Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Design of a Quantum Low-Code Framework

Sharon-Naemi Stiliadou

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Daniel Georg, M.Sc. Dipl.-Inf. Alexander Mandl
Commenced:	December 20, 2024
Completed:	June 20, 2025

Abstract

Quantum computing is a promising paradigm that offers reduced energy consumption, enhanced computational precision, and exponential speedups for certain problems compared to classical counterparts. However, manually building quantum applications that combine classical and quantum parts requires not only specialized knowledge in quantum computing but also significant programming expertise, creating a substantial barrier for many users. Classical low-code platforms can abstract away much of the coding effort, but they are typically domain-specific and not suitable for designing quantum algorithms. Moreover, simply reducing the amount of code is often insufficient, as developers still require a deep understanding of the underlying quantum concepts to model and implement quantum applications correctly. Therefore, it is equally important to provide abstractions that shield users from complex domain knowledge, allowing them to focus on the logical structure of their applications, regardless of their level of expertise in quantum computing. To address this gap, this work develops an open-source low-code modeling tool for designing quantum applications. The objective is to identify and define essential modeling constructs that provide higher-level abstractions, as well as lower-level modeling blocks that represent quantum gates, in order to support the development of quantum applications while ensuring accessibility for users with varying levels of expertise. The tool enables the modeling of both classical and quantum parts, allowing developers to build quantum applications. Additionally, the tool employs a standardized format for saving and loading models, ensuring portability and interoperability across different systems and platforms. The proposed concept will be implemented in a prototype and evaluated through use cases. In parallel, a low-code backend project will prototype the transformation of models into OpenQASM code based on the model structure and its standardized format.

Kurzfassung

Quantencomputing ist ein vielversprechendes Paradigma, das im Vergleich zu klassischen Ansätzen einen geringeren Energieverbrauch, eine höhere Rechenpräzision und exponentielle Beschleunigungen bei bestimmten Problemen bietet. Das manuelle Erstellen von Quantenanwendungen, die klassische und Quantenanteile kombinieren, erfordert jedoch nicht nur spezielles Wissen im Bereich des Quantencomputings, sondern auch umfangreiche Programmierkenntnisse, was für viele Nutzer eine erhebliche Herausforderung darstellt. Klassische Low-Code Plattformen können einen Großteil des Programmieraufwands abstrahieren, sind jedoch typischerweise domänenspezifisch und daher für die Gestaltung von Quantenalgorithmen nicht geeignet. Zudem reicht eine Reduzierung der Code Menge oft nicht aus, da Entwickler weiterhin ein tiefgehendes Verständnis der zugrunde liegenden Quantenkonzepte benötigen, um Quantenalgorithmen korrekt modellieren und implementieren zu können. Daher ist es ebenso wichtig, Abstraktionen bereitzustellen, die Nutzer von komplexem Domänenwissen entlasten und es ihnen ermöglichen, sich auf die logische Struktur ihrer Algorithmen zu konzentrieren, unabhängig von ihrem Fachwissen im Quantencomputing. Um diese Lücke zu schließen, entwickelt diese Arbeit ein Open Source Low-Code Modellierungstool zur Gestaltung von Quantenanwendungen. Ziel ist es, wesentliche Modellierungskonstrukte zu identifizieren und zu definieren, die sowohl höherstufige Abstraktionen als auch niederstufige Modellblöcke für Quanten Gatter bereitstellen, um die Entwicklung von Quantenalgorithmen zu unterstützen und gleichzeitig die Zugänglichkeit für Nutzer mit unterschiedlichem Fachwissen zu gewährleisten. Das Tool ermöglicht die Modellierung sowohl klassischer als auch quanten Teile, sodass Entwickler Quantenalgorithmen erstellen können. Darüber hinaus verwendet das Tool ein standardisiertes Format zum Speichern und Laden von Modellen, um Portabilität und Interoperabilität zwischen verschiedenen Systemen und Plattformen sicherzustellen. Das vorgeschlagene Konzept wird in einem Prototyp implementiert und anhand Anwendungsfälle evaluiert. Parallel dazu wird ein Low-Code Backend Projekt prototypisch die Transformation der Modelle in OpenQASM Code basierend auf der Modellstruktur und dem standardisierten Format umsetzen.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

CNOT Controlled-NOT. 22

GUI Graphical User Interface. 29

M2M Model-to-Model. 17

MDE Model-driven Engineering. 17

NISQ Noisy Intermediate-Scale Quantum. 23

OpenQASM Open Quantum Assembly Language. 16

QAOA Quantum Approximate Optimization Algorithm. 54

QFT Quantum Fourier Transform. 20

QIR Quantum Intermediate Representation. 40

QML Quantum Machine Learning. 27

QNN Quantum Neural Network. 28, 29

QSVC Quantum Support Vector Classifier. 28

QSVM Quantum Support Vector Machine. 28

QSVR Quantum Support Vector Regression. 28, 29

UML Unified Modeling Language. 65

UUID Universally Unique Identifier. 48

VQR Variational Quantum Regression. 28, 29

1 Introduction

Quantum computing has emerged in the past few years as a new paradigm providing enhanced energy efficiency and improved accuracy [Bar22; NC12]. It also offers the potential to solve certain problems faster than classical computers [HTC18]. As a result, quantum computing presents opportunities across various domains, including cryptography, chemistry, and finance [CRA18]. Currently, quantum computers are limited in the number of qubits and are error-prone [Pre18]. Furthermore, classical parts are used inside quantum algorithms, making them hybrid [WBLZ21]. However, the development of these algorithms is complex and time-consuming, requiring specialized knowledge in software engineering, quantum algorithms, and physics [GBB+23].

To address this knowledge gap, solutions are needed that abstract the technical complexities of classical programming, as well as the specialized expertise required for quantum computing. Therefore, low-code platforms such as Scratch [RMM+09] and Google Blockly [Fra14] have been introduced to enable users with limited technical expertise to contribute to the creation of classical programs. These platforms significantly lower the barrier to entry and simplify the development process by abstracting underlying complexities through visual modeling. However, while many existing low-code platforms are tailored to specific domains or use cases, they generally lack the capabilities required for quantum application development [Ebe24].

In this context, quantum low-code tools have emerged, operating at various levels of abstraction, each providing a distinct approach to managing system complexity. Tools such as IBM Quantum Composer [IBM24b] and Quantum Programming Studio [Qua19] support the visual modeling of quantum circuits. A quantum circuit is a sequence of operations applied to qubits, which are the basic units of quantum information. While these tools eliminate the need to write quantum code directly, they do not reduce the complexity involved in designing quantum algorithms. To address this limitation, higher-level abstraction frameworks such as Classiq [Min22] have been developed. Classiq allows users to construct quantum algorithms using functional models and drag-and-drop interfaces, automatically generating executable quantum circuits. These circuits can be executed on various platforms, including IBM [IBM24a] and Amazon Braket [AWS24]. However, since Classiq relies on proprietary technology, it can result in vendor lock-in and reduced flexibility. Additionally, the tool cannot express classical logic. This highlights the need for an open-source tool that supports the design of quantum algo, combining simplicity with the cognitive effectiveness required by established design guidelines [Moo09].

This master's thesis focuses on the design and development of the frontend of an open-source low-code tool for modeling quantum applications. To this end, fundamental components and their characteristics will be identified, covering both quantum and classical parts, in order to support the creation of quantum algorithms. In particular, quantum modeling constructs for higher-level abstractions and lower-level modeling blocks representing quantum gates, while supporting users across a broad spectrum of quantum computing expertise. Furthermore, this work aims to establish a method for integrating existing or customized implementation artifacts into the model.

This streamlines their serialization into a standardized format, thereby ensuring the portability and interoperability of the model across various systems and platforms. Finally, a prototypical implementation of the concept will be created. Along with this work, a separate student project will develop a low-code backend to prototype the transformation of models into executable Open Quantum Assembly Language (OpenQASM) code based on the frontend model.

The remainder of the thesis is organized as follows. Chapter 2 covers the fundamentals of low-code development, quantum computation, and quantum computing patterns. Chapter 3 compares quantum low-code tools based on criteria such as general functionality, hardware support, and usability. Chapter 4 summarizes the current situation, presents the problem in detail, and formulates the requirements for the modeling constructs and blocks, as well as for the modeling tool developed in this thesis. Next, Chapter 5 contains the concept for the low-code modeling tool, its architecture, and the different modeling constructs based on the tool comparison and requirements. Chapter 6 presents a prototypical implementation of the concept. Chapter 7 assesses the feasibility of the concept based on the implementation. Chapter 8 provides an overview of related work and places it in the context of this thesis. Finally, Chapter 9 concludes the thesis and provides an outlook.

2 Fundamentals

This chapter establishes the foundation for the rest of the thesis. First, low-code development and its context within Model-driven Engineering (MDE) are defined. Subsequently, design principles for modeling tools and components are introduced. In the following, we address the basic mathematical concepts of quantum computing. Finally, we explore how patterns can be applied to support the development of quantum algorithms.

2.1 Model-Driven Engineering and Low-Code Development

In MDE models serve as a foundational representation of the system, providing a structured description of its architecture, behavior, and interactions [Sch06]. By leveraging these models, different views or perspectives of a system can be represented, facilitating a separation of concerns. This separation improves manageability by reducing cost, shortening development time, and enhancing overall quality [KR03]. This separation of concerns is often realized by using modeling languages such as UML or formal specification languages such as Alloy and B. A formal definition of models enables Model-to-Model (M2M) transformations, where one model serves as input to produce another. M2M transformations are a core part of MDE development processes, which typically span the requirements, analysis, design, and implementation phases. These transformations progressively refine the system across different abstraction levels, ensuring traceability and maintainability throughout the development lifecycle [BFV12]. The transition from design models to executable software is handled by model-to-code transformations, which automatically generate implementation code from models. This automation accelerates development and reduces errors.

Unlike traditional MDE, which is often targeted at software engineers and relies on formal modeling languages, low-code platforms emphasize accessibility by minimizing the need for handwritten code, allowing domain experts and less technical users to contribute directly to software creation [Fra14; RMM+09]. By abstracting system complexity through visual modeling, low-code platforms lower the barrier to entry and accelerate the software development process. The automated generation of code within these platforms also helps reduce development errors, costs, and release times.

2.2 Design Principles of Modeling Interfaces and Blocks

Modeling tools vary significantly not only in terms of their application domains but also in the design of their user interfaces. This variability affects how standard functionalities are both presented and interacted with, directly impacting usability and user efficiency. Similar principles of interface design and interaction are applied to modeling components in low-code platforms. However, there is no uniform standard. Different platforms use various visual and interactive representations of components, resulting in diverse user experiences and learning curves.

Creating a modeling tool requires standardizing rules and requirements for users, primarily focusing on usability. Usability is a quality attribute that evaluates how easy user interfaces are to use [Gro12]. It encompasses methods for improving ease of use during the design process and comprises five quality components: *learnability*, *efficiency*, *memorability*, *errors*, and *satisfaction*. Learnability refers to how easily users can accomplish basic tasks when they first encounter the design. Efficiency measures how quickly users can perform tasks after learning the design. Memorability assesses how easily users can regain proficiency after not using the design for a period of time. Errors evaluate the number and severity of errors users make, as well as how easily they can recover from them. Satisfaction measures how pleasant the design is to use.

To satisfy these quality attributes, principles for UI design should be used. Thereby, factors such as design and interactions significantly influence the quality components of a user interface. For the design of an intuitive UI, elements such as spacing, sizing, and grouping of visual components are crucial. Various guidelines, including the GNOME Guidelines [GNO25] and Apple Human Interface Guidelines [App25], have proven effective in creating user-friendly interfaces.

The layout for a modeling tool typically features four main areas: the toolbar, the left and right sidebars, and the canvas [JXWL22]. While the toolbar is consistently placed at the top, the specific elements and their arrangement can differ, with some interfaces prioritizing certain tools or functions based on user needs. The left and right sidebars often contain additional tools, panels, or navigation elements, with their content and organization varying depending on the software's specific functionalities and user requirements. The central canvas area, where the primary interaction and content creation occur, can also differ significantly in layout and features, influencing how users interact with the interface.

Based on UI design principles, e.g. Shneidermans golden rules, maintaining consistency across all buttons and interactive elements is essential [Shn87]. This consistency includes uniform styling, predictable behavior, and clear labeling, which collectively enhance the user experience by making the interface more intuitive and easier to navigate. Thereby, principles for interaction with the tool are a critical aspect of user interface design, encompassing various functionalities that enhance user engagement and efficiency. Therefore, enabling shortcuts for experienced users, providing informative feedback, designing dialogues to yield closure, error prevention and handling, permitting easy reversal of actions, supporting internal locus of control, and reducing short-term memory load [Shn87]. Basic interaction features such as zooming, filtering, and feedback play a crucial role in supporting users during model development [Shn96]. Zooming allows users to focus on specific details or gain a broader overview of the model. Filtering facilitates the handling of complex datasets by displaying only relevant information. Additionally, error correction mechanisms such as undo/redo functions and clear, informative error messages enable users to efficiently identify and resolve mistakes, thereby enhancing the overall usability of the system.

While these principles establish essential guidelines for designing the overall modeling tool, they must also be carefully applied to the individual modeling constructs used within the explorer to ensure a consistent, intuitive, and effective user experience. According to the principle of good gestalt, visual representations should be based on simple, familiar, and distinguishable shapes [Gub15]. Shapes such as circles, squares, triangles, and rectangles are more easily perceived and retained. Constructs with similar semantic or functional properties are assigned shapes from the same family, such as squares and kites, to visually demonstrate their relationship [Moo09]. Conversely, distinct families of shapes, such as circles and squares, are used to differentiate constructs with different roles, supporting rapid and unambiguous recognition. Furthermore, incorporating a variety of visual

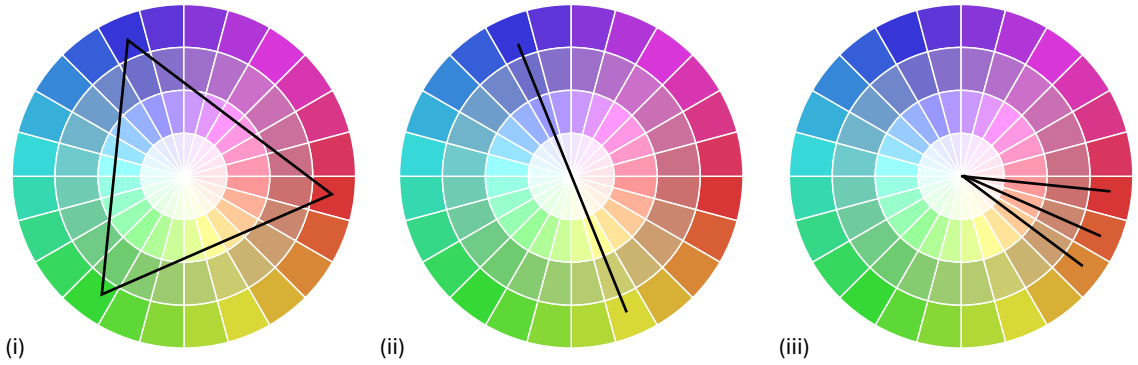

Figure 2.1: (i) Triadic, (ii) complementary, (iii) analogous color schemes [McK18]

variables, such as shapes and colors, can enhance visualization. While shape provides structural cues, color adds an additional semantic layer that can encode state, type, or grouping. The selection of colors should adhere to principles of color theory, employing schemes that promote clarity, contrast, and harmony [McK18]. Triadic, complementary, and analogous color palettes shown in Figure 2.1 can be used to highlight similarities, distinguish differences, and convey relationships within the model. Moreover, saturation and brightness can indicate the importance or optionality of constructs, such as whether a construct is mandatory or not. Cultural considerations are also essential in color selection. Since color associations can vary significantly across cultural contexts. Beyond the visualization of individual modeling constructs, it is essential to consider how these constructs are composed within the broader modeling tool. Thereby, a modeling tool consists of three common composition types: block-based, flow-based, and tree-based structures [GP96; Moo09]. The block-based structure is widely used for process modeling and visual programming, where constructs are represented as discrete, modular blocks that can be arranged in a grid-like manner. The flow-based structure emphasizes the direction of execution or data flow between components, often visualized as connected nodes and edges. This approach is particularly effective for representing sequential and parallel processes. A tree structure is a method of organizing data that utilizes parent-child relationships to present information in a hierarchical manner.

2.3 Quantum Computation

Quantum computation promises computational advantages over classical methods for certain problems. While various models of quantum computation exist, this work focuses on the *gate-based* model, which is the most widely used [NWA21; PKG24]. In the following, the fundamental concepts and operations of the gate-based quantum computation model are introduced.

2.3.1 Qubit

The basic unit of information in classical computation is a bit, which can be in one of two possible states: either 0 or 1. Unlike classical bits, which are limited to discrete values, a qubit can exist in a *superposition* of basis states [NC12]. A general qubit state is a unit vector in a two-dimensional Hilbert space $\mathcal{H}$, and can be expressed as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix} \tag{2.1}$$

where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$. A more general parameterization of the qubit state, up to a global phase $e^{i\gamma}$, is given by:

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle\,, \quad \theta \in [0,\pi],\ \phi \in [0,2\pi], \tag{2.2}$$

which allows a geometric representation on the *Bloch sphere*, depicted in Figure 2.2, where θ and ϕ define the angles. A system of multiple qubits can be described by a *quantum register*, whose state is in the tensor product of individual qubit spaces. The state of a quantum register is called *separable* if it can be written as a product of single-qubit states. If it cannot, the state is considered *entangled*. Several multi-qubit states play a central role in quantum information theory due to their unique entanglement properties. Among the most fundamental are the *Bell states*, which represent the maximally entangled states of two qubits. These are given by:

$$|\phi^{\pm}\rangle = \frac{1}{\sqrt{2}}(|00\rangle \pm |11\rangle), \quad |\psi^{\pm}\rangle = \frac{1}{\sqrt{2}}(|01\rangle \pm |10\rangle) \tag{2.3}$$

These states form an orthonormal basis for the two-qubit Hilbert space and are widely used in quantum teleportation [NC12].

For systems with more than two qubits, the *GHZ state* is a prominent example of a maximally entangled state. In the three-qubit case, it is defined as

$$|GHZ_3\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle) \tag{2.4}$$

Another important three-qubit state is the *W state*, given by

$$|W\rangle = \frac{1}{\sqrt{3}}(|001\rangle + |010\rangle + |100\rangle) \tag{2.5}$$

which, unlike the GHZ state, remains partially entangled even when one qubit is lost.

Finally, the *uniform superposition state* represents an equal superposition over all computational basis states [Ley19]. For an n-qubit system, it is written as

$$|+\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle \tag{2.6}$$

and is prepared by applying Hadamard gates to each qubit initialized in the $|0\rangle$ state. This state serves as the initial configuration in many quantum algorithms, such as Grover [Gro96] and the Quantum Fourier Transform (QFT) [Sho94].

Name	Symbol	Matrix
Identity	I	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
Hadamard	H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Pauli-X	X	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Y	Y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli-Z	Z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
$\pi/8$	T	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$

Table 2.1: Single-qubit Gates

2.3.2 Operations

In quantum computing, computations are performed on qubits using unitary transformations because they are inherently invertible, and operations on a gate-based quantum computer must be reversible [NC12]. Specifically, quantum operations can be categorized into two main types: *single-qubit* gates, which act on individual qubits, and *multi-qubit* gates, which operate on two or more qubits to enable entanglement.

Single-Qubit Gates

Single-qubit gates are represented by 2×2 unitary matrices. Table 2.11 lists several commonly used single-qubit gates, along with their names, circuit symbols, and corresponding matrix representations. The *Identity* gate leaves the quantum state unchanged. The *Hadamard* gate creates superposition by mapping the basis states $|0\rangle$ and $|1\rangle$ to a uniform superposition. The *Pauli-X* gate acts as a quantum NOT gate, flipping the states $|0\rangle$ and $|1\rangle$. The *Pauli-Y* gate performs a bit and phase flip simultaneously. The *Pauli-Z* gate applies a phase flip by changing the sign of the $|1\rangle$ component. Finally, the $\pi/8$ gate applies a $\pi/4$ phase shift to the $|1\rangle$.

Any unitary operation acting on a single qubit can be decomposed into a sequence of rotations generated by the Pauli matrices. Consequently, parameterized rotations around the x-, y-, and z-axes of the Bloch sphere serve as fundamental building blocks. These blocks are used for manipulating quantum states within quantum circuits:

$$R_x(\theta) = e^{-i\theta X/2}, R_y(\theta) = e^{-i\theta Y/2}, R_z(\theta) = e^{-i\theta Z/2} \tag{2.7}$$

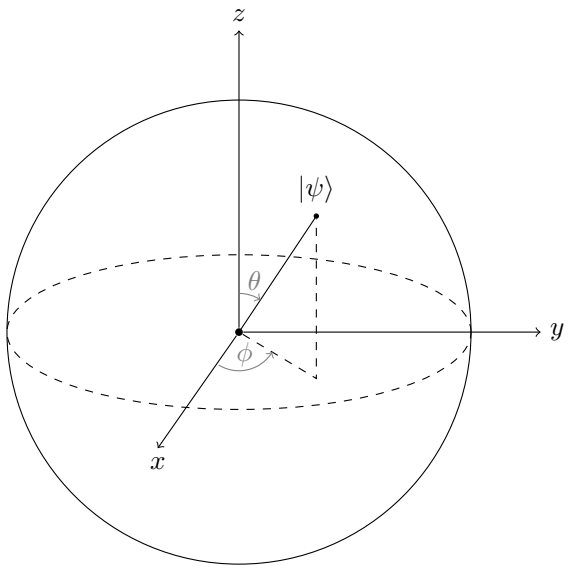

Figure 2.2: Geometric visualization of a qubit on the Bloch sphere

Multi-Qubit Gates

Multi-qubit gates are required to enable interaction between qubits to generate entanglement.

The Controlled-NOT (CNOT) gate operates on two qubits: a control c and a target t. It flips the state of the target qubit if and only if the control qubit is in the state $|1\rangle$. Its action can be described as:

$$\text{CNOT: } |c,t\rangle \rightarrow |c, t \oplus c\rangle\,, \quad U_{\text{CNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.8}$$

The Toffoli gate is universal for classical reversible computation and a three-qubit gate that flips the target qubit if both control qubits are $|1\rangle$:

$$\text{Toffoli: } |c_1, c_2, t\rangle \rightarrow |c_1, c_2, t \oplus (c_1 \cdot c_2)\rangle \tag{2.9}$$

An arbitrary unitary operator acting on n qubits can be decomposed into a sequence of $O(n^2 \cdot 4^n)$ single-qubit and CNOT gates [BBC+95]. Due to the exponential scaling with respect to the number of qubits, such a direct decomposition is computationally inefficient and impractical for large systems. Therefore, the Solovay-Kitaev theorem [KSV02] states that such a decomposition can be approximated by using only {H, T, CNOT} with polylogarithmic operations.

2.3.3 Quantum Circuits

Quantum circuits are the foundational model for quantum computation [PKG24]. A quantum circuit is composed of three elements: qubits (wires), quantum gates, and measurements. Each wire represents a qubit. Each box is either an operation or a measurement that is applied to a set of

qubits. Two important characteristics of quantum circuits are *width* and *depth*. The width refers to the number of qubits used in the circuit, e.g., how many wires it spans. The depth is the number of sequential layers of operations, that is, the longest chain of gates that must be applied one after another, without parallelization. The circuit from Figure 2.3 has a width of 3 and a depth of 3. A quantum circuit can be expressed in multiple quantum SDKs, each offering its own syntax and abstractions. Despite these differences, the underlying logic and structure of the circuit remain the same. However, since SDKs are tied to different execution tools, interoperability is limited. A circuit written in Braket cannot be directly executed on IBM, and vice versa, without conversion, a process that can lead to compatibility issues. This highlights the importance of OpenQASM as a standardized, hardware-agnostic format for describing quantum circuits [CJA+22].

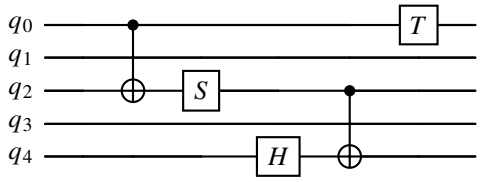

Figure 2.3: Quantum circuit with 5 qubits including S, T, H, and CNOT gates

2.3.4 Measurement

Measurement irreversibly alters the state of a qubit by destroying its superposition and yielding a definite classical outcome. A qubit in superposition collapses to one of the computational basis states upon measurement. Specifically, the outcome $|0\rangle$ is observed with probability $|\alpha|^2$, while $|1\rangle$ is observed with probability $|\beta|^2$. Typically, qubits are measured in the *computational basis*, also known as the Z basis, where the basis states are $|0\rangle$ and $|1\rangle$. However, certain quantum algorithms or subroutines require measurements in alternative bases, such as the X or Y basis, particularly when the *Z-basis* is not *sharp* for the given state, that is, when the state is not an eigenstate of the Z operator. Measurements in alternative bases are performed by first applying a unitary rotation to map the desired basis to the computational basis, followed by a standard Z-basis measurement.

2.3.5 Quantum Algorithm

Today's quantum computers are so-called Noisy Intermediate-Scale Quantum (NISQ) devices, which are limited in qubit count and prone to errors [Pre18]. While these constraints have led to the development of hybrid quantum-classical algorithms, it is important to note that not all algorithms are hybrid solely because of hardware limitations [LB20]. In a broad sense, *hybrid* refers to any algorithm that involves both classical and quantum computational steps. A prominent example is Shor's algorithm [Sho94], which includes classical pre- and post-processing stages but is not suitable for NISQ devices due to its resource requirements. However, when referring to hybrid algorithms in the context of NISQ, what is usually meant are variational quantum algorithms. These involve a feedback loop where a classical optimizer interacts with a parameterized quantum circuit, aiming to minimize or maximize a function [CAB+21]. The goal of such algorithms is to leverage quantum processing while keeping circuit depth and width manageable. The structure of a quantum algorithm is illustrated in Figure 2.4. First, *classical pre-processing* is often employed, with techniques such as

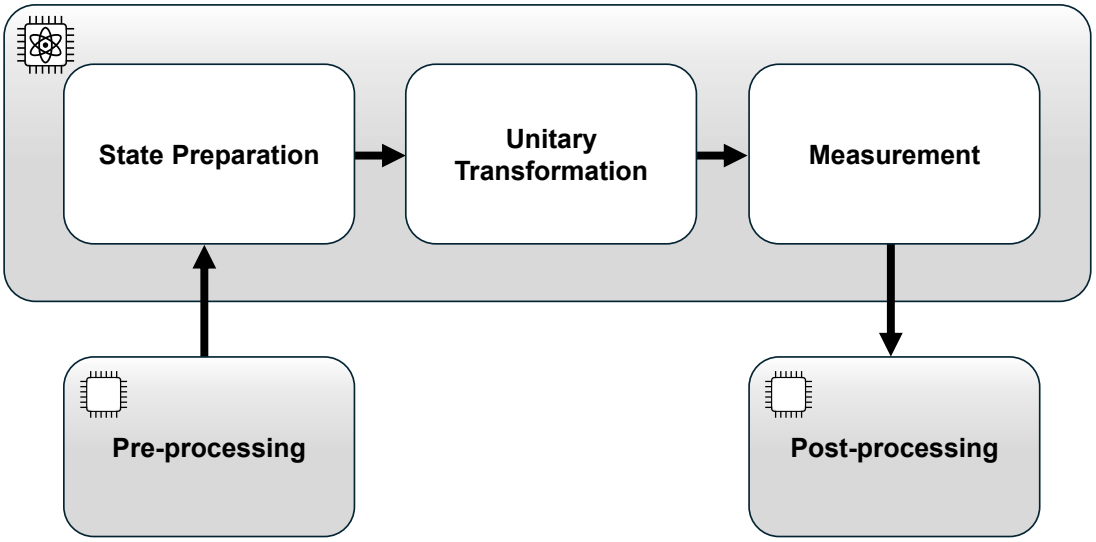

Figure 2.4: Structure of a Quantum Algorithm [LB20]

principal component analysis used for dimensionality reduction and feature selection. The first step also involves decisions regarding the selection of data encoding into quantum states and the algorithm parameters. Subsequently, *state preparation* takes place, where classical data is mapped into a quantum representation. Common encoding methods are *Basis Encoding, Amplitude Encoding, Angle Encoding, Schmidt Decomposition, and Matrix Encoding* [WBLS21]. Basis Encoding maps classical bitstrings directly to computational basis states. Amplitude Encoding uses the amplitudes of a quantum state to encode data. Angle Encoding represents data by encoding classical values into rotation angles of quantum gates. The Schmidt Decomposition is used to express a quantum state as a sum of tensor products of orthonormal basis vectors from two subsystems. Matrix Encoding embeds non-unitary data into a quantum state by constructing a unitary of the data. After state preparation, *unitary transformations* are applied. These transformations are chosen with the goal of guiding the system toward a target quantum state from which *measurements* yield useful information for classical post-processing. Finally, the measurement outcomes are *post-processed classically* to obtain the final result.

It is important to note that, unlike classical systems, arbitrary quantum states cannot be copied due to the *no-cloning theorem* [NC12]. This fundamental limitation of quantum mechanics states that no quantum algorithm is capable of perfectly duplicating an unknown quantum state.

2.3.6 Quantum Computing Patterns

Patterns present proven solutions to recurring problems in a structured and comprehensible manner [AIS77]. They contain abstract solutions that users can adapt to fit their specific needs. As quantum computing demands a deep understanding of theoretical and practical aspects, developing quantum applications remains a complex task. To manage this complexity and promote reuse, a quantum computing pattern language was introduced [Ley19]. Within this pattern language, four development patterns are defined to support reusability and flexibility, enabling the creation and

implementation of quantum algorithms [BBB+23]. The *Quantum Module* pattern encapsulates the quantum part of an algorithm into a reusable component to create the quantum circuit once the input parameters are received. The *Quantum Module Template* pattern provides a template that allows developers to insert custom behavior into generated circuits. The *Hybrid Module* pattern packages quantum and classical parts of an algorithm to enable its integration into applications. The *Classical-Quantum Interface* pattern defines how quantum algorithms are made accessible via an interface, hiding their internal implementation from the user.

3 Tool Comparison

Low-code platforms are designed to simplify software development by enabling users to build programs through graphical interfaces and minimal hand-written code. Classical low-code tools such as Scratch [RMM+09] and Google Blockly [Fra14] have proven effective in different contexts, particularly in education [TKH+24]. Both tools use a drag-and-drop interface with building blocks, making programming more accessible, especially for beginners, since users create code without text-based coding. Scratch enables users to create interactive programs, including games and animations. In contrast, Blockly is a block-based visual language that can be integrated into different applications and translates visual blocks into programming languages such as JavaScript and Python. With the growing importance of machine learning, several domain-specific tools such as Trinity [IHW+21] and Pyrus [ZS21] have been developed to support users in developing their own Machine Learning models and applications [LW22]. For instance, Trinity enables users to construct neural networks using pre-built components. However, these tools are not well-suited for quantum computing due to the fundamentally different computational paradigm it follows [Ebe24]. As a result, low-code platforms for quantum computing have emerged.

Therefore, we compare the existing quantum low-code platforms Classiq Modeler and QuantuMoonLight with the goal of identifying their differences, similarities, and respective features. In Table 3.1, the comparison for each criterion for the tools is listed. First, we compare the Classiq Modeler and QuantuMoonLight on general information aspects such as open-source availability, active maintenance, account requirements, and third-party dependencies. Classiq [Min22] is a proprietary quantum software platform focused on arithmetic operations, with its own SDK and with the Classiq Modeler as one of its core components. In contrast, QuantuMoonLight is a low-code platform designed to support various Quantum Machine Learning (QML) algorithms [ACC+23]. In terms of open-source availability, which allows users to access and modify the code, Classiq is currently free to use but remains closed-source. On the other hand, QuantuMoonLight is free and open-source. However, a significant limitation of QuantuMoonLight is that it has not been actively maintained since 2023, whereas Classiq receives constant updates and maintenance. For both tools, account requirements exist and are defined by the minimum necessary to model and create executable code. This includes having an account on the respective platform with a valid email address. Additionally, QuantuMoonLight requires an IBM account, and starting in July 2025, users will also need to provide credit card information. No model can be created without this account, as the code generation relies on the QML library tied to the IBM platform. Furthermore, this also impacts the criterion of third-party dependencies, which determines whether the tools can be used on their own or require further platforms. QuantuMoonLight depends on IBM's platform, requiring users to have an IBM account to run computations. This dependency introduces constraints, as users cannot switch to alternative quantum cloud offerings and are dependent on IBM's services. Classiq, on the other hand, can be used independently, as it includes its own simulators and provides access to various quantum cloud platforms.

Category / Feature	Classiq Modeler	QuantuMoonLight
General Information		
Open-Source	No	Yes
Actively maintained	Yes	No
Account Requirement	Yes	Yes
Third-party dependency	No	Yes
Functionality		
Primary Focus	Arithmetic operations	QML algorithms
Gatelevel constructs	H, X, Y, Z, RX, RY, RZ, CX, CY, CZ	-
Operator Set	Prepare State Amplitude, Prepare int, Unitary, Suzuki Trotter, QFT, Assignment, Output	Predefined Algorithms
Algorithm Set	QFT	QSVM, QSVC, QNN, QSVR, VQR
State Preparation	Limited to integers and probability distributions	Through IBM
Custom Operators	Limited support	Yes
Classical Constructs	No	Pre-processing and post-processing
Circuit Optimization Goals	Yes	No
Export Formats	QPROG, OpenQASM	Results of pre-processing and post-processing steps
Hardware Integration		
Target Hardware Specification	Yes	Yes
Supported quantum cloud Offerings	Alice and Bob, AWS Braket, Azure Quantum, Google, IBM, Intel, IonQ	IBM
User Experience		
Usage	Website	Local installation
Error Messages	Displayed at synthesis step	After execution

Table 3.1: Comparison of Quantum Low-Code Platforms

Since both tools have different primary focuses, they differ in functionality options. Classiq Modeler allows specifying a quantum ansatz by providing a fixed set of operators, including standard quantum gates and higher-level constructs such as state preparation and algorithms such as QFT. Additionally, Classiq supports expression blocks using constructs like assignment, enabling users to specify operations such as addition directly. However, while classical control constructs like if and repeat are available through the SDK, they are not currently supported in the modeler. Furthermore, support for custom operators is limited. It can only be applied to a single register, but there is no way to merge different registers. In contrast, QuantuMoonLight does not include gate-level constructs but offers algorithms for quantum clustering and quantum classification. Its algorithm set comprises the Quantum Support Vector Machine (QSVM), Quantum Support Vector Classifier

(QSVC), Quantum Neural Network (QNN), Quantum Support Vector Regression (QSVR), and Variational Quantum Regression (VQR). Through its Graphical User Interface (GUI), users can upload and process their own datasets.

The state preparation for the dataset is handled entirely by IBM. While QuantuMoonlight focuses primarily on algorithms, new functionality can be added, as it is open-source. In addition to quantum processing, it also supports essential classical tasks such as pre-processing and post-processing.

One of the features of the Classiq Modeler is the ability to define optimization goals to minimize circuit depth or width, which cannot be specified in QuantuMoonLight. Additionally, the Classiq Modeler supports exporting quantum circuits in multiple formats, including its internal representation, QPROG, and OpenQASM, facilitating interoperability with other tools and platforms. However, QuantuMoonLight only provides the results of the pre-processing and post-processing as diagrams, for example, in PNG format.

Moreover, the tools can be filtered based on the hardware integration they offer. Both the Classiq Modeler and QuantuMoonLight allow users to select a target hardware. However, neither tool can automatically select an appropriate quantum device based on the circuit implementation or predefined algorithm. As previously mentioned, QuantuMoonLight is limited solely to IBM offerings, resulting in vendor lock-in, whereas Classiq supports various different quantum cloud vendors such as Braket and Google.

Finally, we can also examine the Classiq Modeler interface in Figure 3.1 and the QuantuMoonLight interface in Figure 3.2, comparing the two based on their user experience and usability, with criteria outlined in Section 2.2. The setup processes for these tools vary significantly. Classiq Modeler is accessible online, while QuantuMoonLight requires local installation, which involves additional setup steps. Furthermore, QuantuMoonLight's requirement for local installation reduces its accessibility, as not all user groups may be capable of completing the setup. Classiq employs the layout depicted in Figure 3.1, featuring a toolbar and a split left and right section. However, it lacks the capability for drag-and-drop of constructs from the left part, resulting in a lack of direct overview of the constructs. The interface also struggles with manual zooming, which affects usability. To create a quantum algorithm, Classiq uses uniformly structured building blocks. The blocks are black, with assignment blocks having a turquoise border and input/output blocks being purple. These colors are close to each other on the color wheel, possibly because both represent quantum elements. However, there is no directionality on the edges when connecting multiple blocks. During modeling, Classiq does not provide feedback, such as when a letter is placed in a number construct. Consequently, error messages are often not displayed until the synthesis step, which can hinder early debugging. In contrast, QuantuMoonLight employs a basic overview for inputting the datasets required for the machine learning process. It includes multiple drop-down menus for algorithm selection, helping to prevent users from entering invalid inputs. However, no feedback is provided during the design phase if the number of features to extract from the dataset is negative. Both tools also do not offer any shortcuts for experienced users, which can decrease efficiency. In summary, while both Classiq and QuantuMoonLight offer valuable quantum low-code solutions, they reflect different design priorities and trade-offs. Classiq emphasizes broad hardware support and circuit optimization features, but has some usability limitations. QuantuMoonLight focuses on QML algorithms and dataset handling, but suffers from maintenance and dependence on IBM. Understanding these strengths and weaknesses can help users select the platform best suited to their specific quantum programming needs.

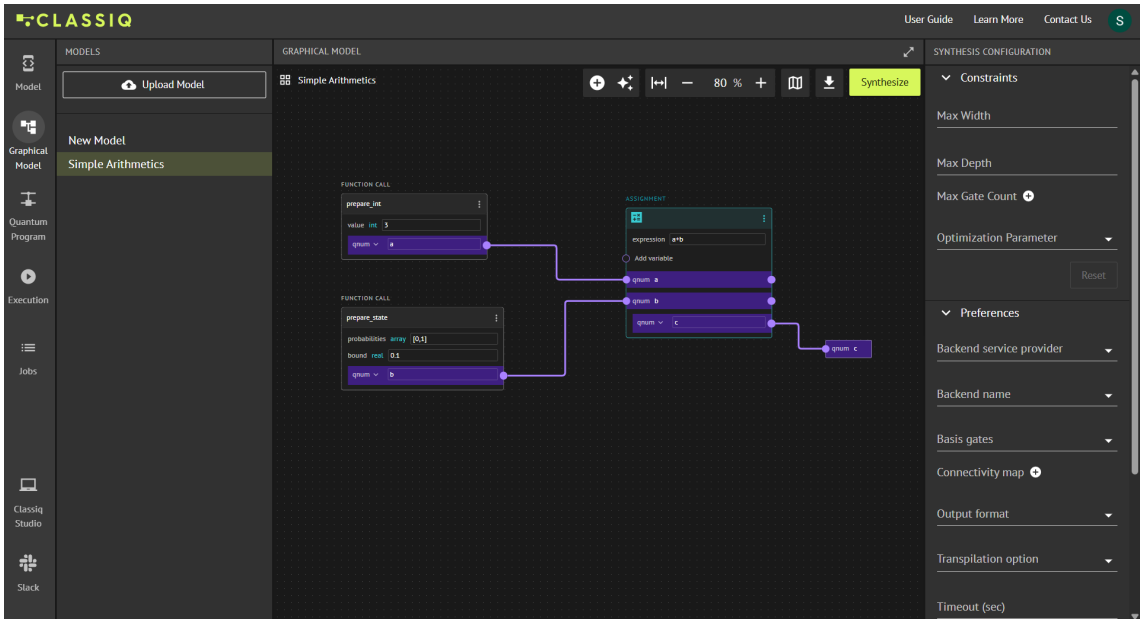

Figure 3.1: Classiq User Interface [Min22]

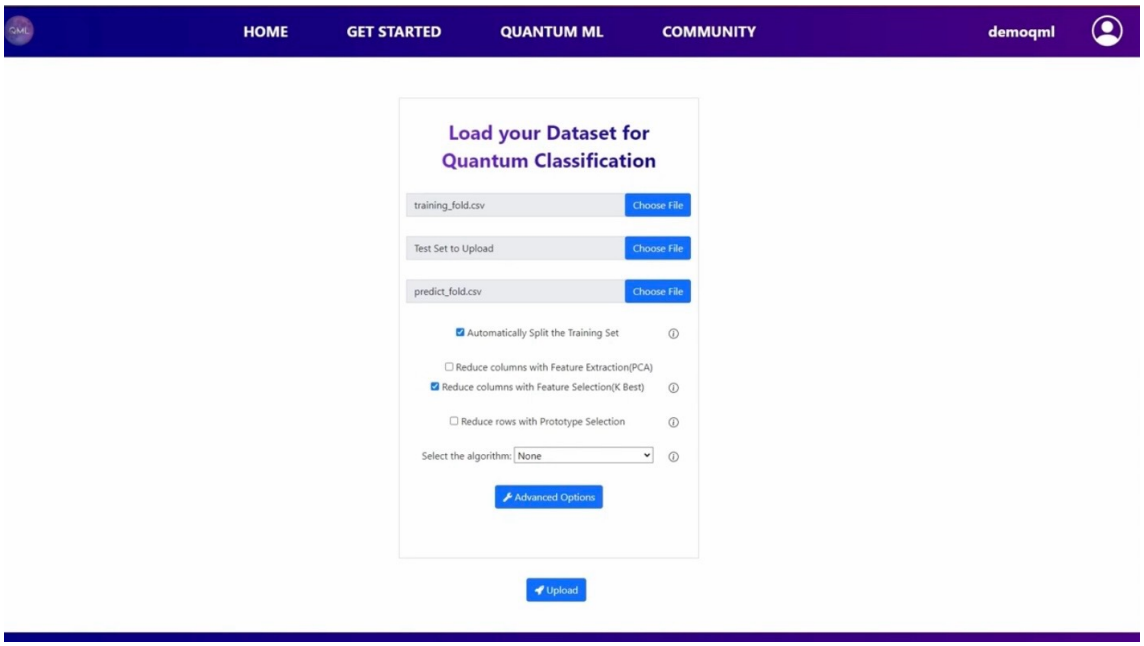

Figure 3.2: QuantuMoonLight User Interface [ACC+23]

4 Problem Statement & Requirements

This chapter presents the problem statement and outlines the requirements to be fulfilled in the context of this work. It discusses the need for a quantum low-code modeling tool capable of supporting the development of quantum algorithms across various levels of abstraction. Based on these considerations, a set of requirements is derived that defines the criteria for the modeling tool proposed in this thesis.

4.1 Problem Statement

Although quantum low-code frameworks exist, as discussed in Chapter 3, they often fall short due to their proprietary or closed-source nature and lack of ongoing maintenance, thereby limiting opportunities for customization. Therefore, designing an open-source modeling tool with relevant constructs is essential to effectively support the development of quantum algorithms. However, identifying a suitable set of modeling constructs is a challenge, as quantum SDKs such as Qiskit [IBM24a] and Amazon Braket [AWS24] differ in their feature sets, supported hardware backends, and levels of abstraction, which affect how algorithms can be expressed. Therefore, standards like OpenQASM are important, as they aim to provide hardware-agnostic representations of quantum algorithms, enabling better interoperability and portability across platforms [CJA+22]. These differences highlight the need for modeling constructs that are not only capable of representing basic quantum structures such as gates and circuits, but are also flexible enough to capture features unique to specific SDKs, while remaining compatible with standards like OpenQASM that promote hardware-agnostic design.

In addition to semantic alignment, it is important to highlight functional semantic differences through appropriate visual representations. The modeling constructs must not only reflect quantum principles accurately but also support a visual syntax that enhances human comprehension, improving the speed, ease, and accuracy of cognitive processing [Moo09]. This visual syntax should be designed to enable intuitive understanding and minimize cognitive load. To achieve this, visual syntax must systematically employ visual variables such as shape, size, orientation, color, texture, and brightness. The combinations of these visual variables can also influence the user's emotional response.

Moreover, quantum algorithm development is inherently multidisciplinary, involving quantum physicists, algorithm designers, and software engineers. These stakeholders differ not only in their roles but also in their levels of experience and familiarity with quantum computing principles. Different stakeholders require different levels of abstraction when working with the same algorithm such as Grover [Gro96]. Quantum experts may focus on the gate-level construction and detailed implementation of the oracle. In contrast, software developers might prefer to use Grover's search as a functional block, integrating it into a larger module. Beginners and learners may only need to understand the high-level purpose and structure of the algorithm without diving into the underlying

mechanics. Therefore, it is important to identify the appropriate set of higher level modeling constructs and lower level modeling blocks such as quantum gates that support different viewpoints and levels of abstraction.

4.2 Requirements

Based on quantum principles, the limitations of existing low-code platforms, and previously identified challenges, the following requirements are derived for both the modeling constructs and blocks, as well as for the overall modeling tool:

Modeling Constructs and Blocks:

- **Requirement 1 (R1):** Each modeling construct and block must adhere to fundamental quantum computing rules, such as the no-cloning theorem and the restriction against feeding classical outputs into quantum inputs.
- **Requirement 2 (R2):** Classical and quantum constructs must be visually distinguishable.
- **Requirement 3 (R3):** Grouping of modeling blocks or modeling constructs on the canvas shall be supported.
- **Requirement 4 (R4):** Classical and quantum modeling constructs must support the specification of their output size.

The modeling constructs and blocks must follow the foundational laws of quantum computing, such as preventing the duplication of unknown arbitrary quantum states and avoiding improper data flow between quantum and classical constructs (**R1**). For example, the input for a state preparation operation, used to encode a classical value into a quantum state, must itself be a classical value, not a qubit. To aid users in distinguishing these differences, visual differentiation is crucial (**R2**). As discussed in Section 2.2, constructs with fundamentally different behaviors, such as classical operations and quantum operations, should appear distinct in diagrams via color and forms. Furthermore, grouping modeling constructs and blocks on the canvas helps structure complex models and improves readability (**R3**). Lastly, specifying output sizes is essential for both design correctness and practical deployment (**R4**). For example, a QFT on an n-qubit register should explicitly declare an n-qubit output, ensuring that subsequent stages in the model match the expected input sizes.

Modeling Tool:

- **Requirement 5 (R5):** The tool must contain sufficient modeling constructs and blocks to model (existing) quantum algorithms.
- **Requirement 6 (R6):** The tool must support drag-and-drop and keyboard shortcuts for editing modeling constructs and blocks.
- **Requirement 7 (R7):** The tool must allow pan and zoom functionality.

- **Requirement 8 (R8):** The tool must support undo and redo functionality.
- **Requirement 9 (R9):** The tool must allow editing of existing diagrams.
- **Requirement 10 (R10):** Models must be checked for compliance with syntactic and semantic rules, with real-time feedback to guide users.
- **Requirement 11 (R11):** The tool must use a standardized format to save and load models.
- **Requirement 12 (R12):** The modeling tool must support integration with existing infrastructure, e.g., through embedding the modeler in an existing web page.
- **Requirement 13 (R13):** The modeling tool must allow users to specify optimization targets with regard to quantum resource usage, including qubit count and circuit depth.
- **Requirement 14 (R14):** The modeling tool should be open-source.

The modeling tool must implement modeling constructs and modeling blocks to ensure the modeling of quantum algorithms (**R5**). A robust set of editing tools facilitates model creation (**R6**). These may include drag-and-drop modeling constructs and blocks, as well as keyboard shortcuts for faster editing. Core interaction features such as pan, zoom, and undo/redo play an important role in enhancing usability (**R7**, **R8**). For instance, zooming allows users to focus on particular parts, while undo/redo functions enable efficient trial-and-error during model development. Additionally, the tool should allow existing diagrams to be loaded and edited (**R9**). Furthermore, real-time validation helps minimize errors and improve usability, as discussed in Section 2.2 (**R10**). For example, the system should flag invalid inputs. In addition, the tool should support serialization to ensure that models can be saved, shared, and reused across different systems and projects (**R11**). Moreover, the system must support integration with external tools such as Qunicorn (**R12**). This enables users to leverage middleware for unified execution across various quantum cloud platforms [WBB+25]. To ensure resource efficiency in quantum execution, the tool should support optimizations for width and depth (**R13**). Finally, extensibility is critical for adapting the tool to new user requirements and emerging new concepts (**R14**).

5 Concept for Modeling Quantum Applications

This section presents the concept enabling the modeling and execution of quantum applications. It begins with a description of the architecture, followed by an overview of the modeling constructs and modeling blocks, including their identification process and properties.

5.1 Architecture

Figure 5.1 presents an overview of the system architecture that supports the modeling and execution of quantum algorithms. In the diagram, components that remain unchanged are displayed in white, expanded components are shaded in gray, and newly introduced elements are highlighted in black. The *Low-Code Modeler* provides an approach for modeling quantum algorithms. It includes several components: a drag-and-drop *editor* with a set of modeling constructs for designing quantum algorithms, a *validator* that checks syntactic and semantic correctness of the models, a *transformer* that prepares models for backend processing, and a *connector* for integrating external services. Additionally, the modeler offers a library of reusable templates to accelerate development. Once the quantum application has been modeled, the transformer removes all data

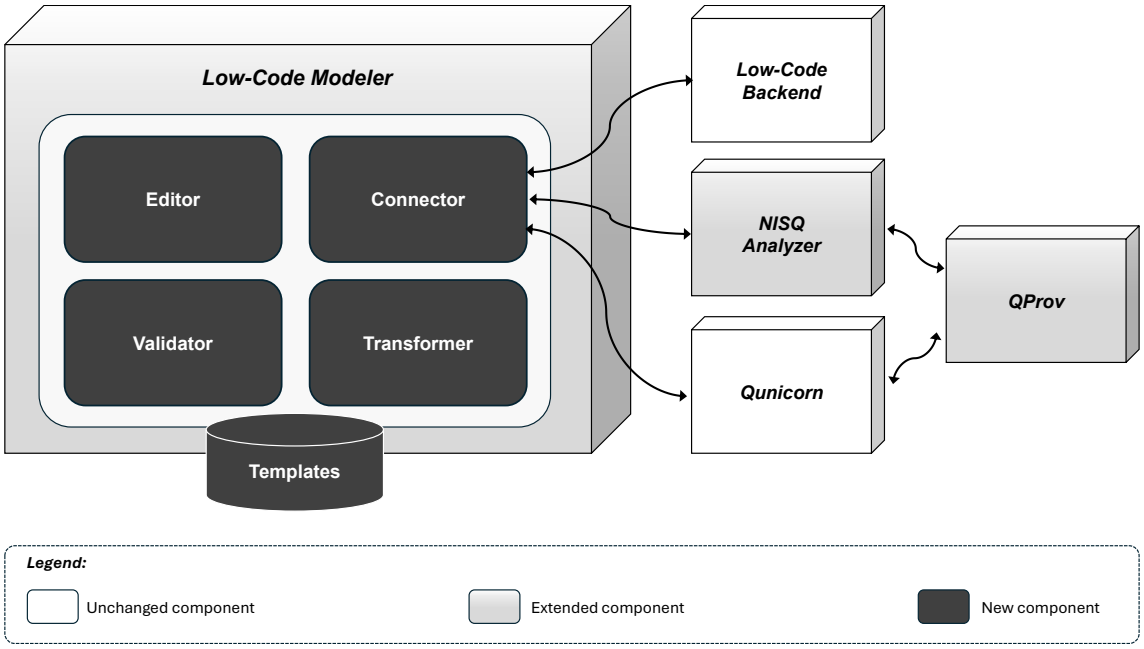

Figure 5.1: Architecture for the Modeling and Execution of Quantum Applications

not needed by the *Low-Code Backend*. In the backend, the modeling constructs are enriched with corresponding OpenQASM3 implementations, which can be selected or adapted based on the required circuit width. The resulting OpenQASM code is then returned to the frontend. From there, the OpenQASM code can either be forwarded to the extended *NISQ Analyzer* [SBB+20] or directly sent to *Qunicorn* [WBB+25]. The NISQ Analyzer assists in selecting an appropriate quantum device based on the specific implementation. It has been extended to filter out backends that do not support conditional operations. To enable this feature, *QProv* [WBL+21], which is responsible for continuously collecting and storing information about available quantum devices, was adapted accordingly. Once a suitable backend is selected, the OpenQASM output can be forwarded to Qunicorn. This component provides a middleware for the unified execution of quantum algorithms across heterogeneous quantum cloud providers [WBB+25].

5.2 Modeling Constructs and Blocks

First, the identification process of the modeling components is described, incorporating both higher-level constructs and low-level blocks. Next, the visualization approach is discussed, detailing how different shapes and colors represent functional semantics to assist the user and reduce the likelihood of errors during interaction. Finally, the properties of each modeling component are presented, including input and output ports.

5.2.1 Identification of Modeling Constructs and Blocks

To determine the modeling components required for representing quantum applications within the modeling tool, we employed a structured, multi-phase research methodology. This approach was grounded in theoretical examination and analysis of implementations. Initially, we examined the quantum computing pattern language, as it encapsulates recurring structures and building blocks commonly used in quantum algorithms [BBB+23]. Subsequently, we analyzed OpenQASM 3, evaluating which constructs are supported within this representation. Furthermore, we identified additional modeling components not explicitly captured in the pattern language and evaluated their implementation and support in widely used SDKs, such as Qiskit and Amazon Braket.

Based on our findings, we identified the following key modeling constructs and blocks as necessary to support the expression and execution of quantum algorithms: *Classical data types* play an important role in the design and execution of quantum algorithms, particularly for control flow. In accordance with the OpenQASM 3 specification, we incorporate eight classical data types: *int*, *float*, *duration*, *array*, *angle*, *bit*, *complex*, and *boolean*. In addition to classical data, quantum programs operate on *quantum data types* such as *qubits* and *ancilla*. Ancilla qubits are temporary qubits used for intermediate steps in a computation. They may be *dirty* or *reset*, and can be *reusable* if properly returned to their original state through a process known as uncomputation.

Circuit-level nodes constitute the fundamental operations within quantum circuits. These include the *Hadamard* gate, the *Pauli* gates and *rotation* gates, the *CNOT* gate, and the *Toffoli* gate. Additionally, as discussed in Section 2.3.2, the T gate is required for the efficient decomposition of unitaries. We also incorporate all gates defined in the OpenQASM 3 standard library, ensuring compatibility with

the language specification. To support flexible manipulation of qubits and registers, we introduce *Splitter* and *Merger* nodes. These enable the conversion between quantum registers and individual qubits and vice versa.

To enable interaction between classical and quantum parts, *Boundary Nodes* are necessary. These nodes handle tasks such as *state preparation*, which encodes classical information into quantum states, and *measurement*, which extracts classical information from quantum computations. *Control Structure* nodes enable conditional and iterative logic within the modeling tool. Constructs such as *if-else* allow for branching, conditionally executing operations based on classical values, typically derived from measurement results. Similarly, the *repeat* block enables the repeated execution of a specified set of operations for a defined number of iterations or until a certain condition is met.

Operations nodes represent higher-level constructs that abstract away from low-level circuit modeling, enabling users to focus on the logical structure of computations rather than gate-by-gate implementation. *Quantum operations* include *arithmetic*, *comparison*, *bitwise logic*, and *min/max* functions. Analogously, these also exist as *classical operations*.

Custom nodes exist for classical and quantum inputs and outputs to provide a mechanism for defining user-specific behaviors and abstractions.

5.2.2 Visualization of the Modeling Constructs

Each modeling block is represented by a unique icon and a distinct name. As discussed in Section 2.2, different functional semantics are visually distinguished to assist the user and help prevent errors. Quantum operations are depicted as blue rectangles, reflecting the box-like shapes commonly used in quantum circuits. Since ancillae are a specialized subtype of qubits used temporarily during computation, they share the same rectangular shape but are colored green. Green was chosen to represent ancilla qubits, maintaining a visual connection to the quantum domain. Within this scheme, dirty ancillae are further distinguished by a darker shade of green to clearly differentiate them from other qubit types in the visualization.

Importantly, every node features dedicated ancilla and dirty ancilla inputs and outputs, each rendered with specialized shapes that visually convey their function. Specifically, the ancilla input is depicted as an arrow-shaped icon, while the corresponding ancilla output has a complementary indentation designed to fit perfectly with the arrow shape of the input. This visual matching reinforces the logical connection between input and output ancillae, helping users intuitively understand their pairing and flow. Similarly, dirty ancilla inputs and outputs follow the same shape-based design principle. The uncompute output for the ancilla is depicted as a horizontally rotated variant of the ancilla output shape, symbolically representing the reversal of the ancilla state.

To emphasize the distinction between classical and quantum constructs, classical operations are shown as rounded shapes colored orange, the compound color to blue on the color wheel, providing a clear visual contrast. Additionally, the flow between nodes is directed to clearly visualize sources and targets. Quantum data flows are shown in blue, consistent with quantum operation coloring, while classical data flows are depicted in orange, matching the classical operations' color scheme. Control structure nodes, which integrate both classical and quantum elements, are represented by purple hexagons, a blend of the forms and colors used for classical and quantum constructs, visually signaling their hybrid nature.

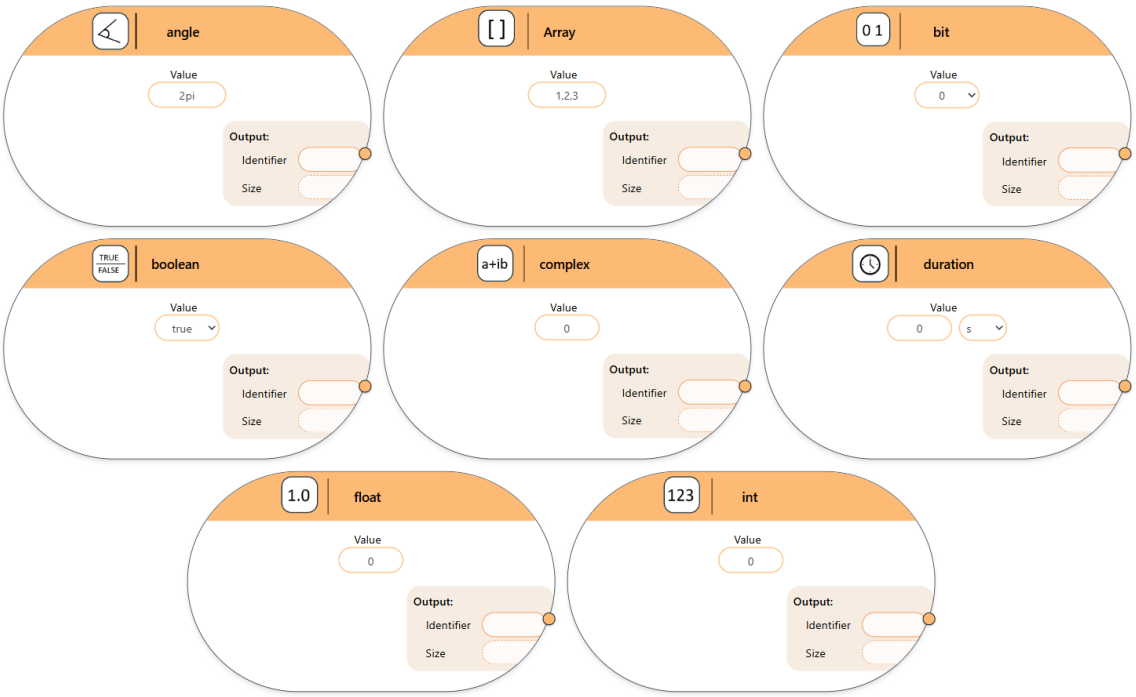

Figure 5.2: Supported Classical Data Types

5.2.3 Data Types

In modeling tools that integrate both classical and quantum paradigms, various data types are needed to represent information, with classical data types illustrated in Figure 5.2 and quantum data types illustrated in Figure 5.3. Each classical data type is associated with an *output identifier* and, optionally, a *size*, which is visually indicated by a dashed border. If a size is explicitly provided, it must be greater than zero. In this case, the data object is created with the specified size. If the size is omitted, it is treated as machine-dependent, and the backend is responsible for determining an appropriate size based on the target architecture. For the classical data types, the *value* can be explicitly specified. Because ancillae and qubits are always initialized in the $|0\rangle$ state, they do not require an explicit value. Both classical and quantum data types are source elements in the model and therefore do not have input ports. Instead, they each expose a single output port, through which their data or state can be connected to subsequent operations.

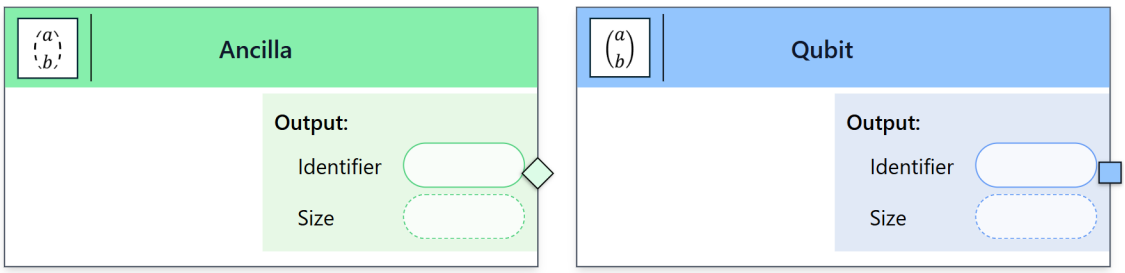

Figure 5.3: Supported Quantum Data Types

5.2.4 Circuit-level Nodes

At the circuit level, the modeler provides a set of quantum gates and a qubit element, which are referred to as modeling blocks, as illustrated in Figure 5.4. The *qubit* is represented as a source node with exactly one output port. Quantum gate nodes are modeled based on their arity. Single-qubit gates have one input and one output. Two-qubit gates have two inputs and two outputs. Furthermore, three-qubit gates have three inputs and three outputs. Gates are categorized into two types: *non-parameterized* and *parameterized*. As shown in Figure 5.4, parameterized gates include a visual representation of their associated parameters, which are float values that must be explicitly provided by the user. The number of required parameters is fixed for each gate type and is also visually indicated on the gate node. Additionally, the *type of parameter* can be selected by the user and is either in radians or degrees.

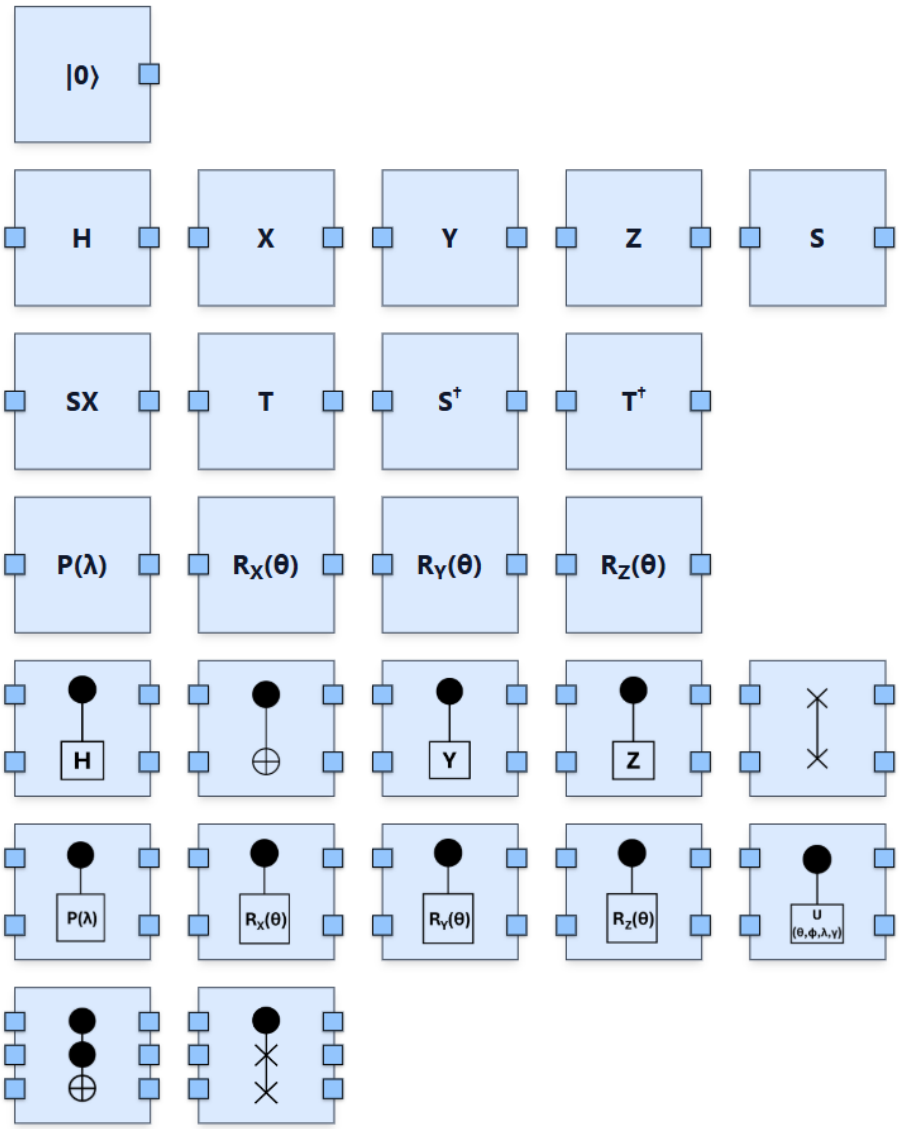

Figure 5.4: Supported Circuit-level Nodes

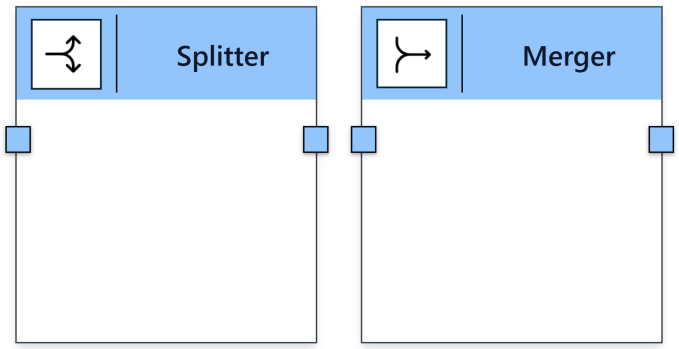

Figure 5.5: Splitter and Merger Nodes

The *Splitter* and *Merger*, visualized in Figure 5.5, are complementary modeling constructs used to manipulate quantum registers. The Splitter takes a quantum register and splits it into individual qubits. The user has to specify the size of the qubit register. Conversely, the Merger block takes multiple single qubits and combines them into a single quantum register.

The circuit model supports *broadcasting*, as introduced in OpenQASM 3 [CJA+22], which allows a single gate operation to be applied element-wise across multiple qubits. Broadcasting is only valid when all involved quantum registers have the same size, ensuring consistent application across corresponding qubit pairs. All gate implementations are managed by the backend, enabling platform-specific optimizations. If a required operation is not natively supported, it can be synthesized using known decomposition techniques, as discussed in Section 2.3.2. Additionally, users can define their own *custom operators*, extending the available gate set with user-defined functionality.

5.2.5 Operations

All modeling constructs in this section share several common properties. Each construct allows the specification of an *implementation type*, such as Quantum Intermediate Representation (QIR) or OpenQASM. These formats support the definition of quantum operations in different intermediate or target-level representations, providing flexibility for execution across various quantum cloud offerings. Additionally, each construct includes an *implementation* attribute, which may contain an *uncompute* section. This refers to the reversal of operations to clean up ancillae. Structurally, each modeling construct except the measurement node features six standardized ports: one input *ancilla port* and one output ancilla port for ancillae, one input *dirty ancilla port* and one output dirty ancilla port for dirty ancillae, one *result port* for operation outputs, and one *uncomputation port* to support reversal of computations. To assist users and reduce the required expertise, the modeler enables users to select an implementation that invokes the enricher of the low-code backend.

Boundary Nodes

The boundary nodes are illustrated in Figure 5.6. State preparation supports two modes: *Encode Value* and *Prepare State*. In the *Encode Value* mode, a classical value is mapped to a quantum state based on the specified *encoding type*, as described in Section 2.3.5. This process is controlled by a user-defined error *bound*. A bound of 0 indicates exact state preparation. However, exact

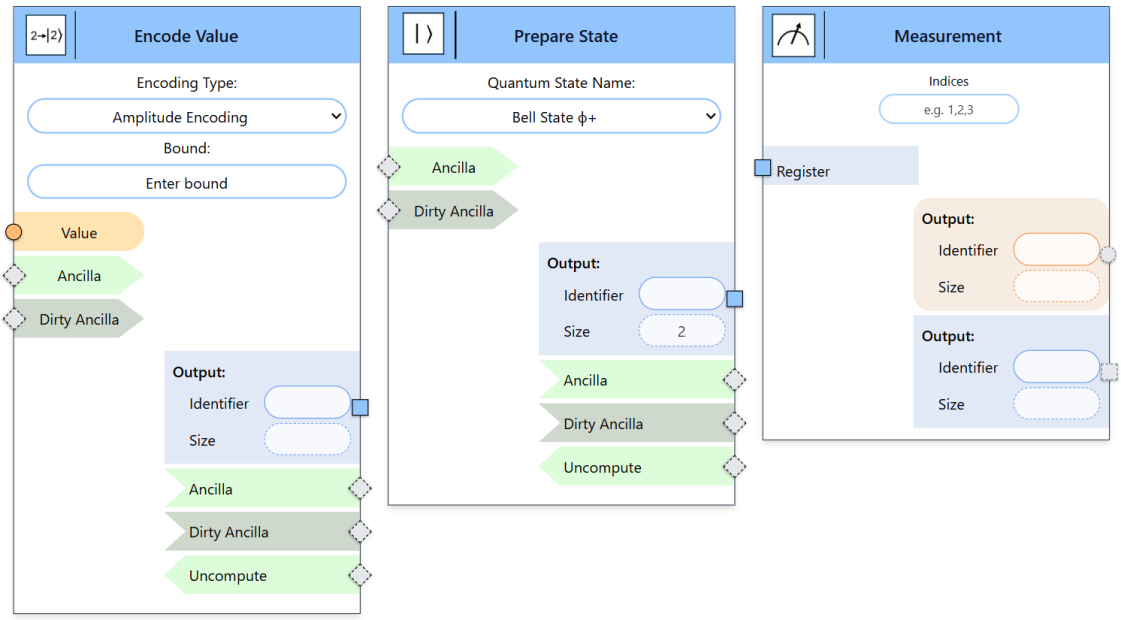

Figure 5.6: Supported Boundary Nodes

preparation may not always be possible for arbitrary states [RYH+23]. In this construct mode, there is one additional input port that receives the classical value to be encoded, and one output port that outputs the resulting quantum state. Furthermore, the modeler allows users to define their own custom encoding schemes, enabling tailored state preparation strategies beyond the predefined types. In the *prepare state* mode, users can initialize a specific quantum state from a set of important states described in Section 2.3.5. Additionally, users have the option to define their own quantum state.

Each measurement node has one input port and two output ports. The input port receives a quantum register. One output port provides the classical result of the measurement, while the other outputs the post-measurement quantum state. To control which qubits are measured, the node includes an *indices* property, specifying the positions within the quantum register to be measured. These indices must lie within the bounds of the register size. This enables *partial measurement*, measuring only selected qubits while leaving unentangled qubits unaffected. The low-code backend automatically generates the implementation corresponding to this measurement behavior. As discussed in Section 2.3.4, measurements are performed in the Z-basis by default. To support alternative measurement schemes, users can specify the *basis* property, which allows selection of Pauli strings to define the measurement basis.

Operators

Operators within the modeling tool are categorized into several classes, including *arithmetic* operators, *bitwise* operators, *comparison* operators, and *min/max* operators, which are depicted for the quantum context in Figure 5.7. The classical counterparts of the operators are shown in Figure 5.8. Arithmetic, bitwise, and comparison operators require two input ports, as they operate on pairs of values. In contrast, min/max operations require only a single input port and operate over a collection of values provided through that input. The output type produced by each operator

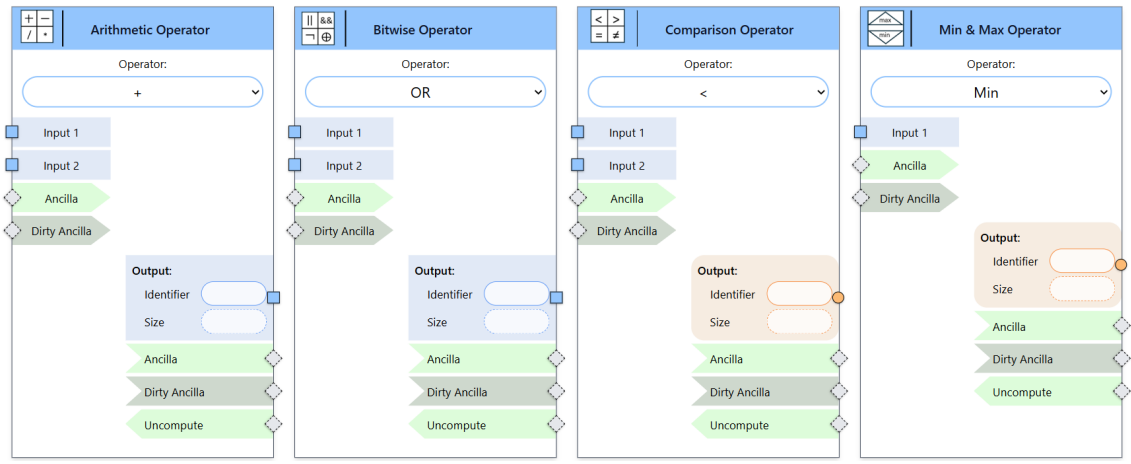

Figure 5.7: Supported Quantum Operators

depends on its category. Arithmetic operators generate a quantum output, as their results may be used in further quantum operations. On the other hand, bitwise, comparison, and min/max operators produce classical outputs, reflecting their deterministic and classical nature.

Each operator node also includes an *operator* property, which explicitly specifies the operation to be performed. The supported arithmetic operators are +, -, *, /, and %. The supported bitwise operators include *AND*, *OR*, *XOR*, *NOT* (unary). Comparison operators include ==, ! =, <, <=, >, and >=. The min/max category supports the operators *min* and *max*.

5.2.6 Control Structure Nodes

The supported control structure nodes are depicted in Figure 5.9. Both node types can accept ancilla inputs. The *repeat* node includes a property called *number*, which specifies how many times the contents of the node are executed. Importantly, the inputs to the node must match the outputs to ensure consistency across iterations. The *if-else* node operates based on a *condition* and requires a single classical input. Both the *if* and *else* branches share the same input and produce outputs of the same type and size. If one branch is left empty, it behaves as an identity operation, effectively leaving the input unchanged. While it is possible to define new variables within these control blocks, such variables are local to their respective scopes and do not persist outside the block.

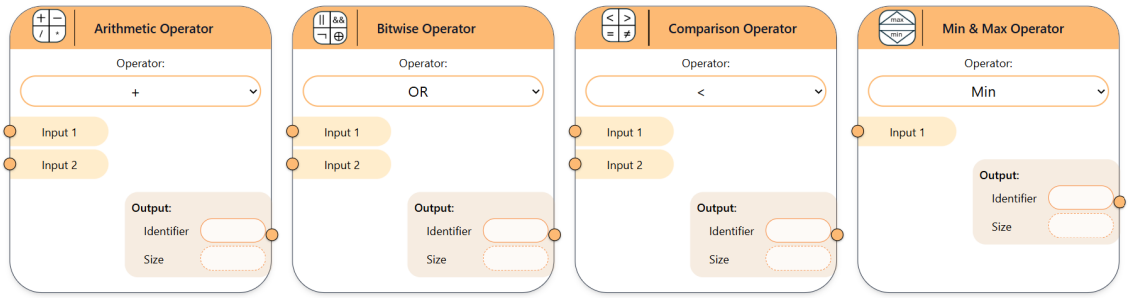

Figure 5.8: Supported Classical Operators

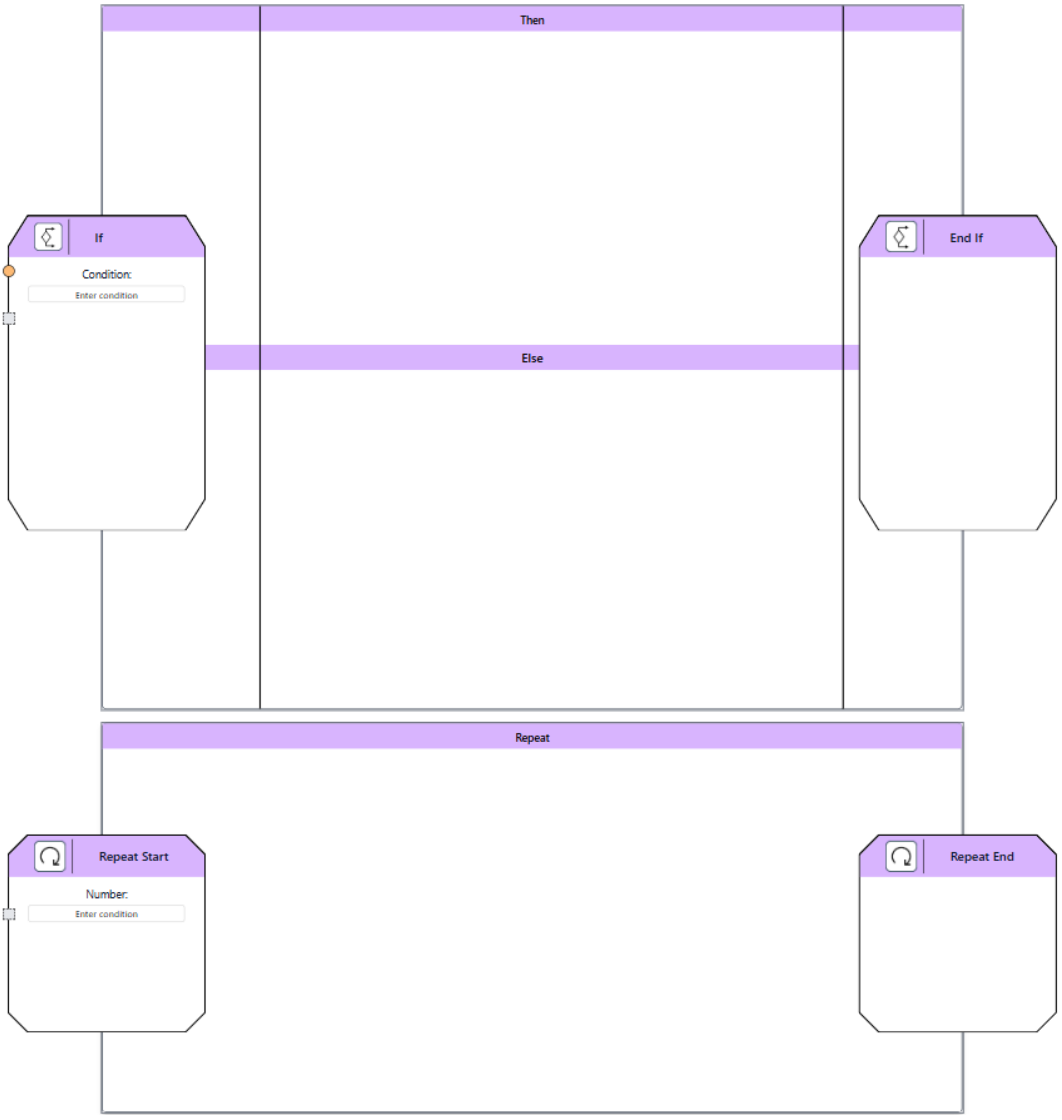

Figure 5.9: Supported Control Structure Nodes

5.2.7 Custom Nodes

Since not every possible operation can be anticipated in advance, the modeling tool allows users to create custom nodes, as shown in Figure 5.10. These nodes can be defined for both classical and quantum contexts, providing flexibility for extending the modeler with user-specific logic. Custom nodes have a configurable *input* and *output* property that allows users to define a positive number of input and output ports. This enables the modeling of complex or non-standard operations that do not fit within the predefined categories, making the tool adaptable to evolving concepts.

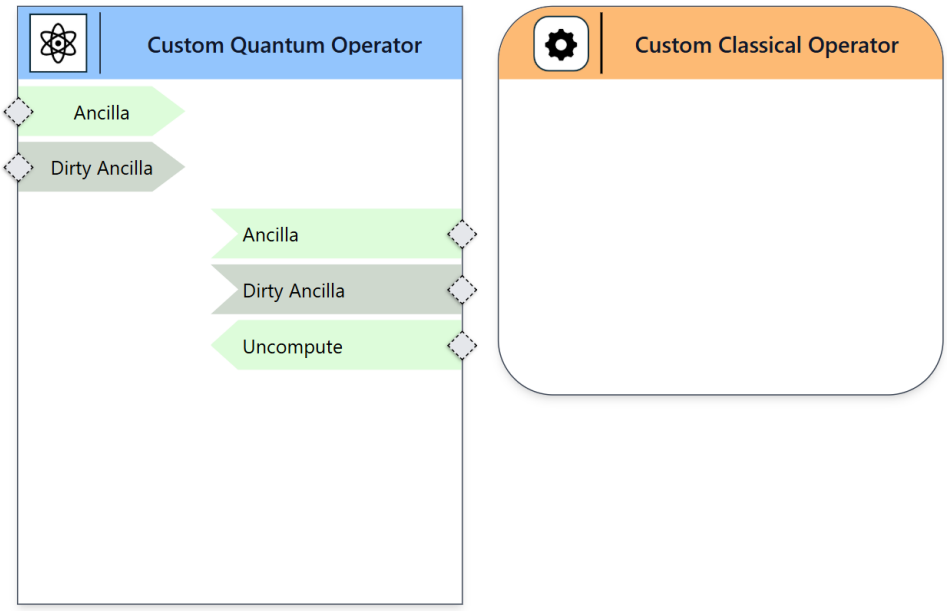

Figure 5.10: Supported Custom Operators

6 Implementation

This chapter presents the prototypical implementation of the quantum low-code modeling tool. First, it discusses the selection of the underlying framework, followed by a detailed description of the implementation of the proposed concept. Finally, the chapter highlights the current limitations of the implementation and provides a foundation for potential future improvements.

6.1 Selection of a Diagramming Library

The selection of a suitable diagramming library was based on a set of selection criteria. These criteria are designed to ensure compatibility with an open-source, community-driven development model and to reduce long-term risks related to maintainability and licensing as discussed in Section 4.1. The selection criteria are:

- **Open-source license:** The library must be freely available and modifiable under a permissive license.
- **Community support:** Actively developed and maintained by a diverse contributor base, rather than being dependent on a single individual or company.
- **Customizability:** Ability to customize nodes and edges.
- **Export capability:** Support for exporting and importing diagram models.
- **Documentation quality:** Technical documentation should be clear, accurate, and up-to-date.
- **Project activity**: Ongoing development and a visible roadmap to ensure continued support and evolution of the library.

Based on these criteria, we conducted a survey of JavaScript-based diagramming libraries. A total of 20 libraries were evaluated. Several popular tools were quickly ruled out due to proprietary licensing. For example, *GoJS* [Nor98] and *JointJS* [Cli10] offer features such as node collapsing, but these are gated behind commercial licenses or limited customization rights, making them incompatible with our open-source objectives. Similarly, tools like *MindFusion* [Min25], *DHTMLX* [Din25], and *Webix Diagram* [XB 25] impose trial periods or require payment for full access.

Libraries such as *Nomnoml* [VL19] and *Mermaid.js* [Sve25] were more promising due to their permissive licenses and simplicity. However, both are tailored to UML diagramming and lack the extensibility required for more complex, flow-based interactions. Tools like *Diagram.js* [bpm25], *State.js* [Far13], and various BPMN-specific libraries were excluded due to limited flexibility or insufficient documentation. We also reviewed general-purpose visualization libraries such as *D3.js* [Mik25], *Fabric.js* [BNE25], and *Cytoscape.js* [FLF+25]. Although powerful, these

frameworks are primarily optimized for data visualization rather than interactive diagram editing, and therefore do not meet our requirements for dynamic user interaction and model manipulation. Frameworks like *mxgraph* [JGr20] were not relevant since their support ended in 2020.

Ultimately, *React Flow* [xyf25] was selected as the most suitable solution, as it fulfills all defined criteria. It is open-source, actively maintained by a broad community, and allows extensive customization of nodes and edges. In addition, it supports export and import of diagrams in JSON format and is purpose-built for building interactive node-based editors. These capabilities make it a robust and future-proof foundation for the implementation described in the following sections.

6.2 Structure of the User Interface

The modeler interface, shown in Figure 6.1, consists of several components that facilitate the creation of quantum algorithms. At the top of the screen is the *toolbar*, which provides functions such as *New Diagram*, *Open*, *Save*, *Save As*, *Upload*, *Configuration*, *Send to Backend*, and *Send to Qunicorn*. These options enable users to manage files, configure the modeling tool, and deploy quantum algorithms. Tooltips and visual feedback confirm actions and enhance usability.

On the left side is the *palette*, offering a searchable, categorized list of modeling constructs and blocks. Categories include *Boundary Nodes*, *Circuit-level Nodes*, *Data Types*, *Operators*, *Control Structure Nodes*, and *Custom Operators*, as described in Section 5.2. Users can drag and drop elements from the palette onto the canvas. Usability features include a responsive search bar, intuitive category labels, and collapsible groups to reduce visual clutter, as discussed in Section 2.2.

The central *canvas area* serves as the main workspace for diagram construction. It features a grid layout to support alignment and design precision. Users can interact with the canvas using keyboard and mouse controls, zooming in and out, resetting the view, or locking the canvas via the controls located at the bottom left. Additional interaction tools include keyboard shortcuts

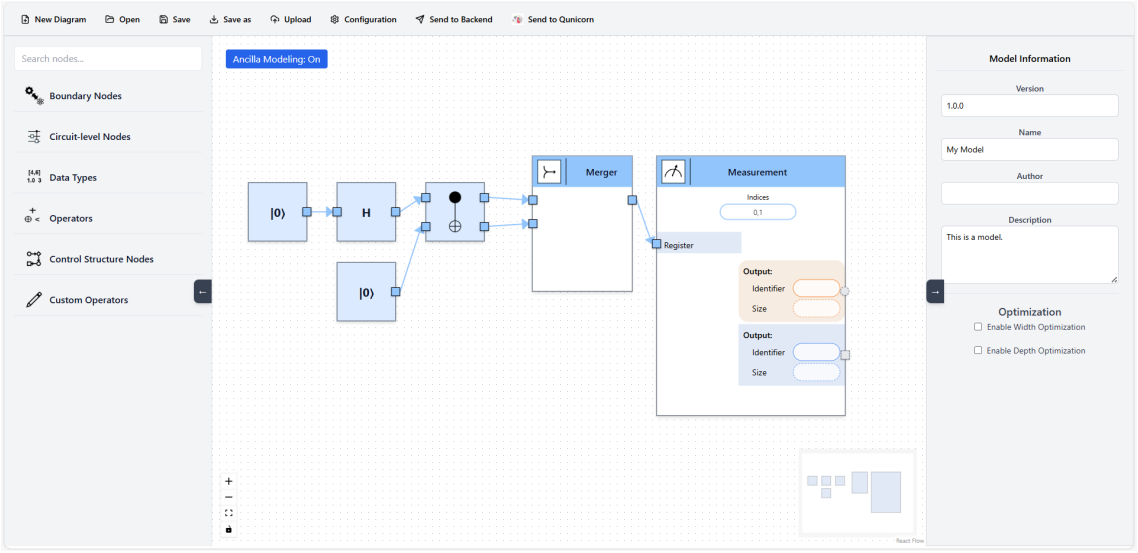

Figure 6.1: User Interface of the Quantum Low-Code Modeler

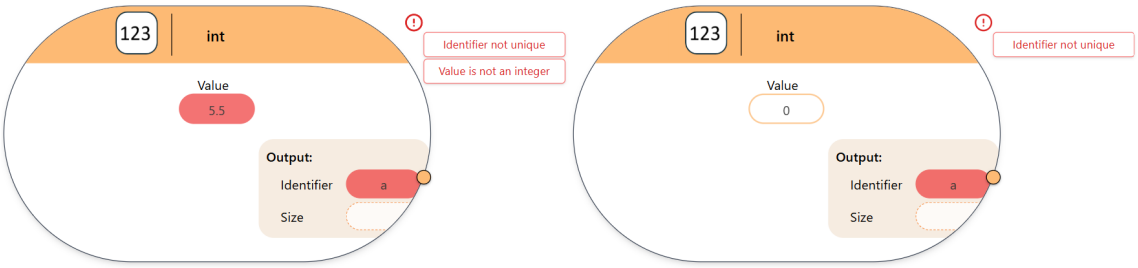

Figure 6.2: Illustration of Data Validation with Unique Identifiers and Integer Value

such as *Delete* to remove nodes, arrow keys for moving elements, a lasso tool activated by holding *Shift* to select multiple nodes, and *Undo/Redo* functionality to support editing. As an example, shown in Figure 6.1, a user can model a simple quantum circuit by placing two qubits on the canvas, applying a *Hadamard* gate to the first qubit, followed by a *CNOT* gate to entangle them. The qubits are then merged, and a measurement operation is added to complete the circuit. In the bottom right corner, a *minimap* provides a zoomed-out view of the entire diagram. This enhances navigation in large or complex models by showing the active workspace area and allowing users to click and drag to reposition the viewport. The minimap also supports color coding and shape differentiation to improve visual context and clarity.

On the right side of the interface is the *properties panel*, labeled *Model Information*. It includes metadata fields such as the model name, version, author, and description. Additionally, it provides configuration options for enabling width or depth optimization. When a node is selected, this panel dynamically updates to display the editable properties of that node.

As discussed in Section 4.1, users have varying levels of experience. To accommodate this, we provide a toggle button that allows users to disable the modeling of ancilla nodes and their associated connections. This feature simplifies the system representation for beginners, while still enabling advanced users to access the full complexity of the model when needed.

As outlined in Section 2.2, it is essential to provide support mechanisms that help users avoid modeling errors. Different quantum modeling tools handle input validation differently. For instance, Classiq displays validation errors only after the user initiates the synthesis process, whereas QuantuMoonLight does not provide feedback during the design phase. To enhance user experience and reduce the likelihood of runtime failures, our modeling tool implements immediate input validation during the design process. Input validation ensures that the values entered by the user conform to the expected data types. If an invalid input is detected, such as a non-integer value in an integer-only field, the system provides real-time visual feedback, using color coding and error messages, to guide the user in correcting the error. This feedback mechanism is illustrated in Figure 6.2, where two data type nodes of type int share the same output identifier, and one of the nodes has a value of type float. Such inconsistencies are detected by validation rules, which enforce the uniqueness of identifiers to prevent naming collisions that could lead to ambiguous behavior in the backend system. When validation fails, the model cannot be submitted for execution, thereby ensuring correctness before runtime. Furthermore, the modeling tool prevents the creation of duplicate edges and disallows connections that violate logical constraints, for example, forbidding edges between classical outputs and quantum inputs.

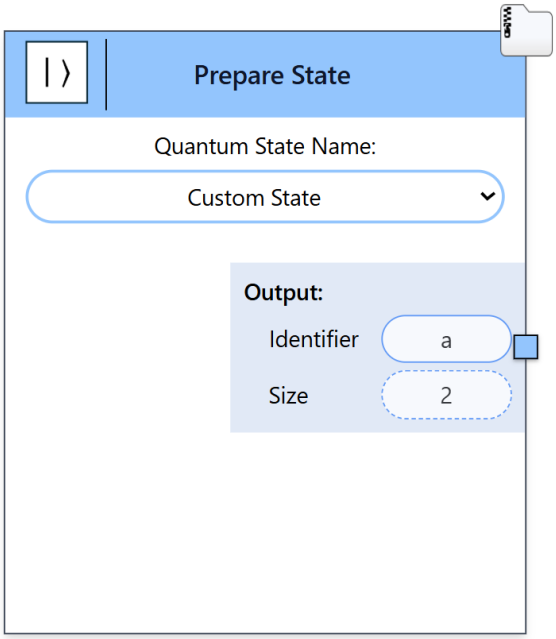

Figure 6.3: Indication of Node Implementation

In Figure 6.2, an example is also shown where an implementation is available for a node. This is indicated by the presence of a ZIP folder icon, located in the top-right corner of the node. Since the type of implementation can vary, this visual cue provides the user with a clear and immediate understanding that an implementation is attached to that particular node.

6.3 Serialization Format and Execution Results

At the top of the JSON in Listing 6.1, there is a `metadata` section (lines 2 to 13) containing information about the model. This includes the version, which in this case is the initial release of the graph representation. The metadata also provides the model's name, author, and description, offering context about the model's purpose. Additionally, an identifier and a timestamp appear, allowing identification of the model instance and recording its creation time. This metadata section enables version control and provides support for applying model transformations in response to changes in the JSON schema.

Following the metadata, the `nodes` array (lines 14 through 40) defines the graph's components. Each node corresponds to a modeling construct with the properties defined in Section 5.2 and is assigned a Universally Unique Identifier (UUID) instead of a simple, incremental string. This use of UUIDs guarantees global uniqueness across models, preventing identifier collisions during graph editing. Along with its UUID, each node contains spatial positioning details (lines 19 through 22) that determine where it appears in the graphical interface, as well as a `data` object (lines 23 to 32) encapsulating the node's operational logic. The `edges` section (lines 41 through 56) currently indicates that connections exist between nodes. Edges define directional links between nodes, representing the flow and dependencies within the quantum model. Finally, the `viewport` section

describes view settings such as x and y coordinates and the zoom level, controlling how the graph is displayed in the user interface. Here, the viewport is positioned at coordinates (343.18, 296.77) with a zoom factor of 0.95.


```
{
  "metadata": {
        "id": "flow-1748805851917",
        "version": "1.0.0",
        "name": "My Model",
        "description": "This is a model of a Bell state.",
        "author": "Sharon",
        "timestamp": "2025-05-30T19:24:11.917Z",
        "optimizeWidth": true,
        "optimizeDepth": false,
        "width": "10"
  },
  "nodes": [
   ...,
   {
      "id": "2a2c37f2-c487-4144-8880-9ccacda8998a",
      "type": "measurementNode",
      "position": {
        "x": 450,
        "y": 420
      },
      "data": {
        "label": "Measurement",
        "inputs": [],
        "implementation": "",
        "implementationType": "",
    "uncomputeImplementationType": "",
        "uncomputeImplementation": "",
        "indices": "0,1,2",
        "outputIdentifier": ""
      },
      "width": 320,
      "height": 400,
      "positionAbsolute": {
        "x": 450,
        "y": 420
      }
    }
  ],
  "edges": [
    {
      "source": "fdde2dc7-bebf-4974-b87e-5070a865318f",
      "sourceHandle": "quantumHandleGateOutput0fdde2dc7-bebf-4974-b87e-5070a865318f",
      "target": "2a2c37f2-c487-4144-8880-9ccacda8998a",
      "targetHandle": "quantumHandleGateInput08ae3a110-0190-4547-b08a-f33be6d3d203",
      "type": "quantumEdge",
      "id": "daf540b2-26c4-4201-bde1-fe48d592d57d",
      "markerEnd": {
        "type": "arrowclosed",
```

```
        "width": 20,
        "height": 20,
        "color": "#93C5FD"
      }
    },
    ...
  ],
  "viewport": {
    "x": 343.1807306131516,
    "y": 296.7762319889813,
    "zoom": 0.9540210602904574
  }
}
```

Listing 6.1: Graph structure in JSON format

After the model is created, as shown in Listing 6.1, it undergoes a pre-processing step to remove information that is not required by the low-code backend. The result of this pre-processing is illustrated in Listing 6.2. In the pre-processed output, all metadata is retained. Each node includes its type and ID. If an implementation exists for a node, its type is marked as implementation, and such nodes are excluded from further enrichment. For all other nodes, the input fields are preserved. Only top-level nodes, those not nested within a parent, are represented directly. Nodes that belong to a parent are encapsulated within their parent node's structure. Regarding edges, the pre-processing includes their source and target identifiers. Additionally, the edge index is derived from the handle connection. Specifically, the first handle in the list corresponds to the first connected port, ensuring consistency in connection tracking.

Once we receive the result from the backend, it can either be sent to the NISQ Analyzer for hardware selection or forwarded to Qunicorn if the target device is already known. A modal window opens in the modeler, allowing the user to start Qunicorn's multi-phase process. Beginning with a deployment phase, followed by the creation of a job using the corresponding deployment ID. A progress bar is displayed inside the modal window to indicate the status of execution. Once the job completes successfully, the results are visualized in a histogram. If the job has not completed after 20 polling requests, a timeout error is triggered, and the user is notified with an error message indicating that the operation has timed out and was unable to complete within the expected timeframe.

6.4 Extension of QProv and the NISQ Analyzer

Due to the introduction of new modeling constructs that support classical operations, the QProv collection had to be adapted to determine whether a quantum device is capable of handling conditional operations. This information became accessible through a request to the IBM device configuration. Since the previous API version of IBM and QProv did not provide access to this data, migrating to the updated interface was necessary. This migration entailed additional setup effort, including the registration of an IBM account linked to valid billing information, to enable access to advanced device capabilities. Previously, the NISQ Analyzer computed combinations for every quantum device and compiler, regardless of whether the device could support conditional operations. To address this, we modified the analyzer's behavior to consider the conditional capability information provided by QProv.

```
{
    "metadata": {
        "id": "flow-1748805851917",
        "version": "1.0.0",
        "name": "My Model",
        "description": "This is a model of a Bell state.",
        "author": "Sharon",
        "timestamp": "2025-05-30T19:24:11.917Z",
        "optimizeWidth": true,
        "optimizeDepth": false,
        "width": "10"
    },
    "nodes": [
     ...
        {
            "id": "fdde2dc7-bebf-4974-b87e-5070a865318f",
            "type": "merger",
            "numberInputs": "2"
        },
        {
            "id": "2a2c37f2-c487-4144-8880-9ccacda8998a",
            "type": "measure",
            "indices": [0,1]
        }
    ],
    "edges": [
        {
            "source": [
                "fdde2dc7-bebf-4974-b87e-5070a865318f",
                0
            ],
            "target": [
                "2a2c37f2-c487-4144-8880-9ccacda8998a",
                0
            ],
            "identifier": null,
            "size": 2
        },
        ...
    ]
}
```

Listing 6.2: Request to Backend

6.5 Limitations

The current modeling tool establishes a robust framework for quantum algorithm design. However, it lacks certain advanced layout functionalities, such as automated edge rearrangement, which have the potential to improve graph readability and enhance the overall user experience. Furthermore,

the canvas flow is effectively optimized for moderate construct sizes, with sequences generally limited to approximately five constructs before interface usability becomes challenging. This constraint impacts the modeling and visualization of more complex quantum algorithms. Although features such as a minimap and zoom capabilities facilitate navigation within larger models, these provide only limited mitigation. Additionally, the absence of a context menu limits user interaction efficiency, as options such as deleting nodes are missing from the right-click interface. Notably, deletion is currently only possible via keyboard input.

7 Validation

This chapter validates the proposed modeling constructs and blocks through a set of representative use cases, demonstrating their applicability. Furthermore, the compliance with the requirements described in Chapter 4 is evaluated.

7.1 Use Cases

This section presents a subset of use cases that serve as representatives of their classes, demonstrating different approaches to modeling quantum algorithms. The other use cases can be found on GitHub [Sti25]. It begins with the creation of a GHZ state, which can be modeled either using individual quantum gates or through a high-level construct that prepares the desired state directly. The chapter then explores how transitions between detailed gate-level modeling and higher-level abstractions can be achieved using components such as Splitter and Merger. To support efficient gate-level modeling, the concept of broadcasting is introduced, allowing a single operation to be applied across multiple qubits. Additionally, use cases involving custom operators show how complex and reusable logic can be integrated into circuit models. Dynamic circuit behavior is addressed through a use case, which demonstrates how conditional control flow can be represented within a quantum model. The chapter also introduces grouping mechanisms that allow related operations to be organized into coherent, modular components, improving both clarity and reusability. Finally, the chapter concludes with examples of several widely known quantum algorithms, including Grover's algorithm, the Deutsch–Jozsa algorithm, and Shor's algorithm. These examples illustrate the expressiveness and flexibility of the modeling tool in representing quantum algorithms.

GHZ State

The first use case, depicted in Figure 7.1, demonstrates the modeling of a GHZ state comprising 7 qubits. This state is maximally entangled, achieved by applying a Hadamard gate to the first qubit, followed by a series of CNOT gates that entangle the remaining qubits in a linear sequence, resulting in linear circuit depth. Modeling such a circuit at the gate level requires a certain level of expertise and can be time-consuming. To support alternative modeling approaches that require less expertise, the *Prepare State* construct is introduced. As illustrated in Figure 7.2, this construct enables users to select the GHZ state and specify the desired number of qubits (e.g., 7), without the need for manual circuit construction.

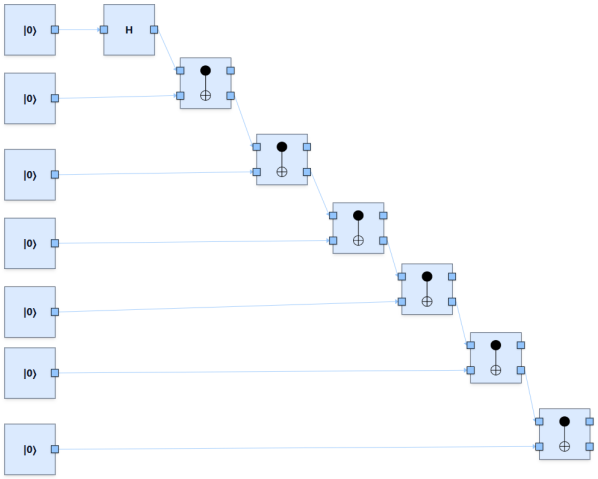

Figure 7.1: GHZ State for 7 Qubits

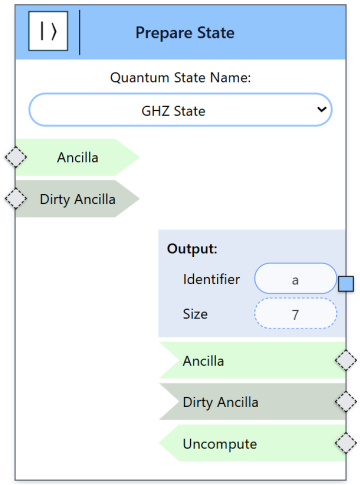

Figure 7.2: GHZ State Preparation through the Prepare State Construct

Circuit-level and Register-based Modeling

In order to support use cases that combine both high-level modeling constructs and low-level modeling blocks, we employ the Splitter and Merger components. These enable a transition between abstract register-based operations and detailed qubit-level operations. One such use case is shown in Figure 7.3, which focuses on modeling the ansatz of the Quantum Approximate Optimization Algorithm (QAOA). The circuit begins with the preparation of a uniform superposition state and proceeds through alternating layers of cost and mixer Hamiltonians, reflecting the core structure of the algorithm. This use case specifically applies QAOA to the Max-Cut problem, a well-known combinatorial optimization task in which the objective is to partition the nodes of a graph to maximize the number of edges between the partitions. QAOA is particularly effective for such NP-hard problems, as it leverages parameterized quantum circuits to approximate optimal solutions [FGRV25]. Here, we consider a graph with three nodes connected by edges between nodes 1 and 2, and 2 and 3. For each edge, we apply an RZZ gate to encode the cost Hamiltonian, followed by an RX gate with a rotation angle of 2β, which implements the mixer Hamiltonian. Since the modeling framework does not natively support the RZZ gate, its standard decomposition

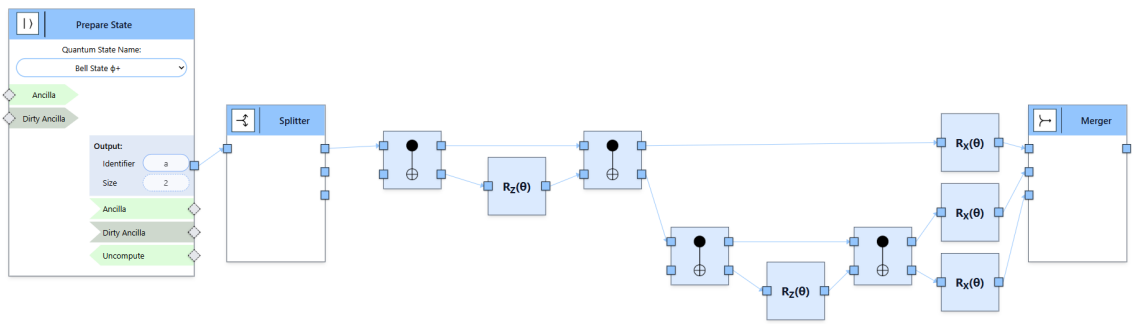

Figure 7.3: QAOA Ansatz using Splitter and Merger

is employed instead, consisting of a sequence of CNOT, RZ, and CNOT operations applied to the corresponding qubit pairs. Alternatively, a custom operator can be used to represent the RZZ gate more compactly within the model.

Broadcasting of Operations

We consider four use cases that demonstrate different approaches to modeling a quantum circuit that prepares and measures a uniform superposition state on two qubits. The first use case, illustrated in Figure 7.4, involves initializing each of the two qubits individually using a Hadamard gate, followed by distinct measurement operations. This results in a circuit with separate nodes for each gate and each qubit.

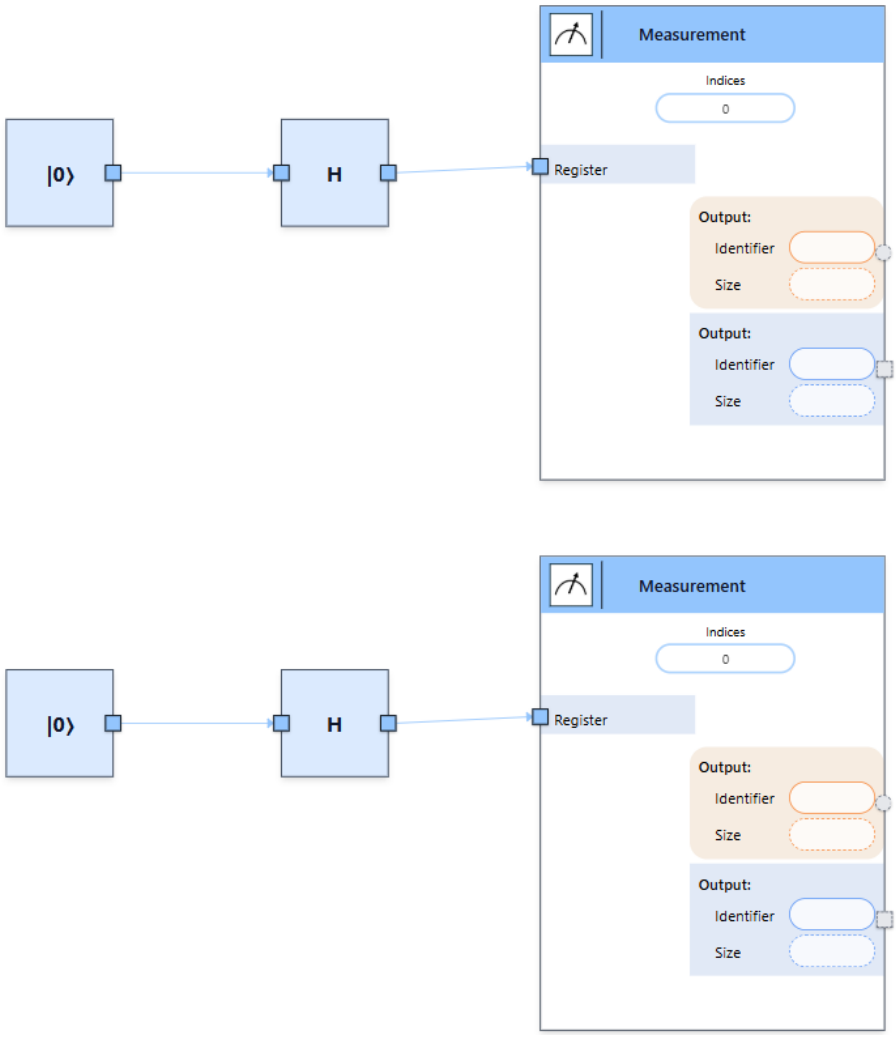

Figure 7.4: Uniform Superposition with 2 Qubits

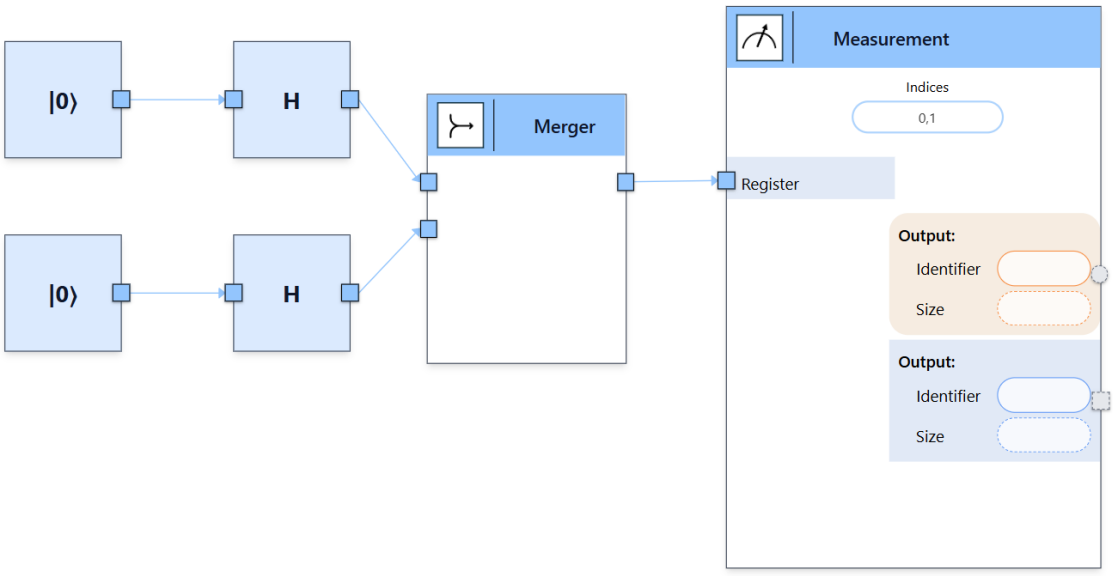

Figure 7.5: Uniform Superposition with 2 Qubits and Merger

In the second use case, depicted in Figure 7.5, we aim to reduce the number of nodes in the circuit by using a merger. After applying the Hadamard gates to each qubit, the two qubits are merged into a single quantum register of size two. A measurement is then performed on the qubits of the register. This approach reduces the number of nodes by one but increases the number of edges by one due to the added merging operation.

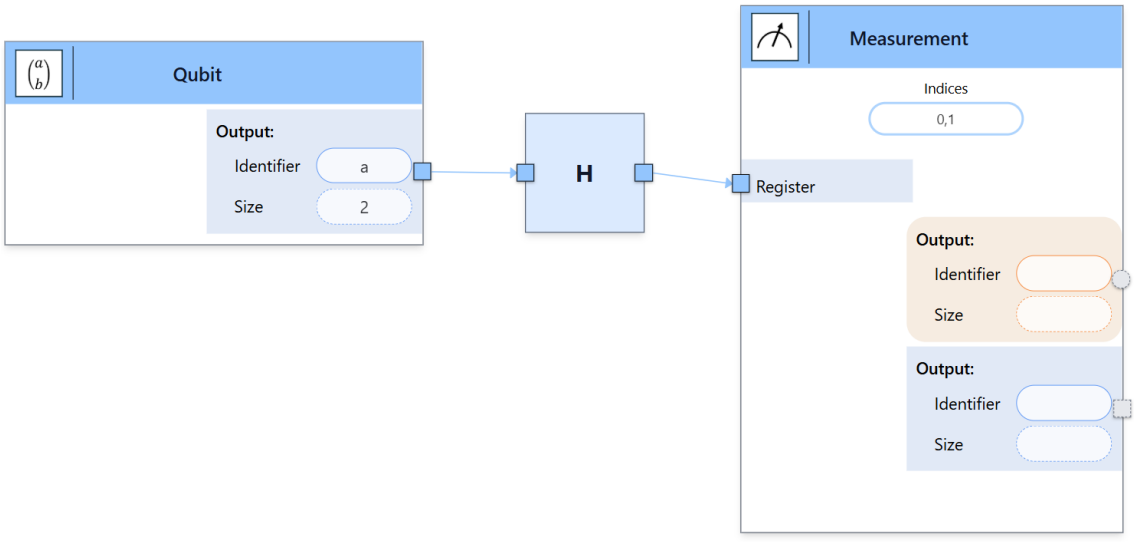

Figure 7.6: Broadcasting of the Hadamard Operation

As shown in Figure 7.6, the third use case models the same behavior using a broadcasting approach, in which a single qubit node of size two represents the entire register. Both the Hadamard and measurement operations are applied to the whole register in a single step, broadcasting the gates across all qubits. This higher-level abstraction simplifies the model.

Finally, the fourth use case, similar to the scenario depicted in Figure 7.2, makes use of the Prepare State block to directly initialize the two-qubit register into a uniform superposition state, followed by a measurement. This approach abstracts away the individual circuit-level blocks entirely, encapsulating the entire state preparation process into a single high-level construct, making it the most concise of the four.

Operators

We consider a set of use cases that demonstrate how classical data can be processed using quantum modeling blocks. In the first use case, depicted in Figure 7.7, we start with three classical values. Each value is encoded in quantum registers using basis encoding. We then perform a quantum addition of the first two values and compare the result to the third value to determine whether the sum is greater. The comparison result is extracted as a classical bit.

The second use case, illustrated in Figure 7.8, involves two classical values on which we apply a bitwise AND operation. Each value is encoded at the basis encoding, and the result of the bitwise logic is computed using quantum gates.

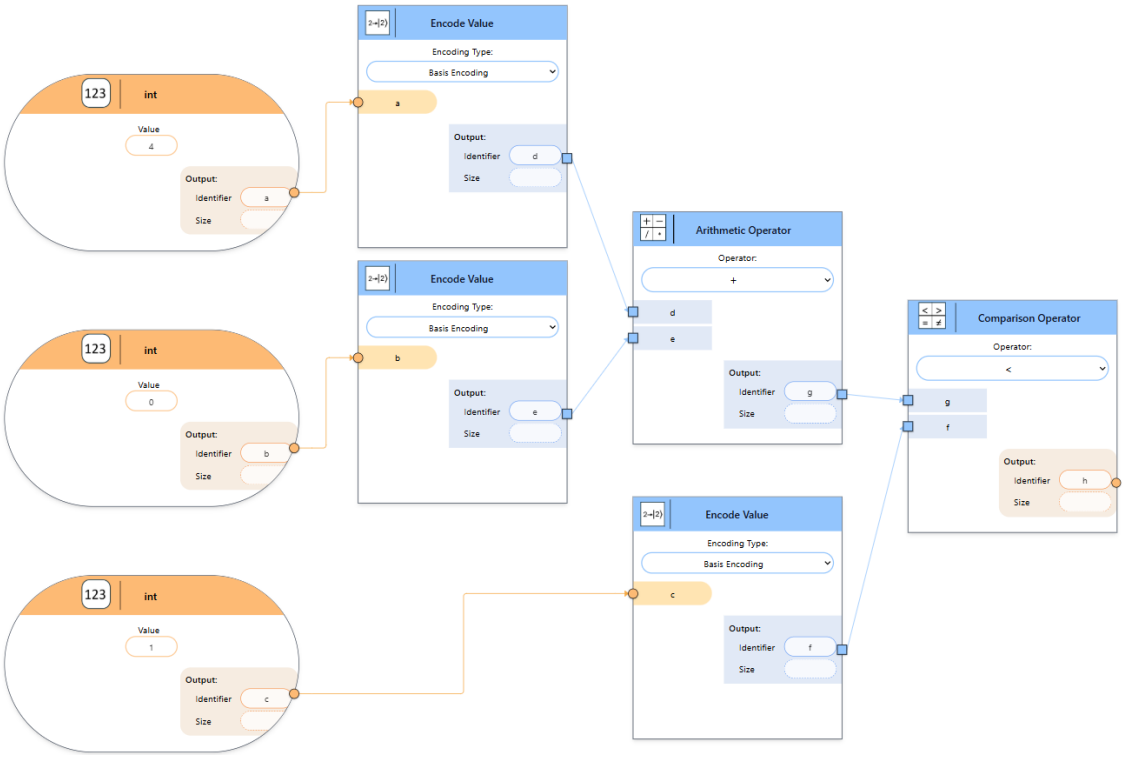

Figure 7.7: Addition and Comparison of Three Values

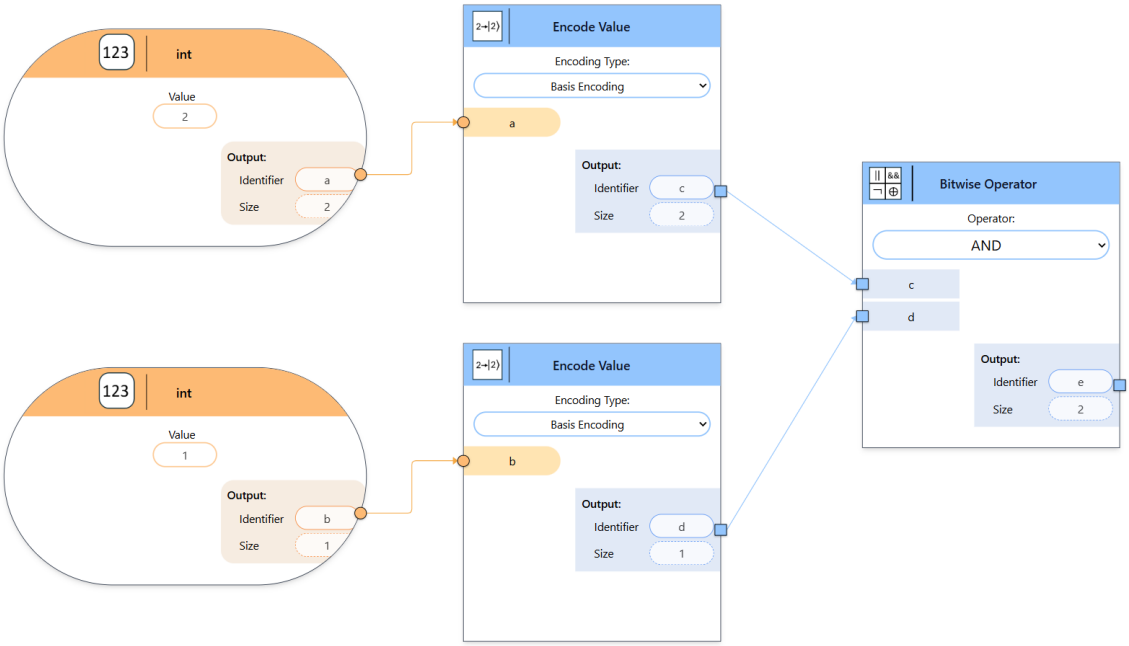

Figure 7.8: Bitwise AND Operation

Figure 7.9 shows the third use case, where an array of classical values is given, and the goal is to determine the minimum value using a min/max operator. The array is amplitude-encoded into a quantum register, and a quantum circuit compares elements pairwise or hierarchically to identify the minimum. The final result is a classical value.

Dynamic Circuits

Current quantum computers are NISQ devices. As a result, circuit depth is a constraint, and techniques that reduce depth are essential for algorithm execution. One such technique is the use of dynamic circuits, which enable mid-circuit measurement, conditional branching, and qubit reset [HJC+22]. These features allow for more efficient circuit constructions. For example, in contrast to the previously discussed modeling of the GHZ state, a dynamic circuit can prepare the

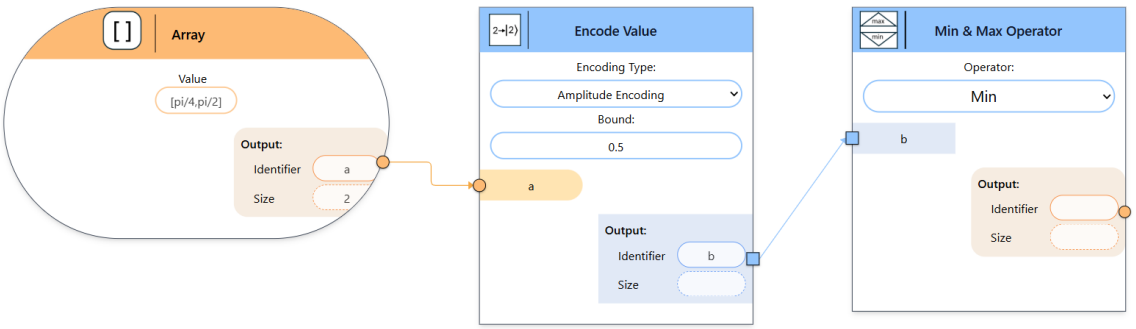

Figure 7.9: Minimum Operator

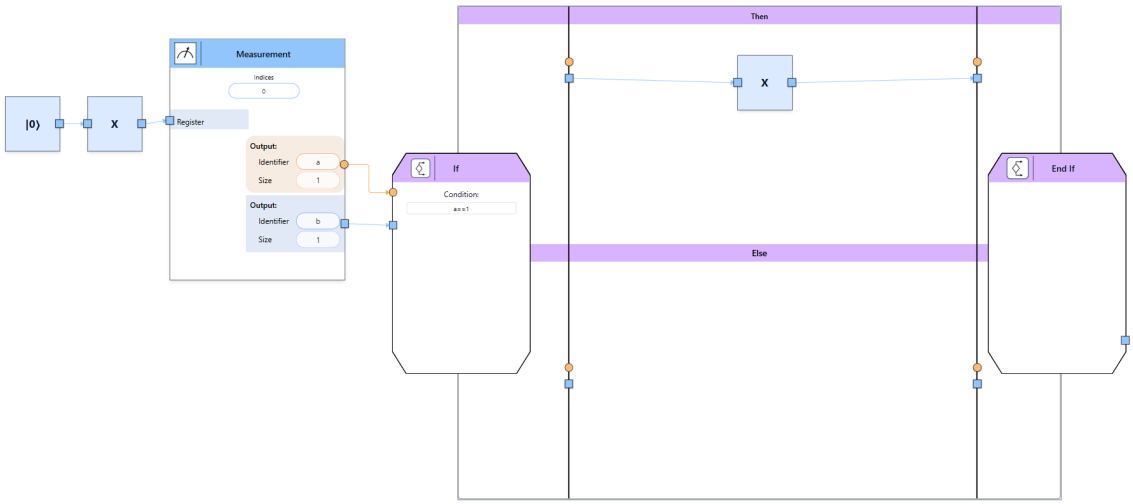

Figure 7.10: Dynamic Circuit for Resetting a Qubit

GHZ state in constant depth by repeatedly measuring and resetting a single qubit, which is then used to entangle the remaining qubits one at a time [HJC+22]. This approach reduces the circuit depth. The qubit reset operation used in this context is illustrated in Figure 7.10. The use case begins by applying a Pauli-X gate to a qubit, followed by a measurement. The measurement result is then used as the condition in an if-statement that initiates the reset procedure. If the outcome is 1, a Pauli-X gate is applied. If the outcome is 0, no operation is performed, which is equivalent to applying the identity gate. This conditional logic ensures that the qubit is deterministically reset to the $|0\rangle$ state, making it available for reuse.

Grouping of Blocks

Grouping is the process of combining multiple operations into a single composite unit. This is depicted in Figure 7.11, where a dynamic circuit featuring one classical and one quantum output is encapsulated into a single group. This approach simplifies complex quantum circuits by treating several gates or nodes that perform a specific task as a single logical construct. Grouping improves modularity and readability, making large models easier to understand and manage by breaking them down into smaller, meaningful components. This concept is analogous to subroutines or gate definitions in OpenQASM, where a sequence of operations is encapsulated as a named gate. Similar to how OpenQASM subroutines organize code by encapsulating complex operations into single gates, grouping operations within quantum circuits allows multiple gates to be treated as a single unit. This improves clarity, maintainability, and efficiency.

Grover Algorithm

The next use case demonstrates the modeling of Grover's Algorithm, which is designed to search an unsorted database or solve black-box problems with a quadratic speedup over classical counterparts [Gro96]. The quantum circuit consists of several components: initialization into a

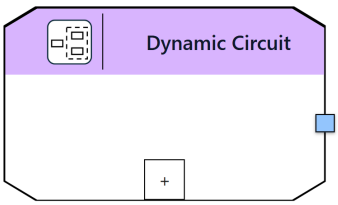

Figure 7.11: Grouped Dynamic Circuit

uniform superposition, the application of a Grover oracle that marks the target state, and a diffusion operator that amplifies its probability amplitude. In our scenario, we have $N = 64$ elements (corresponding to all possible six-bit binary numbers), and our goal is to find the specific value `101010`, as depicted in Figure 7.12. We begin by applying Hadamard gates to all qubits, creating a uniform superposition over all $2^6 = 64$ basis states. This prepares the input state for Grover's search algorithm. Next, we use a `repeat` block to perform the Grover iteration multiple times. We choose six repetitions because the number of required Grover iterations is in $O(\sqrt{N})$ [Gro96]. After the final iteration, measurement collapses the state, and with high probability, the output will be the desired target state $|101010\rangle$.

Deutsch-Jozsa and Shor Algorithm

These use cases are part of a broader class of well-known quantum algorithms that can be modeled using similar constructs. These include the Deutsch–Jozsa algorithm, which determines whether a given function is constant or balanced using only a single query, as demonstrated in the Figure 7.13 with a 3-qubit input register and one ancilla qubit. In this case, the constant oracle is applied, and the expected measurement outcome on the input qubits is 000, indicating a constant function. In Shor's algorithm, illustrated in Figure 7.14, let n be the integer to be factored. One chooses a number m, which is the smallest integer such that 2^m is at least n^2. The function used in Shor's algorithm computes t to the power x modulo n, where t is an integer coprime to n. The quantum circuit uses two registers: the first with m qubits to represent the values of x, and the second to store the results of the function by using ancillae. By applying a quantum Fourier transform followed by a measurement on the first register.

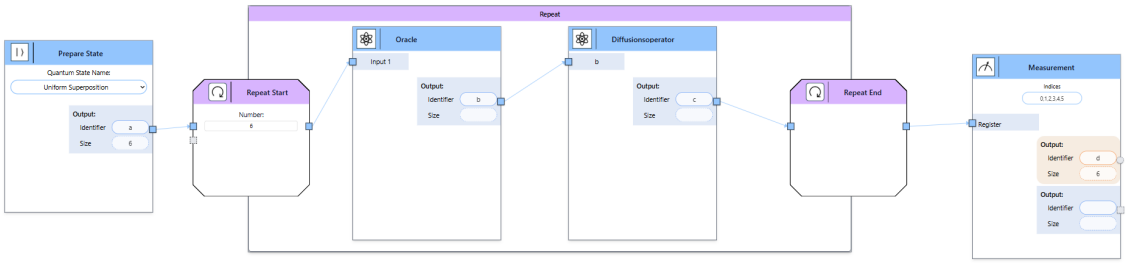

Figure 7.12: Grover's Algorithm

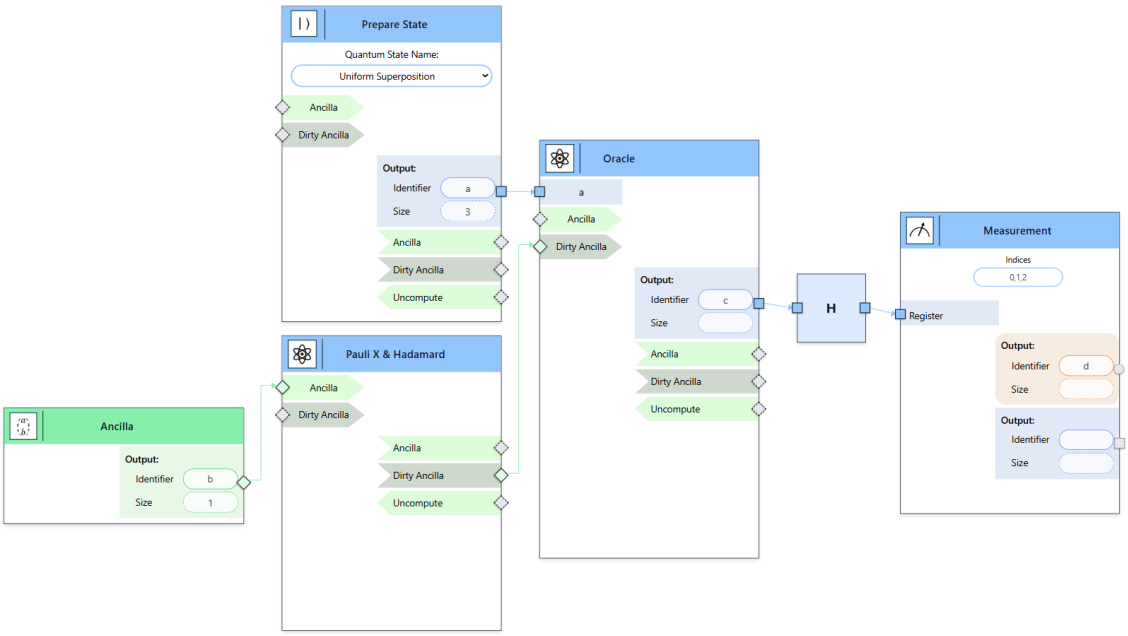

Figure 7.13: Deutsch-Jozsa Algorithm

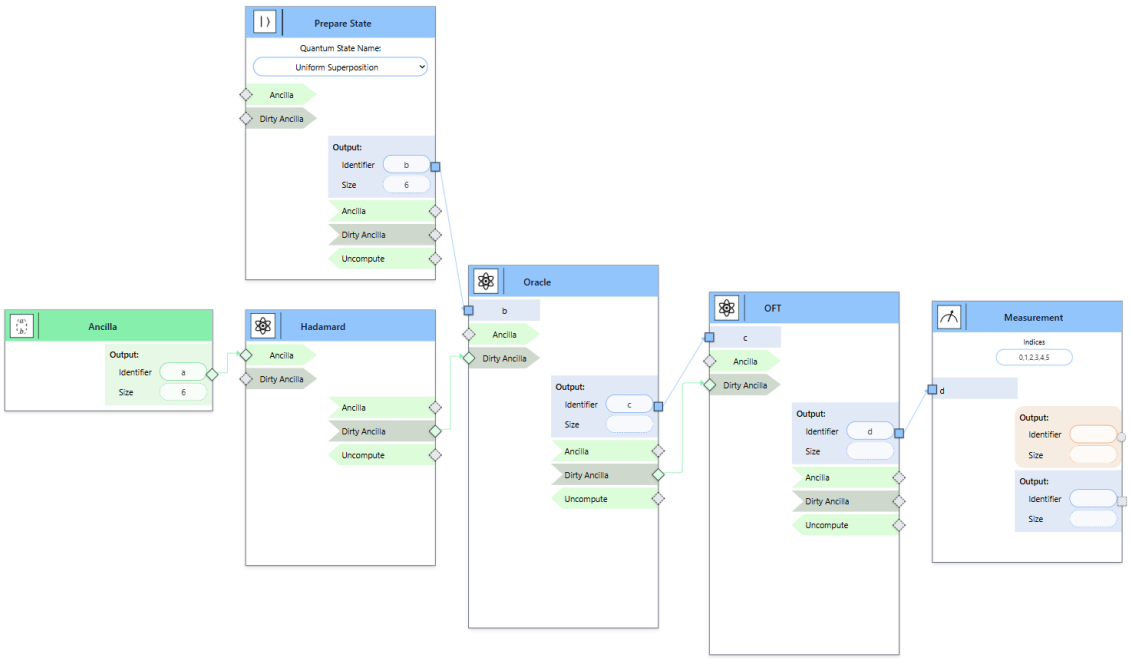

Figure 7.14: Quantum Part of Shor's Algorithm

7.2 Verification of the Requirements

This section presents the validation of the requirements. Table 7.1 illustrates the evaluation of each requirement with respect to its feasibility. In the first column, the requirements from Section 4.2 are listed along with a brief description. The second column displays the outcome of the validation process. A check mark (✓) indicates that the respective requirement has been fulfilled, whereas the absence of a check mark signifies that it could not be met.

Each requirement listed in Table 7.1 has been fulfilled based on the features and capabilities implemented in the modeling tool. Compliance with quantum rules is ensured by enforcing the principles of quantum mechanics within the modeling constructs and blocks, and by preventing invalid connections (R1). The visual distinction of classical and quantum parts is addressed by using different colors and shapes to clearly differentiate between them, improving user clarity (R2). Grouping support is met by enabling users to combine multiple nodes into composite units, which enhances modularity and reusability, much like subroutines in OpenQASM 3 (R3). The system allows annotating the output size of classical and quantum constructs, enabling appropriate selection of implementation through the low-code backend enricher (R4).

Requirement	Fulfilled
Modeling Blocks	
R1(Compliance with Quantum Rules)	✓
R2 (Visual Distinction of classical and quantum parts)	✓
R3 (Grouping Support)	✓
R4 (Output Size Specification)	✓
Modeling Tool	
R5 (Support for Modeling Quantum Algorithms through Modeling Constructs and Blocks)	✓
R6 (Diagram Editing Capabilities)	✓
R7 (Support for Pan and Zoom Functionality)	✓
R8 (Support for Undo and Redo Functionality)	✓
R9 (Editing of Existing Diagrams)	✓
R10 (Real-time Syntactic and Semantic Validation)	✓
R11 (Standardized Model Serialization)	✓
R12 (Integration with Existing Infrastructure)	✓
R13 (Quantum Resource Optimization Support)	✓
R14 (Open-Source)	✓

Table 7.1: Assessment of the Requirements of the Work for their Fulfilment

Requirements R5 through R12 relate to the modeling tool's functionality and user interaction. Support for quantum algorithm modeling is fulfilled by providing modeling blocks that represent gates, along with modeling constructs such as state preparation and other essential model constructs mentioned in Section 5.2 (R5). To build these algorithms, the tool allows users to drag and drop modeling constructs and blocks, connect them according to quantum rules, and, for experienced users, employ keyboard shortcuts for increased efficiency (R6). R7 and R8 are satisfied through interactive navigation and diagram editing capabilities, allowing users to intuitively build and modify diagrams through shortcuts. Furthermore, opening and editing of existing diagrams is enabled (R9). Real-time syntactic and semantic validation has been implemented to ensure immediate feedback on correctness and prevent errors early in the modeling process, for example through input validation mechanisms (R10). The system supports standardized model serialization, facilitating interoperability and persistence of models (R11). Integration with existing infrastructure is enabled through usage with tools such as the NISQ-Analyzer and Qunicorn (R12). Quantum resource optimization support is provided by the specification of the width and depth for the model (R13). Finally, extensibility for custom components, allows users to define new gates and constructs, adapting the tool to evolving quantum algorithms and user needs (R14).

7.3 Limitations

Models that contain a cycle cannot be transformed into executable code. This restriction arises because the low-code backend requires an acyclic graph structure for its enrichment and processing steps. Additionally, while the modeling tool supports classical variables, different quantum computing platforms handle classical variables in varying ways. For example, Amazon Braket does not wait for the quantum output to evaluate the classical variables during execution, which can limit certain classical-quantum interactions. Furthermore, not all target devices fully support all data types or operations specified in OpenQASM 3. For instance, some devices do not support the *complex* data type or mid-circuit measurements, restricting the range of algorithms that can be implemented directly. Our current modeling framework offers limited capabilities for representing the classical runtime environment that quantum algorithms require. Addressing these gaps remains an important direction for future work.

8 Related Work

MDE models are used to describe complex systems, often utilizing modeling languages such as the Unified Modeling Language (UML) to depict behavior and architecture. In the context of quantum computing, Perez-Castillo et al. [PJP21] present an approach to modeling quantum circuits using UML, leveraging activity diagrams to represent quantum operations and interactions. In this approach, each qubit is represented as a pool within an activity diagram. The interactions between qubits are modeled using various UML actions: standard actions, send signal actions, and accept event actions. Single qubit operations are depicted as actions stereotyped with «QuantumGate». For multiple qubit gates, the control bit is represented using a send signal action, while the target qubits are represented using accept event actions. This modeling technique captures the dynamics of quantum circuits, providing a visual and structured representation of quantum algorithms.

Furthermore, UML activity diagrams can be used with class diagrams to generate executable code for hybrid classical-quantum programs [PJCP23]. Class diagrams model the classical components of the system and are translated into Python code. Meanwhile, the activity diagrams representing quantum circuits are transformed into Qiskit code. However, this method has its limitations. Creating and interpreting these UML diagrams requires significant technical expertise in software engineering and quantum computing. Additionally, the code generation process is constrained by the capabilities of the modeling tools and the precision of the diagrams. These limitations highlight the necessity for more user-friendly development tools, such as low-code platforms.

Ebert [Ebe24] conducted a systematic literature review to identify research gaps in the development of low-code platforms for quantum algorithms. The study emphasized the need for platforms that lower entry barriers, particularly for users without deep quantum expertise, and highlighted the importance of open-source solutions. Several tools and frameworks have been developed to support quantum programming and modeling, each with different goals and levels of abstraction, as discussed in Chapter 3. Tools such as IBM Composer and Quantum Programming Studio focus primarily on modeling quantum circuits. These platforms are tightly coupled to the gate-level representation and do not provide higher levels of abstraction. As a result, they require users to have a deep knowledge of quantum computing concepts.

In contrast, frameworks like Qrisp [SBZ+24] and Silq [BBGV20] aim to simplify quantum programming by introducing high-level type systems. While these approaches improve accessibility by abstracting away from low-level circuit design, they do not support graphical modeling. Their modularization strategies operate at a higher conceptual level, which makes them more approachable for users without a deep background in quantum computing. However, they are less accessible since operating those tools also requires software programming skills.

Beyond circuit-level and code-based tools, other approaches focus on automating the construction and orchestration of quantum applications. Vietz et al. [VBB+25] propose an approach in which users provide a problem description, and relevant quantum computing patterns are automatically identified. Based on the user's non-functional requirements, suitable implementations are selected

and then aggregated into an executable quantum application. In contrast, in the approach by Beisel et al. [BBL+25], the user actively selects the patterns. Their method then automates the generation and adaptation of quantum workflows based on these selected patterns. Since patterns represent abstract solutions to problems, they introduce an additional level of complexity for users.

To bridge this gap, concrete solutions are employed [FBB+14], which are organized within solution languages to enable easier navigation [FL17]. Aldekal [Ald24] propose a method to aggregate these concrete solutions into a complete quantum application. However, this approach is currently limited to OpenQASM 2 and is different from low-code platforms, as it focuses on domain-specific quantum solution aggregation rather than providing a quantum low-code environment.

9 Conclusion and Outlook

Building quantum algorithms manually is complex, time-consuming, and error-prone due to the specialized knowledge required and the difficulty of designing quantum algorithms. To address these challenges, low-code platforms have emerged to make quantum development more accessible by minimizing the need for deep expertise and enabling code generation through the drag-and-drop of modeling components within a visual interface. However, although quantum low-code platforms like Classiq and QuantuMoonLight provide such abstraction, they are either closed-source, leading to vendor lock-in, or are no longer actively maintained.

This thesis introduces an open-source low-code modeling tool designed for quantum algorithm development. The modeling tool and its modeling components must adhere to both established design principles and fundamental quantum computing principles. First, relevant modeling components were identified through an analysis of established quantum programming standards, such as OpenQASM, the capabilities of quantum SDKs like Qiskit, and a review of existing low-code platforms, including Classiq and QuantuMoonLight. To support different user groups, the modeling components must include both high-level abstraction constructs and lower-level blocks. Thus, a set of modeling constructs consisting of data types, higher-level operations, and control structure nodes, was defined. Data type constructs are needed to represent information. Operations consist of high-level abstractions of quantum parts, such as encoding values and preparing quantum states, as well as operators and measurements. Furthermore, control structure nodes enable the repetition of constructs and blocks, as well as conditional flow. A set of low-level blocks was incorporated to allow modeling quantum gates and specify the algorithm in detail. The Splitter and Merger allow switching between abstraction layers and between high-level constructs and the qubit level. All modeling elements are accessible through a graphical interface where users can construct quantum algorithms by placing and connecting components interactively. Once a model is created, it is sent to the low-code backend to transform it into an executable OpenQASM representation that enables its execution on Qunicorn.

The quantum low-code modeling tool has been validated through application scenarios, demonstrating the use of each modeling construct and block. These scenarios showcase well-known quantum algorithms such as QAOA, Shor's algorithm, and the Deutsch–Jozsa algorithm. They also demonstrate how quantum states, such as GHZ states, can be defined either through individual gate operations or via high-level constructs like Prepare State.

In summary, this thesis introduces a flexible and accessible modeling tool for quantum algorithm development. It bridges the gap between conceptual design and detailed algorithm modeling, supporting both beginners and experts, while providing a structured and extensible tool for the evolving field of quantum software engineering.

Future developments should focus on enhancing collaboration features, enabling multiple users to work together seamlessly on quantum algorithm design, while leveraging AI-based tools to assist users in generating quantum algorithms from natural language descriptions. This approach will

simplify and accelerate the development process, making quantum programming more accessible to a broader audience. At the same time, addressing current limitations within the modeling framework remains a priority. For instance, features such as copying and reusing modeling elements may currently be unavailable and should be enhanced to streamline the modeling experience. Additionally, while classical variables are supported, their behavior can vary significantly across quantum computing platforms. Future work may explore integration with alternative intermediate representations such as the QIR, offering enhanced flexibility and broader hardware compatibility.

Bibliography

[ACC+23] F. Amato, M. Cicalese, L. Contrasto, G. Cubicciotti, G. D'Ambola, A. La Marca, G. Pagano, F. Tomeo, G. A. Robertazzi, G. Vassallo, G. Acampora, A. Vitiello, G. Catolino, G. Giordano, S. Lambiase, V. Pontillo, G. Sellitto, F. Ferrucci, F. Palomba. "QuantuMoonLight: A low-code platform to experiment with quantum machine learning". In: *SoftwareX* 22 (2023), p. 101399. ISSN: 2352-7110. DOI: 10.1016/j.softx.2023.101399 (cit. on pp. 27, 30).

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. "A Pattern Language: Towns, Buildings, Construction". In: *Oxford University Press* (1977) (cit. on p. 24).

[Ald24] A. E. Aldekal. "Entwicklung eines Frameworks für die Nutzung von konkreten Lösungen der Quantencomputing Mustersprache". Bachelor's Thesis. University of Stuttgart, 2024. DOI: 10.18419/opus-14030 (cit. on p. 66).

[App25] Apple Inc. *Apple Human Interface Guidelines*. 2025. URL: https://developer.apple.com/design/human-interface-guidelines (cit. on p. 18).

[AWS24] AWS. *Amazon Braket Usage Overview*. 2024. URL: https://docs.aws.amazon.com/braket (cit. on pp. 15, 31).

[Bar22] J. Barzen. "From Digital Humanities to Quantum Humanities: Potentials and Applications". In: *Quantum Computing in the Arts and Humanities: An Introduction to Core Concepts, Theory and Applications*. Ed. by E. R. Miranda. Springer International Publishing, 2022, pp. 1–52. DOI: 10.1007/978-3-030-95538-0_1 (cit. on p. 15).

[BBB+23] F. Bühler, J. Barzen, M. Beisel, D. Georg, F. Leymann, K. Wild. "Patterns for Quantum Software Development". In: *Proceedings of the 15th International Conference on Pervasive Patterns and Applications (PATTERNS 2023)*. Xpert Publishing Services (XPS), 2023, pp. 30–39. ISBN: 978-1-68558-049-0 (cit. on pp. 25, 36).

[BBC+95] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, H. Weinfurter. "Elementary gates for quantum computation". In: *Physical review A* 52.5 (1995), p. 3457. DOI: 10.1103/PhysRevA.52.3457 (cit. on p. 22).

[BBGV20] B. Bichsel, M. Baader, T. Gehr, M. Vechev. "Silq: a high-level quantum language with safe uncomputation and intuitive semantics". In: *Proceedings of the* 41^{st} *ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 286–300. ISBN: 9781450376136. DOI: 10.1145/3385412.3386007 (cit. on p. 65).

[BBL+25] M. Beisel, J. Barzen, F. Leymann, L. Stiliadou, D. Vietz, B. Weder. " Pattern-based Generation and Adaptation of Quantum Workflows ". In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2025, pp. 733–733 (cit. on p. 66).

[BFV12] M. Brambilla, P. Fraternali, C. Vaca. "BPMN and design patterns for engineering social BPM solutions". In: *Business Process Management Workshops: BPM 2011 International Workshops, Clermont-Ferrand, France, 2011, Revised Selected Papers, Part I 9*. Springer. 2012, pp. 219–230. DOI: 10.1007/978-3-642-28108-2_22 (cit. on p. 17).

[BNE25] A. Bogazzi, S. Nencel, S. Eberhardt. *Fabric.js: JavaScript Canvas Library*. 2025. URL: https://fabricjs.com (cit. on p. 45).

[bpm25] bpmn-io. *diagram-js: JavaScript Library for Diagram Display and Modification*. 2025. URL: https://github.com/bpmn-io/diagram-js (cit. on p. 45).

[CAB+21] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, et al. "Variational quantum algorithms". In: *Nature Reviews Physics* 3.9 (2021), pp. 625–644. DOI: 10.1038/s42254-021-00348-9 (cit. on p. 23).

[CJA+22] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, B. R. Johnson. "OpenQASM 3: A Broader and Deeper Quantum Assembly Language". In: *ACM Transactions on Quantum Computing* 3.3 (2022), pp. 1–50. ISSN: 2643-6817. DOI: 10.1145/3505636 (cit. on pp. 23, 31, 40).

[Cli10] Client IO. *JointJS: JavaScript Diagramming Library*. 2010. URL: https://www.jointjs.com (cit. on p. 45).

[CRA18] Y. Cao, J. Romero, A. Aspuru-Guzik. "Potential of quantum computing for drug discovery". In: *IBM Journal of Research and Development* 62.6 (2018), 6:1–6:20. DOI: 10.1147/jrd.2018.2888987 (cit. on p. 15).

[Din25] Dinamenta UAB. *DHTMLX Diagram: JavaScript Diagramming Library*. 2025. URL: https://dhtmlx.com (cit. on p. 45).

[Ebe24] T. Ebert. "Konzeption und Entwicklung eines Low-Code-Frameworks für Quantencomputing-Anwendungen". Bachelor's Thesis. University of Stuttgart, 2024. DOI: 10.18419/opus-14994 (cit. on pp. 15, 27, 65).

[Far13] N. Fargo. *State.js: JavaScript Library for First-Class States*. 2013. URL: https://statejs.org (cit. on p. 45).

[FBB+14] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. "From Pattern Languages to Solution Implementations". In: *Proceedings of the 6th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2014, pp. 12–21 (cit. on p. 66).

[FGRV25] E. Farhi, S. Gutmann, D. Ranard, B. Villalonga. *Lower bounding the MaxCut of high girth 3-regular graphs using the QAOA*. 2025. arXiv: 2503.12789 [quant-ph] (cit. on p. 54).

[FL17] M. Falkenthal, F. Leymann. "Easing Pattern Application by Means of Solution Languages". In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 58–64 (cit. on p. 66).

[FLF+25] M. Franz, C. Lopes, D. Fong, M. Kucera, G. Bader, et al. *Cytoscape.js: A Graph Theory Library for Visualization and Analysis*. 2025. URL: https://js.cytoscape.org (cit. on p. 45).

[Fra14] N. Fraser. *Google blockly-a visual programming editor*. 2014. URL: http://developers.google.com/blockly (cit. on pp. 15, 17, 27).

[GBB+23] D. Georg, J. Barzen, M. Beisel, F. Leymann, J. Obst, D. Vietz, B. Weder, V. Yussupov. "Execution Patterns for Quantum Applications". In: *Proceedings of the 18th International Conference on Software Technologies - ICSOFT*. SciTePress, 2023, pp. 258–268. ISBN: 978-989-758-665-1. DOI: 10.5220/0012057700003538 (cit. on p. 15).

[GNO25] GNOME Project. *GNOME Human Interface Guidelines*. 2025. URL: https://developer.gnome.org/hig (cit. on p. 18).

[GP96] T. Green, M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages & Computing* 7.2 (1996), pp. 131–174. ISSN: 1045-926X. DOI: 10.1006/jvlc.1996.0009 (cit. on p. 19).

[Gro12] N. N. Group. *Usability 101: Introduction to Usability*. 2012. URL: https://www.nngroup.com/articles/usability-101-introduction-to-usability (cit. on p. 18).

[Gro96] L. K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the 28th annual ACM symposium on Theory of computing*. 1996, pp. 212–219. DOI: 10.1145/237814.237866 (cit. on pp. 20, 31, 59, 60).

[Gub15] S. Guberman. "On Gestalt theory principles". In: *Gestalt Theory* 37.1 (2015), pp. 25–44 (cit. on p. 18).

[HJC+22] F. Hua, Y. Jin, Y. Chen, S. Vittal, K. Krsulich, L. S. Bishop, J. Lapeyre, A. Javadi-Abhari, E. Z. Zhang. "Exploiting qubit reuse through mid-circuit measurement and reset". In: *arXiv preprint arXiv:2211.01925* (2022). DOI: 10.48550/arXiv.2211.01925 (cit. on pp. 58, 59).

[HTC18] C. Havenstein, D. Thomas, S. Chandrasekaran. "Comparisons of performance between quantum and classical machine learning". In: *SMU Data Science Review* 1.4 (2018), p. 11 (cit. on p. 15).

[IBM24a] IBM. *IBM Quantum*. 2024. URL: https://quantum-computing.ibm.com (cit. on pp. 15, 31).

[IBM24b] IBM. *IBM Quantum Composer*. 2024. URL: https://quantum.ibm.com/composer (cit. on p. 15).

[IHW+21] C. K. Iyer, F. Hou, H. Wang, Y. Wang, K. Oh, S. Ganguli, V. Pandey. "Trinity: A No-Code AI platform for complex spatial datasets". In: *Proceedings of the 4th ACM SIGSPATIAL International Workshop on AI for Geographic Knowledge Discovery*. 2021, pp. 33–42. DOI: 10.1145/3486635.3491072 (cit. on p. 27).

[JGr20] JGraph Ltd. *mxGraph: JavaScript Diagramming Library*. 2020. URL: https://jgraph.github.io/mxgraph (cit. on p. 46).

[JXWL22] W. Junfeng, Z. Xu, X. Wang, J. Lu. "A Comparative Research on Usability and User Experience of User Interface Design Software". In: *International Journal of Advanced Computer Science and Applications* 13 (2022). DOI: 10.14569/ijacsa.2022.0130804 (cit. on p. 18).

[KR03] V. Kulkarni, S. Reddy. "Separation of concerns in model-driven development". In: *IEEE Software* 20.5 (2003), pp. 64–69. DOI: 10.1109/ms.2003.1231154 (cit. on p. 17).

[KSV02] A. Kitaev, A. Shen, M. Vyalyi. "Correspondence between classical and quantum computation". In: *Classical and Quantum Computation*. American Mathematical Society, 2002, pp. 60–65. DOI: 10.1090/gsm/047/10 (cit. on p. 22).

[LB20] F. Leymann, J. Barzen. "The bitter truth about gate-based quantum algorithms in the NISQ era". In: *Quantum Science and Technology* 5.4 (2020), pp. 1–28. DOI: 10.1088/2058-9565/abae7d (cit. on pp. 23, 24).

[Ley19] F. Leymann. "Towards a Pattern Language for Quantum Algorithms". In: *Proceedings of the 1st International Workshop on Quantum Technology and Optimization Problems (QTOP)*. Springer, 2019. DOI: 10.1007/978-3-030-14082-3_19 (cit. on pp. 20, 24).

[LW22] L. Li, Z. Wu. "How Can No/Low Code Platforms Help End-Users Develop ML Applications? - A Systematic Review". In: *HCI International 2022 – Late Breaking Papers: Interacting with eXtended Reality and Artificial Intelligence*. Springer Nature Switzerland, 2022, pp. 338–356. ISBN: 9783031217074. DOI: 10.1007/978-3-031-21707-4_25 (cit. on p. 27).

[McK18] J. McKinley. *A Guide to Color*. College of Agricultural, Consumer and Environmental Sciences, 2018 (cit. on p. 19).

[Mik25] Mike Bostock and Observable, Inc. *D3.js – Data-Driven Documents*. 2025. URL: https://d3js.org (cit. on p. 45).

[Min22] N. Minerbi. "Quantum Software Development with Classiq". In: *Quantum Software Engineering*. Springer International Publishing, 2022, pp. 269–280. ISBN: 9783031053245. DOI: 10.1007/978-3-031-05324-5_14 (cit. on pp. 15, 27, 30).

[Min25] MindFusion Ltd. *MindFusion Diagramming for JavaScript*. 2025. URL: https://mindfusion.eu (cit. on p. 45).

[Moo09] D. Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 756–779. ISSN: 0098-5589. DOI: 10.1109/tse.2009.67 (cit. on pp. 15, 18, 19, 31).

[NC12] M. A. Nielsen, I. L. Chuang. "Quantum Computation and Quantum Information: 10th Anniversary Edition". In: Cambridge University Press, 2012. ISBN: 9780511976667. DOI: 10.1017/cbo9780511976667 (cit. on pp. 15, 20, 21, 24).

[Nor98] Northwoods Software. *Introduction to GoJS Diagramming Components*. 1998. URL: https://gojs.net (cit. on p. 45).

[NWA21] P. Nimbe, B. A. Weyori, A. F. Adekoya. "Models in quantum computing: a systematic review". In: *Quantum Information Processing* 20.2 (2021). ISSN: 1573-1332. DOI: 10.1007/s11128-021-03021-3 (cit. on p. 19).

[PJCP23] R. Pérez-Castillo, L. Jiménez-Navajas, I. Cantalejo, M. Piattini. "Generation of Classical-Quantum Code from UML models". In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 2023, pp. 165–168. DOI: 10.1109/qce57702.2023.10202 (cit. on p. 65).

[PJP21] R. Perez-Castillo, L. Jimenez-Navajas, M. Piattini. "Modelling Quantum Circuits with UML". In: *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. IEEE, 2021, pp. 7–12. DOI: 10.1109/q-se52541.2021.00009 (cit. on p. 65).

[PKG24] S. M.-L. Pfaendler, K. Konson, F. Greinert. "Advancements in Quantum Computing—Viewpoint: Building Adoption and Competency in Industry". In: *Datenbank-Spektrum* 24.1 (2024), pp. 5–20. ISSN: 1610-1995. DOI: 10.1007/s13222-024-00467-4 (cit. on pp. 19, 22).

[Pre18] J. Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79 (cit. on pp. 15, 23).

[Qua19] Quantastica. *Quantum Programming Studio*. 2019. URL: https://quantum-circuit.com (cit. on p. 15).

[RMM+09] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, Y. Kafai. "Scratch: programming for all". In: *Communications of the ACM* 52.11 (2009), pp. 60–67. ISSN: 1557-7317. DOI: 10.1145/1592761.1592779 (cit. on pp. 15, 17, 27).

[RYH+23] T. Rindell, B. Yenilen, N. Halonen, A. Pönni, I. Tittonen, M. Raasakka. "Exploring the optimality of approximate state preparation quantum circuits with a genetic algorithm". In: *Physics Letters A* 475 (2023), p. 128860. ISSN: 0375-9601. DOI: 10.1016/j.physleta.2023.128860 (cit. on p. 41).

[SBB+20] M. Salm, J. Barzen, U. Breitenbücher, F. Leymann, B. Weder, K. Wild. "The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms". In: *Service-Oriented Computing*. Springer International Publishing, 2020, pp. 66–85. ISBN: 9783030648466. DOI: 10.1007/978-3-030-64846-6_5 (cit. on p. 36).

[SBZ+24] R. Seidel, S. Bock, R. Zander, M. Petrič, N. Steinmann, N. Tcholtchev, M. Hauswirth. "Qrisp: A Framework for Compilable High-Level Programming of Gate-Based Quantum Computers". In: *PlanQC-Programming Languages for Quan tum Computing* (2024). DOI: 10.48550/arXiv.2406.14792 (cit. on p. 65).

[Sch06] D. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2 (2006), pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/mc.2006.58 (cit. on p. 17).

[Shn87] B. Shneiderman. "Designing the user interface strategies for effective human-computer interaction". In: *ACM SIGBIO Newsletter* 9.1 (1987), p. 6. ISSN: 0163-5697. DOI: 10.1145/25065.950626 (cit. on p. 18).

[Shn96] B. Shneiderman. "The eyes have it: a task by data type taxonomy for information visualizations". In: *Proceedings 1996 IEEE Symposium on Visual Languages*. VL-96. IEEE Comput. Soc. Press, 1996. DOI: 10.1109/vl.1996.545307 (cit. on p. 18).

[Sho94] P. W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE. 1994, pp. 124–134. DOI: 10.1109/sfcs.1994.365700 (cit. on pp. 20, 23).

[Sti25] S.-N. Stiliadou. *Quantum Low-Code Modeler*. 2025. URL: https://github.com/LEQO-Framework/low-code-modeler (cit. on p. 53).

[Sve25] K. Sveidqvist. *Mermaid.js: JavaScript Diagramming and Charting Tool*. 2025. URL: https://mermaid.js.org (cit. on p. 45).

[TKH+24] J. Theam Lim, C. Kang Lee, E. Heng Lim, P. Voon Wong, K. Tak Yew, C. Yee Ooi. "Preliminary Study on the Accessibility and Learning Experience of Low-Code Development Platforms". In: *2024 5th International Conference on Artificial Intelligence and Data Sciences (AiDAS)*. IEEE, 2024, pp. 486–491. DOI: 10.1109/aidas63860.2024.10730306 (cit. on p. 27).

[VBB+25] D. Vietz, J. Barzen, M. Beisel, F. Leymann, L. Stiliadou, B. Weder. "Harnessing Patterns to Support the Development of Hybrid Quantum Applications". In: *Proceedings of the 14th International Conference on Software and Computer Applications (ICSCA)*. ACM, 2025, pp. 1–11 (cit. on p. 65).

[VL19] A. de Vries, J. Luraschi. *nomnoml: Sassy "UML" Diagrams*. 2019. DOI: 10.32614/cran.package.nomnoml (cit. on p. 45).

[WBB+25] B. Weder, J. Barzen, M. Beisel, F. Bühler, D. Georg, F. Leymann, L. Stiliadou. "Qunicorn: A Middleware for the Unified Execution Across Heterogeneous Quantum Cloud Offerings". In: *2025 IEEE/ACM International Workshop on Quantum Software Engineering (Q-SE)* (2025), pp. 17–24. DOI: 10.1109/q-se66736.2025.00009 (cit. on pp. 33, 36).

[WBL+21] B. Weder, J. Barzen, F. Leymann, M. Salm, K. Wild. "QProv: A provenance system for quantum computing". In: *IET Quantum Communication* 2.4 (2021), pp. 171–181. ISSN: 2632-8925. DOI: 10.1049/qtc2.12012 (cit. on p. 36).

[WBLS21] M. Weigold, J. Barzen, F. Leymann, M. Salm. "Encoding patterns for quantum algorithms". In: *IET Quantum Communication* 2.4 (2021), pp. 141–152. ISSN: 2632-8925. DOI: 10.1049/qtc2.12032 (cit. on p. 24).

[WBLZ21] B. Weder, J. Barzen, F. Leymann, M. Zimmermann. "Hybrid Quantum Applications Need Two Orchestrations in Superposition: A Software Architecture Perspective". In: *2021 IEEE International Conference on Web Services (ICWS)*. IEEE, 2021, pp. 1–13. DOI: 10.1109/icws53863.2021.00015 (cit. on p. 15).

[XB 25] XB Software Ltd. *Webix: JavaScript UI Library*. 2025. URL: https://webix.com (cit. on p. 45).

[xyf25] xyflow. *React Flow: JavaScript Library for Node-Based UIs*. 2025. URL: https://reactflow.dev (cit. on p. 46).

[ZS21] P. Zweihoff, B. Steffen. "Pyrus: An Online Modeling Environment for No-Code Data-Analytics Service Composition". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Springer International Publishing, 2021, pp. 18–40. ISBN: 9783030891596. DOI: 10.1007/978-3-030-89159-6_2 (cit. on p. 27).

All links were last followed on June 20, 2025.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature