A Model-Driven Framework for Composition-Based Quantum Circuit Design

FELIX GEMEINHARDT, Johannes Kepler University Linz, Institute for Business Informatics - Software Engineering, CDL-MINT, Austria

ANTONIO GARMENDIA, Johannes Kepler University Linz, Institute for Business Informatics - Software Engineering, Austria

MANUEL WIMMER, Johannes Kepler University Linz, Institute for Business Informatics - Software Engineering, CDL-MINT, Austria

ROBERT WILLE, Technical University of Munich, Chair for Design Automation, Germany

Quantum programming languages support the design of quantum applications. However, to create such programs, one still needs to understand fundamental characteristics of quantum computing and quantum information theory. Furthermore, quantum algorithms frequently make use of abstract operations with a hidden low-level realization (e.g., Quantum Fourier Transform). Thus, turning from elementary quantum operations to a higher-level view on quantum circuit design not only reduces the complexity, but also lowers the entry barriers for non quantum computing experts.

To this end, this paper proposes a modeling language and design framework for quantum circuits. This allows the definition of composite operators advocating a higher-level quantum algorithm design, together with automated code generation for the circuit execution. The proposed approach comes with a separation of the quantum operation definitions from the quantum circuit syntax, which allows for an independent design and the use of customized libraries. To demonstrate the benefits of the proposed approach, coined Composition-Based Quantum Circuit Designer, we realized the Quantum Counting algorithm as well as the Quantum Approximate Optimization Algorithm with it. This shows that, compared to an existing state-of-the-art editor, the proposed approach allows for the realization of both quantum algorithms on a high-level with a substantially reduced development effort.

CCS Concepts: • Computer systems organization → Quantum computing; • Software and its engineering → Abstraction, modeling and modularity.

Additional Key Words and Phrases: Quantum Computing, Quantum Software Engineering, Quantum Circuits, Model-Driven Engineering, Quantum Software Languages

ACM Reference Format:

Authors’ addresses: Felix Gemeinhardt, felix.gemeinhardt@jku.at, Johannes Kepler University Linz, Institute for Business Informatics - Software Engineering, CDL-MINT, Linz, Austria; Antonio Garmendia, antonio.garmendia@jku.at, Johannes Kepler University Linz, Institute for Business Informatics - Software Engineering, Linz, Austria; Manuel Wimmer, manuel.wimmer@jku.at, Johannes Kepler University Linz, Institute for Business Informatics - Software Engineering, CDL-MINT, Linz, Austria; Robert Wille, robert.wille@tum.de, Technical University of Munich, Chair for Design Automation, Munich, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
Manuscript submitted to ACM
1 INTRODUCTION

Quantum Computing (QC) is an interdisciplinary field which relies on quantum mechanical phenomena to process information. Continuous developments in the field justify to expect near-term superiority compared to classical means of computation at least for certain applications such as simulations in chemistry, optimization problems, or machine learning approaches [9, 20, 36].

Computations performed on a quantum computer are implemented with operations of quantum gates, in analogy to classical gates for conventional computation [13]. Such reversible quantum gates, together with irreversible operations and concurrent classical computation, applied on quantum data (e.g., qubits) in an ordered manner represent a quantum circuit. This so-called quantum circuit model of QC is regarded the most commonly used realistic model to run quantum programs [51].

A universal fault-tolerant quantum computer would require millions of qubits of highest quality [24]. Whereas experimental realizations of such computers will potentially still take decades of research, so-called Noisy Intermediate-Scale Quantum (NISQ) computers already exist today and, therefore, may enable the bespoke near-term superiority of QC with respect to classical computation [58]. Hybrid quantum-classical algorithms, called Variational Quantum Algorithms (VQAs), have been proposed to cope with the limitations given in the NISQ era [9], where the parameters of the quantum circuit are optimized with classical means of computation. Therefore, the resulting two research streams consider quantum algorithms specifically for perfect, or noisy qubits [7].

Nowadays, quantum programming languages, like IBM’s Qiskit\(^1\), Google’s Cirq\(^2\), Microsoft’s Q#\(^3\), or Amazon’s Braket\(^4\) offer the possibility to efficiently program and access quantum computers provided by Cloud services. Furthermore, the programs can be executed on quantum simulators locally or also via Cloud access. The field of Quantum Software Engineering (QSE) is emerging and new tools are published on a regular basis as, e.g., recent pen-based programming solutions [3]. However, code is usually written at the qubit level and requires to understand basic fundamental concepts of quantum physics, like entanglement and superposition. Exceptions are represented by emerging libraries and software development kits (e.g., IBM Qiskit) which offer higher level functionalities.

Such functionalities allow the definition of more abstract quantum operations (e.g., Quantum Fourier Transform (QFT) [51]) which occur frequently in quantum algorithms. One example is the Quantum Phase Estimation (QPE) [51], which is depicted in Figure 1. The illustration highlights the use of higher-level quantum operations and iterative patterns for the definition of quantum algorithms. The QPE-algorithm determines the eigenphase of a given quantum operation (\(U\)-gate). This quantum operation is usually a higher-level, composed gate. A controlled version is iteratively applied a certain number of times (twice for \(U^2\), three times for \(U^3\), etc.) for each control qubit. Thereafter, the bespoke QFT as another example of a higher-level, composed operation is applied to the circuit, before the quantum state is measured.

Therefore, utilizing more abstract design concepts enables to hide the low-level realization and also promotes flexibility and complexity reduction. Furthermore, turning from elementary quantum operations to such a higher-level design perspective also lowers the entry barriers for non-quantum computing experts. Within this process towards higher abstraction and automation in the design of quantum software, it seems reasonable to apply the lessons learned from decades of research on classical software engineering to the field of quantum computing in order to avoid repeating

\(^1\)https://qiskit.org
\(^2\)https://quantumai.google/cirq
\(^3\)https://docs.microsoft.com/en-us/azure/quantum/overview-what-is-qsharp-and-qdk
\(^4\)https://aws.amazon.com/braket/?nc1=h_ls

Manuscript submitted to ACM
the evolution on the software side. Furthermore, due to its nascent character, the field is widely lacking commonly accepted standards which calls for high levels of flexibility and extensibility of the designed software artifacts.

In this work, we build on existing knowledge from the foundations of Model-Driven Engineering (MDE) [10], and Software Language Engineering (SLE) [16] and transfer it to QSE. We present an extensible language for creating quantum circuits which goes beyond the basic concepts at the qubit level and an according modeling framework which we term Composition-Based Quantum Circuit Designer (CoQuaDe). The proposed approach allows to generate modelling environments which support a high-level quantum circuit design by the use of composite operations. These composite operations may represent specific oracles, but also more general, frequently occurring operations like, e.g., amplitude amplification and QFT. The latter kind can be defined dynamically promoting reusability and variation.

The level of abstraction and automation is further increased by accounting for iterative patterns in quantum algorithms as well as automated generation of quantum operations from classical data. Moreover, the proposed approach is based on a separation between the semantics concerning the quantum circuit itself and the specific quantum operations, which enhances portability and flexibility. Therefore, we present two declarative modeling languages to account for the separation of concerns. Note, that the proposed framework is by design modular concerning the utilized backends, the quantum programming language for lower-level code generation, and the editor that is build on top as a frontend. Therefore, it does not rely on commonly accepted standards in the field, which are still lacking.

Our contributions can be summarized as follows: (i) We provide modelling languages and an according framework for the generation of modelling environments; (ii) we provide a framework that allows for quantum circuit design on a higher-level of abstraction and supported automated code generation; (iii) we demonstrate the proposed approach for two well-known quantum algorithms; (iv) we compare the resulting framework with a state-of-the-art editor for quantum circuits regarding the development effort.

The remainder of this paper is structured as follows. Section 2 presents the related work. Section 3 presents an overview of the proposed framework. Details on its prototypical implementation are provided in Section 4 and Section 5. In Section 6, we demonstrate the proposed approach using the realization of the Quantum Counting algorithm [51] and the Quantum Approximate Optimization Algorithm (QAOA) [25]. We conclude the paper and provide future research directions in Section 7.

2 RELATED WORK

Many vendors of quantum computing provide quantum programming languages and software development kits (e.g., IBM’s Qiskit, Google’s Cirq, Microsoft’s Q#, Amazon’s Braaket). Furthermore, vendor-agnostic tools have emerged for higher portability (e.g., XACC [48], Project Q [63], QuantumPath [35]) with an steadily increasing number of upcoming tools.
Concerning graphical editors, the IBM Quantum Composer\(^5\) provides a set of customizable tools that allow to build, visualize and run quantum circuits, where a direct code generation to OpenQASM 2.0 and Qiskit is supported. Similar features are offered within the QI Editor in Quantum Inspire [46], and the QPS quantum circuit modeler which supports circuit execution on multiple platforms\(^6\). The Quirk\(^7\) graphical modeler on the other hand comes with a large set of applicable gates and also allows to create composite operations, but does not provide automatic code generation from the built circuit. The QuAntiL\(^8\) circuit transformer enables the translation of a given circuit into different languages as well as modifications on a qubit and gate level of abstraction. Available graphical quantum circuit editors are summarized and evaluated in Table 1 regarding their features of

- automatically generating code from the built quantum circuit (F1), and
- the possibility to define composite gates (F2).

<table>
<thead>
<tr>
<th>Graphical Editor</th>
<th>F1</th>
<th>F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Quantum Composer [15.07.2022]</td>
<td>✔</td>
<td>∼</td>
</tr>
<tr>
<td>QI editor [v1.0]</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>QPS modeler [0.9.53]</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Quirk [v2.3]</td>
<td>✗</td>
<td>∼</td>
</tr>
<tr>
<td>QuAntiL [v1.0.1]</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>

In Table 1, F1 has been evaluated as ✔ if at least one code generator is provided. The support of composite gates has to comprise the possibility of defining such gates in a manner which is independent of the number of qubits, besides a plain static definition, in order to be evaluated as ✔. The support of pure static definitions, which would be sufficient for a certain oracle but not e.g., for the general QFT, results in a ∼. From Table 1 it can be seen that the majority of available graphical editors does not support composite gate definitions. Particularly, when it comes to such convenient definitions of custom blocks, and other higher-level functionalities of quantum algorithm design, graphical editors are inferior to available textual solutions.

Continuing with such non-graphical solutions for quantum circuit manipulation, QUANTIFY [53] is an open-source framework for the analysis, verification, and optimization of quantum circuits based on Google Cirq. It offers the choice between different Toffoli gate decompositions and semi-automatic circuit modification methods. The Quantum Algorithm Design (QAD) platform of Classiq\(^9\) focuses on the automatic synthesis of complete quantum circuits from high-level textual inputs. From such high-level models and user-defined constraints, the engine generates code in lower level programming languages (e.g., Qiskit, Cirq, Q#) for the execution on a quantum machine. With a focus on building higher level workflows, the Zapata Orchestra\(^10\) software tool allows to orchestrate quantum- as well as classical programs for real-world applications where also quantum annealing facilities may be utilized.

The application of software engineering methods and principles from MDE to the field of QC has been discussed several times in the literature. In this regard, modeling approaches for the design of quantum software have been suggested, e.g., by Pérez-Delgado et al. [56] who proposed a Unified Modeling Language (UML) [52] extension to allow

---

6https://quantum-circuit.com/docs
7https://algassert.com/quirk
9https://www.classiq.io
10https://www.zapatacomputing.com/orquestra-platform

Manuscript submitted to ACM
for the addition of basic quantum elements. Furthermore, the use of UML-profiles has been suggested by Pérez-Castillo et al. [54]. In contrast, Ali et al. [1] developed a conceptual model of quantum programs, whereas in previous work we presented a domain-specific language for the development of hybrid algorithms [27]. Finally, the role of MDE for software modernization towards quantum software has been investigated [38, 55], and it has also been discussed and envisioned in the context of Model-Driven Architecture [49]. Finally, we would like to mention reviews on quantum programming frameworks (e.g., [25, 45, 62]) and quantum software engineering in general [67]. Overall, there exists a variety of graphical as well as non-graphical solutions for the manipulation of quantum circuits where only the latter kind promotes high-level design features and automation. Furthermore, first attempts have been made in applying the principles of MDE to the field of QC. In this work, we continue this line of research and provide an extensible modeling language together with a modeling framework which (i) allows for a flexible and convenient definition and application of composite operations including iterative patterns, and (ii) provides automated code generation. Besides that, the proposed approach also comes with a separation between the quantum circuit syntax and the definitions of the quantum operations which allows to build and use customized libraries.

3 OVERVIEW ON COMPOSITION-BASED QUANTUM CIRCUIT DESIGNER

This section describes the proposed approach to develop modelling environments for quantum circuits. Figure 2 provides a corresponding overview. The approach allows the quantum language designer to extend the language with a set of quantum operations with code generation facilities (label 1), such as elementary quantum gates (e.g., Hadamard and RZ), state preparation operations (e.g., reset gates), measurement (e.g., in computational basis), composite quantum gates (e.g., amplitude amplification and oracles), and iterative quantum operations. These quantum operations may
be provided within specific libraries, e.g., for quantum chemistry, optimization, or machine learning. The quantum language designer can extend the quantum modelling language with as many quantum operations as required.

After the customization of the quantum operations, the framework is able to automatically synthesize a custom modelling environment for quantum circuits (label 2). In this way, the quantum language users can design quantum circuits with the quantum operations defined by the designer of the quantum language (label 3).

When the user has completed designing the quantum circuits, the framework will be able to automatically generate the artifacts (label 4), to execute these circuits on a specific quantum platform (label 5).

In the following, we describe the proposed language (Section 4) as well as the tool support (Section 5) to realize the overall framework structured in Figure 2 in more detail.

4 QUANTUM CIRCUIT MODELLING LANGUAGE

The proposed approach, comes with the separation of the quantum operation definitions, from the quantum circuit syntax. Therefore, first the meta-model for the quantum circuit design is introduced (Section 4.1), before we continue with a description of the quantum library which comprises the bespoke definitions of quantum operations (Section 4.2). Then, we provide information on certain implemented quantum operations (Section 4.3), and an extension for classical problem-specific inputs for operation definitions (Section 4.4). Finally, we show how quantum circuits can be represented using the proposed framework with a simple example (Section 4.5).

4.1 Quantum circuit meta-model

The meta-model for the proposed language is depicted in Figure 3, by using an object-oriented meta-modelling language. The representation of the language is structured into (i) classes which regard definitions of the quantum circuit itself, i.e., excluding the quantum gates, and (ii) classes regarding the quantum operations which are applied to the circuit. The language for the quantum circuit design is inspired by current functionalities of state-of-the-art software development kits for quantum computing (e.g., Qiskit), fundamental quantum information theory [51], as well as identified patterns in quantum computing11.

The QuantumCircuit may contain Registers, either of QuantumRegister or ClassicRegister type. Indeed, the quantum circuit should contain at least one QuantumRegister. This restriction is defined through an OCL constraint [12]. The possibility of having multiple QuantumRegisters in a QuantumCircuit allows a conceptual separation of qubits according to their function, and should simplify the procedure of merging and partitioning of quantum circuits.

Furthermore, a QuantumCircuit consists of multiple Layers, reflecting the sequenced nature of quantum computation. One Layer may include QuantumOperations, which may take controlQubits but take at least one targetQubit. The selection of qubits happens via the Selector class with a combination of ElementSelector, referring to single qubits, and RangeSelector, referring to a range of qubits (e.g., from 0 to 5). The reference to the abstract Register class allows to address different QuantumRegisters.

Regarding the QuantumOperation, stating one controlQubit means that the respective gate is converted to its single-controlled version, whereas a size of controlQubits, which is greater than 1, results in a multi-controlled gate. Furthermore, this class takes the inverseForm attribute, which causes a transformation to the inversed form of a given quantum operation if set to True. A QuantumOperation may be further conditioned on a ClassicControl object, which in turn has a reference to the binary value of a selected single classical bit, or the binary encoded value of a selected ClassicRegister.

11https://patterns.platform.planqk.de/pattern-languages/af7780d5-1f97-4536-8da7-4194b093ab1d

Manuscript submitted to ACM
Furthermore, the relation to the Operation class serves as the link to the definition of the concrete quantum operation as described in Section 4.2, as well as classical information inputs in Quadratic Unconstrained Binary Optimization (QUBO) form as described in Section 4.4.

We made a distinction of different kind of QuantumOperations such as ElementaryQuantumGate, Measurement, StatePreparation, and CompositeQuantumOperation.

The ElementaryQuantumGate class represents the elementary quantum operations, i.e., single-qubit gates, which may also be parameterized. The three angles theta, phi, and lambda are sufficient to define any elementary qubit rotation in this regard [51]. Specifying multiple targetQubits results in an iterative application of the respective ElementaryQuantumGate to the qubits given by targetQubits. This definition should ease the design of frequently occurring layers, where the same gate is applied to each qubit. Such patterns may be used, e.g., to avoid repeated parameter specification, and for initializing the quantum state to the state of equal superposition [47].

The quantum operations which are irreversible quantum gates by definition are StatePreparation and Measurement operations. These classes may not only comprise common instructions, e.g., resetting qubits to $|0\rangle$ or measuring in the computational basis, but also more general irreversible operations. Examples include the preparation of a certain state which is taken to be given at the beginning of a particular quantum algorithm, or the measurement in a basis other than the computational basis.
The Measurement type of gates additionally require classicBits to save the qubits information. The reference to Register allows for a proper assignment to the specific QuantumRegister and ClassicRegister, respectively. Stating multiple targetQubits and classicBits results in the same iterative application as for the ElementaryQuantumGate.

The CompositeQuantumOperation is a composed gate to aggregate arbitrary elements in its composition. This gate may consist of multiple Layers, representing its decomposed form. These Layers in turn comprise abstract QuantumOperations, which closes the cycle. Note that to avoid infinite loops, a constraint is defined that an operation cannot admit a layer which contains an operation equals to any of the parent operations.

The CompositeLoopQuantumOperations enables to represent iterative patterns as a single composite quantum operation. Such iterative patterns occur frequently, e.g., in VQAs [9, 23, 57], Quantum Arithmetics [40], Shor’s Algorithm [6], or QPE and QFT [51]. The CompositeLoopQuantumOperation requires some additional references to Selector for specification. The fixedTargetQubits and fixedControlQubits specify the qubits which serve as target- and control qubits of the loop operation, but do not change between the iterations of the loop. The loopTargetQubits and loopControlQubit describe the overall target- and control qubits for the gate which is iteratively applied within the CompositeLoopQuantumOperation. They must not be confused with the targetQubits and controlQubits of the CompositeLoopQuantumOperation itself. In order to ensure high flexibility of the realized concrete CompositeLoopQuantumOperations, the class in the meta-model of the quantum circuit has several attributes. Depending on the required functionality of the respective concrete CompositeLoopQuantumOperation, these attributes are internally handled in different ways and are therefore further illustrated in Section 4.3.

Additional restrictions to prohibit errors when using the proposed framework are introduced with OCL constraints [12]. Constraints of this kind ensure (i) that QuantumRegisters do not overlap, and (ii) within a single operation,
A targetQubit must not be a controlQubit at the same time. The latter does not hold true for CompositeLoopQuantumOperations where the bespoke constraint is only required for each iteration but not the whole CompositeLoopQuantumOperation itself.

Note that this meta-model does not have the concrete definition of any quantum gate. This is because we promote a flexible approach to dynamically add QuantumOperations. This requirement is due to the large number of quantum operations and the possibility of working with quantum libraries which may be specifically tailored for certain purposes. Obviously, the use of inheritance to extend the quantum circuit meta-model may be a solution, but this involves the frequent modification of the quantum circuit meta-model. In order to avoid this issue, there are several solutions, such as: the application of the type object pattern [39], multi-level modeling [44], among others. The proposed solution is based on the type object pattern by the use of a library meta-model to define quantum operations dynamically [26].

4.2 Quantum library meta-model

Figure 4 shows the meta-model that describes how to define the concrete quantum operations. The root of this meta-model is the QuantumOperationLibrary which may include several QuantumOperationDefinitions. The latter class takes the Boolean attribute reversible. This attribute ensures that manipulations which are unique to reversible gates, like reversing or controlling, only act on reversible quantum operations. To introduce the required restrictions, we use OCL constraints. The reference to the class itself (reverse) allows to easily define the inversed form of a certain quantum operation. Setting certain values for targetQubits or controlQubits allows to fix the number of qubits in the gate definition. Therefore, the proposed framework allows to define QuantumOperations either for an arbitrary or fixed number of qubits. The former is preferable in terms of reusability because the defined operation is independent of the number of qubits it should act on. The latter on the other hand is required for specific quantum operations, e.g., oracles, which are defined only for a certain application.

A QuantumOperationDefinition may be either a ConcreteLoopOperation or a ConcreteQuantumOperation. The ConcreteLoopOperations within the QuantumOperationLibrary may make use of several attributes, which are specified by the according allows~-Booleans (cf. Figure 4). These attributes have been chosen to allow a high degree of expressiveness concerning the possible specific operations. However, to avoid an extensive list of sparsely used attributes, these may be internally handled in different ways by the different ConcreteLoopOperations. Examples hereof are shown in Section 4.3. Furthermore, the number of loopTargetQubits, loopControlQubits, fixedTargetQubits, and fixedControlQubits can be fixed to certain integer values in the definition of the ConcreteLoopOperation.

The ConcreteQuantumOperation takes a Boolean which denotes whether a classical input in QUBO-form is allowed for the creation of the respective ConcreteQuantumOperation. Furthermore, for Measurement operations, the number of classicBits may be fixed analogously to the targetQubits and controlQubits for the QuantumOperationDefinition. The restriction, that classicBits must not be stated for operations other than Measurements, is again realized with an OCL constraint. Finally, a ConcreteQuantumOperation which represents a parameterized gate, can take three angle parameters (theta, phi, lambda) for its definition.

4.3 Implemented CompositeLoopQuantumOperations

In the following, the three currently implemented concrete CompositeLoopQuantumOperations are described. Whereas two of them (StaticLoop, Power2Loop) allow for a high-level realization of frequently occurring patterns in quantum circuits, the third one (GeneralLoop) is designed to be more expressive in order to realize also highly specific loop
patterns. The description of their usage and the implemented CompositeQuantumOperations will follow in Section 6 as the latter are more specific to the provided use cases compared to the CompositeLoopQuantumOperations.

The first operation is the StaticLoop which represents an iterative application of certain QuantumOperations where the targetQubits and controlQubits for the applied gates do not change between iterations. It allows iterations, i.e., the number of times the gates are appended to the QuantumCircuit. It shall be further noted, that the StaticLoop is the only implemented CompositeLoopQuantumOperations that allows multiple QuantumOperations as input (allowsMultipleOperations=True). All other CompositeLoopQuantumOperations-specific parameters (allows*) are False.

The second CompositeLoopQuantumOperations is the Power2Loop, which is useful to realize loop patterns as they occur, e.g., within QPE, QFT, Quantum Arithmetics, and Shor’s Algorithm. Here, the respective gate is applied 2^x times, with x ∈ N, to fixed targetQubits and the controlQubit changes in each iteration. Within each iteration of the Power2Loop, the StaticLoop is utilized for the repeated applications to unchanged qubits. The following additional parameters specify the Power2Loop:

- incrementControlQubits: A Boolean which specifies whether the controlQubit is incremented or decremented between successive iterations.
- incrementTargetQubits: A Boolean which specifies the number of gate applications for each iteration. Here, True results in an increasing number of gate applications for each controlQubit, i.e., in the first iteration the single controlled gate is appended 2^0 times and in the last (z-th) iteration 2^{z-1} times, where z is given by the number of stated controlQubits. Analogously, False reverses the number of applications starting with 2^{z-1} for the first and 2^0 for the last iteration and controlQubit, respectively.

The StaticLoop and Power2Loop already cover iterative patterns of quantum algorithms, as they occur, e.g., within VQAs [9, 23, 57], or QPE and QFT [9]. However, to facilitate and provide higher expressiveness, we implemented a third, more exhaustive CompositeLoopQuantumOperations, called GeneralLoop. This operation allows to realize less well specified loops as they occur, e.g., in ansätze for VQAs or Quantum Arithmetics. To avoid an excessive amount of parameters, those are internally handled in different ways even within distinct forms of the GeneralLoop as described bellow. By investigating various loop patterns (e.g., from [40], [51], the PlanQK Pattern Atlas\textsuperscript{12}, the Qiskit Textbook\textsuperscript{13}) we figured out the following minimum set of additional parameters:

- targetQubitsIterationType: Qubits can change according to different schemes between iterations. SHIFT causes a block of targetQubits to be shifted by incrementBy after each iteration. The size of the qubit-block and whether the shift happens in an incremental or decremental manner is specified by parameters that are discussed bellow (targetQubitsBlockSize, incrementTargetQubits). In the CHANGE_BLOCK method, incrementBy qubits are added or removed from the targetQubits. Details of this change are described bellow. Lastly, NONE keeps the targetQubits without any changes between iterations.
- controlQubitsIterationType: Same as with targetQubitsIterationType but for the control qubits of the gate applications.
- targetQubitsBlockSize: This parameter is handled differently in the SHIFT and CHANGE_BLOCK method. In the SHIFT method, it specifies the size of the block of targetQubits that is shifted. In the CHANGE_BLOCK method, it denotes the minimal amount of targetQubits. For example, if the stated targetQubits for the gate application are (0, 1, 2, 3, 4) and the number of qubits should be reduced in each iteration, targetQubitsBlockSize= 2 would

\textsuperscript{12}https://patterns.platform.planqk.de/pattern-languages/af7780d5-1f97-4536-8da7-4194b093ab1d
\textsuperscript{13}https://qiskit.org/textbook/preface.html
result in a loop of four iterations where the targetQubits of the last iteration are (0, 1) (provided incrementBy= 1, and incrementBlockTargetQubits= False and incrementTargetQubits= False as described below).

- controlQubitsBlockSize: Same as with targetQubitBlockSize but for the control qubits of the gate applications.
- incrementBlockTargetQubits: A Boolean which specifies whether a block of targetQubits should be incremented or decremented between iterations, i.e., whether targetQubits are added to or removed from the block. It can only be stated for the CHANGE_BLOCK method as the block size remains constant in the SHIFT method. Together with the Boolean incrementTargetQubits it specifies the four possible variants of how the block of qubits is modified.
- incrementBlockControlQubits: Same as incrementBlockTargetQubits but for the control qubits of the gate applications.
- incrementTargetQubits: A Boolean which denotes whether targetQubits are addressed in a ascending or descending manner. Within the CHANGE_BLOCK method, together with the Boolean incrementBlockTargetQubits it specifies the four possible variants of how the block of qubits is modified. For example, stating targetQubits (0, 1, 2, 3, 4), incrementBlockTargetQubits= True, and incrementTargetQubits= False would yield the following targetQubits for the respective iterations: (4), (3, 4), (2, 3, 4), (1, 2, 3, 4), (0, 1, 2, 3, 4) (provided targetQubitsBlockSize= 1). Within the SHIFT method this parameter simply specifies whether the targetQubits are increased (e.g., (0, 1), (1, 2), (2, 3), (3, 4)) or decreased (e.g., (3, 4), (2, 3), (1, 2), (0, 1)).
- incrementControlQubits: Same as incrementTargetQubits but for the control qubits of the gate applications.
- fixedTargetQubits: A subset of targetQubits for the gate applications which denote the qubits that remain the same for each iteration. The gates are applied to those qubits but the qubits do not change between iterations, i.e., they are not considered in the SHIFT or CHANGE_BLOCK method.
- fixedControlQubits: Same as fixedTargetQubits but for the control qubits of the gate applications.
- iterations: The number of iterations that should be applied. In contrast to the StaticLoop, this parameter is not mandatory. As default, our tool would automatically determine the maximum number of iterations possible based on the stated parameters.

4.4 Extension for QUBO-inputs

The features of the proposed approach described above allow for the design of quantum circuits that may be parameterized. Therefore, in principle, circuits for VQAs can be implemented. However, the ansatz of a VQA may not be fixed, as for example the hardware-efficient ansatz of VQE [42], but rather be defined by problem-specific information like, e.g., the cost function in the case of QAOA [23]. In order to automate the creation of ConcreteQuantumOperations based on this problem-specific input, the framework is extended at the meta-model level with the Operation class (cf. Figure 3) and the additional allowsQubo parameter for ConcreteQuantumOperations (cf. Figure 4). The Operation class serves as the link for the cost function input in QUBO-form, i.e., a matrix where the entries represent the coefficients of the cost function.

Note that the described extension is rather specific to QAOA and combinatorial optimization problems, whereas the features of the proposed framework described in the previous sections are more generally applicable. Nevertheless, the former is included in the framework to allow the creation of parameterized quantum circuit for QAOA, which represents a prominent VQA [9], at a high level of automation and abstraction. It should be highlighted that VQAs, which do not require problem-specific information in their ansatz definition, can be represented with the proposed framework without the described extension for QUBO-inputs.
Overall, the proposed approach promotes (i) abstraction by hiding low-level gates, (ii) variation due to the possibility of a flexible definition of CompositeQuantumOperations and of having multiple targetQubits and controlQubits, (iii) composition with the concept of CompositeQuantumOperations and CompositeLoopQuantumOperations, and (iv) library support by the use of the type object pattern. In the next section, we demonstrate these features with a simple example.

### 4.5 Representation of quantum circuits

The chosen example to demonstrate the application of the proposed approach is the standard circuit to generate the GHZ-state [30]. This fully entangled state is important, e.g., for distributed quantum information processing and quantum communication [21]. Taking the quantum circuit for generating the GHZ-state for 5 qubits (Figure 5), the required quantum operations comprise a Hadamard gate on the first qubit, followed by a series of single-controlled Pauli-X gates (CNOTs). Therefore, this minimal example comprises elementary quantum gates (Hadamard), as well as iterative components (CNOTs).

The according instructions to implement this circuit with the proposed framework are given in Listing 1. The QuantumCircuit contains one QuantumRegister with five qubits, and two Layers. The first Layer contains an ElementaryQuantumGate, specifically the Hadamard gate (ConcreteQuantumOperation) which acts on the first qubit (targetQubits [0]). In the second layer, the CNOT gates are implemented using the concrete GeneralLoop operation, which acts on the whole quantum circuit (targetQubits [0-4]). The required parameters for the loop result from its definition as a ConcreteLoopOperation with the according allows* statements, where only non-default values for these parameters have to be stated by the user. The CNOTs inside the GeneralLoop have control qubits 0-3 (loopControlQubits) and target qubits 1-4 (loopTargetQubits). Because the CNOT only takes one control qubit and target qubit, blocks of targetQubitsBlockSize=1 and controlQubitsBlockSize=1 are applied, where the selected qubits are SHIFTed in each iteration (targetQubitsIterationType, controlQubitsIterationType). Here, the incrementTargetQubits and incrementControlQubits statements result in an ascending shift of qubits with each iteration. Note that the chosen example solely serves to demonstrate the application of the proposed framework to a very minimal example. Some of the given instructions would not be necessary for a full specification but have been stated to explain the parameters of the GeneralLoop. More sophisticated demonstration cases are presented in Section 6.
Listing 1. Implementation of 5-qubit GHZ-state quantum circuit

```
QuantumCircuit GHZ {
  QuantumRegister qr {
    NumberOfQubits 5
  }
  Layer L1 {
    ElementaryQuantumGate {
      operation Hadamard
      targetQubits [0]
    }
  }
  Layer L2 {
    CompositeLoopQuantumOperation {
      loop GeneralLoop
      targetQubits [(0-4)]
      operations (Pauli-X)
      loopControlQubits [(1-3)]
      incrementControlQubits
      loopTargetQubits [(0-3)]
      incrementTargetQubits
      targetQubitsBlockSize 1
      controlQubitsBlockSize 1
      targetQubitsIterationType SHIFT
      controlQubitsIterationType SHIFT
    }
  }
}
```

5 TOOL SUPPORT

We implemented the proposed approach, called CoQuaDe, atop of the Eclipse Modeling Framework (EMF) [64] as an Eclipse plug-in available at: https://github.com/jku-win-se/composition-quantum-circuit. The meta-models introduced above are implemented in Ecore, which is the meta-modeling language provided by EMF. In addition, we also built a textual editor for quantum circuits atop of Xtext [8], which is a framework compatible with EMF to develop programming languages.

As explained in Section 4, the main objective of designing the library meta-model is due to the fact that the quantum operations can be added dynamically. To do this, we implemented an Eclipse Extension Point [65] in which the developer is able to add ElementaryQuantumGates, CompositeQuantumOperations, StatePreparation, and Measurement operations. Of course, the developer should provide all the data related in order to add a ConcreteQuantumOperation or ConcreteLoopOperation. To demonstrate the feasibility of the approach, we implemented the following operations: Reset (StatePreparation); Measurement in computational basis; Hadamard, Pauli-Z, Pauli-X, Swap, and RZ as ElementaryQuantumGates; a Grover unitary, a general cost unitary and mixing unitary, a QFT gate, as well as two QFT-element gates as CompositeQuantumOperations; and a StaticLoop, Power2Loop, and GeneralLoop as CompositeLoopQuantumOperations.

We demonstrate the feasibility of the resulting tool by implementing two use cases, namely the Quantum Counting algorithm and QAOA, which will be explained in the next section. In both cases, we were able to directly generate Qiskit code from each designed circuit. It should be further highlighted at this point that the proposed approach is modular concerning the lower-level quantum programming language. However, for demonstration purposes we rely on the Qiskit SDK [2] as described bellow.

6 DEMONSTRATION AND EVALUATION

In the following, we will assess the potential of the proposed composition-based approach (CoQuaDe) for reducing the development effort regarding (i) non-parameterized quantum circuits for fault-tolerant quantum computing, as well as
(ii) parameterized quantum circuits for algorithms of the NISQ era (VQAs). Therefore, the following research questions (RQs) will be answered:

- **RQ1:** How are non-parameterized quantum circuits implemented using CoQuaDe?
- **RQ2:** How are parameterized quantum circuits for VQAs implemented using CoQuaDe?
- **RQ3:** What is the succinctness of the proposed approach?

To assess **RQ1**, we apply the approach to the QPE algorithm, which is a prominent representative of quantum algorithms for fault-tolerant quantum computation [61], and a central building block of many other quantum algorithms (e.g., HHL algorithm [33], Shor’s algorithm [6]). Specifically, we will treat the Quantum Counting algorithm [51] (cf. Subsection 6.1), which represents an instance of QPE. **RQ2** will be assessed by implementing the QAOA algorithm [23] as a representative of VQAs, where the quantum circuit is parameterized (cf. Subsection 6.2). In contrast to other VQAs (e.g., VQE), in QAOA the concrete form of the circuit is furthermore only specified by additional classical input in QUBO-form. Regarding **RQ1** and **RQ2**, we will propose two alternatives for modelling the respective quantum circuits. Finally, we evaluate the succinctness of the proposed language for both demonstration cases by comparing the number of required actions with the IBM Quantum Composer (**RQ3**). The reason for the latter lies in the design of our language as a declarative one and our intention to build a graphical editor on top of our presented framework in the future. Concerning the latter, we envision our framework as a quantum blended modelling environment [14]. The results of our evaluation are presented and discussed in Subsection 6.3. The IBM Quantum Composer has been preferred over other graphical editors (cf. Section 2) as it supports composite gates and it is well documented and maintained.

Regarding the presented demonstration case implementations, it should be noted that advancing to higher levels of abstraction is always possible, if the according operation definitions are provided. The latter would get arbitrarily specific though, and reusability would be lost. Therefore, we will justify the chosen level of composition for a fair comparison in Section 6.3.

### 6.1 Quantum Counting

The Quantum Counting algorithm outputs the approximate number of solutions $M$ of a given search problem, which is generally unknown in advance. The algorithm basically represents a combination of the Grover iteration with the phase estimation technique based upon the QFT [51]. Being an application of the QPE procedure [51], Quantum Counting estimates the eigenphase of the Grover unitary, with a certain accuracy, and success probability. From the eigenphase, $M$ can be calculated with classical means. The quantum registers for the circuit are made up by counting qubits, where the required number depends on the desired success probability and qubits for implementing the Grover unitary. Next, we illustrate and describe the implemented quantum circuit.

#### 6.1.1 Overview on the Quantum Circuit

The first step in the Quantum Counting algorithm is the state initialization, which consists of Hadamard gates applied to all qubits. The subsequent gates of the circuit represent the QPE algorithm for Quantum Counting via several Grover unitaries which are controlled on the counting qubits, and the inverse QFT on those qubits. One Grover unitary is composed of (i) Hadamards applied to each targetQubit, (ii) a problem-specific oracle, and (iii) an amplitude amplification operation. The repeated application of controlled Grover unitaries with different repeats for different control qubits encodes the phase of this unitary to the control qubits in the Fourier basis via the phase kickback mechanism [51]. The inverse QFT is finally used to translate this information to the computational basis before the state is being measured.

$^{14}$https://quantum-computing.ibm.com/composer/docs/iqx/new
6.1.2 Implementation of the Quantum Circuit. The described demonstration case is taken from the IBM Qiskit Textbook. Such textbook examples serve educational and demonstration purposes very well but come with the disadvantage of using insufficiently small numbers of qubits for realistic applications. Therefore, our evaluation is limited to a demonstration case, where we expect smaller benefits of our approach, compared to large quantum circuits of the same kind.

The generated quantum circuit is depicted for various levels of abstraction in Figures 6-7, which are described next.

The state initialization can be realized with a single Hadamard (ElementaryQuantumGate) which takes all qubits from the circuit as targetQubits (label 1).

For the subsequent phase encoding via repeated applications of the controlled Grover unitary, the Power2Loop has been utilized (label 2). Here, incrementControlQubits as well as incrementTargetQubits has been set to True. The Grover unitary itself has been implemented as a ConcreteQuantumOperation with a fixed number of targetQubits= 4, where stating one controlQubit results in a single controlled version of the respective CompositeQuantumOperation.

The inverse QFT has been implemented for two alternatives. Regarding the first one, the swap and rotation part are implemented separately (Alternative 1). For this purpose, the GeneralLoop operation has been utilized to generate the swap block (Figure 6, 7: label 3) with the Swap gate (ElementaryQuantumGate) as the applied gate and the attributes of the CompositeLoopQuantumOperations being specified as given in Listing 2. No fixedControlQubits, fixedTargetQubits, and Iterations have been defined. Next, the GeneralLoop is again used to realize the rotations (Figure 6, 7: label 4) within the inverse QFT. The gate, which is iteratively applied four times within the loop, is given by the implemented QFT_Element (CompositeQuantumOperation). It shall be noted, that two versions for this composed gate are possible: first, as an object which just utilizes concepts and methods from the Qiskit SDK [2] in its definition, and second as an object which relies on the concepts of our proposed approach (e.g., Power2Loop) in its definition. With the required

Fig. 6. High level view of generated quantum circuit for Quantum Counting (Alternative 1); visualization conducted with [2]

Fig. 7. First order decomposition of generated quantum circuit for Quantum Counting (Alternative 1); visualization conducted with [2]

\[\text{https://qiskit.org/textbook/ch-algorithms/quantum-counting.html} \]

Manuscript submitted to ACM
Listing 2. Implementation of Layer 3 for Quantum Counting (label 3) using CoQuaDe

```plaintext
Layer L3 {
  CompositeLoopQuantumOperation {
    loop GeneralLoop
    targetQubits [(0-3)]
    operations (Swap)
    loopTargetQubits [(0-1)]
    loopControlQubits [(2-3)]
    incrementControlQubits
    targetQubitsBlockSize 1
    controlQubitsBlockSize 1
    controlQubitsIterationType SHIFT
    targetQubitsIterationType SHIFT
  }
}
```

Listing 3. Implementation of Layer 4 for Quantum Counting (label 4) using CoQuaDe

```plaintext
Layer L4 {
  CompositeLoopQuantumOperation {
    loop GeneralLoop
    targetQubits [(0-3)]
    operations (QFTElement)
    loopTargetQubits [(0-3)]
    incrementTargetQubits
    incrementBlockTargetQubits
    targetQubitsBlockSize 1
    targetQubitsIterationType CHANGE_BLOCK
  }
}
```

CompositeQuantumOperation being specified, the rotation part of the inverse QFT is generated with the attributes for the GeneralLoop as presented in Listing 3 (note that no controlQubits are given for the CompositeLoopQuantumOperation).

Again, no fixedControlQubits, fixedTargetQubits, and Iterations are specified. After their creation, the swap and rotation part of QFT are applied to the counting qubits of the quantum circuit.

An alternative way of obtaining the inverse QFT is possible in case a dedicated CompositeQuantumOperation is provided in the QuantumOperationLibrary, where the attribute inverseForm = True causes a conversion of the original QFT to its inversed version (Alternative 2). The final element of the QuantumCircuit is represented by a single Measurement (label 5) with the counting qubits of the circuit being defined as its targetQubits.

Note that all mentioned CompositeQuantumOperations are defined for an arbitrary number of qubits, and only fully specified when being applied to the circuit with the given targetQubits and controlQubits. The only exception is the Grover unitary, which includes a specific oracle, and is therefore defined as a ConcreteQuantumOperation with a fixed number of targetQubits.

Overall, we implemented a quantum circuit for the Quantum Counting algorithm as an instance of QPE at different levels of abstraction. Within Alternative 1, the inverse QFT gate is explicitly built using our framework, whereas in Alternative 2 we suppose to have a QFT gate provided in the quantum library. Finally, it should be noted that the CoQuaDe is expressive enough to realize dynamic quantum circuits, with the dynamic QPE [11, 17] as one example. However, we refrain from going into the details of treating dynamic quantum circuits at this point, as they are more concerned about efficient low-level implementation and compilation of circuits, rather than high-level functionalities.

---

16 Gemeinhardt, et al.


Manuscript submitted to ACM
6.2 QAOA

The application of VQAs has been shown useful for exploiting the potential of current NISQ devices [9]. Such algorithms take a certain parameterized quantum circuit, called ansatz, where the parameters of the circuit are classically optimized for a particular optimization function. The final output is then obtained based on measurement results from the optimized quantum circuit. One prominent example of VQAs is the QAOA, which has been specifically developed for combinatorial optimization problems. Being inspired by the adiabatic evolution of the quantum system given in quantum annealing [23], QAOA integrates information from the cost function of the optimization problem, for the definition of its ansatz.

6.2.1 Overview on the Quantum Circuit. The parametrized quantum circuit of QAOA comprises two unitaries: the cost unitary and the mixing unitary. The cost unitary is defined by the cost function of the combinatorial optimization problem, which is usually stated as a QUBO problem [29], whereas the mixing unitary does not require further information for its definition. The resulting ansatz, which acts on the quantum system, is given by an alternating application of these two unitaries for a certain number of times. It should be noted, that there are multiple adaptations to the original QAOA, which may either address the cost unitary (e.g., [66]) or the mixing unitary (e.g., [31]). In its original version, with the choice of the mixing unitary mentioned above, the initial state of the quantum system is represented by the state of equal superposition.

6.2.2 Implementation of the Quantum Circuit. Again, the investigated demonstration case is based on the small example provided in the IBM Qiskit Textbook17. In this particular case, the combinatorial optimization problem takes only four variables, resulting in a quantum circuit size of four qubits. The implemented circuits are depicted at different levels of abstraction in Figure 8-9. The implementation of the quantum circuit for QAOA is presented in Listing 4.

In order to realize the described circuit with our framework, the first step is to create the initial state. This happens again by applying a Hadamard gate (Listing 4: Layer L1) with all qubits defined as targetQubits (Figure 8-9: label 1). Thereafter, the cost and mixing unitary have to be specified. As described above, the cost unitary can be built based on the cost function coefficients. In order to automate this process for arbitrary coefficients, we make use of the language extension described in Section 4.4. The output of the routine is a ConcreteQuantumOperation representing the cost unitary that is automatically stored to the QuantumOperationLibrary. Using this routine, therefore, relieves the user from the knowledge of how to build the respective unitary based on the problem information. The mixing unitary for the original QAOA, due to its generality, is supposed to be readily available as a CompositeQuantumOperation in

17https://qiskit.org/textbook/ch-applications/qaoa.html
the used library. At this point, it is possible to proceed in different ways. First, the new cost unitary and the mixing unitary can be applied to a `QuantumCircuit`, which is subsequently stored. This `QuantumCircuit` can now be used like a `ConcreteQuantumOperation` within the `StaticLoop` to be iterated for a specified number of times (Alternative 1). Alternatively, one can pass a list of `QuantumOperations` to the `StaticLoop` (label 2) and thereby circumvent the additional step of creating an intermediate `Quantum Circuit` (Alternative 2). The latter alternative is represented in `Layer L2` of Listing 4. Finally, the measurement is conducted by a single `Measurement` gate (Listing 4: `Layer L3`) with multiple `targetQubits` (label 3).

In summary, a quantum circuit for QAOA can be implemented in two alternative ways. Within the first, an intermediate `QuantumCircuit` is created, stored, and subsequently iteratively applied to the main circuit. The second alternative does not require this intermediate step and allows for a direct application of the respective unitaries.
6.3 Evaluation and Discussion

We compare the proposed approach to the IBM Quantum Composer. Therefore, we evaluate the development effort to design quantum circuits for QPE and QAOA. The comparison of textual and graphical declarative languages requires an according metric to measure the development effort. For this purpose, we interpret the declaration of a quantum circuit as an attributed typed graph [34]. Based on this representation, the required number of actions taken by the user is defined as the sum of (i) created objects, (ii) user-specified non-default attributes, and (iii) links between objects.

Regarding the quantum circuit for QPE, we have chosen a level of composition where still only (i) unspecified and generally applicable CompositeLoopQuantumOperations, and (ii) frequently occurring composite gates are utilized. One example of the latter is the QFT gate, which is an integral part of the HHL algorithm [33], Shor’s algorithm [6], and QPE [51]. The problem-specific, non-reusable Grover unitary represents the only necessary exception to the statement above. Therefore, we analogously build this unitary in advance with the IBM Quantum Composer and view its generation and application just as two actions to ensure a fair comparison. We conducted analogously with elementary quantum gates that are not supported by the IBM Quantum Composer to avoid artificially high number of actions in its evaluation. We want to highlight at this point, that the creation of controlled composed gates is currently not supported by the IBM Quantum Composer. It is only feasible by utilizing OpenQASM code, which is generated in advance with the Qiskit SDK. In contrast, the CoQuaDe allows for a very simple application of composite gates in their controlled version.

Concerning the quantum circuit for the QAOA algorithm, the situation is slightly different. Besides the generally applicable StaticLoop, we utilize two unitaries which are specific to the standard version of the QAOA algorithm: the cost unitary and the mixing unitary. The former, is only specified given the QUBO-input as described in Section 4.4, whereas the latter is independent of the optimization problem at hand. Adaptations to the original QAOA, which regard different cost and mixer unitaries are a field of active research (e.g., [4, 32, 59, 60, 66, 68]). Therefore, we aim to build a QuantumOperationLibrary specifically for quantum combinatorial optimization, with the two implemented unitaries as a starting point. Further included quantum operations may comprise adaptations to the standard QAOA, but also unitaries for other VQAs (e.g., VQE) and non-VQAs (e.g., Grover Adaptive Search [22, 28]). In contrast to the QPE circuit, for QAOA we counted the required actions for the composite gate definitions in the implementation with the IBM Quantum Composer. The results of the evaluation are summarized in Table 2. It has to be considered, that the illustrated demonstration cases represent small examples of quantum circuits.

Discussion. In summary, using the CoQuaDe we were able to develop quantum circuits for QPE (RQ1) as well as QAOA (RQ2) for different alternatives. Regarding RQ3, the required numbers of actions for these two demonstration cases could be reduced by 72% (QAOA) and 29% (QPE) compared to the state-of-the-art. Further scaling advantages are supposed for larger quantum circuits. Here, the utilization of composite gates results in a constant scaling of required actions using the CoQuaDe, whereas the scaling for the IBM Quantum Composer would be at least linear, depending on the specific composite gate. Therefore, using the CoQuaDe allows for quantum circuit design on a higher-level of abstraction with a significantly reduced development effort.

Table 2. Required number of actions (#objects/#links/#non-default parameters/total)

<table>
<thead>
<tr>
<th>Circuit</th>
<th>IBM Quantum Composer</th>
<th>CoQuaDe</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPE (Alt.2)</td>
<td>35/39/9/83</td>
<td>32/21/6/59</td>
</tr>
<tr>
<td>QAOA (Alt.2)</td>
<td>42/50/16/108</td>
<td>18/9/3/30</td>
</tr>
</tbody>
</table>
Limitations. It should be noted, that the evaluated demonstration cases represent prominent and sophisticated examples of non-parameterized as well as parameterized quantum circuits, respectively. Nevertheless we cannot generalize our findings regarding the implementation possibilities to arbitrary quantum circuits of the bespoke kinds.

7 CONCLUSION AND FUTURE WORK

We presented a composition-oriented modeling language for creating quantum circuits. By incorporating concepts which go beyond the qubit-level of software design, the proposal provides the use of composed quantum operations and automated code generation from the built quantum circuits. This allows to hide low-level implementation details in the design of such circuits. Furthermore, we have demonstrated the feasibility and succinctness benefits of the proposed approach via the application to the Quantum Counting algorithm and QAOA. We found significantly reduced development efforts compared to using existing state-of-the-art quantum circuit designers.

Future Work. The proposed approach, being work in progress, offers several immediate extension possibilities. In the future, we will explore frameworks like the Eclipse Sirus or JavaFX for the implementation of a graphical editor for our presented approach. In this sense, we plan to provide a quantum blended modelling environment build atop of the presented quantum languages [14]. In addition, we plan to enable the import of quantum circuits and subsequent manipulation of the circuit with our framework.

Furthermore, the repertoire of quantum operations will be extended in the future to cope with more advanced quantum circuits. In this regard, we aim to build a library for quantum operations specifically for the purpose of quantum combinatorial optimization as described in Section 6.3. This will allow for fast experimentation with different variational and non-variational solution approaches. Concerning VQAs, we intend to add the possibility of stating initial parameters for the generated parametrized quantum circuit.

The proposed model will also be extended for higher-level circuit design and optimization. In this regard, a first step will be to include facilities for automated quantum operator discovery, utilizing techniques from genetic programming and reinforcement learning. Here, the goal is to automatically create a CompositeQuantumGate that yields a certain target output state. Furthermore, the circuit synthesis may comprise model-based circuit aggregation and partitioning [19], and the framework may incorporate generic as well as NISQ-specific circuit optimization procedures (e.g., [50]). Applying concepts from MDE also allows to use well-known model-based transformation tools [41] for quantum circuit transformations to different representations. The later are required, for example, when using the ZX-calculus [43] and the LOv-calculus [15]. Finally, as the bespoke procedures may produce errors, a subsequent verification step [37] might be necessary to guarantee that the resulting quantum circuits are correct.

ACKNOWLEDGMENTS

Financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development and by the Austrian Science Fund (P 30525-N31) is gratefully acknowledged.

DATA AVAILABILITY

All code and data is available at: https://github.com/jku-win-se/composition-quantum-circuit. In this repository, we published both explained meta-models and the implementation of the demonstration cases.

Manuscript submitted to ACM
A Model-Driven Framework for Composition-Based Quantum Circuit Design

REFERENCES


