathbb{N}$, are considered. The computational basis states $|j\rangle \in \mathcal{H}^n$, for $0 \leq j < 2^n$, are used as an orthonormal basis within the space, where $(|j\rangle)_l = \delta_{jl}$.[3] General quantum states, $|\phi\rangle \in \mathcal{H}^n$, can then be written in the form

$$|\phi\rangle = \sum_{j=0}^{2^n-1} (z)_j \, |j\rangle \,,$$

[3] δ_{jl} is the Kronecker delta, which is 1 if $j = l$ and 0 otherwise.

where $(z)_j \in \mathbb{C}$ and $\sum_{j=0}^{2^n-1} |(z)_j|^2 = 1$.[4] Quantum states reside within the unit circle of $\mathbb{C}^{2^n}$. For simplicity, quantum systems that evolve according to the Schrödinger equation are considered:

$$\frac{\mathrm{d}\,|\phi\rangle}{\mathrm{d}t} = -\mathrm{i}\hat{H}\,|\phi\rangle\,,$$

where $\hat{H}$ is a Hamiltonian, a complex matrix such that $\hat{H} = \hat{H}^\dagger = \overline{\hat{H}^\top}$; and $|\phi\rangle$ is a quantum state.[5] In the rest of this section, Algorithm 1 is used to find suitable barrier certificates for operations that are commonly used in quantum computers.

Code Availability The code that implements Algorithm 1 and the following examples is available at:

`https://github.com/marco-lewis/quantum-barrier-certificates`.

4.5.1 Hadamard Operation Example

The evolution of the Hadamard operation, $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, is given by $\hat{H}_H = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ and $|\phi\rangle$ is one qubit, $(z)_0\,|0\rangle + (z)_1\,|1\rangle$. So, $z(t) = \begin{pmatrix} (z)_0(t) \\ (z)_1(t) \end{pmatrix}$ and

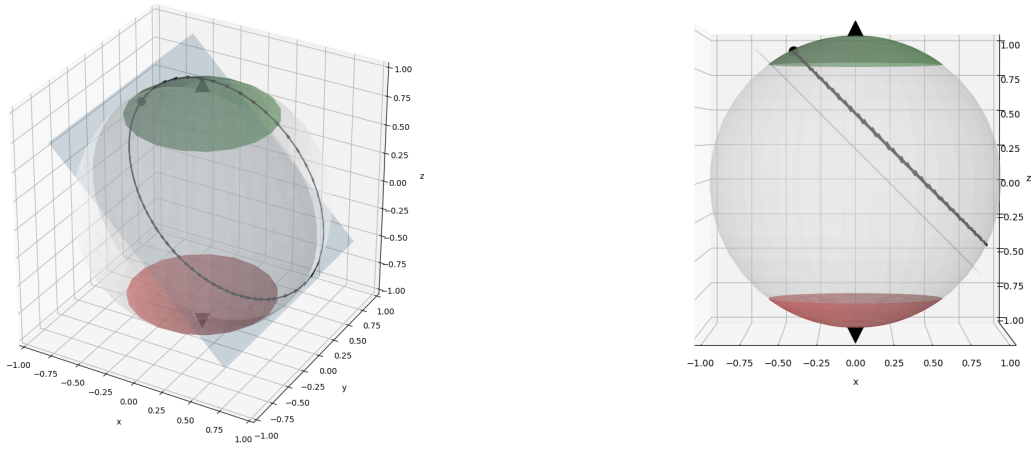
$$\dot{z} = -\mathrm{i}\hat{H}_H z = -\mathrm{i}\begin{pmatrix} (z)_0 + (z)_1 \\ (z)_0 - (z)_1 \end{pmatrix}.$$

The system evolves over the surface of the unit sphere, $\mathcal{Z} = \{z \in \mathbb{C}^2 : |(z)_0|^2 + |(z)_1|^2 = 1\}$. The initial set is defined as $\mathcal{Z}_0 = \{z \in \mathcal{Z} : |(z)_0|^2 \geq 0.9\}$ and the unsafe set as $\mathcal{Z}_u = \{z \in \mathcal{Z} : |(z)_0|^2 \leq 0.1\}$. This can be thought of as starting in a quantum state that is close to $|0\rangle$, *i.e.*, has a greater than 90% chance of being measured, and avoiding entering a state where $|0\rangle$ is unlikely, *i.e.*, has less than 10% chance of being measured. Note that the definitions of $\mathcal{Z}_0$ and $\mathcal{Z}_u$ are restricted by $\mathcal{Z}$, therefore $|(z)_1|^2 \leq 0.1$ and $|(z)_1|^2 \geq 0.9$ for $\mathcal{Z}_0$ and $\mathcal{Z}_u$ respectively. A barrier function computed by Algorithm 1 is

$$B(z) = \frac{11}{5} - 3(z)_0\overline{(z)_0} - (z)_0\overline{(z)_1} - \overline{(z)_0}(z)_1 - (z)_1\overline{(z)_1}.$$

[4] In Dirac notation, $(z)_j = \langle j|\phi\rangle$ and $\overline{(z)_j} = \langle\phi|j\rangle$.

[5] The Planck constant, $\hbar$, is set to 1 in the Schrödinger equation.



(a) Isometric view of system

(b) Top down view of system

Figure 4.2: System evolution on a Bloch sphere. The initial state of the system is $\sqrt{0.9}\,|0\rangle + \mathrm{i}\sqrt{0.1}\,|1\rangle$ (the black dot) and evolves according to the black line (in an anti-clockwise rotation with a period of $t = \pi$). The green surface around the north pole ($|0\rangle$) is the initial region, $\mathcal{Z}_0$, and the red surface around the south pole ($|1\rangle$) is the unsafe region, $\mathcal{Z}_u$. The blue surface is the plane of the barrier function when $B(z) = 0$, with $x < -z$ being the unsafe region.

By rearranging and using properties of the complex conjugate, we find that

$$B(z) = 2(\frac{1}{10} - |(z)_0|^2 + \frac{1}{2} - \mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\}).$$

The derivation is given in Appendix B.1.3. The first term of the barrier ($\frac{1}{10} - |(z)_0|^2$) acts as a restriction on how close to $|0\rangle$ as $|\phi\rangle$ evolves, whereas the second term ($\frac{1}{2} - \mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\}$) is a restriction on the phase of the quantum state. Next, it is checked if B is indeed a barrier certificate.

Proposition 4.4. *The system evolving according to Equation* (4.5.1)*, initial set* $(z)_0$ *and unsafe set* Z_u *is safe.*

The proposition is proved in Appendix B.1.4. A visualisation on a Bloch sphere representation of the example system and its associate barrier are given in Figure 4.2.

4.5.2 Phase Operation Example

The evolution of the phase operation $S = \begin{pmatrix} 1 & 0 \\ 0 & \mathrm{i} \end{pmatrix}$ is given by the Hamiltonian $\hat{H}_S = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ for a single qubit $(z)_0 |0\rangle + (z)_1 |1\rangle$. Thus, the evolution of the system for $z(t) = \begin{pmatrix} (z)_0(t) \\ (z)_1(t) \end{pmatrix}$ is

$$\dot{z} = -\mathrm{i} \begin{pmatrix} (z)_0 \\ -(z)_1 \end{pmatrix}. \tag{4.17}$$

Again, $\mathcal{Z}$ represents the unit sphere as described previously. Two pairs of safe and unsafe regions are given. The first pair, $\mathcal{Z}^1 = (\mathcal{Z}_0^1, \mathcal{Z}_u^1)$, is given by

$$\mathcal{Z}_0^1 = \{z \in \mathcal{Z} : |(z)_0|^2 \geq 0.9\}, \quad \mathcal{Z}_u^1 = \{z \in \mathcal{Z} : |(z)_0|^2 < 0.9 - err\};$$

and the second pair, $\mathcal{Z}^2 = (\mathcal{Z}_0^2, \mathcal{Z}_u^2)$, is given by

$$\mathcal{Z}_0^2 = \{z \in \mathcal{Z} : |(z)_1|^2 \geq 0.9\}, \quad \mathcal{Z}_u^2 = \{z \in \mathcal{Z} : |(z)_1|^2 < 0.9 - err\},$$

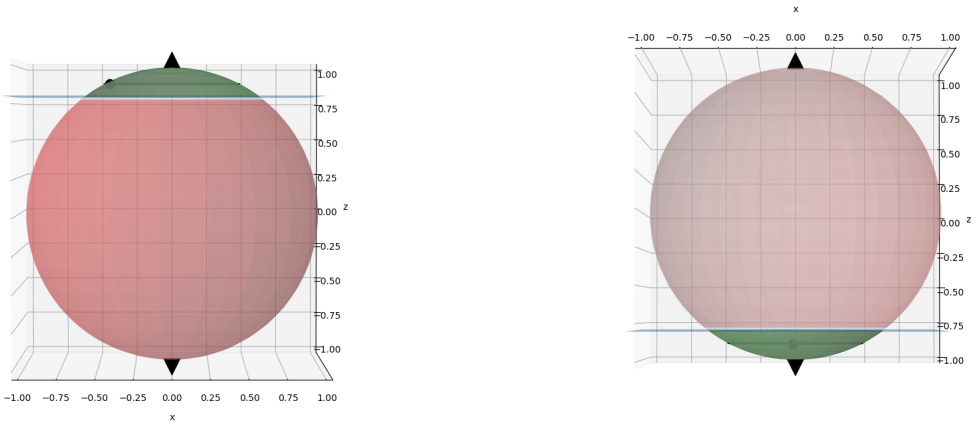
where, in both cases, $err = 0.01$ acts as an error term. The pair $\mathcal{Z}^1$ starts with a system that is close to the $|0\rangle$ state and ensures that the system remains close to $|0\rangle$. This is encoded by making the unsafe region be the states whose probability of measuring $|0\rangle$ below 0.9, hence err is used to provide a small buffer between the initial and unsafe region. The pair $\mathcal{Z}^2$ has similar behaviour with $|1\rangle$ as the respective state.

The system for each pair of constraints is considered safe by the following barriers computed by Algorithm 1:

$$B_1(z) = 0.9 - (z)_0\overline{(z)_0}, \quad B_2(z) = 0.9 - (z)_1\overline{(z)_1},$$

where B_1 is the barrier for $\mathcal{Z}^1$ and B_2 is the barrier for $\mathcal{Z}^2$.[6] The system with different pairs of regions can be seen on Bloch spheres in Figure 4.3. Again, both functions B_1 and B_2 are valid barrier certificates.

[6]These barriers can similarly be written using the Dirac notation.



(a) Evolution with initial and unsafe states $\mathcal{Z}^1$. The barrier at $B_1(z) = 0$ is a flat plane that borders $(z)_0{}^2 = 0.9$.

(b) Evolution with initial and unsafe states $\mathcal{Z}^2$. Similarly, $B_2(z) = 0$ is a flat plane that borders $(z)_1{}^2 = 0.9$.

Figure 4.3: State evolution of (4.17) demonstrated on a Bloch sphere.

Proposition 4.5. *The system given by Equation 4.17 with the set of initial states* $\mathcal{Z}_0^1$ *and the unsafe set* $\mathcal{Z}_u^1$ *is safe.*

Proposition 4.6. *The system given by Equation 4.17 with the set of initial states* $\mathcal{Z}_0^2$ *and the unsafe set* $\mathcal{Z}_u^2$ *is safe.*

The proofs are provided in Appendix B.1.5, they are similar to each other and to the proof of Proposition 4.4. These barriers give bounds on how the system evolves, *i.e.*, the system must only change the phase of the system and not the amplitude. This can be applied in general by combining barriers to show how a (disturbed) system is restricted in its evolution.

4.5.3 Controlled-NOT Operation Example

The final example considered is the controlled-NOT (CNOT) operation acting on two qubits: a control qubit, $|\phi_c\rangle$, and a target qubit, $|\phi_t\rangle$, with the full quantum state being $|\phi_c\phi_t\rangle$. The CNOT operation performs the NOT operation on a target qubit ($|0\rangle \to |1\rangle$ and $|1\rangle \to |0\rangle$) if the control qubit is set to $|1\rangle$ and does nothing if the control qubit is set to $|0\rangle$. The CNOT operation and its associated Hamiltonian

are given by

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad \hat{H}_{\text{CNOT}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix}.$$

The system $z(t) = (z_j(t))_{j=0,\ldots,3}$ evolves according to

$$\dot{z} = -\mathrm{i} \begin{pmatrix} 0 \\ 0 \\ (z)_2 - (z)_3 \\ -(z)_2 + (z)_3 \end{pmatrix}.$$

This system evolves over $\mathcal{Z} = \{z \in \mathbb{C}^4 : \sum_{j=0}^{3} |(z)_j|^2 = 1\}$. Using this as the system, various initial and unsafe regions can be set up to reason about the behaviour of the CNOT operation.

Control in $|0\rangle$

Here, consider the following initial and unsafe regions

$$\mathcal{Z}_0 = \{z \in \mathbb{C}^4 : |(z)_0|^2 \geq 0.9\},$$
$$\mathcal{Z}_u = \{z \in \mathbb{C}^4 : |(z)_0|^2 \leq 0.9 - err\},$$

where, again, $err = 0.01$. The initial set, $\mathcal{Z}_0$, encapsulates the quantum states that start in the $|00\rangle$ state with high probability and $\mathcal{Z}_u$ captures the states that are not in the initial region. Note that the unsafe system can also be specified with $|(z)_1|^2 + |(z)_2|^2 + |(z)_3|^2 \geq 0.1 + err$ due to the properties of the unit sphere. These regions capture the behaviour that the quantum state should not change much when the control qubit is in the $|0\rangle$ state.

Using Algorithm 1, the barrier $B(z) = 0.9 - (z)_0\overline{(z)_0}$ can be generated to show that the system is safe.

A similar example can be considered where the initial state $|00\rangle$ is replaced with $|01\rangle$ instead (replace $(z)_0$ with $(z)_1$ in $\mathcal{Z}_0$ and $\mathcal{Z}_u$). The behaviour that the state of the system should not change much is still desired; the function $B(z) = 0.9 - (z)_1\overline{(z)_1}$ is computed as a barrier to show this behaviour is met.

Control in $|1\rangle$

Now consider when the initial region has the control qubit near the state $|1\rangle$. The following regions are considered:

$$\mathcal{Z}_0 = \{z \in \mathbb{C}^4 : |(z)_2|^2 \geq 0.9\},$$
$$\mathcal{Z}_u = \{z \in \mathbb{C}^4 : |(z)_0|^2 + |(z)_1|^2 \geq 0.1 + err\},$$

with $err = 0.01$. This system starts close to the $|10\rangle$ state and the evolution should do nothing to the control qubit, *i.e.*, the quantum state may change the amplitudes of $|10\rangle$ and $|11\rangle$, but may not change the amplitudes of $|00\rangle$ or $|01\rangle$ much (with err acting as a buffer again). Note that the specified behaviour does not captures the NOT behaviour on the target qubit. Algorithm 1 considers this system safe by outputting the barrier certificate $B(z) = 0.9 - (z)_2\overline{(z)_2} - (z)_3\overline{(z)_3}$. This is also the barrier if the system were to start near the $|11\rangle$ state instead.

4.6 Conclusions

In this Chapter, the theory of barrier certificates was extended to handle complex variables. It was then shown how one can automatically generate simple complex-valued barrier certificates using polynomial functions and linear programming techniques. Finally, the application of the developed techniques was studied to check properties of time-independent quantum systems.

There are numerous directions for this research to take. In particular, one can consider (quantum) systems that are time-dependent, have a control or noise component, or are discrete-time, *i.e.*, quantum circuits. The main difficulty is that while it has been shown that barrier certificates can be extended to complex domain, the generation process provided in this Chapter is very specific to the system given. The classical methods for handling real versions of the extended systems, such as [4, 13, 93, 125, 144], need to be investigated for possible extensions to handle the complex versions. Data-driven approaches for generating barrier certificates, *e.g.*, [135], based on measurements of a quantum system can also be considered. A final challenge to consider is how to verify large quantum systems. Techniques, such as Trotterization, allow Hamiltonians to be simulated either by simpler Hamiltonians of the same size or of lower dimension. How barrier certificates can ensure safety of such systems is a route to explore.

This Chapter has demonstrated the potential for barrier certificates to be used for verifying properties of quantum gates. The next Chapter provides a further investigation into the generation process for barrier certificates and looks at how this technique can be used to verify quantum circuits.

Chapter 5

Verification of Quantum Circuits using Discrete Barrier Certificates

This Chapter further explores the usage of barrier certificates to verify properties in quantum systems first developed in Chapter 4. The standard generation process for barrier certificates is extended to handle different types of quantum systems, with particular focus in this Chapter on looking at the quantum circuit level in the quantum stack.

5.1 Introduction

In Chapter 4, the notions of barrier certificates for complex systems and a method for producing a barrier certificate was produced. The linear programming technique used is flawed in that it is specific to the system worked with. If a different type of dynamical system were considered, then the same technique could not be used to generate a barrier. Thus, while the definition of barrier certificate has been extended to the complex domain, the method to generate barrier certificates was not extended.

In the literature of barrier certificates, a common technique known as Sum of Squares (SOS) is used as a generation technique. Commonly, a set of barrier certificate constraints is converted into equivalent SOS equations. These equations are then given to a SOS solver such as SOSTOOLS [118], which is a tool built on semidefinite program solvers [119], in order to find a valid barrier certificate. The main reason SOS is used as a generation technique is that the conversion of constraints to SOS equations can be applied to almost any barrier certificate scheme.

Additionally, the availability and efficiency of SOS solvers make it easy to implement the technique.

In this Chapter, it will be demonstrated how the technique of using SOS can be generalised to the complex domain. The generalisation of barrier certificates into the complex domain from the previous Chapter is used and the SOS technique is adapted by making use of Hermitian Sum of Squares (HSOS), a complex extension of SOS [126]. This technique is applied to a discrete-time dynamical system by analysing how barrier certificates can be applied to quantum circuits.

5.2 Quantum Circuits as Dynamical Systems

Firstly, the dynamical system considered to represent quantum circuits is introduced.

Definition 5.1 (Discrete-time complex-space system)**.** A discrete-time complex-space system is a tuple $S = (\mathcal{Z}, \mathcal{Z}_0, F, f)$, where

- $\mathcal{Z} \subseteq \mathbb{C}^n$ is the continuous (complex-valued) state space;
- $\mathcal{Z}_0 \subseteq \mathcal{Z}$ is the set of initial states;
- F is a *finite* set of functions that contain all the possible dynamics the system can perform;
- and $f : \mathbb{N}_{\geq 0} \to F$ is a function that assigns at each time step the dynamics of the system.

At each time step, t, the state of the system is z_t and the dynamics of the system is defined by

$$z_{t+1} = f(t)(z_t) = f_t(z_t).$$

Conversion of quantum circuits into the dynamical system given above is straightforward. For an n-qubit system, quantum states are restricted to points on the unit circle of $\mathbb{C}^{2^n}$, *i.e.*, $\mathcal{Z} = \{z \in \mathbb{C}^{2^n} : \sum_j |(z)_j|^2 = 1\}$. Assume that for any quantum circuit, there is some initial state $|\phi\rangle = \sum_{j=0}^{2^n} (z)_j |j\rangle$ or a set of initial points that can be chosen from. Again, as discussed in Section 4.1, this may be from taking an region in the state space where the initial states has a high chance of occurring. One can include error in the initial state through noise that occurs in preparing the initial state. Thus, $\mathcal{Z}_0$ can be specified by the user depending on how noisy preparing the

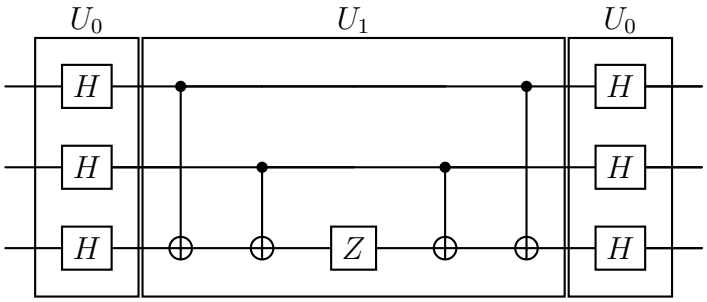


Figure 5.1: Quantum circuit with 3 unitary operations consisting of grouped gates.

initial state is. Finally, say the quantum circuit is of the form $U = U_{k-1} \dots U_1 U_0$, where $U_0, U_1, \dots$ are unitary operations. Then $F = \{U_t\}_{t=0,\dots,k-1}$, and $f(t) = U_t$ for $t < k$ and $f(t) = I_{2^n}$ otherwise.

There is flexibility in what level the unitary operations are chosen, since one can let unitary operations be individual gates or they can be a combination of several gates at different depths. For example, the quantum circuit in Figure 5.1 is equivalent to the system $S = (\mathcal{Z}, \mathcal{Z}_0, F, f)$ where

$$\mathcal{Z} = \{z \in \mathbb{C}^8 : \sum_j |(z)_j|^2 = 1\};$$
$$\mathcal{Z}_0 = \{z \in \mathcal{Z} : |(z)_0|^2 \geq 0.99, \mathrm{Im}\{(z)_0\} = 0\};$$
$$F = \{U_0, U_1\}; \text{ and}$$
$$f_t(z) = \begin{cases} U_0 z & \text{for } t = 0, 2, \\ U_1 z & \text{for } t = 1, \\ z & \text{otherwise;} \end{cases}$$

and the actual unitary transformation corresponding to U_0 and U_1 can be easily deduced from Figure 5.1. In this example, $\mathcal{Z}_0$ corresponds to the set of initial quantum states having $|0\rangle$ be measured with at least 99% probability and no complex phase. This corresponds with the errors that devices can face when preparing a quantum state.

5.3 Complex Discrete-Time Barrier Certificates

Next, the notion of safety for the complex system is reintroduced.

Definition 5.2 (Safety)**.** Let $S = (\mathcal{Z}, \mathcal{Z}_0, F, f)$ be a discrete-time complex-space

system and $\mathcal{Z}_u \subseteq \mathcal{Z}$ denote the unsafe set. Then S is safe if for all $z_0 \in \mathcal{Z}_0$, $z_t \notin \mathcal{Z}_u$ for any $t \in \mathbb{N}_{\geq 0}$.

To solve this problem of safety using barrier certificates the ideas behind barrier certificates for hybrid systems [124] and k-inductive barrier certificates [8] are combined, adapting them to our dynamical system as well as the complex domain. Additionally, the certificate to handle different dynamics that occur during evolution is adapted.

The barrier certificates for [124] are capable of handling systems with dynamics that occur on both a continuous and discrete space. For our system $S = (\mathcal{Z}, \mathcal{Z}_0, F, f)$, these spaces corresponds to our complex space, $\mathcal{Z}$, and the time steps, $\mathbb{N}_{\geq 0}$, respectively. The k-inductive barrier certificates introduced in [8] work on the basis that the system is allowed to evolve slightly towards the unsafe region but after k steps the system will be further away from the unsafe region than before taking the k steps.

Theorem 5.1 (k-Inductive Hybrid Barrier Certificate)**.** *Let $S = (\mathcal{Z}, \mathcal{Z}_0, F, f)$ and the unsafe set be $\mathcal{Z}_u$. Suppose there exists a barrier certificate: a collection of functions $\{B_t(z)\}_{t\in\mathbb{N}_{\geq 0}}$, where $B_t(z) \in \mathbb{R}$ for all $t \geq 0$, $z \in \mathcal{Z}$; and constants $k \geq \mathbb{Z}_{\geq 1}$, $\epsilon, \gamma \in \mathbb{R}_{\geq 0}$ and $d > k(\epsilon + \gamma)$ that satisfy the following equations*

$$B_0(z) \leq 0, \forall z \in \mathcal{Z}_0; \tag{5.1a}$$

$$B_t(z) \geq d, \forall z \in \mathcal{Z}_u, \forall t \in \mathbb{N}_{\geq 0}; \tag{5.1b}$$

$$B_t(f_t(z)) - B_t(z) \leq \epsilon, \forall z \in \mathcal{Z}, t \in \mathbb{N}_{\geq 0}; \tag{5.1c}$$

$$B_{t+1}(z) - B_t(z) \leq \gamma, \forall z \in \mathcal{Z}, t \in \mathbb{N}_{\geq 0}; \tag{5.1d}$$

$$B_{t+k}(f_{t+k-1}(\dots f_{t+1}(f_t(z)))) - B_t(z) \leq 0, \forall z \in \mathcal{Z}, t \in \mathbb{N}_{\geq 0} \tag{5.1e}$$
$$\text{such that } t = rk \text{ for } r \in \mathbb{N}_{\geq 0}.$$

Then the safety of S with respect to $\mathcal{Z}_u$ is guaranteed.

Proof. The theorem is proved by contradiction. Let the system have a k-inductive hybrid barrier certificate, *i.e.*, $\{B_t(z)\}, k, \epsilon, \gamma, d$ that satisfy the conditions given in (5.1), and assume that the system is not safe.

Let $(z_0, \dots, z_T)$ be a trace, where $z_{t+1} = f_t(z_t)$, that reaches an unsafe state, $z_T \in Z_u$. Let $T = t + m$, where $t, m \in \mathbb{N}_{\geq 0}$, $t = rk$ for some $r \in \mathbb{N}_{\geq 0}$ and $m < k$. For the initial state and unsafe state, $B_0(z_0) \leq 0$ and $B_{t+m}(z_{t+m}) \geq d$ respectively.

Using (5.1c) and (5.1d), then

$$\begin{aligned}
&B_{t+m}(z_{t+m}) = B_{t+m}(f_{t+m-1}(z_{t+m-1})) \\
&\leq B_{t+m-1}(f_{t+m-1}(z_{t+m-1})) + \gamma \\
&\leq B_{t+m-1}(z_{t+m-1}) + \epsilon + \gamma \\
&\leq \dots \\
&\leq B_t(z_t) + m(\epsilon + \gamma) \leq B_t(z_t) + k(\epsilon + \gamma).
\end{aligned}$$

Using (5.1e) and induction, $B_t(z_t) \leq B_{t-k}(z_{t-k}) \leq \cdots \leq B_0(z_0)$. Therefore,

$$B_{t+m}(z_{t+m}) \leq B_0(z_0) + k(\epsilon + \gamma) < d.$$

This is a contradiction to (5.1b) and therefore $z_T \notin Z_u$. Therefore, the system S is safe. □

Remark 5.1. Note that with certain values of k, ϵ and γ, the number of equations to satisfy can be reduced. For example, if $k = 1$ and $\gamma = 0$, then Equation (5.1c) is implied through Equations (5.1d) and (5.1e) for any value of ϵ.

One of the challenges with Theorem 5.1 is that the functions, B_t, have complex variables but are required to return a real value. Therefore, the functions are restricted to specific classes that can be easily defined and will be useful for finding a barrier. To do this, the notion of conjugate-flattening is used as given in Definition 4.4. In addition, a definition for polynomials is used.

Definition 5.3 (Conjugate-flattening Polynomial)**.** A conjugate-flattening function, $P(z) = p(z, \overline{z}) \in \mathbb{R}$, is a conjugate-flattening polynomial if $p(z, u)$ is a polynomial with complex variables and coefficients.

These definitions can be used in Theorem 5.1 for the collection of functions to ensure that $B_t(z) \in \mathbb{R}$ for any $t \geq 0, z \in \mathcal{Z}$. Additionally, being able to differentiate between conjugate-flattening functions and polynomials will be useful when in the discussion of the generation of barrier certificates in Section 5.4.3.

5.4 Computation of Barrier Certificates through Hermitian Sum of Squares

There exist several approaches for computing a barrier certificate given the specification and dynamics of a system. Most approaches are automatic and include the usage of neural networks and SMT (Satisfiability Modulo Theory) solvers [1, 60] to compute a barrier. However, the standard approach is to make use of sum of squares (SOS) optimisation in order to find a suitable barrier [119]. Unlike other techniques, SOS optimisation provides an efficient method whilst remaining theoretically sound. In this Section, the SOS technique is adapted to complex variables.

5.4.1 Sum of Squares for Complex Numbers

A polynomial with real variables and coefficients, $p(x)$, is referred to as a sum of squares (SOS) if $p(x) = \sum_k p_k(x)^2$, where p_k are polynomials (of any degree) for $k \geq 1$. There are two reasons that SOS polynomials are useful for generating barrier certificates:

i SOS polynomials are real and positive. This makes SOS polynomials useful for ensuring generated functions obey theorems for safety (such as Theorem 5.1).

ii There is an equivalence between SOS polynomials and positive semidefinite matrices: $p(x)$ is a SOS *iff* there exists a positive semidefinite matrix Q such that $p(x) = v(x)^\intercal Q v(x)$ where $v(x)$ is a vector of monomial terms. Such matrices can be found using semidefinite programming [119].

Combining these two properties gives us an efficient and sound method for finding real polynomial barriers.

A function with complex variables and coefficients, p, may produce complex values, *i.e.*, $p : \mathbb{C}^n \to \mathbb{C}$. This means that p may not produce only positive, or even real, values. Thus, a method is needed to ensure $p(z)$ is real and positive for any $z \in \mathbb{C}^n$. This can be done by considering a variation of sum of squares.

Definition 5.4 (Hermitian Sum of Squares [126])**.** A complex function, $p(z) : \mathbb{C}^n \to \mathbb{C}$ is a Hermitian sum of squares (HSOS) if $p(z) = \sum_k p_k(z)\overline{p_k(z)}$ where $p_k : \mathbb{C}^n \to \mathbb{C}$ are complex polynomials and $k \geq 1$.

Remark 5.2. Note that the main difference between HSOS and SOS is that the conjugate ($\overline{z}$) is used for HSOS. If a HSOS polynomial is restricted to have only real variables and coefficients, then the standard SOS definition can be used since $p_k(x)\overline{p_k(x)} = p_k(x)^2$.

The properties previously described for SOS polynomials hold for HSOS polynomials with some slight modifications for the complex domain.

Proposition 5.2. *Let $p(z)$ be a HSOS. Then $p(z) \in \mathbb{R}$ for all $z \in \mathbb{C}^n$ and $p(z)$ is positive ($p(z) \geq 0$).*

Proof.

$$p(z) = \sum_k p_k(z)\overline{p_k(z)} = \sum_k |p_k(z)|^2$$

It is clear that the right-hand side is a real value and is also positive. □

Additionally, there is a notion of positive semidefinite matrices for complex vectors as well.

Definition 5.5 (Positive semidefinite [91])**.** A complex $n \times n$ matrix Q is (Hermitian) positive semidefinite if $z^\dagger Q z \geq 0, \forall z \in \mathbb{C}^n$.

With these definitions, an equivalence between HSOS polynomials and (Hermitian) positive semidefinite matrices can be shown.

Proposition 5.3. *Let $p(z) : \mathbb{C}^n \to \mathbb{C}$ be a conjugate-flattening polynomial of degree $2d$. Then, $p(z)$ is a HSOS iff there exists a (Hermitian) positive semidefinite matrix, Q, such that $p(z) = v(z)^\dagger Q v(z)$, where $v(z)$ is a vector of all monomials of degree less than d.*

Proof. Firstly, assume that $p(z)$ is a HSOS. Therefore, $p(z) = \sum_k p_k(z)\overline{p_k(z)}$ where p_k is a polynomial. Since p has degree $2d$ and $\deg(p_k(z)) = \deg(\overline{p_k(z)})$, then each p_k is of degree up to d. Therefore, $p_k(z) = B_k v(z)$, where B_k is a (row) vector of complex coefficients.

Let B be the matrix whose rows are B_k and elements are $(B)_{kl}$.

By expanding $p(z)$, then

$$\begin{aligned}
p(z) &= \sum_k B_k v(z) \overline{B_k v(z)} \\
&= \sum_k \sum_{l,m} (B_k v(z))_l (\overline{B_k v(z)})_m \\
&= \sum_k \sum_{l,m} (B)_{kl} v(z)_l (\overline{B})_{km} (\overline{v(z)})_m \\
&= \sum_k \sum_{l,m} (B^\dagger)_{mk} (B)_{kl} (v(z))_l (\overline{v(z)})_m \\
&= \sum_{l,m} (\sum_k (B^\dagger)_{mk} (B)_{kl}) (v(z))_l \overline{(v(z))_m} \\
&= \sum_{l,m} (B^\dagger B)_{ml} (v(z))_l \overline{(v(z))_m} \\
&= v(z)^\dagger (B^\dagger B) v(z).
\end{aligned}$$

Since B is a complex matrix, then $Q = B^\dagger B$ is positive semi-definite.

Now, assume that $p(z) = v(z)^\dagger Q v(z)$ with Q being positive semi-definite, then there exists a complex matrix B such that $Q = B^\dagger B$. Simply by performing the summation in reverse gives us that $p(z) = \sum_k p_k(z)\overline{p_k(z)}$ where $p_k(z) = B_k v(z)$. □

Since properties of SOS polynomials are shared by HSOS polynomials, computation techniques can be used to find barrier certificates for real systems and they can be adapted for complex systems.

5.4.2 Semi-algebraic Sets

In the usual method for computing barrier certificates, the sets used are assumed to be semi-algebraic [29], *i.e.*, they can be described using vectors of polynomials. For example, if $X \subseteq \mathbb{R}^n$ is semi-algebraic, it can be written as

$$X = \{x \in \mathbb{R}^n : g(x) \geq \mathbf{0}\},$$

where $g(x)$ is a vector of polynomials and the ordering is applied element-wise ($g_j(x) \geq 0$ for all j).

This definition does not immediately hold for complex variables, since complex values cannot be ordered. Thus, semi-algebraic sets for complex numbers must be defined by a vector of conjugate-flattening polynomials, *i.e.*, $g(z) \in \mathbb{R}^n$ for all

$z \in \mathbb{C}^n$. Then write that $Z \subseteq \mathbb{C}^n$ is (complex) semi-algebraic if

$$Z = \{z \in \mathbb{C}^n : g(z) \geq \mathbf{0}\};$$

where g is a vector of conjugate-flattening polynomials and, again, the ordering is applied element-wise. From now on, assume that the sets $\mathcal{Z}, \mathcal{Z}_0$, and $\mathcal{Z}_u$ are (complex) semi-algebraic and use the vectors g_I, g_0, and g_u to describe their elements respectively.

5.4.3 HSOS Equations for Barrier Certificate Constraints

Now the HSOS equations that need to be computed to get a function that satisfies the constraints of a barrier certificate can be stated.

Lemma 5.4. *Let $S = (\mathcal{Z}, \mathcal{Z}_0, F, f)$; g_I, g_0, g_u be given vectors of conjugate-flattening polynomials describing $\mathcal{Z}, \mathcal{Z}_0$, and $\mathcal{Z}_u$, respectively. Suppose there exists a collection of (conjugate-flattening) polynomials $\{B_t(z) = b_t(z, \overline{z})\}_{t \in \mathbb{N}_{\geq 0}}$; positive numbers $k \in \mathbb{N}_{\geq 0}$, $\epsilon, \gamma \in \mathbb{R}_{\geq 0}$, $d > k(\epsilon + \gamma)$; and vectors of Hermitian sum of squares $\lambda_{U;t}(z)$, $\lambda_{Init}(z)$, $\lambda_t(z)$, $\lambda_{t,t'}(z)$ and $\hat{\lambda}_t(z)$ such that the expressions:*

$$-B_0(z) - \lambda_{Init}(z)^\intercal g_0(z); \tag{5.2a}$$

$$B_t(z) - \lambda_{U;t}(z)^\intercal g_u(z) - d, \forall t \in \mathbb{N}_{\geq 0}; \tag{5.2b}$$

$$-B_t(f_t(z)) + B_t(z) - \lambda_t(z)^\intercal g_I(z) + \epsilon, \forall t \in \mathbb{N}_{\geq 0}; \tag{5.2c}$$

$$-B_{t+1}(z) + B_t(z) - \lambda_{t,t+1}(z)^\intercal g_I(z) + \gamma, \forall t \in \mathbb{N}_{\geq 0}; \tag{5.2d}$$

$$-B_{t+k}(f_{t+k-1}(\dots f_{t+1}(f_t(z)))) + B_t(z) - \hat{\lambda}_t(z)^\intercal g_I(z), \quad \forall t \in \mathbb{N}_{\geq 0} \text{ such that } t = rk \text{ for } r \in \mathbb{N}_{\geq 0}; \tag{5.2e}$$

are Hermitian sum of squares. Then B satisfies Theorem 5.1 and the safety of S is guaranteed.

Proof. It is shown that expression (5.2b) being HSOS implies that B satisfies constraint (5.1b) of Theorem 5.1. Firstly, note that since $\lambda_{U;t}$ is a vector of HSOSs, then $\lambda_{U;t}(z) \geq 0, \forall z \in \mathcal{Z}$. Additionally, for $z_u \in \mathcal{Z}_u$, we have that $g(z_u) \geq 0$. Therefore, $\lambda_{U;t}(z_u)^\intercal g(z_u) \geq 0$ and further that $B_t(z_u) - d \geq 0$ as Equation (5.2b) is HSOS. Thus, constraint (5.1b) is satisfied.

Similar reasoning can be applied to the other expressions in (5.2) and their

Algorithm 2: Finding a barrier certificate using HSOS

Input: Constants: $k, \delta, \epsilon, \gamma$; discrete-time complex-space dynamical system: $S = (\mathcal{Z}, \mathcal{Z}_0, F, f)$; vectors describing associated semi-algebraic sets: g_I, g_0, g_u

1 Set $d > k(\epsilon + \gamma)$.
2 Setup symbolic function: Set $B(z) = b(z, \overline{z})$ to be a parameterized conjugate-flattening polynomial of degree up to δ.
3 Define symbolic lambda polynomials: parameterized conjugate-flattening polynomials $\lambda_{\text{Init}}(z)$, $\lambda_{U;t}(z)$, $\lambda_t(z)$, $\lambda_{t,t'}(z)$ and $\hat{\lambda}_t(z)$ for each t, t'.
4 Add to HSOS solver: Add the various λ polynomials and the equations in (5.2) as HSOS constraints to the HSOS solver.
5 Run the HSOS solver.
6 **if** *feasible* **then**
7 | **return** $B(z)$ with coefficients set to the values from the HSOS solver
8 **else**
9 | **error:** barrier does not exist for system with constants given
10 **end**

respective counterparts in (5.1). This results in the conditions in Theorem 5.1 being satisfied and, therefore, safety is guaranteed. □

In the same way HSOS is related to SOS, Lemma 5.4 is similar to theorems and lemmas for real dynamical systems, with the major difference being the usage of HSOS equations rather than SOS equations to allow us to use complex variables.

Remark 5.3. Note that the barrier generated is a conjugate-flattening polynomial, but the definition of k-inductive hybrid barrier certificates given in Theorem 5.1 does not require the barrier function to be a polynomial. Simply by restricting the barrier to a conjugate-flattening polynomial allows us to easily adapt the SOS methods to the complex domain while retaining safety of the system.

5.4.4 Algorithm for finding a Barrier Certificate

An algorithm is provided for finding a barrier certificate for a discrete-time complex-space system $S = (\mathcal{Z}, \mathcal{Z}_0, F, f)$, given in Algorithm 2. In order to compute a barrier, consider $B(z)$ as a δ-degree conjugate-flattening polynomial. The coefficients of B are parameterised as a vector of complex values $\beta \in \mathbb{C}^\kappa$, where $\kappa \approx \sum_{j=0}^{\delta}(2n)^j = \frac{1-(2n)^{\delta+1}}{1-2n}$ is the number of coefficients in a δ-degree polynomial with $2n$ variables ($2n$ comes from the fact that z and $\overline{z}$ terms need to be considered). Then $B(z)$ is

input with its parameterised coefficients β into the equations given in (5.2). These equations are then given to an appropriate HSOS solver, which attempts to find appropriate values for β.

Remark 5.4. For specific hyper-parameters a barrier may not be produced, hence why the HSOS may return *infeasible* and there is an error in Algorithm 2 (line 9). This can be due to several reasons, for example the barrier may have a degree that is higher than δ or the values of ϵ and γ need to be changed. The system may simply be unsafe. Alternatively, the required barrier for the system and unsafe region given may need to be non-polynomial, in which case the HSOS technique cannot be used to generate a barrier. The work in [3] shows that a (real) continuous system with polynomial dynamics is safe but does not admit a polynomial Lyapunov function, *i.e.*, a barrier certificate.

5.5 Case Studies

In this Section, it is shown how the theory that has been developed can be used in practice. Details are provided of the implementation of a toy HSOS solver and demonstrate its usage for barrier certificates on some example quantum circuits. Note that in the quantum circuit setting, $\mathcal{Z} = \{z \in \mathbb{C}^{2^n} : \sum_{j=0}^{2^n-1} |(z)_j|^2 = 1\}$ as discussed in Section 5.2.

5.5.1 Experimental Setup

For these experiments, the dynamical system $f_t(z)$; functions describing the semi-algebraic sets: g_I, g_0, g_u; parameters k, ϵ, γ from Lemma 5.4; and the degree of the barrier function, represented by $\deg(B)$, are provided.

HSOS Solver

Standard SOS solvers, such as SOSTOOLS [118], use semidefinite programs to compute the polynomials in the summation while the solver handles the conversion to and from SOS equations. Unfortunately, there does not exist any HSOS solver or even a complex semidefinite optimiser that makes use of the speedups shown in [81], as current semidefinite solvers that allow for complex programs convert them to equivalent real programs.

With the equivalence shown between HSOS and positive semidefinite matrices in Proposition 5.3, a simple HSOS solver has been developed in Python that converts the HSOS equations into (complex) semidefinite programs. The SymPy [110] package is used to represent polynomials, and the semidefinite programs are solved using PICOS [133], a Python interface to relevant semidefinite solvers, giving appropriate coefficients for the HSOS equation.[1]

Verification

To ensure the correctness of the barrier certificate generated, SMT solvers are used to formally check that the generated barrier certificate satisfies Equations (5.1). For a collection of barriers, $\{B_t(z)\}_t$ that are generated, the equations in (5.1) are encoded as proof obligations and a counter-example is searched for. Additionally, the barrier is encoded to produce real values.

The equations are encoded by removing the existential quantifier ($\forall$), replacing set membership ($\in$) with satisfaction of the relevant semi-algebraic polynomial vector (g), and negating the rest of the statement. For example, Equation (5.1a) ($B_0(z) \leq 0, \forall z \in \mathcal{Z}_0$) is encoded by the proof obligation $B_0(z) > 0 \wedge g_0(z) \geq 0$ (again the ordering of $g_0(z)$ is applied element-wise).

The Z3 [58] Python package is made use of to setup the proof obligations, and dReal [78] to verify them.[2] If a proof obligation is *unsatisfiable*, then that means the relevant equation is true. However, if a *satisfiable* or *δ-satisfiable* (in the case of dReal) result is received, then a counter-example is found and so the relevant equation is not satisfied (in the case of dReal, being a δ-sat solver, the equation may not be satisfied). Each equation and barrier function is checked in a separate call to the SMT solver as this makes it faster to check each equation and the reason why a certain barrier fails can be found if a counter-example is received.

Device Details

The experiments were performed on a laptop with an Intel(R) Core(TM) i5-10310U CPU @ 1.70GHz x 8 cores processor and 16GB of RAM using Ubuntu 20.04.3 LTS.

[1] The complex semidefinite programs are converted into real programs by PICOS.

[2] Functionality to verify using Z3 is included as well.

Code Availability

The code for Algorithm 2, the conversion of HSOS to semidefinite programs and verifying generated barrier certificates is available at:
`https://github.com/marco-lewis/discrete-quantum-bc`.

5.5.2 Phase (Z) Gate

A simple example is considered, the Z-gate introduced in Section 2.1.1. Consider a simple circuit with one qubit, $\mathcal{Z} = \{z \in \mathbb{C}^2 : |(z)_0|^2 + |(z)_1|^2 = 1\}$, that applies the Z-gate repeatedly, with dynamics described by

$$\forall t \in \mathbb{N}_{\geq 0}, f_t(z) = Zz.$$

The initial and unsafe region are specified as

$$\mathcal{Z}_0 = \{z \in \mathcal{Z} : |(z)_0|^2 \geq 0.9\}, \text{ and}$$
$$\mathcal{Z}_u = \{z \in \mathcal{Z} : |(z)_0|^2 \leq 0.9 - err\},$$

where $err = 0.1$. These regions can be thought of quantum states with a certain state being measured with a set probability. Here, $\mathcal{Z}_0$ is the region where $|0\rangle$ is likely to be measured with at least 90% probability (similarly $\mathcal{Z}_u$, $|1\rangle$ and 20% respectively due to properties of $\mathcal{Z}$.). These sets capture the behaviour of a qubit not moving too far from the initial region as it evolves provided (through the buffer space provided by err).

By using Algorithm 2 with $k = 1, \epsilon = 0.01, \gamma = 0.01$; the barrier (rounded to 3 d.p.)

$$B(z) = 4.453 - 0.848{(z)_0}^2 - 3.871(z)_0\overline{(z)_0} + 2.274(z)_1\overline{(z)_1} - 0.848\overline{(z)_0}^2$$

separates the two regions and ensures safety.

5.5.3 NOT (X) Gate

In this example, consider a X-gate that is being applied repeatedly to a set of n qubits (as a reminder $\mathcal{Z} = \{z \in \mathbb{C}^{2^n} : \sum_{j=0}^{2^n-1} |(z)_j|^2 = 1\}$). The dynamical system is described by

$$\forall t \in \mathbb{N}_{\geq 0}, f_t(z) = X^{\otimes n}z,$$

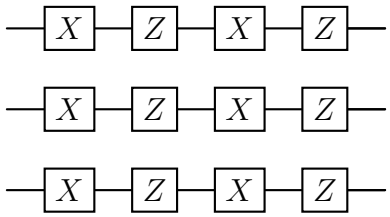


Figure 5.2: Quantum circuit for alternating between X and Z gates with 3 qubits.

with the initial and unsafe regions,

$$\begin{aligned} \mathcal{Z}_0 &= \{z \in \mathcal{Z} : \frac{1}{2^n} - err \leq |(z)_j|^2 \leq \frac{1}{2^n} + err\}, \text{ and} \\ \mathcal{Z}_u^p &= \{z \in \mathcal{Z} : |(z)_p|^2 \geq \frac{1}{2^n} + 2err\}, \end{aligned} \tag{5.3}$$

where $err = \frac{1}{10^{n+1}}$ and $p \in \{0, \ldots, 2^n - 1\}$. The initial region represents the set of quantum states that is close to the uniform superposition of quantum states, $|+\rangle^n = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} |j\rangle$. The unsafe region determines that the system should avoid exiting the region for some target state, $|p\rangle$. The err term provides an error for starting in the initial region, as well as providing a small space the system may evolve into before hitting the unsafe region.

The barrier certificate generated for a system with 2 qubits ($n = 2$) and $|0\rangle$ as a target state ($p = 0$) system with $k = 2, \epsilon = 0.01, \gamma = 0$; is

$$\begin{aligned} B(z) = &- 51.56 + 204.89(z)_0(\overline{z})_0 + 0.03(z)_1(\overline{z})_1 + 0.03(z)_2(\overline{z})_2 + 0.03(z)_3(\overline{z})_3 \\ &- 0.03(z)_1(\overline{z})_2 - 0.03(z)_2(\overline{z})_1 - 0.03(z)_0(\overline{z})_3 - 0.03(z)_3(\overline{z})_0, \end{aligned}$$

with which safety is ensured.

5.5.4 Alternating X and Z Gates

Here a quantum circuit is considered that alternates between the two operations used in previous examples. The dynamical system is described by

$$\forall t \in \mathbb{N}_{\geq 0}, f_t(z) = \begin{cases} X^{\otimes n} z, & \text{if } t \text{ is even;} \\ Z^{\otimes n} z, & \text{otherwise.} \end{cases}$$

The associated quantum circuit, with $n = 3$, is given in Figure 5.2.

The behaviour is specified to be such that the initial region is near one of the

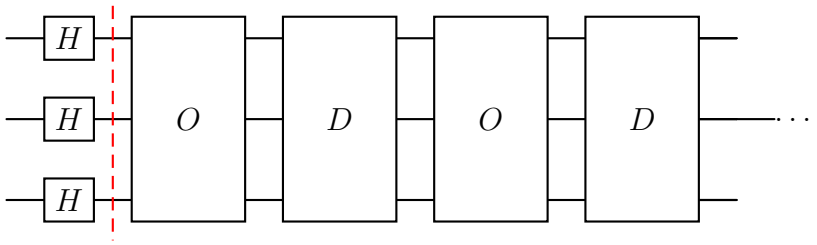


Figure 5.3: The quantum circuit for a 3-qubit version of Grover's algorithm. The initial Hadamard gates applied to the state $|000\rangle$ prepare the quantum state for the repeated operations applied to it.

basis states and the system is set to behave in a way such that the basis state either occurs with high probability or low probability. To be more specific, the safe regions are when $|(z)_p|^2 \geq 0.9$ and $|(z)_p|^2 \leq 0.1$. The behaviour is specified by the regions

$$\begin{aligned}\mathcal{Z}_0^p &= \{z \in \mathcal{Z} : |(z)_p|^2 \geq 0.9\}, \text{ and}\\ \mathcal{Z}_u^p &= \{z \in \mathcal{Z} : 0.1 + err \leq |(z)_p|^2 \leq 0.9 - err\},\end{aligned}$$

for $p \in \{0, 1, \ldots, 2^n - 1\}$, where $err = 0.1$ provides a buffer.

The generated barrier certificate (which has multiple functions) that ensure the safety of a system using 2-qubits ($n = 2$) and $|0\rangle$ as a target state ($p = 0$) with $k = 2, \epsilon = 0.01, \gamma = 0.01$, are

$$\begin{aligned}B_0(z) =& 0.9117 - 1.0307(z)_0(\overline{z})_0 - 0.0095(z)_0(\overline{z})_3 + 0.0219(z)_1(\overline{z})_1\\ &+ 0.0011(z)_1(\overline{z})_2 + 0.0011(z)_2(\overline{z})_1 - 0.0004(z)_2(\overline{z})_2\\ &- 0.0095(z)_3(\overline{z})_0 + 0.0066(z)_3(\overline{z})_3,\\ B_1(z) =& 0.902 - 1.0212(z)_0(\overline{z})_0 - 0.0136(z)_0(\overline{z})_3 + 0.0289(z)_1(\overline{z})_1\\ &+ 0.0058(z)_1(\overline{z})_2 + 0.0058(z)_2(\overline{z})_1 + 0.0083(z)_2(\overline{z})_2\\ &- 0.0136(z)_3(\overline{z})_0 + 0.0136(z)_3(\overline{z})_3,\end{aligned}$$

where B_0 is the function for the dynamics evolving according to X and B_1 for Z respectively.

5.5.5 Grover's Algorithm

Now a property of Grover's search algorithm [85] is verified. The database search problem is: given a function $h : \{0, 1\}^n \to \{0, 1\}$ such that $h(m) = 1$ for a unique

$m \in \{0,1\}^n$ and $h(x) = 0$ for $x \neq m$, find m with as few calls to h as possible.[3] Grover's algorithm solves this by putting the quantum state into superposition (using Hadamard gates) and then alternating between an oracle, $O|x\rangle = (-1)^{h(x)}|x\rangle$, and a diffusion step, $D = H^{\otimes n}(2|0^n\rangle\langle 0^n| - I_n)H^{\otimes n}$. *I.e.,* the initial quantum state is $|\phi\rangle = \frac{1}{\sqrt{2^n}}\sum_x |x\rangle$ and the operation $G = D.O$ is applied to the quantum state r times, where r depends on the number of qubits. This moves the quantum state into a state where $|m\rangle$ is measured with high probability. The circuit for Grover's algorithm is given in Figure 5.3.

The evolution of the quantum state can be viewed geometrically [2, 114] as

$$G^r|\phi\rangle = \cos\left(\frac{2r+1}{2}\theta\right)|\neg m\rangle + \sin\left(\frac{2r+1}{2}\theta\right)|m\rangle,$$

where $0 \leq \theta \leq \pi/2$ such that $\sin(\theta) = \frac{2\sqrt{2^n-1}}{2^n}$ and $|\neg m\rangle = \frac{1}{\sqrt{2^n-1}}\sum_{x\neq m}|x\rangle$. In a setting with no noise and $n > 1$, it can be seen that only $|m\rangle$ can be measured with high probability no matter the value of r. Any unmarked element, x, will have at most probability $\frac{1}{2^n-1}$. However, if the initial state is slightly disturbed, is this still the case? Previous work has shown that Grover's algorithm with more iterations can still return the marked state even when the initial state is not in equal superposition [27] but it does not show if an unmarked state is restricted.

The dynamical system for Grover's algorithm is given as

$$\forall t \in \mathbb{N}_{\geq 0}, f_t(z) = \begin{cases} Oz, & \text{if } t \text{ is even;} \\ Dz, & \text{otherwise.} \end{cases}$$

The initial region, $\mathcal{Z}_0$, is set to be a region close to the superposition of states with a slight disturbance to its amplitude. For an n-qubit system, the initial region is

$$\begin{aligned}\mathcal{Z}_0 = &\{z \in \mathcal{Z} : \frac{1}{2^n} - err \leq |(z)_j|^2 \leq \frac{1}{2^n} + err, \\ &-\sqrt{err} \leq \mathrm{Im}\{(z)_j\} \leq \sqrt{err} \text{ for } 0 \leq j \leq 2^n - 1\},\end{aligned}$$

with $err = \frac{1}{10^{n+1}}$. This follows the definition of the initial region given in Equation (5.3) with an additional constraint on the imaginary value.

[3] There is a version of the problem that has several marked elements, but only a single marked element is considered.

The goal is to verify that no single unmarked state is ever likely, *i.e.*, an unmarked state will never be the most likely result. This specification, for some unmarked element p, is given by the region

$$\mathcal{Z}_u^p = \{z \in \mathcal{Z} : |(z)_p|^2 \geq 0.9\}.$$

The value 0.9 is to represent the unmarked state being chosen with high probability and for simplicity's sake.

While, the quantum state for $n = 2$ could be specified, it was found that no barrier could be produced with a degree less than 4 and trying several different hyper-parameters. Running the algorithm with higher degrees simply takes too long given the complexity of the initial state (see the discussion in the next section).

Remark 5.5. Additionally, the dynamical system was changed into a single equation in an attempt to find a barrier. It was experimented defining the system starting from the initial state evolving according to the dynamics $f_t(z) = D.Oz$ (even steps), and also tried a system where the initial state starts after applying the oracle to the initial state and the dynamics evolve according to $f_t(z) = O.Dz$ (odd steps). However, no barrier could be found for either systems using the unsafe system as described before.

5.5.6 Discussion

The experiments were extended to a higher number of qubits to see how the implementation would perform. Details of how the Z gate experiment is extended are given in Appendix B.2. The runtimes for the experiments are shown in Table 5.1 using the following headers: *Experiment* refers to the relevant experiment discussed in this section; *# Qubits* refers to the number of qubits used; *Target State* refers to the state that is used for safety properties (where $|p\rangle$ modifies $\mathcal{Z}_0^p$ and $\mathcal{Z}_u^p$); *Generation Time* refers to the amount of time spent generating the barrier certificate, split between time performing setup and post-processing (*S&P*) and time in PICOS (*PICOS*); and *Verification Time* refers to the amount of time taken to verify the barrier using SMT solvers.

While most experiments used a 2-degree barrier, the *X and Z Gate* experiment used a 4-degree barrier for the targets $|1\rangle$, $|2\rangle$, and $|3\rangle$. Raising the degree of the barrier has a large affect on the setup for the barrier. Further, most of the generation time was spent setting up the various polynomials rather than running

Table 5.1: Average runtimes for 5 runs of each experiment. For verification a timeout was set to 300s for each call to the SMT solver. Entries with a * in them faced some timeout during the verification. Grover experiments used an oracle function where 0 is the marked state (*i.e.*, $h(0) = 1$).

Experiment	*# Qubits*	*Target State* ($\|p\rangle$)	*Generation Time (s)*			*Verification Time (s)*
			S&P	*PICOS*	*Status*	
Z Gate	1	$\|0\rangle$	2.62	1.72	solved	0.15
		$\|1\rangle$	2.53	1.56	solved	0.16
	2	$\|0\rangle$	86.23	33.85	solved	0.93
		$\|1\rangle$	87.90	34.34	solved	21.42
		$\|2\rangle$	89.62	34.32	solved	22.06
		$\|3\rangle$	89.02	34.28	solved	4.77
X Gate	1	$\|0\rangle$	2.99	2.14	solved	0.41
		$\|1\rangle$	3.20	2.48	solved	131.26
	2	$\|0\rangle$	248.71	95.96	solved	900.37*
		$\|1\rangle$	246.20	96.19	solved	900.42*
		$\|2\rangle$	247.17	70.88	**unsolved**	-
		$\|3\rangle$	241.01	69.34	**unsolved**	-
X and Z Gates	1	$\|0\rangle$	4.31	7.06	solved	0.317
		$\|1\rangle$	4.53	7.42	solved	1.20
	2	$\|0\rangle$	98.45	297.78	solved	160.73
		$\|1\rangle$	3,955.97	579.68	solved	2,104.04*
		$\|2\rangle$	3,890.78	757.57	**unsolved**	-
		$\|3\rangle$	3,996.65	877.00	solved	2,105.83*
	3	$\|0\rangle$	-	-	**killed**	-
Grover (k=1)	2	$\|1\rangle$	1,297.32	523.15	**unsolved**	-
Grover (k=2)	2	$\|1\rangle$	1,288.85	590.36	**unsolved**	-
Grover (even)	2	$\|1\rangle$	1,257.80	98.26	**unsolved**	-
Grover (odd)	2	$\|1\rangle$	1,261.36	97.90	**unsolved**	-

the semidefinite solver in PICOS. This time can be reduced by reducing the number of terms in the polynomial for the barrier, meaning a change in the form of the barrier could lead to faster runtimes but at the cost of not being able to generate barriers. An alternative approach would be to cache or store the symbolic representations of functions so that they can be easily reused.

Additionally, it was found that while the implementation could find barriers for some examples, it could not for others (noted by those that were **unsolved** or **killed**). The barriers that were not timed out during verification could also be incorrect. This is likely due to only considering a low degree polynomial for the barrier. However, using a higher degree polynomial would take much longer in the setup phase. It could be the case that a non-polynomial barrier function or a different barrier scheme needs to be used to ensure safety of some quantum systems, however this should not be the case for these simple examples.

An alternative approach would be to consider a change in variables to reduce the dimensionality of the variables. As seen in the results for Grover's algorithm, changing the dynamics of the system to use only a single function in the dynamics vastly reduced the time spent solving the semidefinite program, reducing the dimensionality could reduce the time further. For instance, the dynamics of Grover's algorithm could be encoded using the geometric representation instead of the standard quantum state representation. This would allow for a reduction in the dimension of the system (from the 4 variables required to 2 variables), and could allow some properties of quantum circuits to be verified, although the properties would need to be specified using this new basis. More generally, this approach could lend an exponential reduction in the number of variables, since the standard representation for an n-qubit Grover's algorithms as a dynamical system requires 2^n variables, but using the geometric representation requires only 2 variables. However, a question remains if this reduction comes at the cost of complexity in the definition of the initial and unsafe regions. As it currently stands though, while some systems can be proven safe, barrier certificates are inefficient if naively applied to quantum circuits with a high number of qubits.

Finally, while the experiments were run on a device with modest resources, it failed to run an example using 3 qubits. Even though a device with more RAM and a faster processor could compute a barrier for a 3-qubit system, the same issue will arise when considering a 4, 5, or 6 qubit system. It is difficult to see barrier certificates being used beyond a low number of qubits without the usage of a change

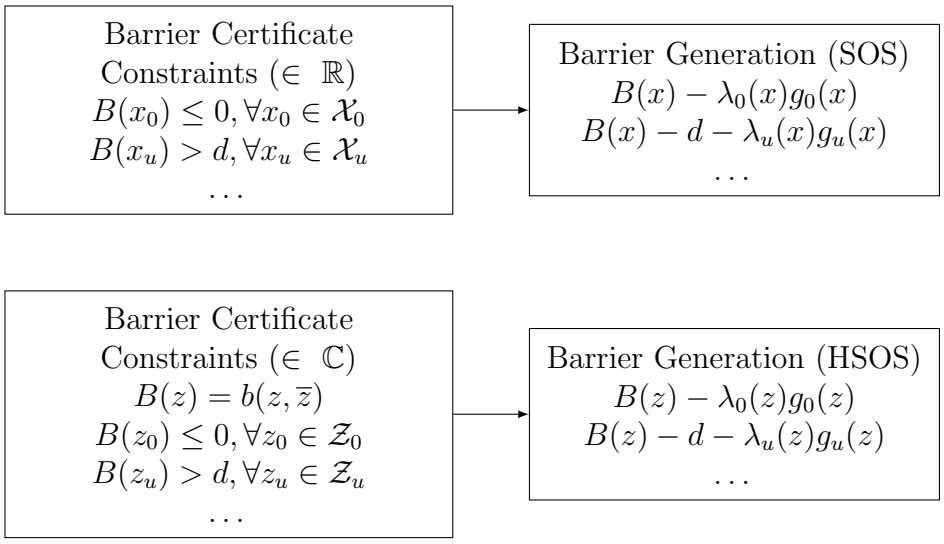


Figure 5.4: Conversion of a real barrier certificate scheme using SOS equations to generate a barrier into a complex scheme using HSOS equations. This is done by replacing the real variables with complex variables, allowing conjugation as an operation, setting the barrier function to be a conjugate-flattening polynomial and requiring the SOS equations to be HSOS equations instead.

in variables. However, barrier certificates sacrifice scalability for flexibility as will be discussed in the conclusion.

5.6 Conclusion

In this article, a different approach has been considered to the verification of quantum circuits through the usage of barrier certificates by treating the quantum circuits as dynamical systems. Firstly, it was shown how to extend the notion of barrier certificates from the real domain into the complex domain. Then it was demonstrated how the standard approach for generating barrier certificates, through sum of squares, can be extended to the complex domain as well, through Hermitian sum of squares. Finally, experiments were performed to show the usage of the developed technique.

Our techniques for extending barrier certificates to the complex domain can be applied to complex dynamical systems in general. Any barrier certificate that uses SOS techniques to generate a barrier can be extended to the complex domain through the use of conjugate-flattening functions and HSOS equations as demonstrated in Figure 5.4. This extension can be done almost freely by including the complex

conjugate as an operation.

The barrier certificate technique is very flexible, but comes at the cost of scalability (as seen in Table 5.1). While the barrier certificate approach is expensive for quantum circuits with a large number of qubits, the automation provided could be useful for verifying certain behaviours of quantum systems considered in Chapter 4. A quantum system can be modelled when there is noisy qubit initialisation and, in the future, how to verify quantum systems with additional properties can be considered, such as noise or control during state evolution, using barrier certificates. For example, a noise model [80] can be used to model the noise a quantum circuit faces as part of the dynamical system and the barrier certificate techniques for handling stochastic systems, [93] or [144], can be adapted to handle complex variables to verify properties on the noisy dynamical system. Further, other behavioural properties can be investigated instead of safety, such as reachability where the system evolves into a specified region rather than avoid it.

This concludes the investigation and analysis of the second automated technique in this thesis. Further discussions on the results of this work are found in Part IV.

Part IV

Conclusion and Supplementary Material

Chapter 6

Conclusion and Future Outlook

This last Chapter provides the main findings of the thesis and an outlook into the future of automated techniques for verifying aspects of quantum computers.

In this thesis, the usage of automated techniques has been explored to verify different properties of the quantum stack. Two techniques have been investigated. The first technique, `SilVer`, makes use of SMT solvers and the standard software verification framework to automatically convert and verify programs for quantum computers, where behaviours are defined through `SilSpeq`, the specification language. The second technique extends barrier certificates to verify properties in the Hamiltonian and quantum circuit model of quantum computing. The challenges and problems of these techniques are discussed, as well as their future and how the techniques can be used together.

6.1 Challenges and Immediate Problems

There are several immediate problems that can be investigated from the work done with this thesis with more time and resources available.

With `SilVer` (and `SilSpeq`), one of the major problems faced was the handling of subroutines, *i.e.*, when one function calls another function. `SilVer` partly handles subroutine calling when handling oracles, but this can be further expanded. The major challenge with function calling is the knowledge of whether a function's input and outputs are classical or quantum in nature and how this affects verification of a function calling a subroutine. This poses a challenge, from a theoretical perspective, in how to specify a program such that the specification can be used to verify another function, as well as from an implementation perspective in how to do such conversion

automatically. With more time, an investigation into how subroutines are handled in `SilVer` could be performed. Some work has already been done into looking at specification of quantum programs with classical variables [71], but an investigation into how to adapt these ideas for subroutines is needed.

On top of that, improving `SilSpeq` to handle quantum behaviours is an investigation that should happen first. Specifying quantum behaviour was a major challenge in the development of `SilVer`. Whilst there are techniques out there using density matrices for quantum specification, *e.g.*, [103], for users it would be more suitable to have a compact way to specify behaviours. Finding a way to write out a specification for quantum behaviours in a compact way would improve the capabilities of `SilSpeq` greatly. For example, one could then verify quantum gates that are defined in Silq or verify the GHZ state more explicitly.

With the work on barrier certificates, a problem remains in providing the assurance of safety for Grover's algorithm. One method to address this problem is to first investigate simple quantum circuits that entangle qubits together and progressively increase the complexity of the entanglement. Some insights could be gained as to why polynomial barrier functions are not suitable for ensuring safety in Grover's algorithm. Alternatively, a different barrier certificate scheme may be required or a non-polynomial barrier function. With the relation between barrier certificates and Hermitian Sum of Squares (HSOS) equations developed in this thesis, a new scheme only requires new constraints on the barrier and a conversion into appropriate HSOS equations.

6.2 Future Outlook

By making use of the software verification techniques, `SilVer` can be extended in several different ways. One aspect where `SilVer` can easily be extended is its modularity. The design of `SilVer` allows any suitable quantum programming language to be converted into the QRAM program model. This means that programs written in languages such as Q# [143] and Qiskit [129] could be verified. Additionally, the QRAM program model can be used as an interface to different forms of verification. This can allow different methods to verify programs, which would have their own trade-offs. For instance, different SMT solvers could be used or further automation could be implemented through the idea of weakest precondition [63, 64] to reduce the number of constraints to be verified in a solver.

Not only can the automation steps be extended, but the ideas used in the specification language, `SilSpeq`, can be further developed. Being able to use the specification of one function in another function would be a large step forward for automation techniques in quantum computers. To develop this, quantum and classical specification would need to be separated or handled carefully, and a way of writing quantum specification compactly would need to be developed. Using vectors would be quite costly, but using Dirac bra-ket or density matrix notations in an algebraic form could provide a compact method for specification.

With the theory developed of Hermitian Sum of Squares (HSOS) for barrier certificates, various systems can be verified for different behaviours. This can include systems that have noise and/or a control component. To truly take advantage of the theory, a complex semidefinite program solver and a HSOS solver would need to be developed. Additionally, some aspects of theory still need to be explored. For example, whilst in theory a semidefinite matrix from a HSOS equation is smaller than one from a transformed SOS equation (by converting complex variables to real ones), it would be good to formally prove this as well as show a computational advantage. An exploration into non-polynomial barrier certificates may provide further utility for verifying behaviours in more complicated quantum systems. Some work and tools have already been developed to investigate the usage of machine learning techniques to create valid barrier certificates with non-polynomial terms [1, 121]. Exploring different methods of representing quantum states may provide speedups for using barrier certificates.

Another prospect is the combination of the automated framework introduced in Part II and the technique of using barrier certificates in Part III. This would involve providing a quantum circuit or system (rather than a Silq program) that safety or reachability properties would be checked by using the HSOS barrier certificate generation technique, which would replace the SMT solver. Additionally, rather than requiring the user to provide the necessary encodings for the sets used by the barrier certificate technique, the user could be provided with a specification language to describe the sets in a concise manner and the type of property they wish to verify. Figure 6.1 shows how the framework from `SilVer` could be adapted for the usage of barrier certificates.

Whilst automated techniques clearly have a place in verifying quantum computers, there is a need to keep up with the physical hardware, which is close to approaching over 1000 qubits. If implemented naively, most automated techniques

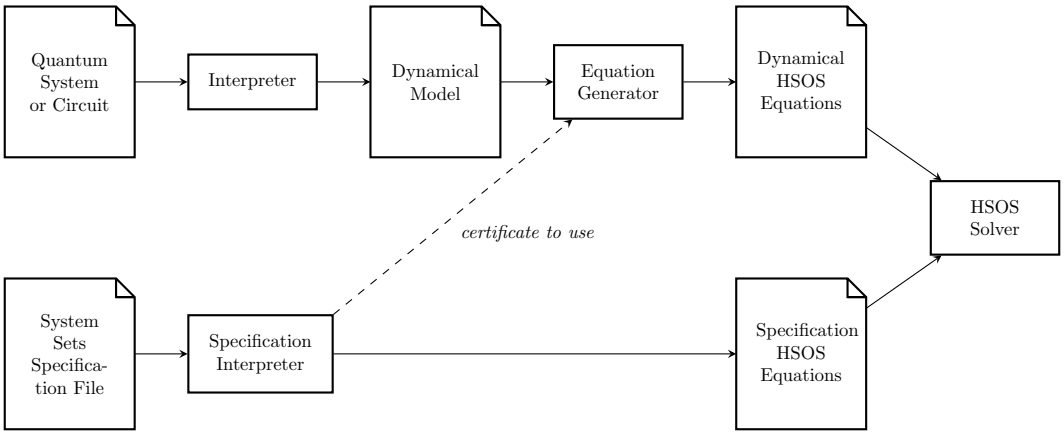


Figure 6.1: `SilVer` framework adapted to verification of safety/reachability properties through barrier certificates.

will be unable to keep up with the number of qubits in quantum computers due to the exponential representation of quantum states. Methods need to be developed to allow for effective verification for large quantum systems. As shown, these methods could be adapted from frameworks that already exist, like the software verification framework for `SilVer`; or an extension of a known method, as seen by extending real to complex variables for barrier certificates.

Bibliography

[1] Alessandro Abate et al. FOSSIL: A software tool for the formal synthesis of Lyapunov functions and barrier certificates using neural networks. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*. ACM, 2021. doi:10.1145/3447928.3456646.

[2] Dorit Aharonov. *Quantum Computation*, volume 6 of *Annual Reviews of Computational Physics*, pages 259–346. World Scientific, March 1999. doi:10.1142/9789812815569_0007.

[3] Amir Ali Ahmadi, Miroslav Krstic, and Pablo A. Parrilo. A globally asymptotically stable polynomial vector field with no polynomial lyapunov function. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*, pages 7579–7580, 2011. doi:10.1109/CDC.2011.6161499.

[4] Aaron D. Ames, Samuel Coogan, Magnus Egerstedt, Gennaro Notomista, Koushil Sreenath, and Paulo Tabuada. Control barrier functions: Theory and applications. In *2019 18th European Control Conference (ECC)*, pages 3420–3431, 2019. doi:10.23919/ECC.2019.8796030.

[5] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. *SIGPLAN Not.*, 36(5):137–147, May 2001. doi:10.1145/381694.378825.

[6] Matthew Amy. *Formal Methods in Quantum Circuit Design*. PhD thesis, University of Waterloo, February 2019. URL: http://hdl.handle.net/10012/14480.

[7] Matthew Amy. Towards large-scale functional verification of universal quantum circuits. *Electronic Proceedings in Theoretical Computer Science*, 287(287):1–21, 2019. doi:10.4204/Eptcs.287.1.

[8] Mahathi Anand, Vishnu Murali, Ashutosh Trivedi, and Majid Zamani. Safety verification of dynamical systems via k-inductive barrier certificates. In *2021 60th IEEE Conference on Decision and Control (CDC)*, pages 1314–1320, 2021. doi:10.1109/CDC45484.2021.9682889.

[9] Linda Anticoli, Carla Piazza, Leonardo Taglialegne, and Paolo Zuliani. Towards quantum programs verification: From Quipper circuits to QPMC. In Simon Devitt and Ivan Lanese, editors, *Reversible Computation*, pages 213–219, Cham, 2016. Springer International Publishing.

[10] Linda Anticoli, Carla Piazza, Leonardo Taglialegne, and Paolo Zuliani. Entangλe: A translation framework from Quipper programs to quantum Markov chains. In Simonetta Balsamo, Andrea Marin, and Enrico Vicario, editors, *New Frontiers in Quantitative Methods in Informatics*, pages 113–126, Cham, 2018. Springer International Publishing.

[11] F. T. Arecchi, Eric Courtens, Robert Gilmore, and Harry Thomas. Atomic coherent states in quantum optics. *Phys. Rev. A*, 6:2211–2237, December 1972. doi:10.1103/PhysRevA.6.2211.

[12] Miriam Backens, Simon Perdrix, and Quanlong Wang. A simplified stabilizer ZX-calculus. *Electronic Proceedings in Theoretical Computer Science*, 236:1–20, January 2017. doi:10.4204/eptcs.236.1.

[13] Stanley Bak. t-barrier certificates: A continuous analogy to k-induction. *IFAC-PapersOnLine*, 51(16):145–150, 2018. 6th IFAC Conference on Analysis and Design of Hybrid Systems ADHS 2018. doi:10.1016/j.ifacol.2018.08.025.

[14] Pedro Baltazar, Rohit Chadha, and Paulo Mateus. Quantum computation tree logic — model checking and complete calculus. *International Journal of Quantum Information*, 06(02):219–236, 2008. doi:10.1142/s0219749908003530.

[15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[16] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-319-10575-8_11`.

[17] Gilles Barthe, Delphine Demange, and David Pichardie. A Formally Verified SSA-Based Middle-End. In Helmut Seidl, editor, *ESOP 2012*, volume 7211 of *LNCS*, pages 47–66. Springer, 2012. `doi:10.1007/978-3-642-28869-2_3`.

[18] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. symQV: Automated symbolic verification of quantum programs. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods*, pages 181–198, Cham, 2023. Springer International Publishing.

[19] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, 22(5):563–591, May 1980. `doi:10.1007/BF01011339`.

[20] Charles H. Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, 560:7–11, 2014. Theoretical Aspects of Quantum Cryptography – celebrating 30 years of BB84. `doi:10.1016/j.tcs.2014.05.025`.

[21] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Lett.*, 70:1895–1899, 1993. `doi:10.1103/PhysRevLett.70.1895`.

[22] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997. `doi:10.1137/S0097539796300921`.

[23] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[24] Amandeep Singh Bhatia and Ajay Kumar. On relation between linear temporal logic and quantum finite automata. *J. Log. Lang. Inf.*, 29(2):109–120, 2020. doi:10.1007/s10849-019-09302-6.

[25] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 286–300, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386007.

[26] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[27] Eli Biham, Ofer Biham, David Biron, Markus Grassl, and Daniel A. Lidar. Grover's quantum search algorithm for an arbitrary initial amplitude distribution. *Phys. Rev. A*, 60:2742–2745, October 1999. doi:10.1103/PhysRevA.60.2742.

[28] F. Bloch. Nuclear induction. *Phys. Rev.*, 70:460–474, October 1946. doi:10.1103/PhysRev.70.460.

[29] J. Bochnak, M. Coste, and M-F. Roy. *Real Algebraic Geometry*. Springer, 1998. doi:10.1007/978-3-662-03718-8.

[30] Anthony Bordg, Hanna Lachnitt, and Yijun He. Certified quantum computation in Isabelle/HOL. *Journal of Automated Reasoning 2020 65:5*, 65:691–709, 12 2020. doi:10.1007/S10817-020-09584-7.

[31] Anthony Bordg, Hanna Lachnitt, and Yijun He. Isabelle Marries Dirac: a library for quantum computation and quantum information. *Archive of Formal Proofs*, November 2020. https://isa-afp.org/entries/Isabelle_Marries_Dirac.html, Formal proof development.

[32] Sergey Bravyi and David Gosset. Improved classical simulation of quantum circuits dominated by clifford gates. *Phys. Rev. Lett.*, 116:250501, June 2016. doi:10.1103/PhysRevLett.116.250501.

[33] Sergey Bravyi, David Gosset, and Robert König. Quantum advantage with shallow circuits. *Science*, 362(6412):308–311, 2018. doi:10.1126/science.aar3106.

[34] Lukas Burgholzer and Robert Wille. Advanced equivalence checking for quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40:1810–1824, 9 2021. doi:10.1109/TCAD.2020.3032630.

[35] Yudong Cao, Jonathan Romero, Jonathan P. Olson, Matthias Degroote, Peter D. Johnson, Mária Kieferová, Ian D. Kivlichan, Tim Menke, Borja Peropadre, Nicolas P. D. Sawaya, Sukin Sim, Libor Veis, and Alán Aspuru-Guzik. Quantum chemistry in the age of quantum computing. *Chemical Reviews*, 119(19):10856–10915, 2019. PMID: 31469277. doi:10.1021/acs.chemrev.8b00803.

[36] Titouan Carette, Yohann D'Anello, and Simon Perdrix. Quantum algorithms and oracles with the scalable ZX-calculus. *Electronic Proceedings in Theoretical Computer Science*, 343:193–209, September 2021. doi:10.4204/eptcs.343.10.

[37] Titouan Carette, Dominic Horsman, and Simon Perdrix. SZX-calculus: Scalable graphical quantum reasoning. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*, volume 138 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 55:1–55:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.MFCS.2019.55.

[38] Rohit Chadha, Paulo Mateus, Amílcar Sernadas, and Cristina Sernadas. Extending classical logic for reasoning about quantum systems. In Kurt Engesser, Dov M. Gabbay, and Daniel Lehmann, editors, *Handbook of Quantum Logic and Quantum Structures*, pages 325 – 371. Elsevier, Amsterdam, 2009. doi:10.1016/B978-0-444-52869-8.50011-6.

[39] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. An automated deductive verification framework for circuit-building quantum programs. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 148–177, Cham, 2021. Springer International Publishing.

[40] Tian-Fu Chen, Jie-Hong R. Jiang, and Min-Hsiu Hsieh. Partial equivalence checking of quantum circuits. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 594–604, 2022. doi:10.1109/QCE53715.2022.00082.

[41] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai. AutoQ: An automata-based quantum circuit verifier. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 139–153, Cham, 2023. Springer Nature Switzerland.

[42] Andrew M. Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, page 59–68, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/780542.780552.

[43] Cirq Developers. Cirq, December 2022. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors. doi:10.5281/zenodo.7465577.

[44] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[45] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[46] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. doi:10.1145/186025.186051.

[47] Edmund M Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, London, Cambridge, 1999.

[48] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 1–30, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-35746-6_1.

[49] John F. Clauser, Michael A. Horne, Abner Shimony, and Richard A. Holt. Proposed experiment to test local hidden-variable theories. *Phys. Rev. Lett.*, 23:880–884, October 1969. doi:10.1103/PhysRevLett.23.880.

[50] Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, April 2011. doi:10.1088/1367-2630/13/4/043016.

[51] Andrea Colledan and Ugo Dal Lago. On Dynamic Lifting and Effect Typing in Circuit Description Languages. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:21, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.TYPES.2022.3.

[52] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. URL: https://hal.inria.fr/hal-01960203.

[53] Patrick Cousot. *Abstract Interpretation Based Formal Methods and Future Challenges*, pages 138–156. Springer Berlin Heidelberg, 2001. doi:10.1007/3-540-44577-3_10.

[54] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, page 238–252, 1977. doi:10.1145/512950.512973.

[55] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3), September 2022. doi:10.1145/3505636.

[56] Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. Code motion of control structures in high-level languages. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, page 70–85. ACM, 1986. doi:10.1145/512644.512651.

[57] Timothy A. S. Davidson, Simon J. Gay, Hynek Mlnarik, Rajagopal Nagarajan, and Nick Papanikolaou. Model checking for communicating quantum processes. *Int. J. Unconv. Comput.*, 8(1):73–98, 2012. URL: http://www.oldcitypublishing.com/journals/ijuc-home/ijuc-issue-contents/ijuc-volume-8-number-1-2012/ijuc-8-1-p-73-98/.

[58] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[59] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In Marcel Vinícius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications*, pages 23–36, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[60] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011. doi:10.1145/1995376.1995394.

[61] D Deutsch, A Barenco, and A Ekert. Universality in quantum computation. *Proc., Math. Phys. Sci.*, 449(1937):669–677, June 1995.

[62] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992. doi:10.1098/rspa.1992.0167.

[63] Ellie D'Hondt and Prakash Panangaden. Quantum weakest preconditions. *Math. Struct. Comput. Sci.*, 16(3):429–451, 2006. doi:10.1017/S0960129506005251.

[64] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. doi:10.1145/360933.360975.

[65] Mnacho Echenim. Quantum projective measurements and the CHSH inequality. *Archive of Formal Proofs*, March 2021. https://isa-afp.org/entries/Projective_Measurements.html, Formal proof development.

[66] Mnacho Echenim and Mehdi Mhalla. Quantum projective measurements and the chsh inequality in Isabelle/HOL, 2021. arXiv:2103.08535.

[67] Tao Fang and Jitao Sun. Stability analysis of complex-valued nonlinear differential system. *Journal of Applied Mathematics*, 2013:621957, 2013. doi:10.1155/2013/621957.

[68] Yuan Feng, Runyao Duan, Zhengfeng Ji, and Mingsheng Ying. Probabilistic bisimulations for quantum processes. *Information and Computation*, 205(11):1608–1639, 2007. doi:10.1016/j.ic.2007.08.001.

[69] Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Shenggang Ying. Model checking omega-regular properties for quantum Markov chains. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CONCUR.2017.35.

[70] Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Lijun Zhang. QPMC: A model checker for quantum programs and protocols. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods*, pages 265–272, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-19249-9_17.

[71] Yuan Feng and Mingsheng Ying. Quantum Hoare logic with classical variables. *ACM Transactions on Quantum Computing*, 2(4), December 2021. doi:10.1145/3456877.

[72] Yuan Feng, Nengkun Yu, and Mingsheng Ying. Model checking quantum Markov chains. *Journal of Computer and System Sciences*, 79(7):1181 – 1198, 2013. doi:10.1016/j.jcss.2013.04.002.

[73] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, June 1982. doi:10.1007/BF02650179.

[74] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[75] Martin Fränzle, Mingshuai Chen, and Paul Kröger. In memory of Oded Maler: Automatic reachability analysis of hybrid-state automata. *ACM SIGLOG News*, 6(1):19–39, 2019. doi:10.1145/3313909.3313913.

[76] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. Proto-Quipper with dynamic lifting, 2022. arXiv:2204.13041.

[77] X. Fu, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels. A heterogeneous quantum computer architecture. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, page 323–330, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2903150.2906827.

[78] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, pages 208–214, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[79] Simon J. Gay and Rajagopal Nagarajan. Communicating quantum processes. *SIGPLAN Not.*, 40(1):145–157, January 2005. doi:10.1145/1047659.1040318.

[80] Konstantinos Georgopoulos, Clive Emary, and Paolo Zuliani. Modeling and simulating the noisy behavior of near-term quantum computers. *Phys. Rev. A*, 104:062432, December 2021. URL: https://link.aps.org/doi/10.1103/PhysRevA.104.062432, doi:10.1103/PhysRevA.104.062432.

[81] Jean Charles Gilbert and Cédric Josz. Plea for a semidefinite optimization solver in complex numbers. Research report, Inria Paris, March 2017. URL: https://inria.hal.science/hal-01422932.

[82] D Gottesman. The heisenberg representation of quantum computers. 6 1998. URL: https://www.osti.gov/biblio/319738.

[83] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. *SIGPLAN Not.*, 48(6):333–342, June 2013. doi:10.1145/2499370.2462177.

[84] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. *Going Beyond Bell's Theorem*, pages 69–72. Springer Netherlands, Dordrecht, 1989. doi:10.1007/978-94-017-0849-4_10.

[85] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237814.237866.

[86] Reiner Hähnle and Marieke Huisman. Deductive software verification: From pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, pages 345–373, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-319-91908-9_18.

[87] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.*, 103:150502, October 2009. doi:10.1103/PhysRevLett.103.150502.

[88] Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. Proving quantum programs correct. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics*

(LIPIcs), pages 21:1–21:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2021.21.

[89] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

[90] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. Property-based testing of quantum programs in Q#. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 430–435, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387940.3391459.

[91] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 2 edition, 2012.

[92] Yipeng Huang and Margaret Martonosi. QDB: From quantum algorithms towards correct quantum programs. In Titus Barik, Joshua Sunshine, and Sarah Chasins, editors, *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*, volume 67 of *OpenAccess Series in Informatics (OASIcs)*, pages 4:1–4:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.PLATEAU.2018.4.

[93] Pushpak Jagtap, Sadegh Soudjani, and Majid Zamani. Formal synthesis of stochastic systems via control barrier certificates. *IEEE Transactions on Automatic Control*, 66(7):3097–3110, 2021. doi:10.1109/TAC.2020.3013916.

[94] Yoshihiko Kakutani. A logic for formal verification of quantum programs. In Anupam Datta, editor, *Advances in Computer Science - ASIAN 2009. Information Security and Privacy*, pages 79–93, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[95] Aleks Kissinger and John van de Wetering. PyZX: Large scale automated diagrammatic reasoning. *Electronic Proceedings in Theoretical Computer Science*, 318:229–241, May 2020. doi:10.4204/eptcs.318.14.

[96] Emanuel Knill. Conventions for quantum pseudocode. 6 1996. doi:10.2172/366453.

[97] Attila Kondacs and John Watrous. On the power of quantum finite state automata. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 66–75. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646094.

[98] Dexter Kozen. Results on the propositional μ-calculus. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming*, pages 348–359, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

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

[100] Abolfazl Lavaei, Sadegh Soudjani, Alessandro Abate, and Majid Zamani. Automated verification and synthesis of stochastic hybrid systems: A survey. 2021. arXiv:2101.07491.

[101] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. Formal verification of quantum programs: Theory, tools, and challenges. *ACM Transactions on Quantum Computing*, 5(1), December 2023. doi:10.1145/3624483.

[102] Marco Lewis, Paolo Zuliani, and Sadegh Soudjani. Verification of quantum systems using barrier certificates. In Nils Jansen and Mirco Tribastone, editors, *Quantitative Evaluation of Systems*, pages 346–362, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-43835-6_24.

[103] Junyi Liu et al. Formal verification of quantum algorithms using quantum Hoare logic. *Lecture Notes in Computer Science*, 11562 LNCS:187–207, 2019. doi:10.1007/978-3-030-25543-5_12.

[104] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Formal verification of quantum algorithms using quantum Hoare logic. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 187–207, Cham, 2019. Springer International Publishing.

[105] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Quantum Hoare logic. *Archive of Formal*

Proofs, March 2019. `https://isa-afp.org/entries/QHLProver.html`, Formal proof development.

[106] Junjie Luo and Jianjun Zhao. Formalization of quantum intermediate representations for code safety, 2023. `arXiv:2303.14500`.

[107] Yuri I. Manin. *Computable and Uncomputable*, pages 13–15. Sovetskoe Radio, Moscow, 1980. Original (in Russian).

[108] Yuri I. Manin. *Introduction to the book Computable and Uncomputable*, pages 69–77. Collected works (American Mathematical Society). American Mathematical Society, 2007.

[109] Paulo Mateus, Jaime Ramos, Amílcar Sernadas, and Cristina Sernadas. *Temporal Logics for Reasoning about Quantum Systems*, page 389–413. Cambridge University Press, 2009. `doi:10.1017/CBO9781139193313.011`.

[110] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017. `doi:10.7717/peerj-cs.103`.

[111] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, February 2010. `doi:10.1145/1646353.1646372`.

[112] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[113] Ian M. Mitchell. Comparing forward and backward reachability as tools for safety analysis. In *Proceedings of the 10th International Conference on Hybrid Systems: Computation and Control*, page 428–443. Springer-Verlag, 2007.

[114] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, USA, 10th edition, 2011.

[115] Philipp Niemann and Robert Wille. *Quantum Multiple-Valued Decision Diagrams*, pages 35–58. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-63724-2_4.

[116] Alexandru Paler and Simon J. Devitt. An introduction into fault-tolerant quantum computing. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2744769.2747911.

[117] Matteo Paltenghi and Michael Pradel. Bugs in quantum computing platforms: an empirical study. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022. doi:10.1145/3527330.

[118] A. Papachristodoulou, J. Anderson, G. Valmorbida, S. Prajna, P. Seiler, P. A. Parrilo, M. M. Peet, and D. Jagt. *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*, 2021. Available from https://github.com/oxfordcontrol/SOSTOOLS. arXiv:1310.4716.

[119] Pablo A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, 96(2):293–320, May 2003. doi:10.1007/s10107-003-0387-5.

[120] Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. *SIGPLAN Not.*, 52(1):846–858, January 2017. doi:10.1145/3093333.3009894.

[121] Andrea Peruffo, Daniele Ahmed, and Alessandro Abate. Automated and formal synthesis of neural barrier certificates for dynamical models. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 370–388. Springer, 2021. doi:10.1007/978-3-030-72016-2_20.

[122] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 46–57, USA, 1977. IEEE Computer Society. doi:10.1109/SFCS.1977.32.

[123] Gabriel Pontolillo and Mohammad Reza Mousavi. Delta debugging for property-based regression testing of quantum programs. In *Proceedings of the*

5th International Workshop on Quantum Software Engineering (Q-SE 2024), held at ICSE 2024, 2024.

[124] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In Rajeev Alur and George J. Pappas, editors, *Hybrid Systems: Computation and Control*, pages 477–492, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[125] Stephen Prajna, Ali Jadbabaie, and George J. Pappas. A framework for worst-case and stochastic safety verification using barrier certificates. *IEEE Transactions on Automatic Control*, 52:1415–1428, 2007. doi:10.1109/TAC.2007.902736.

[126] Mihai Putinar. *Chapter 9: Sums of Hermitian Squares: Old and New*, pages 407–446. Society for Industrial and Applied Mathematics, 2012. doi:10.1137/1.9781611972290.ch9.

[127] Xudong Qin, Yuxin Deng, and Wenjie Du. Verifying quantum communication protocols with ground bisimulation. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–38, Cham, 2020. Springer International Publishing.

[128] QIR Alliance. QIR Alliance. https://www.qir-alliance.org/, 2023.

[129] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023. doi:10.5281/zenodo.2573505.

[130] Stefan Ratschan. Converse theorems for safety and barrier certificates. *IEEE Transactions on Automatic Control*, 63(8):2628–2632, 2018. doi:10.1109/TAC.2018.2792325.

[131] Joschka Roffe. Quantum error correction: an introductory guide. *Contemporary Physics*, 60(3):226–245, 2019. doi:10.1080/00107514.2019.1667078.

[132] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27. ACM, 1988. doi:10.1145/73560.73562.

[133] Guillaume Sagnol and Maximilian Stahlberg. PICOS: A Python interface to conic optimization solvers. *Journal of Open Source Software*, 7(70):3915, February 2022. doi:10.21105/joss.03915.

[134] Jeff W. Sanders and Paolo Zuliani. Quantum programming. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction*, pages 80–99, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[135] Oliver Schön, Zhengang Zhong, and Sadegh Soudjani. Data-driven distributionally robust safety verification using barrier certificates and conditional mean embeddings, 2024. arXiv:2403.10497.

[136] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004. doi:10.1017/S0960129504004256.

[137] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, USA, 1994.

[138] Yunong Shi, Runzhou Tao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. CertiQ: A mostly-automated verification of a realistic quantum compiler, 2020. arXiv:1908.08963.

[139] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997. doi:10.1137/S0097539795293172.

[140] Kartik Singhal, Kesha Hietala, Sarah Marshall, and Robert Rand. Q# as a quantum algorithmic language. In *Proceedings of the 19th International Conference on Quantum Physics and Logic (QPL), Oxford, U.K., June 27–July 1, 2022*. Open Publishing Association, June 2022. arXiv:2206.03532.

[141] Sadegh Soudjani and Alessandro Abate. Precise approximations of the probability distribution of a Markov process in time: an application to probabilistic invariance. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 547–561. Springer, 2014. doi:10.1007/978-3-642-54862-8_45.

[142] Sadegh Soudjani and Alessandro Abate. Quantitative approximation of the probability distribution of a Markov process by formal abstractions. *Logical Methods in Computer Science*, 11, 2015. doi:10.2168/LMCS-11(3:8)2015.

[143] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3183895.3183901.

[144] Rin Takano, Hiroyuki Oyama, and Masaki Yamakita. Application of robust control barrier function with stochastic disturbance model for discrete time systems. *IFAC-PapersOnLine*, 51(31):46–51, 2018. 5th IFAC Conference on Engine and Powertrain Control, Simulation and Modeling E-COSM 2018. doi:10.1016/j.ifacol.2018.10.009.

[145] Runzhou Tao et al. Giallar: Push-button verification for the Qiskit quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 641–656, New York, NY, USA, 2022. ACM. doi:10.1145/3519939.3523431.

[146] Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T. Chong, and Ronghui Gu. Gleipnir: toward practical error analysis for quantum programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 48–64, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454029.

[147] Yuan Hung Tsai, Jie Hong R. Jiang, and Chiao Shan Jhang. Bit-slicing the Hilbert space: Scaling up accurate quantum circuit simulation. *Proceedings - Design Automation Conference*, 2021-December:439–444, 12 2021. doi:10.1109/DAC18074.2021.9586191.

[148] John van de Wetering. ZX-calculus for the working quantum computer scientist, 2020. arXiv:2012.13966.

[149] Finn Voichick and Michael Hicks. Toward a quantum programming language for higher-level formal verification. In *Informal Proceedings of the Workshop on Programming Languages and Quantum Computing (PLanQC)*, June 2021.

[150] X. Wang, P. Arcaini, T. Yue, and S. Ali. QuCAT: A combinatorial testing tool for quantum software. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2066–2069, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society. doi:10.1109/ASE56229.2023.00062.

[151] Chun-Yu Wei, Yuan-Hung Tsai, Chiao-Shan Jhang, and Jie-Hong R. Jiang. Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 523–528, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3489517.3530481.

[152] James D. Whitfield, Jacob Biamonte, and Alán Aspuru-Guzik. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics*, 109(5):735–750, 2011. doi:10.1080/00268976.2011.552441.

[153] Freek Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[154] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. Tools for quantum computing based on decision diagrams. *ACM Transactions on Quantum Computing*, 3(3), July 2022. doi:10.1145/3491246.

[155] Rafael Wisniewski and Christoffer Sloth. Converse barrier certificate theorem. In *52nd IEEE Conference on Decision and Control*, pages 4713–4718, 2013. doi:10.1109/CDC.2013.6760627.

[156] William K. Wootters and Wojciech H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, 1982. doi:10.1038/299802a0.

[157] Ming Xu, Jianling Fu, Jingyi Mei, and Yuxin Deng. An algebraic method to fidelity-based model checking over quantum Markov chains. *Theoretical Computer Science*, 935:61–81, 2022. doi:10.1016/j.tcs.2022.08.016.

[158] Mingsheng Ying. Floyd-hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.*, 33(6):19:1–19:49, 2011. doi:10.1145/2049706.2049708.

[159] Mingsheng Ying and Yuan Feng. Model checking quantum systems — a survey, 2018. arXiv:1807.09466.

[160] Mingsheng Ying, Yuan Feng, Runyao Duan, and Zhengfeng Ji. An algebra of quantum processes. *ACM Trans. Comput. Logic*, 10(3), April 2009. doi:10.1145/1507244.1507249.

[161] Nengkun Yu. Quantum temporal logic, 2019. arXiv:1908.00158.

[162] Nengkun Yu and Jens Palsberg. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, page 542–558, 2021. doi:10.1145/3453483.3454061.

[163] Charles Yuan, Christopher McNally, and Michael Carbin. Twist: Sound reasoning for purity and entanglement in quantum programs. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498691.

[164] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 427–440, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2103656.2103709.

[165] Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. Bugs4q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs. *Journal of Systems and Software*, 205:111805, 2023. URL: https://www.sciencedirect.com/science/article/pii/S0164121223002005, doi:10.1016/j.jss.2023.111805.

[166] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. CoqQ: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571222.

Appendix A

Background Appendix

The contents of this Appendix are excerpts from a survey paper published in the ACM Transactions on Quantum Computing under the title "Formal Verification of Quantum Programs: Theory, Tools, and Challenges" [101]. This work was done in collaboration with my supervisors (Paolo Zuliani and Sadegh Soudjani).

A.1 Quantum Computing and Formal Verification

A.1.1 Clifford Gates

The Clifford gates [82] are the following gate operations:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{A.1}$$

Circuits that can be generated from Clifford gates are referred to as being in the Clifford group and can be simulated efficiently on a classical computer, as stated by the Gottesman-Knill theorem [82]. It is unknown if all non-Clifford circuits (i.e. circuits containing operations that cannot be broken down to Clifford operations) can be simulated efficiently. If this were the case, then quantum computers could be efficiently simulated on classical computers.

The operation S can be generalised to R_k, where

$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^k}} \end{pmatrix}, \tag{A.2}$$

and it can easily be seen that if $k \geq 2$, then $(R_k)^{2^{k-2}} = S$ (with $R_2 = S$). For $k \geq 3$, R_k is a non-Clifford operation. The circuits generated by using Clifford gates and the R_k gate (for a fixed k) are referred to as the Clifford+R_k group. In particular, denote $T = R_3 = \begin{pmatrix} 1 & 0 \\ 0 & e^{\pi i/4} \end{pmatrix}$ and then Clifford + R_3 = Clifford + T. Algorithms for simulating Clifford+T gates are discussed in [32]. The algorithms' runtimes are exponential based on the number of T gates.

A.1.2 Kripke Structures and CTL

Kripke structures model software or systems by describing transitions between states in a similar way to finite state machines. But Kripke structures also model properties that hold in each state. Formally:

Definition A.1. A Kripke structure is given by a 4-tuple $M = (S, S_0, R, L)$ where

- S is a finite set of states;
- $S_0 \subseteq S$ is the set of initial states;
- $R \subseteq S \times S$ is a total transition relation, where for all $s \in S$, there exists $s' \in S$ such that $(s, s') \in R$;
- $L : S \to 2^{AP}$ is a labelling function that gives the set of propositions ($p \in AP$) that hold within a given state.

A common type of logic used to specify behaviour is temporal logic, which can be used to describe what propositions may hold about the system over time. Examples of temporal logics include Linear Temporal Logic (LTL) [122], Computation Tree Logic (CTL) [45] and the μ-calculus [98]. The definition of CTL is given and briefly studied as it will be useful for understanding Appendix A.2.3.

Before giving formal semantics of CTL, paths on a Kripke structure are first defined. Throughout, let $M = (S, S_0, R, L)$ be a Kripke structure.

Definition A.2. A path is a tuple $\sigma = (s_0, s_1, s_2, \dots)$ where $s_i \in S$ and we have that $(s_i, s_{i+1}) \in R$ for all i.

Note that a path can have infinite or finite length as long as there are suitable transitions.

Terms in CTL are given by state formulae, θ, and temporal operators, T, that only exist bound with path quantifiers. These formulae are defined inductively by

$$\begin{aligned} \theta &::= p \mid \neg\theta \mid \theta \vee \theta \mid \mathbf{E}T \mid \mathbf{A}T \\ T &::= \mathbf{X}\theta \mid \mathbf{F}\theta \mid \mathbf{G}\theta \mid \theta\mathbf{U}\theta, \end{aligned}$$

where $p \in AP$ is an atomic proposition in the model. The terms **X** ("next"), **F** ("eventually"), **G** ("always") and **U** ("until") denote basic temporal operators. The terms **A** (for all paths) and **E** (there exists a path) are path quantifiers. The semantics of the state and temporal operators described for a Kripke structure are given in Equation (A.3), where $\sigma = (s_0, s_1, \dots)$ denotes a path.

$$\begin{aligned} &[\![p]\!]_M = \{s \in S : p \in L(s)\} \\ &[\![\neg\theta]\!]_M = S / [\![\theta]\!]_M \\ &[\![\theta_1 \vee \theta_2]\!]_M = [\![\theta_1]\!]_M \cup [\![\theta_2]\!]_M \\ &[\![\mathbf{EX}\theta]\!]_M = \{s \in S : \exists t \in S \text{ such that } t \in [\![\theta]\!]_M \text{ and } (s,t) \in R\} \\ &[\![\mathbf{EG}\theta]\!]_M = \{s \in S : \exists \sigma \text{ such that } s = s_0 \text{ and } \forall k \in \mathbb{N}, s_k \in [\![\theta]\!]_M\} \\ &[\![\mathbf{EF}\theta]\!]_M = \{s \in S : \exists \sigma \text{ and } \exists k \in \mathbb{N} \text{ such that } s = s_0 \text{ and } s_k \in [\![\theta]\!]_M\} \\ &[\![\mathbf{E}\theta_1\mathbf{U}\theta_2]\!]_M = \{s \in S : \exists \sigma \text{ and } \exists k \in \mathbb{N} \text{ such that } s = s_0 \text{ and} \\ &\qquad\qquad s_i \in [\![\theta_1]\!]_M, s_j \in [\![\theta_2]\!]_M \text{ for } i < k, j \geq k\} \end{aligned} \tag{A.3}$$

Note that we can get the semantic formulas for $\mathbf{A}T$ for all the temporal operators, T, by replacing $\exists\sigma$ for $\forall\sigma$ (similarly replace $\exists t \in S$ for $\forall t \in S$) in the set definitions for the semantics.

Further, the operators **F** and **G** are obtained from **U** simply as $\mathbf{F}\theta = \mathit{true}\,\mathbf{U}\,\theta$ and $\mathbf{G}\theta = \neg\mathbf{F}\neg\theta$, where *true* is the atomic proposition true in all states.

It should be noted that CTL can be described with a subset of temporal and logical expressions as it is possible to create formulae from different terms. For example, the statements "there is no path such that eventually θ holds" and "for all paths $\neg\theta$ always holds" are equivalent and specified in CTL by $\neg\mathbf{EF}\theta = \mathbf{AG}(\neg\theta)$.

Example A.1. Figure A.1 gives a model and the relative computation tree is given in Figure A.2 with an example CTL operation.

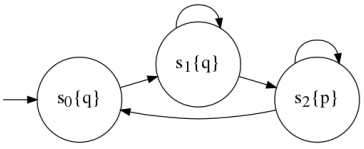


Figure A.1: An example of a Kripke structure with $S = \{s_0, s_1, s_2\}$.

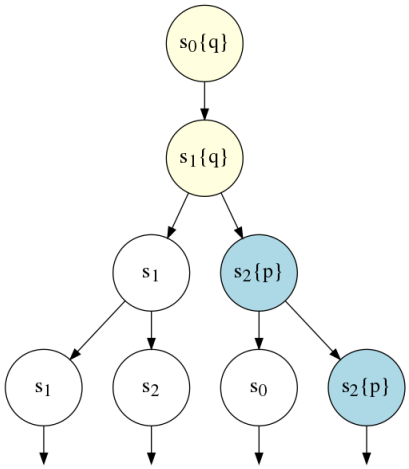


Figure A.2: A computation tree for the automata given in Figure A.1 that shows the CTL operation $\mathbf{E}(q\mathbf{U}p)$. The proposition q holds in states s_0 and s_1, but the proposition p holds in state s_2.

The model checking problem is to find all valid states that satisfy a temporal logic formula. Alternatively, one could just ask whether the formula is true in the initial states. Given M, a transition system using a set of states, S; and θ, a temporal logic formula; then find all $s \in S$ such that M and s model θ, the semantics of which is denoted by $M, s \models \theta \iff s \in [\![\theta]\!]_M$. Model checking has been explored in usage for verifying quantum programs against extensions of temporal logics. This can be seen in Appendix A.2.3 and Section 2.2.5.

A.2 Formal Quantum Verification Methods

This section aims to introduce the theoretical ideas that have been used in pursuit of the verification of quantum programs. While this section covers some theories, it is not a complete list. Further theories include quantum Markov chains [72] and quantum automata [97], which are given a brief introduction in a previous survey [159] with further references therein.

A.2.1 Quantum Weakest Preconditions

In deterministic programming, the weakest precondition [64] gives a method of transforming the problem of checking whether a program is valid in Hoare logic into a problem of determining whether a precondition implies the said weakest precondition. More formally for deterministic programs, given a program S and a predicate postcondition Q, then the weakest precondition $\text{wp}(S)(Q)$ is the precondition to S such that for all preconditions P with $\{P\}S\{Q\}$, then $P \implies \text{wp}(S)(Q)$.

Whilst a probabilistic version of the Hoare logic has been developed and can be used as a means to verify quantum programs [134], D'Hondt and Panangaden [63] demonstrated one can develop a quantum Hoare-style logic using density matrices. This then allows for the notion of a quantum weakest precondition. The difference in definition is that now the program S is a quantum program, where $S(\rho)$ is the density matrix after applying program S to density matrix ρ; the precondition P and postcondition Q are each a quantum predicate, which is a Hermitian operator with positive eigenvalues upper bounded by 1; and a valid precondition P must satisfy $\text{tr}(P\rho) \leq \text{tr}(Q\ S(\rho))$ for all density matrices ρ. Write $\{P\}S\{Q\}$ if P, Q and S follow the final inequality. Thus, the quantum weakest precondition $\text{wp}(S)(Q)$ is defined such that for all valid preconditions P, $\text{tr}(P\rho) \leq \text{tr}(\text{wp}(S)(Q)\rho))$ for all density matrices ρ.

With this notion, it is possible to change the verification problem of quantum programs to that of calculating quantum preconditions. In a sense, the quantum weakest precondition gives the most "general" precondition for a postcondition, meaning that as long as we have a "specific" precondition we can always reach the same postcondition as the "general" precondition. The quantum weakest precondition is a concept that can see usage in different verification systems depending on the language and design used. In Section A.2.2, we will see an example of its usage.

A.2.2 Quantum Hoare Logic

Ying [158] has been developing the Quantum Hoare Logic (QHL) over the last decade, as an extension of the standard Floyd-Hoare logic. Introduced in his original work, the classical while-language is extended to a quantum version and the Floyd-Hoare logic is amended to verify the extension. The quantum-while language is:

$$S ::= \textbf{skip} \mid q := 0 \mid \bar{q} := U\bar{q} \mid S_1; S_2 \mid \textbf{measure}\ M[\bar{q}] : \bar{S} \mid \textbf{while}\ M[\bar{q}] = 1\ \textbf{do}\ S,$$

where q is a qubit, $\bar{q}$ is a quantum register and $\bar{S} = \{S_m\}_{m\in\mathbb{M}}$ is an indexed set of quantum programs, where $\mathbb{M}$ is the set of measurement results with associated measurement operators $\{M_m\}_{m\in\mathbb{M}}$. The commands do the following operations: **skip** does nothing; $q := 0$ initialises a qubit; $\bar{q} := U\bar{q}$ performs a unitary operation on a number of qubits; $S_1; S_2$ is composition of statements; **measure** $M[\bar{q}] : \bar{S}$ measures several qubits and performs a program from $\bar{S}$ depending on the result of measurement, *i.e.*, if the measurement result is m, the program follows the commands given by $S_m \in \bar{S}$; and **while** $M[\bar{q}] = 1$ **do** S performs S until a "false" measurement is read.

The quantum-while language stands out because it does not define programs in terms of describing quantum circuits. This allows for an imperative approach for writing quantum programs, rather than the very low-level idea of constructing a circuit. Another feature to highlight is the use of measurement within the language. The **measure** command replaces the classical **if** statement and the **while** statement requires measurement on a set of qubits during each iteration.

The Quantum Hoare Logic then extends the Hoare triple $\{P\}S\{Q\}$, where S is a program written in the quantum-while language and P, Q are quantum predicates (as defined in Section A.2.1). However, these predicates are additionally upper bounded by the identity operator, I, and lower bounded by the zero operator, 0. They are bounded in that for any predicate P (used in QHL), then for all density matrices ρ we have $\mathrm{tr}(0\rho) \leq \mathrm{tr}(P\rho) \leq \mathrm{tr}(I\rho)$, thus $0 \leq \mathrm{tr}(P\rho) \leq 1$.

Inference rules can be used to create Hoare triples for quantum programs depending on the statement. Beyond that, the notion of weakest precondition can be used to generate valid Hoare triples. If a desired postcondition is wanted, then rules can be used to find the weakest precondition for a program. The correctness of evaluating the Quantum Hoare Logic on a program and the condition that evaluation terminates (notions of partial and total correctness respectively) are given in [158].

Work has continued on this logic over the last decade to further improve it and create an implementation, as will be seen in Section 2.2.2. As an example of an improvement to be made on the grammar, it is clear to see that there is no classical functionality defined. This prevents some quantum algorithms, such as Shor's algorithm [139], being fully implemented within the quantum-while language. A recent work [71] has extended the quantum-while language to include classical variables and the logic has been reworked to show the extension is verifiable. Another version of Quantum Hoare logic has been proposed in [94], which is not designed around

the usage of weakest precondition. Further, this other version verifies Selinger's QPL [136], a quantum programming language based on flowcharts and featuring classical bits as well as **if** statements.

Example A.2. An example of using Quantum Hoare logic and the quantum weakest precondition to verify Deutsch's algorithm (a one qubit version of the Deutsch-Jozsa algorithm [62]) is given. Given $f : \{0,1\} \to \{0,1\}$, recall that Deutsch's algorithm determines the value of $f(0) \oplus f(1)$ with a single evaluation of f. It should be noted that this example does not make use of an ancillary qubit for the sake of simplicity. Deutsch's algorithm in the quantum-while language is given as

$$\mathit{Deutsch} = [q := 0; q := Hq; q := O_f q; q := Hq; \textbf{measure } M[q] : \bar{S}_D],$$

where $\bar{S}_D = \{S_0 = \textbf{skip}, S_1 = \textbf{skip}\}$, H is the single-qubit Hadamard gate and O_f is the quantum oracle defined by the matrix

$$\begin{pmatrix} (-1)^{f(0)} & 0 \\ 0 & (-1)^{f(1)} \end{pmatrix}.$$

Further, the measurement operators M consist of measurements on the computational basis:

$$\left\{ M_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, M_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \right\}.$$

The result of receiving either measurement outcome in the **measurement** statement is to simply **skip**. Using the definition of the weakest precondition for Quantum Hoare Logic, the goal is to find the weakest precondition of the postcondition $\mathit{Post} = (1 - f(0) \oplus f(1)) |0\rangle \langle 0| + (f(0) \oplus f(1)) |1\rangle \langle 1|$. This postcondition states that q should be in the appropriate state depending on the value of $f(0) \oplus f(1)$. Here, the weakest precondition of the **measurement** statement is calculated and the rest of the calculation for the *Deutsch* program is given in Appendix A.3.1.

From Proposition 7.1 in [158],

$$wp.(\textbf{measure } M[\bar{q}] : \bar{S}).P := \sum_m M_m^\dagger (wp.(S_m).P) M_m,$$

and therefore,

$$wp.(\textbf{measure } M[q] : \bar{S}_D).Post := M_0^\dagger(wp.(S_0).Post)M_0 + M_1^\dagger(wp.(S_1).Post)M_1$$
$$= M_0^\dagger(wp.(\textbf{skip}).Post)M_0 + M_1^\dagger(wp.(\textbf{skip}).Post)M_1.$$

Further to this, the weakest precondition of **skip** is simply $wp.(\textbf{skip}).P = P$. Thus,

$$\begin{aligned} wp.(\textbf{measure } M[q] : \bar{S}_D).Post &:= M_0^\dagger(Post)M_0 + M_1^\dagger(Post)M_1 \\ &= (1 - f(0) \oplus f(1)) \left|0\right\rangle \left\langle 0\right| + (f(0) \oplus f(1)) \left|1\right\rangle \left\langle 1\right| \\ &= Post, \end{aligned}$$

giving the Hoare triple $\{Post\}\textbf{measure } M[q] : \bar{S}_D\{Post\}$.

Following the rules for quantum weakest precondition of the Quantum Hoare Logic, the resulting Hoare triple for the entire program is $\{I\}Deutsch\{Post\}$, where I is the single qubit identity operator. This weakest precondition means that *Deutsch* can have any precondition and the program will always produce the correct result. This is because quantum predicates are upper bounded by I and I is the most general precondition we can have such that *Post* is the postcondition.

A.2.3 Quantum Computation Tree Logic

Various notions of extending Computation Tree Logic (CTL) to the quantum case have been studied, and implemented into model checking algorithms. For example, [72] investigated a quantum extension of probabilistic CTL (PCTL). Recently, in [157] the authors created a different extension of CTL that uses the concept of fidelity (which measures how much a density matrix state is changed after being acted on by a super-operator).

Other temporal logics have been studied both as quantum extensions of the original logic [109, 161] and how temporal logics can model behaviour in quantum systems; examples of investigating linear temporal properties can be found in [24] and more general ω-regular properties in [69]. In this section, the notion of quantum computation tree logic given in [14] is presented.

The Quantum Computation Tree Logic (QCTL) is a temporal logic used to reason about the behaviour of a quantum Kripke structure. Formally:

Definition A.3. A (finite) quantum Kripke structure over a set of qubits qB and variables X is a tuple (S, R) where:

- $S \subset \mathcal{H}_{\text{qB}} \times \mathbb{R}^X$ is a set of pairs (ϕ, ρ), where ϕ is a quantum state of the qubits qB and ρ is an assignment of variables to reals;
- $R \subseteq S \times S$ is a relation such that for any (ϕ, ρ), there exists (ϕ', ρ') such that $((\phi, \rho), (\phi', \rho')) \in R$.

Note that in comparison to the standard definition of Kripke structures, propositions are embedded as variables into the state rather than labels on specific states. In the literature for model checking, structures vary between Kripke structures, Markov chains, transition systems and variations on those structures (such as the quantum Kripke structure described). These structures vary in definition but will follow the structure of a state transition system that has some labels (propositions) associated with the states of the structure. The main difference between such structures is the transition relation between states and the temporal logic used for specifying desired behaviour, which changes how these models are checked.

By combining the decidable fragment of the exogenous quantum propositional logic (dEQPL) [38] with the classical CTL used in model checking, we get QCTL. The grammar for dEQPL used in [14] is:

$$
\begin{aligned}
&\text{Classical Formulae}\\
\alpha ::= {} & \bot_c \mid \text{qb} \mid \alpha \Rightarrow_c \alpha,\\
&\text{Terms}\\
t ::= {} & x(\in Var) \mid m(\in \mathbb{Z}) \mid (t + t) \mid (tt) \mid \text{Re}(|\mathsf{T}\rangle_A) \mid \text{Im}(|\mathsf{T}\rangle_A) \mid \textstyle\int \alpha,\\
&\text{Quantum Formulae}\\
\gamma ::= {} & t \leq t \mid \bot_q \mid \gamma \Rightarrow_q \gamma.
\end{aligned}
$$

The logic on classical formulae, α, and quantum formulae, γ, are distinguished by using subscript c and q respectively. Further, other connectives can be abbreviated $(\neg, \wedge, \vee, \Leftrightarrow, \top)$ using $\bot$ and $\Rightarrow$.[1]

Classical formulae describe the set of qubits to measure from using classical logic statements and qubit symbols qb from qB. Terms describe numerical expressions that can be made with additional variables, constants, and functions for getting information about the quantum state. The term $\text{Re}(|\mathsf{T}\rangle_A)$ denotes the real part of the amplitude of the quantum state from a subset of qubit symbols, $A \subseteq$ qB

[1]Note that in [14] classical formulae do not use the subscript and different symbols are used for quantum formulae: $\neg_q$ is $\boxminus$, $\wedge_q$ is $\sqcap$, $\vee_q$ is $\sqcup$, $\Leftrightarrow_q$ is $\equiv$.

(similarly $\text{Im}(|\mathsf{T}\rangle_A)$ for the imaginary part). The term $\int\alpha$ denotes the probability that α holds when measuring all qubits. Finally, quantum formulae allow us to reason about terms by using logical expressions and comparison formulae. This allows us to reason about the state of a quantum system at a specific time step.

In [14], it is shown that QCTL is sound and (weakly) complete. Further, an algorithm was developed that checks if a QCTL formula is satisfiable by extending an algorithm used for model checking CTL. The grammar for QCTL is:

$$\theta ::= \gamma \mid (\theta \Rightarrow_q \theta) \mid \mathbf{EX}\theta \mid \mathbf{AF}\theta \mid \mathbf{E}[\theta \mathbf{U} \theta],$$
$$\text{where } \gamma \text{ is a dEQPL quantum formula.}$$

The semantics of the QCTL given above for the temporal operators are similar to CTL formula except they act over paths on a quantum Kripke structure. Note that QCTL uses a subset of the temporal logic operations from CTL and the other temporal operations can be derived from this subset.

Unlike some of the other formal methods discussed in this section, this logic would be used in a similar way to the model checking described in Section 2.1.2. This would involve converting properties of a quantum program or circuit into the QCTL language. By providing a specification for the program, the QCTL formula can be checked for model satisfiability using the algorithm given in [14]. The other QCTLs introduced at the start of this section [72, 157] found use in model checking quantum Markov chains.

Two examples of QCTL [14] formulae are given.

Example A.3. Denote $\Box\alpha$ as $(\int\alpha) = 1$. This is a dEQPL formula that states a classical formula α holds with probability 1 after measuring all qubits. The following formula is used in [14] as the formula for verifying a single bit version of the BB84 protocol [20]:

$$B = (\Box(b_A \Leftrightarrow_c b_B)) \Rightarrow_q \mathbf{A}[(\neg_q(\Box e))\mathbf{U}((\Box e) \wedge_q ((\Box k) \Leftrightarrow_q (\int m = 1)))].$$

The formula means "If Alice and Bob are in the same basis $(\Box(b_A \Leftrightarrow_c b_B))$, then down all (quantum) paths the protocol has not ended $(\neg_q(\Box e)))$ until it has ended $(\Box e)$ and the generated key bit is equal to the value of the qubit used $((\Box k) \Leftrightarrow_q (\int m = 1))$".

Example A.4. As a second example, consider Deutsch's algorithm. Denote $b =$

$f(0) \oplus f(1)$ as a classical bit. Further, use m as a classical bit to say when the algorithm has performed the measurement operation and let q denote the qubit used for the algorithm. Usage of m is required to specify temporal behaviour for Deutsch's algorithm. We then have the following formula:

$$D = \mathbf{A}[(\neg_q(\Box m))\mathbf{U}((\Box m) \wedge_q ((\Box b) \Leftrightarrow_q (\textstyle\int q = 1)))]. \tag{A.4}$$

The formula reads "for all (quantum) paths ($\mathbf{A}$), the qubit is not measured ($\neg_q(\Box m)$) until it is measured ($\Box m$) and the measured qubit gives us the correct result with certainty ($(\Box b) \Leftrightarrow_q (\int q = 1)$)". A quantum Kripke structure in Appendix A.3.2 is given that is based on Deutsch's algorithm and satisfies the formula D. That is, a pair (S, R) such that all states in the structure satisfy D is described.

A.2.4 Path Sums

Path sums are a representation of unitary operators in terms of a summation of exponential polynomials with different quantum states. This representation was used in [7] to show the equivalence of quantum programs. Equation (A.5) shows a path sum represented as a unitary operator. Note that $\mathbf{x} = (x_1, \ldots, x_n)$ is a collection of Boolean variables or constants (the input signature), P is a (phase) polynomial with inputs $\mathbf{x}$ and $\mathbf{y}$, and $f : \mathbb{Z}_2^n \times \mathbb{Z}_2^m \to \mathbb{Z}_2^n$ is a multi-variable Boolean function (the output signature):

$$U : |\mathbf{x}\rangle \to \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y} \in \mathbb{Z}_2^m} e^{2\pi i P(\mathbf{x},\mathbf{y})} \, |f(\mathbf{x}, \mathbf{y})\rangle \, . \tag{A.5}$$

Path sums are used to represent the semantics of Clifford+R_k circuits[2] for a fixed k. Reduction rules can be applied to decrease the size of the circuit and so it is easy to check the equivalence of quantum circuits. Experiments in [7] have shown that Clifford+T circuits with a large number of qubits and gates can be efficiently and automatically verified. This gives a useful representation to efficiently verify quantum circuits. However, it will require support through a translation tool to be used to verify circuits written in a high-level programming language, which could include classical components. An extension of path sums has been used for verifying a quantum programming language in the `QBricks` verification framework, which is

[2] See Appendix A.1.1 for information on Clifford circuits.

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \qquad \begin{pmatrix}1 & 0\\ 0 & e^{i\alpha}\end{pmatrix} \qquad e^{\frac{i\alpha}{2}}\begin{pmatrix}\cos\frac{\alpha}{2} & -i\sin\frac{\alpha}{2}\\ -i\sin\frac{\alpha}{2} & \cos\frac{\alpha}{2}\end{pmatrix}$$

Figure A.3: The Hadamard node (left), green spider (middle) and red spider (right) with their matrix representation. The vertices can have more than one edge enter and exit them, but for simplicity I consider the case when only one edge enters a vertex. The spiders represent rotations on an axis, with the green spider rotating around the Z-axis and the red spider rotating around the X-axis. Note that when $\alpha = 0$, the spiders are simply identity operators and when $\alpha = \pi$ the spiders represent the pauli-Z and pauli-X gate respectively.

generally discussed in Section 2.2.4.

Example A.5. The path sum representation of the Pauli gates are listed below:

$$\begin{aligned} X &: |x\rangle \to |1-x\rangle\,, \\ Y &: |x\rangle \to e^{2\pi i\frac{(2x+1)}{4}}\,|1-x\rangle\,, \\ Z &: |x\rangle \to e^{2\pi i\frac{x}{2}}\,|x\rangle\,. \end{aligned}$$

A.2.5 The ZX-Calculus

The ZX-calculus [50] can be utilised as an alternate form of verification. The ZX-calculus is a graphical tool designed to convert quantum circuits into a graphical model and back again (but not all graphs are quantum circuits). These graphical models or networks model wires in the form of edges and some operations in the form of vertices.

The ZX-calculus consists of three main vertices: the Hadamard node, green spiders and red spiders (shown in Figure A.3). Using a set of rewrite rules, it is possible to add or remove various vertices. This allows the creation of optimised circuits and comparison between circuits.

While the ZX-calculus is a useful tool for low-level verification, it is an example in highlighting the difference in what the other tools described are trying to achieve. The ZX-calculus is useful for optimising circuits and showing the equivalence of circuits. It can be used to verify properties of simple circuits, such as the teleportation protocol [50].

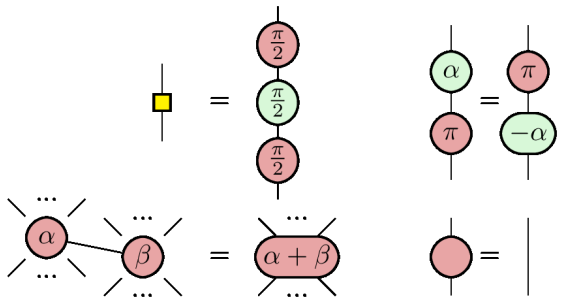


Figure A.4: A number of different rewrite rules [12]. It is important to note that the colours of spiders can be swapped (so green spiders replace red spiders and vice versa). The summation of scalars is modulo 2π.

However, recently, the ZX-calculus has also been used to verify some properties of oracle-based algorithms [36]. As research continues into the ZX-calculus, the calculus may reach a point where it can be used to represent programs. The Scalable ZX-calculus [37] can represent circuits with a parameterised number of qubits and is used in [36] to represent quantum algorithms. The ZX-calculus has several other applications[3] and a much more in-depth introduction can be found in [148].

Example A.6. Here a few rewrite rules from [12] are given in Figure A.4. Whilst many different rewrite rules can be used, these are a few key ones for this example.

Using these rules, it is now possible to show optimisations of different circuits. For example, we can use the rewrite rules to show that applying two Hadamard gates to a circuit is the same as doing nothing (performing the identity operation). This derivation is given in Figure A.5.

A.3 Examples

A.3.1 Quantum Hoare Logic Example

As seen earlier, $\{Post\}\textbf{measure}\ M[q] : \bar{S}_D\{Post\}$. The calculation of $wp.(q := Hq).Post$ is now given. For ease of notation, denote $c = 1 - f(0) \oplus f(1)$ and $b = f(0) \oplus f(1)$ (so $Post = (c\,|0\rangle\langle 0| + b\,|1\rangle\langle 1|)$. Using the weakest precondition rules

[3]Such as usage in circuit optimisation, surface codes and measurement-based quantum computation.

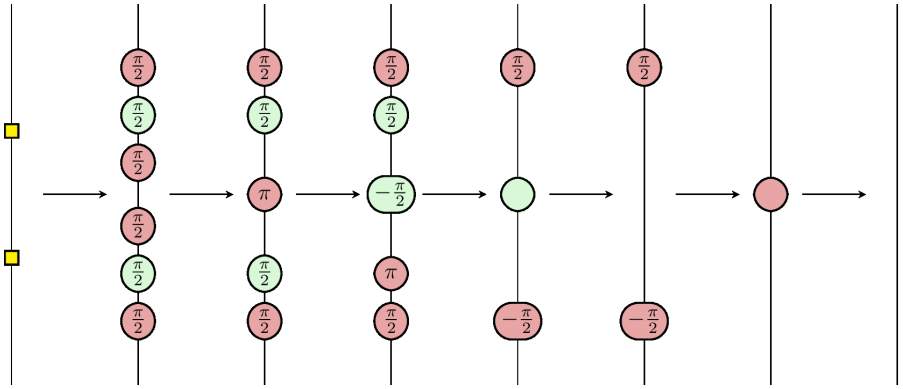


Figure A.5: Derivation of two Hadamard gates being equivalent to the identity operation using the rules of the ZX-calculus. The derivation uses instances of the rewrite rules given in Figure A.4. Firstly, the Hadamard gates are decomposed and then the red spiders are merged. Two of the spiders are swapped and then the green spiders are merged. This combined green spider can be removed by the identity rule and the final red spiders can be similarly merged and removed. An alternative way of proving the Hadamard gate is self-inverse can be found in [12].

from Proposition 7.1 in [158],

$$\begin{aligned} wp.(q := Hq).Post &= H^\dagger Post\ H \\ &= H(c\,|0\rangle\langle 0| + b\,|1\rangle\langle 1|)H \\ &= c\,|+\rangle\langle +| + b\,|-\rangle\langle -|\,, \end{aligned}$$

and so, the Hoare triple $\{c\,|+\rangle\langle +| + b\,|-\rangle\langle -|\}q := Hq\{Post\}$ holds. Next we find the weakest precondition of $q := O_f q$ for $\{c\,|+\rangle\langle +| + b\,|-\rangle\langle -|\}$. Using the same

rule,

$$\begin{aligned}
wp.(q := O_f q).(c\,|+\rangle\langle+| + b\,|-\rangle\langle-|) &= O_f^\dagger(c\,|+\rangle\langle+| + b\,|-\rangle\langle-|)O_f \\
&= O_f(c\,|+\rangle\langle+| + b\,|-\rangle\langle-|)O_f \\
&= ((-1)^{f(0)}\,|0\rangle\langle 0| + (-1)^{f(1)}\,|1\rangle\langle 1|) \\
&(c\,|+\rangle\langle+| + b\,|-\rangle\langle-|) \\
&((-1)^{f(0)}\,|0\rangle\langle 0| + (-1)^{f(1)}\,|1\rangle\langle 1|) \\
&= c((-1)^{f(0)+f(0)}\,|0\rangle\langle 0| + (-1)^{f(0)+f(1)}\,|0\rangle\langle 1| \\
&+ (-1)^{f(1)+f(0)}\,|1\rangle\langle 0| + (-1)^{f(1)+f(1)}\,|1\rangle\langle 1|) \\
&+ b((-1)^{f(0)+f(0)}\,|0\rangle\langle 0| - (-1)^{f(0)+f(1)}\,|0\rangle\langle 1| \\
&- (-1)^{f(1)+f(0)}\,|1\rangle\langle 0| + (-1)^{f(1)+f(1)}\,|1\rangle\langle 1|) \\
&= (c+b)(-1)^{2f(0)}\,|0\rangle\langle 0| \\
&+ (c-b)(-1)^{f(0)+f(1)}\,|0\rangle\langle 1| \\
&+ (c-b)(-1)^{f(1)+f(0)}\,|1\rangle\langle 0| \\
&+ (c+b)(-1)^{2f(1)}\,|1\rangle\langle 1| \\
&= |0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1| = |+\rangle\langle+|\,.
\end{aligned}$$

The penultimate equality holds because $c - b = 1 - 2(f(x) \oplus f(y)) = (-1)^{f(x)+f(y)}$. Thus, the Hoare triple $\{|+\rangle\langle+|\}q := O_f q\{c\,|+\rangle\langle+| + b\,|-\rangle\langle-|\}$ holds.

It is not hard to see that $\{|0\rangle\langle 0|\}q := Hq\{|+\rangle\langle+|\}$ with $|0\rangle\langle 0|$ being the weakest precondition of the given statement and postcondition.

Finally, the weakest precondition rule for initialisation can be used:

$$wp.(q := 0).P = |0\rangle_q\langle 0|\,P\,|0\rangle_q\langle 0| + |1\rangle_q\langle 0|\,P\,|0\rangle_q\langle 1|\,.$$

For simplicity, we can drop the q notation from the equation, since we only have a 1-qubit program. Thus, it is easy to see that

$$\begin{aligned}
wp.(q := 0).\,|0\rangle\langle 0| &= |0\rangle\langle 0|0\rangle\langle 0|0\rangle\langle 0| + |1\rangle\langle 0|0\rangle\langle 0|0\rangle\langle 1| \\
&= |0\rangle\langle 0| + |1\rangle\langle 1| = I,
\end{aligned}$$

and therefore the Hoare triple $\{I\}q := 0\{|0\rangle\langle 0|\}$ holds. By following the sequential rule, it is easy to see that $wp.(Deutsch).Post = I$ and the desired Hoare triple of $\{I\}Deutsch\{Post\}$ is achieved.

A.3.2 QCTL Deutsch Example

Firstly, the quantum Kripke structure is setup. For simplicity, variables are not used and the ρ terms from states are omitted. The set of qubits is denoted $\mathrm{qB} = \{m, b, q\}$ and the initial state of qubits is denoted $I = \{|0, b, 0\rangle : b \in \{0, 1\}\}$. The operation for Deutsch's algorithm is introduced. Again, for simplicity, all the operations are performed in a single unitary

$$U_D = (X \otimes I_2 \otimes I_2)(I_2 \otimes I_2 \otimes H)(I_2 \otimes I_2 \otimes O_f)(I_2 \otimes I_2 \otimes H) = X \otimes I_2 \otimes HO_fH.$$

Thus, the set of states is $S = \{{U_D}^n a\,|\psi\rangle : n \in \mathbb{N}_0, |\psi\rangle \in I, a \in \{1, -1\}\}$ and, since U_D is self-inverse, $S = \{\pm\,|0, 0, 0\rangle, \pm\,|0, 1, 0\rangle, \pm\,|1, 0, 0\rangle, \pm\,|1, 1, 1\rangle\}$. The transition relation is then given by $R = \{(|\psi\rangle, U_D\,|\psi\rangle) : |\psi\rangle \in S\}$ and the quantum Kripke structure is (S, R).

The specification given in Equation A.4 can be used but the temporal operators are converted to ones used in [14]:

$$\begin{aligned}
\theta_1 =& \neg_q(\Box m),\\
\theta_2 =& (\Box m) \wedge_q ((\Box b) \Leftrightarrow_q (\textstyle\int q = 1)),\\
D =& \mathbf{A}[\theta_1 \mathbf{U} \theta_2]\\
=& \mathbf{AF}\theta_2 \wedge_q \neg_q \mathbf{E}(\neg_q \theta_2 \mathbf{U} \neg_q(\theta_1 \vee_q \theta_2)).
\end{aligned}$$

The set of states from S that satisfy D can now be checked, which are written as

$$Sat_{(S,R)}(D) \subseteq S,$$

and can use the algorithm in Table 9 from [14] to determine the set. Firstly, note that

$$\begin{aligned}
Sat_{(S,R)}(D) =& \mathbf{FixedPoint}[\lambda X.\{R^{-1}X\} \cup X, Sat_{(S,R)}(\theta_2)] \wedge_q\\
& S \setminus \mathbf{FixedPoint}[\lambda Y.\{R^{-1}Y \cap Sat_{(S,R)}(\neg_q \theta_2)\}, Sat_{(S,R)}(\neg_q(\theta_1 \vee_q \theta_2))],
\end{aligned}$$

where $\mathbf{FixedPoint}[f, x_0]$ performs the computation $x_{i+1} = f(x_i)$ until $x_{i+1} = x_i$. Denote the set of states that satisfy a dEQPL formula, γ, by $[\![\gamma]\!]_{\mathrm{dEQPL}} = \{|\psi\rangle \in S : s \Vdash_{\mathrm{dEQPL}} \gamma\}$. The following evaluations of dEQPL formula give

$$[\![\Box m]\!]_{\mathrm{dEQPL}} = \{|\psi\rangle \in S : \sum_{b,q} \|\langle 1, b, q|\psi\rangle\|^2 = 1\},$$

$$\begin{aligned}
[\![\Box b]\!]_{\text{dEQPL}} =& \{|\psi\rangle \in S : \sum_{m,q} \|\langle m, 1, q|\psi\rangle\|^2 = 1\},\\
[\![\int q = 1]\!]_{\text{dEQPL}} =& \{|\psi\rangle \in S : \sum_{m,b} \|\langle m, b, 1|\psi\rangle\|^2 = 1\},\\
[\![(\Box b) \Leftrightarrow_q (\int q = 1)]\!]_{\text{dEQPL}} =& \{|\psi\rangle \in S : \sum_{m} \|\langle m, c, c|\psi\rangle\|^2 = 1 \text{ for } c \in \{0,1\}\}.
\end{aligned}$$

Next the set of states that satisfy the quantum formulas are found:

$$\begin{aligned}
Sat_{(S,R)}(\theta_1) =& S \setminus [\![\Box m]\!]_{\text{dEQPL}}\\
=& S \setminus \{|\psi\rangle \in S : \sum_{b,q} \|\langle 1, b, q|\psi\rangle\|^2 = 1\}\\
=& \{\pm|0,0,0\rangle, \pm|0,1,0\rangle\};\\
Sat_{(S,R)}(\theta_2) =& [\![\theta_2]\!]_{\text{dEQPL}}\\
=& \{|\psi\rangle \in S : \|\langle 1, c, c|\psi\rangle\|^2 = 1 \text{ for } c \in \{0,1\}\}\\
=& \{\pm|1,0,0\rangle, \pm|1,1,1\rangle\};\\
Sat_{(S,R)}(\neg_q \theta_2) =& S \setminus \{\pm|1,0,0\rangle, \pm|1,1,1\rangle\} = \{\pm|0,0,0\rangle, \pm|0,1,0\rangle\};\\
Sat_{(S,R)}(\neg_q(\theta_1 \vee_q \theta_2)) =& S \setminus ([\![\theta_1]\!]_{\text{dEQPL}} \cup [\![\theta_2]\!]_{\text{dEQPL}})\\
=& S \setminus (\{\pm|0,0,0\rangle, \pm|0,1,0\rangle\} \cup \{\pm|1,0,0\rangle, \pm|1,1,1\rangle\}) = \emptyset.
\end{aligned}$$

The evaluations of the fixed point evaluations are

$$\begin{aligned}
&\textbf{FixedPoint}[\lambda X.\{R^{-1}X\} \cup X, Sat_{(S,R)}(\theta_2)]\\
&\quad = \{\pm|0,0,0\rangle, \pm|0,1,0\rangle, \pm|1,0,0\rangle, \pm|1,1,1\rangle\} = S,
\end{aligned}$$

and

$$\begin{aligned}
&\textbf{FixedPoint}[\lambda Y.\{R^{-1}Y \cap Sat_{(S,R)}(\neg_q\theta_2)\}, Sat_{(S,R)}(\neg_q(\theta_1 \vee_q \theta_2))]\\
&= \textbf{FixedPoint}[\lambda Y.\{R^{-1}Y \cap \{\pm|0,0,0\rangle, \pm|0,1,0\rangle\}\}, \emptyset] = \emptyset.
\end{aligned}$$

Finally,

$$Sat_{(S,R)}(D) = S \cap (S \setminus \emptyset) = S \cap S = S.$$

All states in (S, R) satisfy D and thus the algorithm correctly returns the value b.

A.4 Design of Verification Frameworks and Quantum Programming Languages

In this section, several trade-offs and properties that are desired from a programming language for quantum verification are given. Whilst the trade-offs are similar to what classical theorem provers need to consider, different requirements need to be met for the language due to the nature of quantum computation. These criteria will later be used to highlight the differences of the available verification frameworks for quantum programming.

A.4.1 Trade-offs

Environment This concerns the environment in which the programmer creates their programs.[4] There are a few options available when considering this. Firstly, the language could be embedded within an available theorem prover. This gives the benefits of the host language, as well as access to libraries and community support. Further, someone familiar with the environment would be able to pick up the language fairly easily. However, embedding within a theorem prover does mean that the quantum verifier also suffers from the limitations of the said theorem prover. Numerous current tools take this approach, such as SQIR and QHLProver, which are discussed in Section 2.2. In general, a quantum programming language could be embedded in a classical language, such as how Quipper is embedded in Haskell [83], and a modified verification framework can be used to verify programs.

An alternative is to create a new environment dedicated to the verification of quantum programs from scratch. This gives more freedom in being able to meet the specification the designer creates. On the downside, it may take longer to develop than other methods. It will take longer to fully build a dedicated tool for the verification of programs versus building off or extending a well-known tool. This approach would be akin to building a tool such as an SMT solver (Z3, dReal, etc.) or model checker (PRISM, nuSMV, etc.).

Another approach is to extend a quantum programming language with a verification framework. This would allow both programs to be verified and executed on a simulator or quantum hardware. However, the choice of language needs to be considered and the framework would need to be updated whenever the underlying

[4]Some of the environment trade-offs discussed are common in other domain specific languages.

language gets updated. One example of this approach in the quantum setting is the `Entang`λ`e` framework [10]. `Entang`λ`e` is capable of translating Quipper [83] programs into quantum Markov chains [70], which can then be checked against temporal logic formulae.

Interactive vs Automated Here, one needs to consider whether the user should interact with the proofs they are constructing and how much should a tool automate the process of generating the proof. One can design a fully interactive prover, a fully automated prover or a mix between the two.

If the framework is designed to be interactive, then it may take more man-hours to construct proofs. However, this brings the benefit of the user obtaining a better understanding as to how a program is (or is not) valid. On the other hand, an automated-focused framework could only require the push of a button to prove programs, possibly leaving the user in the dark as to why the program is correct.

The size of systems requiring verification are a factor in which type of tool to use. A large system might not be suitable for an interactive approach, requiring the use of an automated approach. In [111], the authors highlight the use of model checkers and SMT solvers for the verification of avionics systems. The case studies provided contain systems with reachable state spaces of up to 1.5×10^{37} states.

Executability and Separability Another property to consider is what should be executable within the framework and how separable a program should be from its specification. For example, should specification be written within the program definition, in the same file as the program or within a separate file.

This will depend on the environment chosen for the framework and how mixed the specification is with the programs created. If a program is to be run on quantum hardware, the specification and verification of a program need to be separated from its definition. This is because the verification of quantum programs occurs classically (at least for now) whereas programs would be executed on quantum hardware. This puts the separation between the definition and execution of programs and specification as a high priority.

Running programs on a simulator is quite different. The reason for running a quantum program on a simulator is to check the outcomes of the program. In verification frameworks that provide counterexamples, simulators can be used to run and test the counterexample. This usage ensures that a counterexample is in fact

correct and is particularly useful for abstraction-based approaches, where the result from an abstraction may be different to the result from running the actual program (akin to CEGAR for model checking described within Section 2.1.2). Simulators can also be used for running small programs and the program is guaranteed to perform as expected. This allows the designer to freely decide how separable program and specification should be.

A.4.2 Limitations of Quantum Programs

Due to the difference in nature between quantum and classical computing, quantum programming languages have different requirements to classical ones. In [92] a discussion on a number of bugs to avoid when implementing a quantum programming language is given. Here a few of the ideas presented in [92] are discussed as are some other thoughts. The paper goes into more details on common bugs that programmers may create due to human error, but this is omitted.

No-cloning One of the key properties to meet is that the no-cloning theorem [156] is adhered to. The rules that quantum mechanics follows do not allow for the copying of arbitrary quantum data. Therefore, any quantum programming language should have inbuilt functionality to forbid (perfect) cloning. For example, if programs and proofs are mixed within the language, then the no-cloning theorem can be proved within the language. Alternatively, no-cloning can be achieved by making it built into the language through the use of types or linear logic.

Limited Classical Functionality A useful approach of designing a quantum computer is by having a classical computer being able to access a small quantum processor, known as quantum random access memory (QRAM), in a similar way to a graphics processing unit (GPU) [96]. The quantum part of the computer should perform purely quantum operations, having very little classical functionality within it. Because of this restriction, quantum programming languages need to be designed so that there is very limited classical functionality within the language as to avoid affecting the quantum system.

It should be noted though that there are some nice features that can come with having classical functionality. For instance, oracles implement classical functions into a quantum circuit and having classical functionality would allow for easy implementation of oracles. Silq [25] implements this in an easy-to-do way. Striking the

right balance of how much classical functionality to have is therefore very important to consider.

Depending on the amount of classical functionality, there are two ways to verify a program with quantum and classical components. One method is to combine a quantum verifier with a classical verifier, each handling the verification of separate parts of the program. The alternative would be for the verifier to have suitable logic to handle classical and quantum functionality together.

Program Parameters Moving away from low-level quantum circuit description languages and into the realm of quantum programs requires the use of parameters. Using parameters allows quantum programmers to easily describe their program in a very general manner and then provide specific values during runtime. For example, Grover's algorithm can use parameters to describe the size of the circuit and a general oracle function that it can take as input. This provides complexity from a verification perspective as the more parameterised a program is, the harder it is to verify for correctness or other properties.

For some quantum programs, it can be useful to use the output of measurement results to influence the control of a program. This concept is a form of *dynamical lifting*, which allows classical data to be used to affect the control of a quantum program. Classical data can be used as a parameter before or during runtime. Types of control flow include determining the number of wires to use, running different circuits after a qubit is measured (*e.g.*, the teleportation protocol) or a notification to redo the computation (*e.g.*, repeat until success loops). Recent extensions of Quipper have shown how dynamic lifting can be implemented in practice [51, 76]. Techniques such as dynamic lifting will require specific methods to be handled by verifiers.

Ancillary Cleaning Ancillaries are qubits that are introduced into the system in an initial state, used for a computation temporarily and then returned to their initial state. This process of returning the ancillary to its initial state is known as cleaning and is important for programmers or language designers to take into account. In classical computation, ancillary bits can be easily removed automatically by the processor through garbage collection, since classical bits can be discarded without affecting the state of the program. However, if ancillary qubits are not cleaned, there is the potential for measurement outcomes to be affected due to entanglement

between the main qubits and the ancillary ones. It should be decided by the designers whether programs should automatically uncompute ancillaries or if the programmer should perform this task.

A key feature in the programming language Silq [25] is that ancillaries are automatically uncomputed. Whether this feature will be seen in future verifiable programming languages is uncertain. From a verification perspective it would be advantageous to verify properties of ancillary qubits. For example, one such property is verifying that an ancillary qubit returns to its initial state so that it can be removed from the computation.

Types for Quantum Variables Another issue that is faced by designers of quantum programming languages is the design of types within their language. This includes problems such as whether measuring a quantum variable causes it to remain the same type, and what properties of a quantum variable need to be embedded into a type.

Another factor to consider when handling types is how to detect entanglement between different variables. Twist [163] features a unique typing system that allows the programmer to change the type of a variable depending on whether it is entangled with other variables or not. This is known as purity checking and it is handled through operations that are run before and during the running of a program. The techniques developed in Twist can be of use for future languages, but advancements need to be made to perform full purity checking before the program is run. Purity checking can be considered a form of verification and is a problem that needs to be explored with different tools.

Algorithm Milestones Designers of quantum languages often show the capabilities of their language by implementing a quantum algorithm. One should think of each algorithm as a milestone that should be reached. The lowest milestone to reach is being able to implement the Deutsch-Jozsa algorithm [62] or the quantum teleportation protocol [21] as these are fairly well-known algorithms. The next milestone would be to write either Grover's [85] or the Quantum Phase Estimation algorithm (many of these algorithms mentioned can be found in [114]). This is because both involve some form of iteration and are slightly more complex than the Deutsch-Jozsa and teleportation algorithms. The hardest algorithms to implement would be some of the 7 algorithms implemented in Quipper [83], which includes quantum walks on

Binary Welded Trees [42] and the Ground State Estimation algorithm [152].

For verifiable quantum programming languages, it is also necessary to be able to prove these programs run correctly. Many quantum programming languages are already able to implement several of the common "textbook" algorithms and a variety of algorithms in other fields, such as quantum chemistry [35]. So far, verifiable languages have only been able to prove about "textbook" algorithms. Whether they can prove facts about more advanced algorithms is yet to be seen. A discussion of two non-textbook algorithms and their challenges to verification is given in Appendix A.5.

A.5 Challenges in Verifying Complex Quantum Algorithms

Whilst many standard quantum algorithms have been verified in a number of the tools described above, specific techniques will be needed to verify complex algorithms. Here the problems that arise for verifying the Harrow-Hassidim-Lloyd algorithm [87] and the Binary Welded Tree quantum walk algorithm [42] are studied.

A.5.1 The Harrow-Hassidim-Lloyd Algorithm

The Harrow-Hassidim-Lloyd (HHL) [87] quantum algorithm solves linear systems of equations. Specifically, given a sparse, Hermitian matrix $\mathbf{A}$ and unit vector $\mathbf{b}$, find $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$.[5] Using a matrix $\mathbf{A}$ as described above, the classical Conjugate-Gradient method can be used to find $\mathbf{x}$ in time $O(N\kappa s \log(1/\epsilon))$, where N is the size of $\mathbf{A}$, κ is the condition number of $\mathbf{A}$, s denotes how sparse $\mathbf{A}$ is and ϵ is the error [137]. In the case of the HHL algorithm, the actual output is an approximation of $\mathbf{x}$ using a measurement matrix $\mathbf{M}$. This approximation can be found classically in $O(N\kappa \text{poly}(1/\epsilon))$. The run time of HHL is $O(\log(N)\kappa^2 s^2/\epsilon^3)$ and returns the answer with high probability, providing an exponential speedup with respect to N. The circuit for the HHL algorithm is given in Figure A.6 and some Silq code is presented in Figure A.7.

Here, a brief explanation of how the HHL algorithm works is given. Let $N = 2^n$ and the N-length unit vector $\mathbf{b}$ be represented by the quantum state $|b\rangle =$

[5] Although it is possible to change a linear system problem that uses a non-Hermitian matrix to one that uses a Hermitian matrix (details are within [87]).

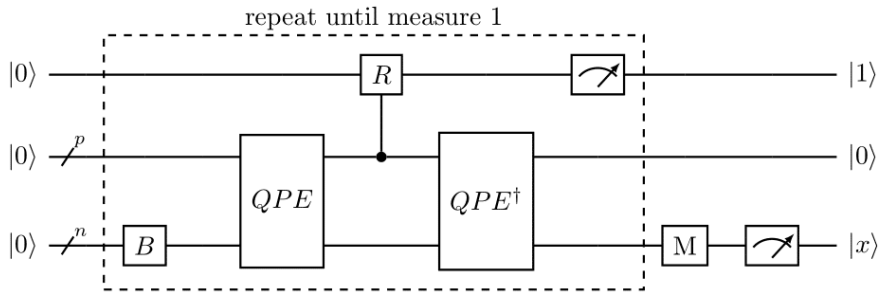


Figure A.6: The quantum circuit for the HHL algorithm.

```
// Init b
b_q := orB(b_q);

// QPE
eigen_qs := H_n[prec](eigen_qs);
b_q := qpe_ham_loop[n, prec](eigen_qs, t, hamU, b_q);
eigen_qs := reverse(qft[prec])(eigen_qs);

// Controlled rotation
anc := control_rot[n, prec](eigen_qs, approx_lambda, t/(2*pi), anc);

// Inverse QPE
eigen_qs := qft[prec](eigen_qs);
b_q := qpe_ham_loop[n, prec](eigen_qs, t, revhamU, b_q);
eigen_qs := H_n[prec](eigen_qs);

// Measure ancillary
repeat_loop = not(measure(anc));
```

Figure A.7: Main loop body for the HHL algorithm written in Silq. Full code available at `https://github.com/marco-lewis/silq-hhl`.

$\sum_{i=0}^{N-1} \mathbf{b}_i |i\rangle$. This is loaded into a n-qubit register using an operator B. Hamiltonian simulation is used to represent $\mathbf{A}$ in a quantum phase estimation call. The unitary matrix U in quantum phase estimation is $e^{-i\mathbf{A}t}$ for HHL and rotates $|b\rangle = \sum_j \beta_j |\mu_j\rangle$ around the eigenvectors of $\mathbf{A}$, which are $\mu_\mathbf{j}$ and $|\mu_j\rangle$ being their quantum state representation (similarly to $|b\rangle$).

Performing the quantum phase estimation (QPE) call entangles the eigenvalue representations of $\mathbf{A}$ in a new quantum register with their associated eigenvectors. This gives a quantum state of the form $\sum_j |\lambda_j\rangle |\mu_j\rangle$. The eigenvalues are then embedded into the phase of the quantum state using controlled rotations on an ancillary

qubit (using an operator R), leaving the quantum state as $\sum_j |\lambda_j\rangle |\mu_j\rangle (C |0\rangle + \frac{1}{\lambda_j} |1\rangle)$.

The quantum phase estimation routine is undone and the ancillary qubit is then measured. If it returns $|0\rangle$, then the entire quantum state is dumped and the algorithm is run again. If it returns $|1\rangle$, then the register that previously contained $|b\rangle$ now contains $|x\rangle = \sum_j \frac{\beta_j}{\lambda_j} |\mu_j\rangle$, a representation of $\mathbf{x}$. This can then be measured after applying $\mathbf{M}$, which changes the basis that $|x\rangle$ is measured in.

As mentioned previously, only the CoqQ tool can verify an instance of HHL with assumptions on the inputs to the algorithm. The HHL algorithm has a number of features that will make formal verification of an implementation challenging. Some of these features are not common in the standard algorithms and are discussed.

Repeat until Success One of the key aspects involved in this algorithm is the Repeat until Success loop that is dependent on the measurement of an ancillary qubit. This feature is also an aspect of Shor's algorithm [139], since there are possibilities for a measured result to be invalidated through classical checks. QHLProver is the only suitable verification framework that can handle Repeat-until-Success loops dependent on measurement outcomes. This is achieved through the verification of the **while** statement in the quantum-while language.

Subroutines As mentioned the HHL algorithm features the quantum phase estimation algorithm. This would need to be verifiable first before much progress could be made on the HHL algorithm. However, it is also important to consider what properties of quantum phase estimation need to be proved for different algorithms. For HHL, it will be important to show that $\mathbf{b}$ has a representation under the eigenvectors of $\mathbf{A}$. It will also be important to consider how the eigenvalue representations affects the phase of the quantum state. Verification will be needed here to prevent side effects within a program.

Hamiltonian Simulation Approximation Hamiltonian simulation is used to give a unitary approximation of the evolution of a Hamiltonian system. This technique has been used with matrices of different types for different purposes (as is the case in HHL). A verification tool will need to ensure that an implementation of an approximation is actually a good approximation of the behaviour that is expected. Techniques for verifying that implementations of a Hamiltonian are correct have not yet been studied. None of the tools given have verified an example of Hamiltonian

simulation. Tools embedded within theorem provers might be able to perform this verification but this will require separate study.

A.5.2 Walking on Binary Welded Trees

The problem statement of the Binary Welded Tree (BWT) algorithm [42] is that there are two trees, of the same depth, that are joined together by a weld at their leaves. The task is to walk from one root node (the entrance node) to the root node on the other tree (the exit node). When at a node, you only have local knowledge, where you cannot ask where you are on the tree but you can identify edges you have been down. For trees of depth n, one can walk to the weld and then walk down edges randomly until the exit node is reached. This is referred to as a random walk and has a worst case run time of $O(2^n)$, as one may visit almost every node in the tree.[6] The random quantum approach allows the walker to traverse down all edges from a node by entering a superposition of staying at the current node and traversing to a new node. This quantum approach achieves an exponential speed up in comparison to the classical approach, running in $O(\text{poly}(n))$ and the quantum system will collapse to the root of the other tree with high probability.

Each leaf node in the weld is connected to two leaves on the other tree. Further, the weld is designed such that it is cyclic and every leaf is in the cycle. An example of a binary welded tree is given in Figure A.8. To identify edges at a node, they are coloured. Edges only need to have a unique colour to the other edges on attached nodes. There is no constraint on the number of colours used, but it is possible to colour a tree using only 4 colours (this can be seen in Figure A.8).

Again, the reader should refer to [42] for full details on how the algorithm works. To explain briefly, a quantum register is used to represent the nodes and the register is initialised to the start node. The algorithm features a number of colour oracles that modify some ancillary register. A colour oracle returns a connected node if a node has an edge of that colour or the error state if not. Further, an additional qubit is used to flag the error state.

For a specific colour, the quantum register is put through the colour oracle. Then a rotation occurs through a Hamiltonian simulation. The entangled node register and ancillary register are rotated so that the connected node is slightly rotated into the quantum state. Then the colour oracle is reversed. The algorithm loops through

[6] One can achieve a polynomial classical algorithm, but this uses global knowledge of the walkers position on the graph.

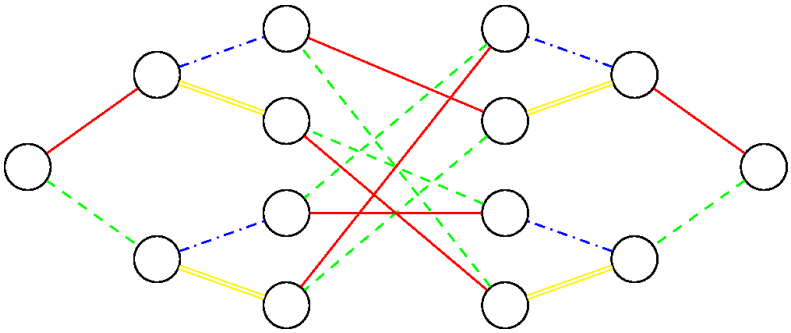


Figure A.8: Examples of a binary welded tree with 4 colours used to colour the edges (solid is red, dashed is greed, dot-dashed is blue and double-line is yellow)

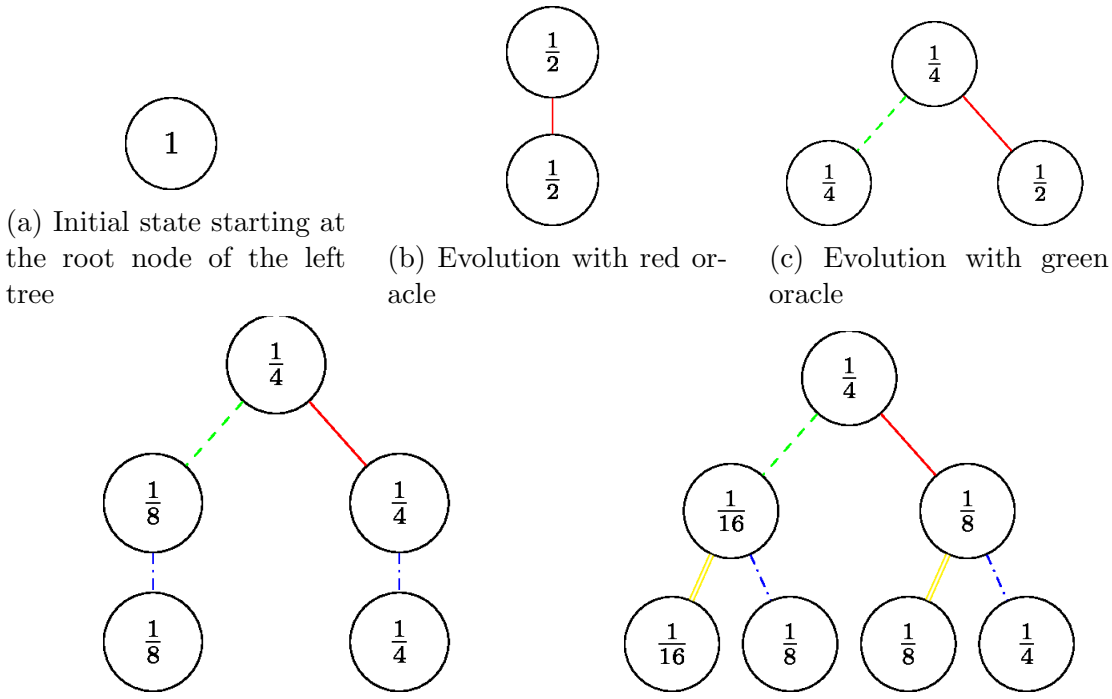
.

```
def Ham[n:!N](const G_or: !N x uint[n]->lifted uint[n], t:!R,
        node_reg:uint[n], neighbour_anc:uint[n], err_anc:B)
{
  for col in [0..numOfCols){
    // Perform first V_c oracle
    [neighbour_anc, err_anc] := V(G_or, col, node_reg,
        neighbour_anc, err_anc);

    // Simulation of T operation
    [node_reg, neighbour_anc, err_anc] := sim_T(t, node_reg,
        neighbour_anc, err_anc);

    // Perform second V_c oracle
    [neighbour_anc, err_anc] := V(G_or, col, node_reg,
        neighbour_anc, err_anc);
  }
  return (node_reg, neighbour_anc, err_anc);
}
```

Figure A.9: A section of Silq code [25] for the BWT Algorithm. The colours are looped over, V represents the colour oracle function and sim_T represents the Hamiltonian simulation that takes places. The colour oracle is used twice to uncompute the ancillaries (full code available at `https://github.com/marco-lewis/silq-binary-welded-trees`).



(a) Initial state starting at the root node of the left tree

(b) Evolution with red oracle

(c) Evolution with green oracle

(d) Evolution with blue oracle

(e) Evolution with yellow oracle

Figure A.10: Example of the first iteration performing the main procedure through each colour oracle. Inside each node is the probability of that node being measured. A variable can be set to change the proportion of probability moved to a connected node. In this example, half of the probability from a node is shared with its neighbour. Changing the order the oracles are called in would change the evolution of the system.

the colour oracles and repeats this loop a specified number of times. By selecting an appropriate time, the algorithm will stop and return the label of the root of the right tree with high probability. Figure A.10 shows a diagrammatic evolution of the quantum state for the first iteration through the colour oracles.

Through the description of the algorithm, one can see that there are various features that will be difficult to formally verify. Some of these features are discussed.

Loop Invariants Unlike the measurement based loops in the HHL algorithm, the BWT algorithm features two classical loops: one for the colour oracles and one for repeating the process. This leads to very deep circuits being created and so making the verification process slow from simply unwinding the entire loop. To verify these loops, either loop invariants need to be made or Bounded Model Checking could be used to unwind the procedure for a number of iterations. `QBricks` already features

an *iter* statement for repeating a quantum circuit a finite number of items. Such a feature could be used to verify aspects of BWT, but this is expected to be a challenging task.

Quantum Objects - Nodes vs Qubits Whereas most of the standard algorithms use qubits to represent values, the BWT algorithm works on a "quantum" graph. While physically this graph would be represented by qubits, it would be helpful for verification to verify about the nodes and graph structure easily. QuantumOne [149] is a recent endeavour into verifying about quantum objects rather than qubits.

Oracle Implementation In standard algorithms (*e.g.*, Deutsch-Jozsa), oracles only refer to a single function with defined constraints. However the oracles for the BWT algorithm are more complex, depending on a classical variable from a loop and featuring an error flag. This provides difficulty in terms of defining the expected behaviour for the oracle function.

Appendix B

Barrier Certificates Appendix

B.1 Proof for Continuous Barrier Certificates

B.1.1 Proof of Theorem 4.2

The proof is similar to the intuition given for Theorem 4.1.

Assume by contradiction that the system has a complex-valued convex barrier certificate, but the system is not safe. Therefore, there is an initial state $z(0) \in \mathcal{Z}_0$ and time $T \in \mathbb{R}^+$ such that $z(T) \in \mathcal{Z}_u$. By the definition of convex barrier certificate, we have that $B(z(0)) \leq 0$ and $B(z(T)) > 0$. Thus, the barrier must grow positively at some point during the system evolution. However, we have that $\frac{\mathrm{d}B(z(t))}{\mathrm{d}t} \leq 0$ for all $t \in \mathbb{R}^+$ based on Equation (4.13). The system cannot grow positively and so we have a contradiction. Therefore, the system must be safe. □

B.1.2 Proof of Proposition 4.3

Let $\dot{z} = f(z)$ be a system over $\mathcal{Z}$ with $\mathcal{Z}_0$ and $\mathcal{Z}_u$ being the initial and unsafe sets as before. Let $\mathcal{B}$ denote the set of (complex-valued convex) barrier certificates such that for any $B \in \mathcal{B}$ the system $f(z)$ is safe. Take $B_1, B_2 \in \mathcal{B}$ and consider the function $B(z) = \lambda B_1(z) + (1-\lambda)B_2(z)$, where $\lambda \in [0,1]$. Since $B_1(z) \leq 0$ and $B_2(z) \leq 0$ for all $z \in \mathcal{Z}_0$, then $B(z) \leq 0$ as well. A similar argument holds for $B(z) > 0$ for all $z \in \mathcal{Z}_u$. Finally, consider the differential equation $\frac{\mathrm{d}B}{\mathrm{d}t}$. It is trivial to see that

$$\frac{\mathrm{d}B}{\mathrm{d}t} = \lambda \frac{\mathrm{d}B_1}{\mathrm{d}t} + (1-\lambda)\frac{\mathrm{d}B_2}{\mathrm{d}t} \leq 0,$$

because differentiation is linear; and $\frac{\mathrm{d}B_1}{\mathrm{d}t}, \frac{\mathrm{d}B_2}{\mathrm{d}t} \leq 0$ for all $z \in \mathcal{Z}$. Therefore, B satisfies the properties of a barrier certificate for $f(z)$ and so $B \in \mathcal{B}$. Hence, $\mathcal{B}$ is convex. □

B.1.3 Derivation of Barrier for Hadamard System

By substituting $z_j\overline{z_j} = |z_j|^2$ and noting that $\mathrm{Re}\{z\} = z + \overline{z}$ for any $z \in \mathbb{C}$, we have that

$$B(z) = \frac{11}{5} - 3|(z)_0|^2 - \mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\} - |(z)_1|^2.$$

Since $|(z)_1|^2 = 1 - |(z)_0|^2$ (due to properties of quantum systems), we then have

$$B(z) = \frac{6}{5} - 2|(z)_0|^2 - \mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\},$$

and by simply rearranging we get

$$B(z) = 2(\frac{1}{10} - |(z)_0|^2 + \frac{1}{2} - \mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\}).$$

B.1.4 Proof of Proposition 4.4

This is proved by showing that B meets the conditions of a convex barrier certificate (given in Definition 4.6). Safety is then guaranteed from Theorem 4.2.

Firstly, consider $z \in \mathcal{Z}_0$. As $|(z)_0|^2 \geq 0.9$, then $B(z) \leq 2(-\frac{4}{5} - \mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\})$. Further, it can be seen that

$$\Big|\mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\}\Big| = |\mathrm{Re}\{z_0\}\,\mathrm{Re}\{z_1\} + \mathrm{Im}\{z_0\}\,\mathrm{Im}\{z_1\}| < 1 \times \sqrt{\frac{1}{10}} + 1 \times \sqrt{\frac{1}{10}} = \sqrt{\frac{2}{5}}.$$

Note that we are taking the maximal possible value of each component and therefore this is larger than the maximal value of $\mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\}$. Thus,

$$B(z) \leq 2(-\frac{4}{5} - \mathrm{Re}\Big\{(z)_0\overline{(z)_1}\Big\}) < 2(-\frac{4}{5} + \sqrt{\frac{2}{5}}) < 0.$$

A similar argument can be made for when $z \in \mathcal{Z}_u$ and it can be shown that $B(z) > 0$.

Finally, Equations (4.10) and (4.5.1) are used to get

$$\begin{aligned}\frac{\mathrm{d}B}{\mathrm{d}t} &= -\mathrm{i}\Big(-(2\overline{(z)_0}+\overline{(z)_1})((z)_0+(z)_1) - \overline{(z)_0}((z)_0-(z)_1) \\ &\quad + (2(z)_0+(z)_1)(\overline{(z)_0}+\overline{(z)_1}) + (z)_0(\overline{(z)_0}-\overline{(z)_1})\Big) \\ &= -\mathrm{i}\Big(-2\overline{(z)_0}(z)_1 - (z)_0\overline{(z)_1} + \overline{(z)_0}(z)_1 + 2(z)_0\overline{(z)_1} + \overline{(z)_0}(z)_1 - (z)_0\overline{(z)_1}\Big) \\ &= 0, \forall z \in \mathcal{Z}.\end{aligned}$$

Therefore, the system meets the conditions given in Equations (4.11), (4.12) and (4.13); the system is safe. □

B.1.5 Proof of Proposition 4.5 and 4.6

This subsection begins by proving Proposition 4.5.

Proof. Again, this is proved by showing that B_1 meets the conditions of a convex barrier certificate (given in Definition 4.6). Firstly, for any $z \in \mathcal{Z}_0^2$, we have that $|(z)_0|^2 \geq 0.9$, and

$$B_1(z) = 0.9 - (z)_0\overline{(z)_0} \leq 0.9 - 0.9 = 0.$$

When $z \in \mathcal{Z}_u$, a similar argument can be made since $|z_0|^2 < 0.89$, and

$$B_1(z) = 0.9 - (z)_0\overline{(z)_0} > 0.9 - 0.89 = 0.01 > 0.$$

Finally, using Equation (4.10) and (4.17), then

$$\begin{aligned}\frac{\mathrm{d}B}{\mathrm{d}t} &= -\mathrm{i}\Big(-\overline{(z)_0}((z)_0) + 0(-(z)_1) - (z)_0(-\overline{(z)_0}) + 0(\overline{(z)_1})\Big) \\ &= -\mathrm{i}\Big(-(z)_0\overline{(z)_0} + (z)_0\overline{(z)_0}\Big) = 0, \forall z \in \mathcal{Z}.\end{aligned}$$

Therefore, the system is safe. □

The proof of Proposition 4.6 is the same as the proof for Proposition 4.5 except swapping $(z)_0$ for $(z)_1$ where appropriate.

B.2 Phase Gate Example Extended

To extend the Z gate example to an n-qubit system, the dynamics are set as

$$f_t(z) = Z^{\otimes n} z.$$

Then,

$$\mathcal{Z}_0^p = \{z \in \mathcal{Z} : |(z)_p|^2 \geq 0.9\}, \text{ and}$$
$$\mathcal{Z}_u^p = \{z \in \mathcal{Z} : \sum_{j \neq p} |(z)_j|^2 \geq 0.2\},$$

for $p \in \{0, \ldots, 2^n - 1\}$ (where $|p\rangle$ is the target state) for the initial and unsafe space respectively.