Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea Triennale in Informatica

Structural Verification of Quantum Circuits: A Haskell Implementation via Hoare Logic

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Presentata da:
Alex Bastianini

Sessione Luglio 2026
Anno Accademico 2025/2026

Introduction

Ever since Richard Feynman first introduced the concept of quantum computing 45 years ago [9], the technology has shown great promise. Many early theoretical results highlighted its advantages over classical computing, most notably through breakthroughs like Shor's algorithm for integer factorization and Grover's algorithm for unstructured search [6, 10]. Thanks to the introduction of the first general use quantum computers in the last decade, some of said advantages have been leveraged in the real world to solve problems in optimization, quantum simulation, chemistry and machine learning [2, 3, 7, 12].

However, even with recent advancements in fault tolerant systems [1, 5], the limited amount of usable qubits (the quantum equivalent of classical bits) is still the main bottleneck in unlocking the full quantum advantage. Because of this, focusing on the efficiency and correctness of quantum programs written to run on these machines is crucial at this time. Due to the lack of reliable quantum hardware and the exponential cost of classical simulation, traditional approaches such as dynamic debugging and testing aren't yet viable ways to compute the structural cost and correctness of quantum programs.

Therefore, in order to analyse quantum code without running it, static analysis based formal methods - like Hoare logic or symbolic execution - are the main alternatives. As we will discuss in the following chapters, such methods can be used to prove structural and logical properties of quantum circuits, while also considering possible optimizations.

This thesis aims to develop a Hoare Logic that is able to reason about struc-

tural properties of quantum circuits created by a simple, `Qiskit`-inspired quantum language. In addition, we look to implement circuit optimisation rules directly into the formal logic.

Contributions of the paper: the main contribution of this thesis is the design and development of a formal verification tool written in Haskell[1]. This program is able to parse and verify a custom quantum language that has been expanded with assertions and Hoare-style preconditions and post-conditions, automatically generating the required verification conditions and proving them using an SMT solver.

Structure: The remainder of the thesis is structured as follows:

Chapter 1: Introduces essential theory of quantum computing (qubits, quantum states, quantum gates, circuits and optimisation) and formal verification (Hoare logic, verification condition generation and SMT solvers)

Chapter 2: Defines the quantum programming language and the syntax and derivation rules of our Hoare logic, while considering how verification can be automated. This chapter also discusses how circuit optimization can be implemented into the logic.

Chapter 3: Outlines the various components of our Haskell program, demonstrating how the logic introduced in the previous chapter can be implemented.

[1]The source code is available at https://github.com/AlexBastia/QisCount

Contents

Chapter 1

Theoretical Background

1.1 Quantum Computing

Whereas classical computers are based on *bits* that can either be switched "on" or "off", quantum computers work by carefully modifying the state of a *quantum* system. This fundamental difference in models of computation requires a complete remodelling of the mathematical representation that describes how a computer behaves.

1.1.1 Qubits

Qubits, short for *quantum bits*, are the fundamental element of quantum information. They are implemented as a binary property of a quantum system. This means that when said property is measured, the output is either "on" or "off", just like a classical bit. The difference is that *quantum superposition* allows the system to be in multiple states at the same time, and when the property is observed it stochastically collapses into a single base state. To model this behaviour, the state of a qubit $|\psi\rangle$ can be represented as a linear combination:

$$|\psi\rangle = c_0 |0\rangle + c_1 |1\rangle$$

where $|0\rangle$ and $|1\rangle$ indicate the "off" and "on" states and $c_0, c_1 \in \mathbb{C}$ such that $|c_0|^2$ and $|c_1|^2$ are the probabilities of the qubit being in the relative state when measured. The reason for using the squared modulus of complex numbers as opposed to directly using probabilities as coefficients is that this allows for addition to lower the resulting probabilities. This property is needed to correctly model *quantum interference*, the phenomenon where quantum particles act like waves that can combine, but that can also cancel each other out. As opposed to a wave however, a single qubit can also interact with itself.
So, by setting the base states of the qubit as the canonical base of $\mathbb{C}^2$ as follows:

$$\left\{ |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$

we can represent the state of a qubit as a normalised vector in this complex vector space:

$$|\psi\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$$

The "$|\cdot\rangle$" symbol, known as a *ket* in Dirac notation, now has an explicit meaning: it represents the column vector associated to the given state.

1.1.2 Multi-Qubit Systems

In classical computing, the state of multi-bit systems can be written as the Cartesian product of independent bit states. For example, the state of a byte can be written as

$$(1,0,0,1,1,0,1,0) \in \{0,1\}^8 = \overbrace{\{0,1\} \times ... \times \{0,1\}}^{8 \text{ times}}$$

However, this is not true in the quantum world because of a unique behaviour known as *entanglement*, which is when the states of two quantum particles become dependant on one another. This implies that the state of a multi-qubit system can't always be described as the product of independent states, forcing the construction of a new complex vector space.

Given two independent quantum systems of n and m qubits respectively, the quantum system obtained by merging them will have $\mathbb{C}^n \otimes \mathbb{C}^m \cong \mathbb{C}^{n \times m}$ as a state space, so the resulting state will be a complex vector of size $n \times m$. This exponential growth in state size is the reason why the classical simulation of quantum computers is inefficient.
The base states of a quantum system are often labelled using their decimal value, forming the *computational base*:

$$\{|\theta\rangle \mid \theta \in \{0, ..., 2^n - 1\}\}$$

Using this base, we can write the state of an n-qubit system as

$$|\psi\rangle = \sum_{\theta=0}^{2^n-1} c_\theta |\theta\rangle$$

1.1.3 Quantum Gates

As described by the Schrödinger equation, a closed quantum system can only undergo unitary evolution, thus the only way we can manipulate our quantum state without it collapsing is through unitary matrices. A square matrix U is unitary if its inverse is equal to its conjugate transpose $U^\dagger$ (often referred to as the Hermitian adjoint):

$$U^\dagger U = U U^\dagger = I$$

where I is the identity matrix. It can be proven that unitary matrices preserve the normalisation of state vectors, so these applications are consistent with our representation.
Therefore, all possible operators that act on a closed quantum system (known as *quantum gates*), are isomorphic to the group of unitary matrices of size n. As a result, quantum gates are limited to only reversible operations. Some of the most common gates include:

- **Pauli Matrices**:

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

X is the equivalent to a classical not gate, Z performs a phase flip and Y combines the two operations.

- **Hadamard**:

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

When applied to the base states, it creates an equal superposition where the probability of measuring $|0\rangle$ or $|1\rangle$ are the same. However, the phases would differ.

- **CNOT**:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

A controlled not operation, also labeled as CX. If the second qubit is $|1\rangle$, then a not operation is performed on the first. Otherwise, the gate acts as the identity.

- **SWAP**:

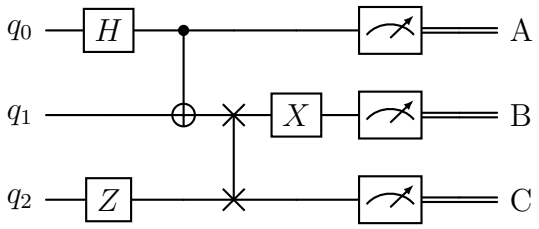
$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Swaps the state of the two qubits.

A set of gates is universal if, through composition, they can simulate any other quantum gate.

1.1.4 Quantum Circuits

A quantum program can be represented as a sequence of gates applied to a set of qubits, that can be depicted as follows:



Where the wires represent the life span of the qubits (usually initialized to $|0\rangle$) and gates are applied in time from left to right. The square with a meter indicates that the qubit has been measured, and the double line connects to the bit where the value is stored. The two "$\times$"s connected by a vertical line indicate a $SWAP$ gate, and the "$\oplus$" connected to the filled circle represents a $CNOT$, where the former is the target and the latter is the control.
Due to the reversible nature of quantum transformations, in a quantum circuit: two qubit lines can never converge into one, one qubit line can never duplicate and there can never be any loops.

We can define some structural properties of a quantum circuit:

- *Gate count*: number of gates.
- *Width*: number of qubits.
- *Depth*: time-units needed to compute the final value of a given qubit.

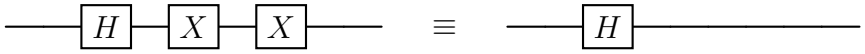
Gates that act on different qubits can be executed simultaneously, so the depth of a quantum circuit isn't necessarily equal to the gate count.

1.1.5 Circuit Optimization

Any circuit can be compressed into a single unitary transformation given by the composition of all the gates. It's possible that the construction of the circuit determined by the quantum program isn't the most efficient decomposition, given a fixed gate set, of the composed transformation with respect to some of the circuit's structural properties. This means that the circuit can

be optimized. There are different ways a quantum circuit can be transformed into an equivalent and more optimized one. Some examples include:

- **Gate Cancellation**: many quantum gates are their own inverse, like X and $CNOT$, meaning that if the gate is applied twice in a row it acts as the identity.

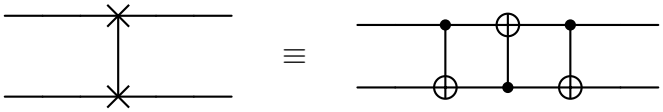


- **Gate Commutation**: two gates, with relative matrices U_1, U_2, are commuting if for any state $|\psi\rangle$:

$$U_1 U_2 |\psi\rangle = U_2 U_1 |\psi\rangle$$

  This means that it's possible to rearrange the order of operations without impacting the final state. This can reduce circuit depth through parallelism and allow more gate cancellations.

- **Template Matching**: this technique involves recognizing specific gate sequences and replacing them with optimized equivalents.



In addition, for some gates it is possible to define a set of conditions on the state of the qubits, known as *triviality conditions* (TCs), such that if these conditions are met for every possible base state that the system can collapse to, then the gate has no effect and can be removed. Formally, given a state $|\psi\rangle = \sum_{\theta=0}^{2^n-1} c_\theta |\theta\rangle$ and a set of TCs C, then the relative gate is trivial if, for each base state $|\theta\rangle$ where C doesn't hold, $c_\theta = 0$.

Gate	Triviality Conditions
$CNOT(q, c)$	$c == 0$
$SWAP(q_1, q_2)$	$q_1 == q_2$

In general, controlled gates are trivial if none of the basis states with non-zero probability have value 1 for all control qubits.

1.2 Formal Verification

Formally verifying a computer program means using formal or mathematical methods to prove its correctness with respect to a formal specification of the programming language used. This is the strongest possible way to determine if a certain block of code works as expected or not, and because of this formal verification is used mostly in mission-critical applications. We are interested in such techniques as they don't require running or simulating quantum code in order to reason about its properties. There exist many frameworks that allow for the formal analysis of code, we will focus on Hoare Logic as it is best suited for the purposes of this thesis.

1.2.1 Hoare Logic

Hoare logic is a formal proof system for proving properties of computer programs, introduced in 1969 by C.A.R. Hoare. The fundamental element of the logic is the *Hoare triple*, a compound assertion written as follows:

$$\{P\}C\{Q\}$$

where:

- C is a program (or a block of commands)
- P and Q are logical assertions, called *conditions*, on the program's state immediately before/after the execution of C. P is known as the

precondition and Q is the *postcondition*. They can be formulas of any arbitrary logic depending on the required expressiveness.

Such an assertion indicates that the program is *partially correct*: any terminating execution of C from a state satisfying A will end in a state satisfying B. If this holds true for any interpretation over the free variables in the assertions, then the triple is *valid* and we can write:

$$\models \{P\}C\{Q\}$$

Hoare logic constructs proofs by using axioms and inference rules, known as *Hoare rules*, that match with specific syntactic elements of a programming language. Some of the core rules are as follows:

- **Assignment Axiom**

$$\frac{}{\{Q[x \mapsto a]\}\ x := a\ \{Q\}}$$

  Where $Q[x \mapsto a]$ is the formula Q with all instances of the variable x substituted with the expression a.

- **Rule of Composition**

$$\frac{\{P\}C_0\{E\} \quad \{E\}C_1\{Q\}}{\{P\}C_0C_1\{Q\}}$$

- **Conditional Rule**

$$\frac{\{P \wedge b\}C_0\{Q\} \quad \{P \wedge \neg b\}C_1\{Q\}}{\{P\}\ \texttt{if}\ b\ \texttt{then}\ C_0\ \texttt{else}\ C_1\ \{Q\}}$$

- **Loop Rule**

$$\frac{\{I \wedge b\}C\{I\}}{\{I\}\ \texttt{while}\ b\ \texttt{do}\ C\ \{I \wedge \neg b\}}$$

- **Consequence Rule**

$$\frac{\models (P \implies P') \quad \{P'\}C\{Q'\} \quad \models (Q' \implies Q)}{\{P\}C\{Q\}}$$

  The notation $\models (P \implies P')$ and $\models (Q' \implies Q)$ indicates that the assertions $P \implies P'$ and $Q' \implies Q$ are valid. Note that we use $A \implies B$ as shorthand for $\neg A \vee B$.

If there exists a derivation for which $\{P\}C\{Q\}$ is a conclusion, then the triple is *provable* and we can write:

$$\vdash \{P\}C\{Q\}$$

It can be proven that Hoare logic is *sound*, meaning that if a triple is provable then it is also valid:

$$\vdash \{P\}C\{Q\} \implies \models \{P\}C\{Q\}$$

While Hoare Logic provides a framework for verifying proofs, it isn't well suited for proof automation. The following section will discuss how we can do just that.

1.2.2 Weakest Precondition Calculus

In order to build a verification engine that would be able to compute the validity of a Hoare triple, Edgar Dijkstra proposed a way to transform the logic into a backward-directed deterministic function.
To surpass the inherit non-determinism of Hoare rules, this function must operate on *annotated* code, which requires the programmer to manually indicate the invariant for while commands.
Given a piece of annotated code C and a postcondition Q, the *weakest precondition* (WP) function $\mathcal{WP}(C, Q)$ calculates the most permissive precondition such that $\{\mathcal{WP}(C, Q)\}C\{Q\}$ is a valid triple. Being the weakest possible assertion implies that:

$$\models \{P\}C\{Q\} \quad \iff \quad (P \implies \mathcal{WP}(C, Q))$$

therefore we can simply verify the right side, known as a *verification condition* (VC), in order to prove the correctness of a triple.
The algorithm used to calculate the WP operates backward from the postcondition, a technique called *backpropagation*. By using deterministic rules for each type of command, it's possible to propagate the WP starting from the postcondition. Some of the main rules are as follows:

- **Assignment**

$$\mathcal{WP}(x \leftarrow E, Q) = Q[x \mapsto E]$$

- **Composition**

$$\mathcal{WP}(C_1; C_2, Q) = \mathcal{WP}(C_1, \mathcal{WP}(C_2, Q))$$

- **Conditional**

$$\mathcal{WP}(\text{if } b \text{ then } C_1 \text{ else } C_2, Q) = (b \implies \mathcal{WP}(C_1, Q) \wedge (\neg b \implies \mathcal{WP}(C_2, Q)))$$

- **Loop**

$$\mathcal{WP}(\text{while } b \text{ do } [I]\ C, Q) = I$$

In addition to calculating the weakest precondition (that can then be propagated to the start of the program to generate the final VC) other VCs need to be proven independently in order to verify the validity of the invariant:

$$I \wedge b \implies \mathcal{WP}(C, I) \quad \text{and} \quad I \wedge \neg b \implies Q$$

Once the algorithm reaches the start of the program, an automated theorem prover can be used to prove the generated VCs.

1.2.3 SMT Solvers

Boolean satisfiability (SAT) is the problem of determining if there exists an interpretation of a Boolean formula for which the formula evaluates to true. If such an interpretation exists, the formula is *satisfiable*. Unfortunately this problem is NP-Complete, meaning there will probably never be an algorithm that solves it in polynomial time. However, thanks to extensive research in this field, some efficient algorithms have been developed that terminate in reasonable time (even if a solution hasn't been found). We can use these algorithms to prove that a formula φ is valid by checking that the negated formula is unsatisfiable:

$$\models \varphi \iff \neg\varphi \notin \text{SAT}$$

The issue that stops us from using SAT solvers to directly verify VCs is the fact that the logic used to write assertions about a program's state often requires non boolean expressions. The generalization of SAT to more complex formulas is known as *satisfiability modulo theories* (SMT). SMT is the problem of deciding if a formula in first-order logic, possibly including predicates over sets of non-binary values, is satisfiable. This allows for formulas involving integers, real numbers and even some data structures like arrays and lists. Predicates over these values are evaluated using their respective theories: for example, linear inequalities with real variables are evaluated using the rules of the theory of linear real arithmetic.
SMT solvers are the algorithms that attempt to solve SMT instances. Because first-order logic is only *semi*-decidable, most SMT solvers restrict the use of quantifiers in the formulas they recognize. In addition, only decidable theories can be properly implemented by an SMT solver.
The modern approach to implementing an SMT solver, known as the DPPL(T) architecture, is to decouple the theory-specific solvers (*T-solvers*) from the SAT problem over the predicates. These two modules interact in the following way:

1. A DPLL-based SAT solver generates a valid truth assignment for all boolean variables and predicates. If this fails, the formula is unsatisfiable.

2. The predicate literals are passed to their respective T-solvers that verify if the constraints can coexist. If they can, the formula is satisfiable.

3. If the T-solver detects a contradiction, it sends an explanation of infeasibility to the SAT solver and the algorithm jumps back to step 1.

Chapter 2

Building a Hoare Logic for a Simple Quantum Language

The goal of this chapter is to define a quantum programming language, followed by the construction of a Hoare logic for said language that allows for the description of structural properties of quantum circuits. We will also explore how circuit optimization can be integrated into the logic.

2.1 Annotated Proto-Qiskit

The language we will use is a simple quantum language based on the Python library `Qiskit`, called *Proto-Qiskit*[1]. To prepare code written in this language for automatic verification, additional syntax for annotations and conditions will be defined. Some slight changes will be made to the original syntax to make the code more readable.

2.1.1 Syntax

The syntax for the language can be divided into three main groups: expressions, commands and the final program triple. The symbol ϕ will be used

[1] A more detailed definition of the language can be found in [11].

to refer to formulae in the assertion logic that will be defined in Subsection 2.2.1.

Expressions

$$e ::= k \mid x \mid f^n(e_1, ..., e_n) \mid \texttt{QuantumCircuit}(e_1, ..., e_n)$$

Where:

- k is a constant.
- x is a program variable.
- f^n is a placeholder for common boolean, arithmetic and comparison operators. Standard programming language syntax is used (eg. `!`, `&&`, `||`, `%`, ...).

Commands

$$\begin{aligned} c ::= {} & x \texttt{ <- } e; \mid \{c_1 ... c_n\} \mid \texttt{if } e\texttt{:}c_1 \texttt{ else:}c_2 \mid \texttt{while } e\texttt{:}[\phi]\ c \mid \\ & x.U(e_1, ..., e_m); \mid x.\texttt{measure}(e_1, e_2); \mid x.\texttt{compose}(e); \mid \\ & x.\texttt{addBits}(e); \mid x.\texttt{addQubits}(e); \end{aligned}$$

Where:

- The assertion between brackets $[\phi]$ in the while command is the loop invariant.
- U is a unitary transformation on qubits $e_1, ..., e_m$.

Hoare Triples

A Hoare triple takes the following form:

$$p ::= \{\phi_1\}c\{\phi_2\}$$

where ϕ_1 is the precondition and ϕ_2 is the postcondition.

2.1.2 Types

The expressions in Annotated Proto-Qiskit will return values in $\mathbb{B} = \{\text{True}, \text{False}\}$, $\mathbb{Z} = \{..., -1, 0, 1, ...\}$ or $\mathbb{CIRC}$. The latter is the set of quantum circuits:

$$\mathbb{CIRC} \coloneqq \{(\text{inst}, \text{qs}, \text{bs}) \mid \text{inst} \in \mathbb{INST}^*, \text{qs} \in \mathbb{N}, \text{bs} \in \mathbb{N}\}$$

where qs is the number of qubits in the circuit, bs is the number of bits and inst is a sequence of instructions which are defined as follows:

$$\mathbb{INST} \coloneqq \{(g, (q_1, ..., q_m)(b)) \mid g \in \mathcal{G}, (q_1, ..., q_m) \in \mathbb{N}^m, b \in \mathbb{N}\}$$

with $g \in \mathcal{G}$ being the name of a gate and $(q_1, ..., q_m)$ being the indexes of the qubits bit that the gate is applied to. In addition, g can also be a measurement: in this case $g = meas$ and the classical bit index b is supplied.

For the sake of brevity we will not formally define the typing rules of Proto-Qiskit[2]. From now on it can be assumed that programs are well-typed.

2.1.3 Operational Semantics

We can now describe the operational semantics of the syntax introduced above.

[2]These can be found in [11].

Definition 1 (Operational Semantics of Proto-Qiskit)**.** *Given the sets VAL* $\coloneqq \mathbb{N} \cup \mathbb{B} \cup \mathbb{CIRC}$ *(all possible values) and VAR (the set of variables in the program), we can define the state as a function* $\sigma : VAR \to VAL$ *that returns the value of a given variable. The semantics of an expression can now be defined as a function* $[\![\cdot]\!]_\sigma : EXPR \to VAL$ *such that:*

$$
\begin{aligned}
[\![c]\!]_\sigma &= c \\
[\![x]\!]_\sigma &= \sigma(x) \\
[\![f^n(e_1, ..., e_n)]\!]_\sigma &= f([\![e_1]\!]_\sigma, ..., [\![e_n]\!]_\sigma) \\
[\![\texttt{QuantumCircuit}(e_1, e_2)]\!]_\sigma &= (\varepsilon, [\![e_1]\!]_\sigma, [\![e_2]\!]_\sigma)
\end{aligned}
$$

where c and f are the interpretations of constants and functions respectively in VAL and $VAL^n \to VAL$*.*

The command semantics can be described in big-step notation, where $(c, \sigma) \Downarrow \sigma'$ indicates that the execution of a command c in the state σ terminates on state σ'. We will now describe some of the most notable inference rules.

Unitary

This rule appends a unitary transformation to the instruction set, provided the referenced qubits are within the circuit's bounds:

$$
\frac{\begin{array}{c}[\![x]\!]_\sigma = (inst, qs, bs) \quad [\![e_1]\!]_\sigma = n_1 \quad \cdots \quad [\![e_m]\!]_\sigma = n_m \\ n_1 \leq qs \quad \cdots \quad n_m \leq qs\end{array}}{(x.U(e_1, \ldots, e_m),\ \sigma) \Downarrow \sigma[x \mapsto (inst; U(n_1, \ldots, n_m)(),\ qs,\ bs)]}
$$

Composition

This rule concatenates the instruction sequences of two circuits, provided the second circuit fits within the bounds of the first:

$$\frac{[\![x]\!]_\sigma = (inst_1, qs_1, bs_1) \quad [\![e]\!]_\sigma = (inst_2, qs_2, bs_2) \quad qs_2 \leq qs_1 \quad bs_2 \leq bs_1}{(x.\,\texttt{compose}(e),\ \sigma) \Downarrow \sigma[x \mapsto (inst_1 :: inst_2,\ qs_1,\ bs_1)]}$$

Measurement

This rule appends a measurement instruction, provided the referenced qubit and bit indices are within the circuit's bounds:

$$\frac{[\![x]\!]_\sigma = (inst, qs, bs) \quad [\![e_1]\!]_\sigma = n_1 \quad [\![e_2]\!]_\sigma = n_2 \quad n_1 \leq qs \quad n_2 \leq bs}{(x.\,\texttt{measure}(e_1, e_2),\ \sigma) \Downarrow \sigma[x \mapsto (inst; \text{meas}(n_1, n_2),\ qs,\ bs)]}$$

Adding Qubits/Bits

These rules increment the qubit or bit count of a circuit by the given amount:

$$\frac{[\![x]\!]_\sigma = (inst, qs, bs) \quad [\![e]\!]_\sigma = n}{(x.\,\texttt{addQubits}(e),\ \sigma) \Downarrow \sigma[x \mapsto (inst,\ qs + n,\ bs)]}$$

$$\frac{[\![x]\!]_\sigma = (inst, qs, bs) \quad [\![e]\!]_\sigma = n}{(x.\,\texttt{addBits}(e),\ \sigma) \Downarrow \sigma[x \mapsto (inst,\ qs,\ bs + n)]}$$

2.2 A Hoare Logic for Annotated Proto-Qiskit

Given the formal definition of our language, we can now build a logic system for it. The construction of a Hoare logic requires a logic for the description of a program's state and a set of Hoare rules that detail how commands modify said state.

2.2.1 Assertion Logic

It is important that the underlying logic used for assertions be expressive enough to describe the useful components of the program's state. For our purposes, this includes comparisons between arithmetic expressions including special functions for circuit size metrics. Therefore, in addition to the boolean and arithmetic expressions defined previously for Proto-Qiskit, the program state logic will have to include descriptions of quantum circuits and operations over them that return structural information. Thus the assertion formulae can be considered a superset of the program's boolean expressions, allowing us to only define the additional circuit functions.

Syntax

The assertion logic is a quantifier-free many-sorted first-order logic over the sorts $\{Num, Bool, Circ\}$, that corresponds to the types included in VAL. In fact, logic expressions and formulae are a superset of Boolean program expressions, so a complete definition is not needed as it can be inferred from the previous section. There are three new functions that describe circuit metrics:

- $width : \mathbb{CIRC} \to \mathbb{N}$, returns the width (total number of qubits and bits) of a circuit
- $gatecount : \mathbb{CIRC} \to \mathbb{N}$, returns the number of gates in a circuit
- $depth : \mathbb{CIRC} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, returns the depth of a circuit relative to a qubit

Furthermore, commands that modify quantum circuits are added to the logic by treating them as constructors:

- $U : \mathbb{CIRC} \times \mathbb{N}^m \to \mathbb{CIRC}$, for every m-qubit unitary operation U. $U(C, q_1, \ldots, q_m)$ represents circuit C with U appended to qubits $q_1, \ldots, q_m$

- *measure* : $\mathbb{CIRC} \times \mathbb{N} \times \mathbb{N} \to \mathbb{CIRC}$. $measure(C, q, b)$ represents C with qubit q measured and the result stored in bit b
- *compose* : $\mathbb{CIRC} \times \mathbb{CIRC} \to \mathbb{CIRC}$. $compose(C, D)$ represents the sequential composition of circuits C and D
- *addQubits* : $\mathbb{CIRC} \times \mathbb{N} \to \mathbb{CIRC}$. $addQubits(C, n)$ represents C with n qubits added
- *addBits* : $\mathbb{CIRC} \times \mathbb{N} \to \mathbb{CIRC}$. $addBits(C, n)$ represents C with n bits added

Both of these setts are added to the functions $f^n(e_1, ..., e_n)$ in logical expressions. To isolate operations allowed in logic formulae, we define predicates $P^n(e_1, ..., e_n)$ as a subset of f^n with Boolean return values but non Boolean operands.

Semantics

Logic expression semantics is defined the same way as with program expressions, with the addition of interpretations for the extra functions. The semantics of our circuit size metrics are evaluated as follows:

- $[\![width]\!]((inst, qs, bs)) = qs + bs$
- $[\![gatecount]\!]((inst, qs, bs)) = |inst|$
- $[\![depth]\!]((\varepsilon, qs, bs), q) = 0$
- $[\![depth]\!]((inst;\ op(q_1, \ldots, q_n)(\ldots), qs, bs), q) = \begin{cases} \hat{D} + 1 & \text{if } q \in \{q_1, \ldots, q_n\} \\ D & \text{otherwise} \end{cases}$
  where $\hat{D} = \max\{[\![depth]\!]((inst, qs, bs), q_{mid}) \mid q_{mid} \in \{q_1, \ldots, q_n\}\}$ and $D = [\![depth]\!]((inst, qs, bs), q)$.

The circuit constructors are interpreted as follows:

- $[\![QuantumCircuit]\!](qs, bs) = (\varepsilon, qs, bs)$

- $[\![U]\!]((inst, qs, bs), q_1, \ldots, q_m) = (inst;\ U(q_1, \ldots, q_m)(),\ qs,\ bs)$
- $[\![measure]\!]((inst, qs, bs), q, b) = (inst;\ \text{meas}(q)(b),\ qs,\ bs)$
- $[\![compose]\!]((inst_1, qs_1, bs_1), (inst_2, qs_2, bs_2)) = (inst_1 :: inst_2,\ qs_1,\ bs_1)$
- $[\![addQubits]\!]((inst, qs, bs), n) = (inst,\ qs + n,\ bs)$
- $[\![addBits]\!]((inst, qs, bs), n) = (inst,\ qs,\ bs + n)$

Definition 2 (Assertion Semantics)**.** *The interpretation of a well-formed formula ϕ (which is assumed to also be well-typed) is defined as a relation $[\![\phi]\!] \subseteq \Delta$, where Δ is the set of well-typed states $\delta : VAR \to VAL$, such that:*

$$
\begin{aligned}
[\![\texttt{true}]\!] &= \Delta \\
[\![\texttt{false}]\!] &= \emptyset \\
[\![P^n(e_1, ..., e_n)]\!] &= \{\delta \in \Delta \mid ([\![e_1]\!]_\delta, ..., [\![e_n]\!]_\delta) \in [\![P^n]\!]\} \\
[\![\phi \ \texttt{\&\&}\ \psi]\!] &= [\![\phi]\!] \cap [\![\psi]\!] \\
[\![\phi \ \texttt{||}\ \psi]\!] &= [\![\phi]\!] \cup [\![\psi]\!] \\
[\![\texttt{!}\phi]\!] &= \Delta \setminus [\![\phi]\!]
\end{aligned}
$$

where $[\![P^n]\!]$ is the set of all (well-typed) n-tuples $(e_1, ..., e_n)$ such that $P^n(e_1, ..., e_n)$ is true.

A state δ satisfies a formula ϕ if $\delta \in [\![\phi]\!]$, this can be written as $\delta \models \phi$. We say that a formula ϕ is *valid* if $\delta \models \phi$ for all well typed states δ, and we write $\models \phi$.

2.2.2 Derivation Rules

The basic Hoare rules for sequences, conditionals, and loops remain unaltered. Since our assertion logic mirrors the circuit mutation commands as pure constructor functions, the non-standard quantum commands can be

treated as assignment operations through substitution.

UNITARY

$$\frac{}{\{\, \phi[x \mapsto U(x, e_1, \ldots, e_m)] \,\} \quad x.\, U(e_1, \ldots, e_m) \quad \{\, \phi \,\}}$$

MEASURE

$$\frac{}{\{\, \phi[x \mapsto \mathit{measure}(x, e_1, e_2)] \,\} \quad x.\, \mathrm{measure}(e_1, e_2) \quad \{\, \phi \,\}}$$

COMPOSE

$$\frac{}{\{\, \phi[x \mapsto \mathit{compose}(x, e)] \,\} \quad x.\, \mathrm{compose}(e) \quad \{\, \phi \,\}}$$

ADD-QUBITS

$$\frac{}{\{\, \phi[x \mapsto \mathit{addQubits}(x, e)] \,\} \quad x.\, \mathrm{addQubits}(e) \quad \{\, \phi \,\}}$$

ADD-BITS

$$\frac{}{\{\, \phi[x \mapsto \mathit{addBits}(x, e)] \,\} \quad x.\, \mathrm{addBits}(e) \quad \{\, \phi \,\}}$$

As a consequence of treating circuit methods as assignments, we can simply use the standard WP rules defined in Subsection 1.2.2 to construct a deterministic function that returns a single logic formula given annotated code, a precondition and a postcondition. If this formula is valid, the Hoare triple is valid. This is the algorithm at the core of our verification tool.

2.3 Optimization-Aware Hoare Logic

As discussed in Subsection 1.1.5, it is sometimes possible for a quantum circuit described by a program to include redundant quantum gates. In this section we will informally discuss how the Hoare logic defined previously can be adapted to reason about some aspects of the evolving quantum state during circuit construction. This addition allows the cancellation of gates based on TCs via modified Hoare rules, while simultaneously proving the correctness of a triple with respect to the optimized circuit. We also note how automated verification of this logic requires a trade-off in complexity or autonomy compared to the version that does not consider optimization.

2.3.1 Quantum State Assertion Logic

To evaluate TCs on the fly, the verification system must maintain a representation of the quantum state. Because verifying a TC only requires knowing which basis states possess non-zero probabilities, tracking the exact state vector is unnecessary.

Instead, we can abstract the quantum state using a simplified Boolean logic that focuses solely on the binary values of specific qubits. This logic is highly expressive as it can describe basic entanglement (for instance, asserting that two distinct qubits will always measure to the same value) in addition to asserting whether a specific qubit is guaranteed to be in a $|0\rangle$ or $|1\rangle$ state. Any qubit not explicitly described by an assertion is simply treated as being in an unknown state.

The logic can also be extended to support programs that construct and compose multiple quantum circuits by augmenting the state assertions to include circuit identifiers to indicate the targeted circuit.

2.3.2 Hoare Rules and Circuit Optimization

By appending these quantum state assertions to our existing logic, we can track the progression of the quantum state alongside classical program vari-

ables. This integration requires updating the Hoare rules for unitary operations to account for possible gate cancellations.
When the program dictates the application of a quantum gate, the updated logic branches into two potential evaluations:

- **Unitary with Cancellation**: First, the logic evaluates whether the current quantum state satisfies the gate's TCs. If these conditions hold, the gate is guaranteed to not alter the final outcome. The gate is therefore discarded, and the quantum state assertion remains unchanged.
- **Standard Unitary Application**: If the TCs are not satisfied, the gate must be applied to the circuit. The logic calculates the new quantum state by updating the assertions to reflect the transformation caused by the gate.

The rule for circuit initialization is updated by adding a quantum state assertion to the postcondition that setts all the qubits of the newly defined circuit to the $|0\rangle$ state. Composing a circuit y to another circuit x modifies the quantum state of the latter circuit by considering the full application of circuit y where the initial state is set as x's current state previous to the composition. Finally, measuring a qubit resets the measured qubit's state by removing it from all assertions, modelling the behaviour of most modern quantum computers that allow for qubit reuse.

2.3.3 Automated Verification

Because new Hoare rules have been introduced that aren't accounted for in the original definition of $\mathcal{WP}$ (Subsection 1.2.2), it is necessary to consider how a deterministic function that calculates the WP of these updated rules can be defined. The main issue is the difference between the natural directions of updating the quantum state and of the assignment rule, both of which are used in the rule for unitary operations. If we were to adapt this rule for WP calculus, it would be necessary to know the precise quantum state immediately prior to that gate's application. In a backward pass, if the

final postcondition is weak or if intermediate gates obscure the state (such as a Hadamard gate), the verification tool will fail to trigger the TCs. Consequently, a forward pass that calculates the *strongest postcondition* (SP), as opposed to the WP, is favoured for tracking the state of a program. While the SP function[3] can be defined to preserve the state context needed to identify removable gates, the SP rules for variable assignment introduce existential quantifiers into the VC that can pose decidability and performance issues for SMT solvers.

[3]The SP function for common commands is known and a detailed description can be found in [4].

Chapter 3

Implementation

In this chpater we describe how the theory introduced previously has been implemented as a tool for the automatic verification of quantum programs written in annotated Proto-Qiskit, in particular regarding structural properties of quantum circuits (gate count, width and depth). Finally, some simple examples are discussed to show the capabilities and the limitations of our approach.

3.1 Overview

The goal of the tool is to automate the verification of Hoare triples $\{P\}\, C\, \{Q\}$ written in Annotated Proto-Qiskit by using the developed WP calculus, together with an SMT solver.

The tool is organised as a small pipeline of independent stages, shown in Figure 3.1. A source file is first turned into an abstract syntax tree (AST) by the parser. The AST is then consumed by the VC generator, which walks the program backwards from the postcondition, applying the WP rule for each construct and, for every `while` loop, immediately checking the validity of the two associated side-conditions through the solver. Finally, the single VC produced for the whole program is translated into a symbolic formula and handed to the SMT solver, whose answer (*valid* or a concrete counterex-

ample) is reported back to the user.

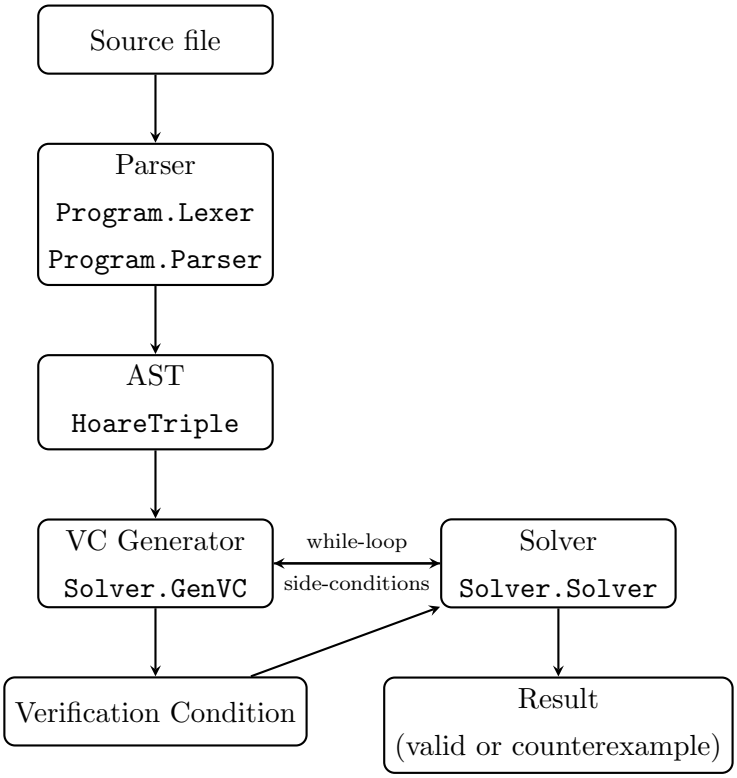


Figure 3.1: High-level architecture of the verification tool.

The tool is implemented in Haskell. This choice is motivated by the nature of the problem, since both the program AST and the WP rules are naturally expressed as algebraic data types and as structurally recursive functions over them. Haskell's purity makes the substitution-based semantics of the WP calculus (Section 3.3) easy to state correctly, since each rule can be written as a direct, side-effect free transcription of its mathematical counterpart, with all non-pure behaviour (calls to the SMT solver, error reporting) confined to explicit `IO`.

Three libraries are central to the implementation. *Megaparsec* is used to build the parser as a set of monadic parser combinators. It gives position-aware error messages and allows the parser monad to be built on top of an arbitrary inner monad via its `ParsecT` transformer, which we use for the integration of a

symbol table during parsing (Section 3.2). The *mtl* library supplies the state monad used for the symbol table. Finally, *SBV* (SMT-Based Verification) is used to translate logical expressions into symbolic values and to drive the *Z3* SMT solver. SBV was chosen over writing SMT-LIB directly because it embeds the solver's theories (linear and non-linear integer arithmetic, arrays, lists) as ordinary Haskell types and operators, and in particular because its `SArray` and `SList` types are what make it possible to model circuits of symbolic length and to index them with symbolic qubit indices (Section 3.4). `Main.hs` simply connects these three stages together. It reads a file path from the command line, parses it, generates the VC and prints the result of calling the solver on it.

3.2 Parsing

The abstract syntax of Annotated Proto-Qiskit is defined in `Program.Syntax`. A `Command` is a direct transcription of a command as defined in Subsection 2.1.1. A whole program is parsed into a `HoareTriple`, a record of a precondition, a command, and a postcondition, mirroring $\{P\}\ C\ \{Q\}$ directly.
Any expression appearing in the program is wrapped in the sum type

```
data Expr = BE BExpr | AE AExpr | QE QExpr
```

categorizing it as a boolean, arithmetic, or a quantum-circuit expression. These three expression languages are defined in `Expression.Syntax` and are parsed by `Expression.Parser`. This separation is what lets the same syntax (an identifier followed by `<-`) be used for assignments of any of the three types, while still letting the rest of the tool work with a single, type-correct `Command` AST. In `pAssign` (`Program.Parser`), the parser tries each of the three sub-parsers and tags whichever one succeeds, simultaneously recording the variable's type in the symbol table the first time it is seen and rejecting later assignments of a different type to the same name.
Currently, only single-qubit gates (`monGate`: `H`, `X`) and two-qubit gates (`binGate`: `CNOT`, `CR`, `SWAP`, `measure`) are accepted, however the gate set can be easily

expanded in future work. Both are wrapped in a single `Gate` type so that `pGate` can offer them as one choice and so that `UnitOp` does not need to be duplicated for arity 1 and arity 2 gates.
Since assertions are a superset of Boolean expressions, the same parser can be used for both syntaxes by adding the `width`, `gatecount` and `depth` functions to the expression parser. This has the downside of allowing these functions in program expressions, when in reality this isn't included in the language definition. However, as the code isn't actually compiled and executed, this doesn't make a difference for our use case.

Parsing is done with a custom monad:

```
type Env    = M.Map String Type
type Parser = ParsecT Void String (S.State Env)
```

where the inner `State` monad carries a symbol table associating variable names to their `Type`. By looking up if a variable's type has been recorded previously, `pAssign` is able to perform the static type check described above purely as a side effect of parsing. Operations inside Boolean and arithmetic formulas are parsed using Megaparsec's operator-table combinators, facilitating the definition of proper precedence and associativity rules. The top-level parser `pProgram` simply assembles a precondition in `[...]`, a block of commands and a postcondition in `[...]` into a `HoareTriple`, and `parseProgram` runs it together with an initially empty symbol table.

3.3 VC Generation

The core of the tool is the WP function in `Solver.GenVC` (`wp :: Command -> BExpr -> IO BExpr`) that implements the predicate-transformer semantics defined in Subsection 1.2.2. The `genVC` function generates the final VC:

```
genVC :: HoareTriple -> IO BExpr
genVC (HoareTriple p c q) = do
```

```
    wpResult <- wp c q
    return $ Imp p wpResult
```

Every operation on quantum circuits (`UnitOp`, `Measure`, `Compose`, `AddQ`, `AddB`) is handled by the same rule as a plain assignment.

```
wp (Assign ide e) q =
  return $ subBool q ide e
wp (UnitOp ide (MonGate g e)) q =
  return $ subBool q ide (QE $ QF1 g (QIde ide) e)
wp (UnitOp ide (BinGate g e1 e2)) q =
  return $ subBool q ide (QE $ QF2 g (QIde ide) e1 e2)
wp (Measure ide e1 e2) q =
  return $ subBool q ide (QE $ QF2 MES (QIde ide) e1 e2)
wp (Program.Syntax.Compose ide qc) q =
  return $ subBool q ide (QE qc)
wp (AddQ ide e) q =
  return $ subBool q ide (QE $ QF1 ADQ (QIde ide) e)
wp (AddB ide e) q =
  return $ subBool q ide (QE $ QF1 ADB (QIde ide) e)
```

The conditional follows the standard rule, with a missing `else` branch treated as a no-op (its branch's WP is simply Q), and sequencing folds the postcondition backwards through the command list one statement at a time, from the last command to the first.

```
wp (If cond c1 mc2) q = do
    wp1 <- wp c1 q
    case mc2 of
        (Just c2) -> do
            wp2 <- wp c2 q
            return $ And (Imp cond wp1) (Imp (Not cond) wp2)
        Nothing ->
            return $ And (Imp cond wp1) (Imp (Not cond) q)
```

```
wp (Seq (s:ss)) q = do
    lastWp <- wp (last (s:ss)) q
    wp (Seq (init (s:ss))) lastWp
wp (Seq []) q = return q
```

The `while` rule is the only command where substitution alone is not enough. Using an annotation in place of the true weakest precondition requires verifying the two side-conditions that check the validity of the provided invariant. The function's type (`Command -> BExpr -> IO BExpr`) is what allows these two checks to be performed immediately by calling out to the solver (`verifyVC`, itself `IO` since it invokes the external Z3 process) right inside the `while` case, rather than collecting them into a list of pending conditions to be checked together at the very end. This has two practical benefits: first, only the invariant itself needs to be returned as the WP of the loop to the rest of the traversal, so every other WP rule can keep the simple type `Command -> BExpr -> IO BExpr` without having to also return and merge additional conditions; second, if an invariant is wrong the failure is reported immediately, attached to the specific loop that caused it (via `error`).

```
wp (While cond inv c) q = do
    bodyWp <- wp c inv
    res1 <- verifyVC (Imp (And inv cond) bodyWp)
    checkResult res1
    res2 <- verifyVC (Imp (And inv (Not cond)) q)
    checkResult res2
    return inv
  where
    checkResult res =
        if isProved res
            then return ()
            else error $ "Verification failed! ..."
```

The actual substitution $Q[x \mapsto e]$ used by the assignment rule is implemented by three functions: `subBool`, `subArithm` and `subQCirc` (one per syntactic class of expression). Each traverses its argument structurally, replacing an identifier node with the substituted value if the identifier's syntactic class matches the tag (`BE`/`AE`/`QE`) of the value being substituted.

3.4 Symbolic Translation

Once a single `BExpr` VC has been produced, it has to be proven by an SMT solver. This translation step (defined in `Solver.Solver`) evaluates the `BExpr`/`AExpr`/`QExpr` syntax into SBV's symbolic values (`SBool`, `SInteger`, and a record of symbolic values for circuits) and lets SBV's `prove` hand the result to Z3.
A circuit is represented symbolically as

```
data SQC =
  SQC { sGates :: SList Gate, sQb :: SInteger, sCb :: SInteger }
```

a symbolic list of gates together with two symbolic integers for the qubit and classical bit register sizes. A gate is represented as a triple `(GateName, Integer, Integer)`. Every gate is given two integer operand slots, with the second set to `0` for single qubit gates. This is so that single qubit and two qubit operations can share one list element type and be prepended to `sGates` by the same code path. Building a fresh circuit (`QC`) starts with an empty gate list, composing two circuits (`Compose`) concatenates their gate lists and takes the maximum of their register sizes, and growing the registers (`ADQ`/`ADB`) only changes `sQb`/`sCb`, leaving the gate list untouched.
A store (`Env = M.Map String SymVar`) is integrated into evaluation with a `StateT Env Symbolic` monad and is used in an equivalent way to the symbol table in the parser. The evaluator (implemented by `evalBExpr`, `evalAExpr` and `evalQExpr`) is essentially a recursive traversal of the formula, and on its own it has no way of knowing that two occurrences of the same variable

name should be tied to the same SBV symbol. The first time a name is encountered a fresh symbolic value is created and recorded in the store with its type, whereas every later occurrence of the same name retrieves that same value instead of creating a new, unconstrained one.
Circuit depth is computed by `calcDepth` by folding over the symbolic gate list while maintaining a symbolic array (`SArray Integer Integer`), mapping each qubit index to its current depth. For each gate, the existing depths of the qubit(s) it is applied to are read out of the array: if the gate is two-qubit, the new depth is one more than the maximum of the two operands' current depths, and is written back to both qubits, whereas a single qubit gate only updates its one operand. SBV's `SArray` is used precisely because it supports indexing by a symbolic key, which Z3 reasons about via its array theory.
The `gatecount` translation also relies on the symbolic array by returning its length (`L.length`), whereas `width` is simply calculated as the sum of the number of qubits and classical bits.

3.5 Evaluation

We illustrate the tool on two example programs, both parametric in the number of qubits `n`.

Conditional in a Single Loop

```
[n > 0]
{
  qc <- QuantumCircuit(n, 0);
  i <- 0;
  while i != n: [gatecount(qc) == i/2 && 0 <= i && i <= n]
  {
    if i % 2 == 1:
    {
      qc.CNOT(i, i + 1);
```

```
    }
    i <- i + 1;
  }
}
[gatecount(qc) == n/2]
```

This program builds a circuit with `n` qubits and iterates a counter `i` from `0` to `n`, applying a `CNOT` only on the odd values of `i`, so that after `i` completed steps exactly $\lfloor i/2 \rfloor$ (only integer division is allowed) gates have been applied. The postcondition asks for $\lfloor n/2 \rfloor$ gates in total. The invariant is preserved in both cases: on an even `i`, $\lfloor i/2 \rfloor$ is unchanged by incrementing to $i+1$; on an odd `i`, incrementing both `gatecount(qc)` and `i` keeps `gatecount` $= \lfloor i/2 \rfloor$ true as well, since $\lfloor i/2 \rfloor + 1 = \lfloor (i+1)/2 \rfloor$ when i is odd. The loop's exit condition follows by substituting `i = n` into the invariant, which is syntactically identical to the postcondition. The only non-trivial arithmetic theory needed is integer division by a literal constant, which, unlike division or multiplication by a variable, stays inside Z3's decidable linear integer arithmetic fragment. Therefore all three VCs are expected to be discharged immediately by the solver.

QFT

```
[n > 0]
{
qc <- QuantumCircuit(n, 0);
i <- 1;
while i != n + 1: [gatecount(qc) == (i-1)*(2*n-i+2)/2 && 1 <= i
                                                    && i <= n + 1]
{
  qc.H(i);
  j <- i + 1;
  while j != n + 1: [gatecount(qc) == (i-1)*(2*n-i+2)/2+j-i &&
```

```
                    1 <= i && i <= n && i + 1 <= j && j <= n + 1]
  {
    qc.CR(i, j);
    j <- j + 1;
  }
  i <- i + 1;
}
}
[gatecount(qc) <= n * n ]
```

This example implements the quantum Fourier transform, consisting of an outer loop over `i` that applies a Hadamard to qubit `i` and then an inner loop over `j` that applies a controlled rotation[1] between `i` and every qubit `j`. A completed outer iteration k contributes $1 + (n - k)$ gates, so after $i - 1$ completed outer iterations the total gate count is the sum $\sum_{k=1}^{i-1}\big(n-k+1\big) = \frac{(i-1)(2n-i+2)}{2}$, which is exactly the closed form used in the outer invariant. The inner invariant simply adds the partial progress `j - i` made through the current outer iteration's inner loop. The postcondition relaxes the exact final count to an upper bound, `gatecount(qc) <= n * n`, which holds. Unlike the first example, the closed-form invariants here are quadratic in `n`, so the four VCs generated for this triple require Z3's non-linear integer arithmetic reasoning. Therefore, the success of the verification is thanks to the solver's heuristics rather than of a decision procedure, and is not guaranteed to scale to more complex cases.

[1] In theory the rotation should be parametrized, however the language doesn't currently support gates with more than two operands. This can easily be achieved in future work.

Conclusion

In this thesis we defined a simple quantum programming language and a Hoare logic that also allows for the verification of structural properties of the circuits that can be built with it. A tool automating such verifications by using weakest precondition calculus and SMT solvers was also introduced. In addition, we discussed how an optimization-aware Hoare logic could be implemented by adding quantum state assertions to the previous logic and by modifying the Hoare rules accordingly. We also explored how such verifications could be automated and identified some of the resulting issues.
The constructed tool is simple and has several limitations. Currently, it only supports a limited gate set and it lacks support for real numbers, which would be needed to represent parametrized rotations. The underlying SMT solver also struggles with non-linear arithmetic where success depended on heuristics rather than a guaranteed decision procedure.
For future work, the tool could be expanded by implementing the optimization-aware Hoare logic that was informally introduced. Support for additional gates could also be added, along with a possible compilation step from non-parametrized circuits to QASM. Finally, the need for user intervention could be reduced by employing techniques based on artificial intelligence for the automatic generation of loop invariants [8].

Bibliography

[1] M. AbuGhanem. Ibm quantum computers: evolution, performance, and future directions. *The Journal of Supercomputing*, 81(5), Apr. 2025. ISSN 1573-0484. doi: 10.1007/s11227-025-07047-7. URL http://dx.doi.org/10.1007/s11227-025-07047-7.

[2] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[3] K. L. Brown, W. J. Munro, and V. M. Kendon. Using quantum computers for quantum simulation. *Entropy*, 12(11):2268–2307, Nov. 2010. ISSN 1099-4300. doi: 10.3390/e12112268. URL http://dx.doi.org/10.3390/e12112268.

[4] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, Heidelberg, 1990. ISBN 978-1-4612-3228-5. doi: 10.1007/978-1-4612-3228-5.

[5] Google Quantum AI and Collaborators. Quantum error correction below the surface code threshold. *Nature*, 638:920–926, 2025. doi: 10.1038/s41586-024-08449-y.

[6] L. K. Grover. A fast quantum mechanical algorithm for database search, 1996. URL https://arxiv.org/abs/quant-ph/9605043.

[7] S. P. Jordan, N. Shutty, M. Wootters, A. Zalcman, A. Schmidhuber, R. King, S. V. Isakov, T. Khattar, and R. Babbush. Optimization by decoded quantum interferometry. *Nature*, 646(8086):831–836, Oct.

2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09527-5. URL http://dx.doi.org/10.1038/s41586-025-09527-5.

[8] R. Liu, M. Chen, L.-I. Wu, J. Ke, and G. Li. Enhancing automated loop invariant generation for complex programs with large language models, 2025. URL https://arxiv.org/abs/2412.10483.

[9] J. Preskill. Quantum computing 40 years later. In *Feynman lectures on computation*, pages 193–244. CRC Press, 2023.

[10] P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.

[11] L. Venturi. *Quantum Circuit Size Estimation via Hoare Logic*. PhD thesis. URL https://amslaurea.unibo.it/id/eprint/34180/.

[12] J. D. Weidman, M. Sajjan, C. Mikolas, Z. J. Stewart, J. Pollanen, S. Kais, and A. K. Wilson. Quantum computing and chemistry. *Cell Reports Physical Science*, 5(9):102105, 2024. ISSN 2666-3864. doi: https://doi.org/10.1016/j.xcrp.2024.102105. URL https://www.sciencedirect.com/science/article/pii/S2666386424003837.